Information Retrieval

INFO 4300 / CS 4300

- Indexing
  - Inverted indexes
  - Compression
  - Index construction
  - Ranking model

Index Construction

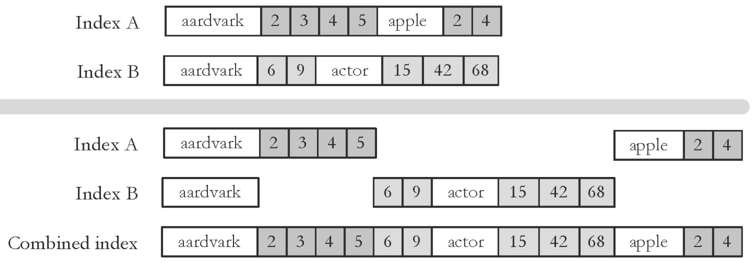
- Simple in-memory indexer

```
procedure BUILDINDEX($D$)                    ▷ $D$ is a set of text documents
    $I \leftarrow$ HashTable()                    ▷ Inverted list storage
    $n \leftarrow 0$                              ▷ Document numbering
    for all documents $d \in D$ do
        $n \leftarrow n + 1$
        $T \leftarrow$ Parse($d$)                ▷ Parse document into tokens
        Remove duplicates from $T$
        for all tokens $t \in T$ do
            if $I_t \notin I$ then
                $I_t \leftarrow$ Array()
            end if
            $I_t$.append($n$)
        end for
    end for
    return $I$
end procedure
```

Merging

- Merging addresses limited memory problem
  - Build the inverted list structure until memory runs out
  - Then write the partial index to disk, start making a new one
  - At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists must be designed so they can be merged in small pieces
  - e.g., storing in alphabetical order

Merging



Index A: aardvark 2 3 4 5 apple 2 4

Index B: aardvark 6 9 actor 15 42 68

Index A: aardvark 2 3 4 5 apple 2 4

Index B: aardvark 6 9 actor 15 42 68

Combined index: aardvark 2 3 4 5 6 9 actor 15 42 68 apple 2 4

Distributed Indexing

- Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)
- Large numbers of inexpensive servers used rather than larger, more expensive machines
- *MapReduce* is a distributed programming tool designed for indexing and analysis tasks

---

Example

- Given a large text file that contains data about credit card transactions
  - Each line of the file contains a credit card number and an amount of money
  - Determine the number of unique credit card numbers
- Could use hash table – memory problems
  - counting is simple with sorted file
- Similar with distributed approach
  - sorting and placement are crucial

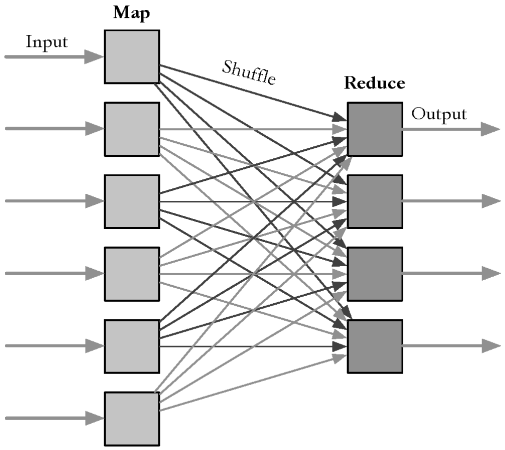
MapReduce

- Distributed programming framework that focuses on data placement and distribution
- *Mapper*
  - Generally, transforms a list of items into another list of items of the same length
- *Reducer*
  - Transforms a list of items into a single item
  - Definitions not so strict in terms of number of outputs
- Many mapper and reducer tasks on a cluster of machines

MapReduce

- Basic process
  - *Map* stage which transforms data records into pairs, each with a key and a value
  - *Shuffle* uses a hash function so that all pairs with the same key end up next to each other and on the same machine
  - *Reduce* stage processes records in batches, where all pairs with the same key are processed at the same time
- *Idempotence* of Mapper and Reducer provides fault tolerance
  - multiple operations on same input gives same output

MapReduce



Example

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure

procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
```

Indexing Example

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(      number :position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```

Result Merging

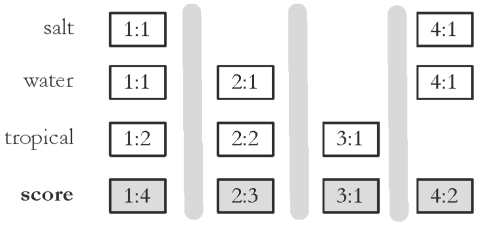
- Index merging is a good strategy for handling updates when they come in large batches
- For small updates this is very inefficient
  - instead, create separate index for new documents, merge *results* from both searches
  - could be in-memory, fast to update and search
- Deletions handled using *delete list*
  - Modifications done by putting old version on delete list, adding new version to new documents index

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

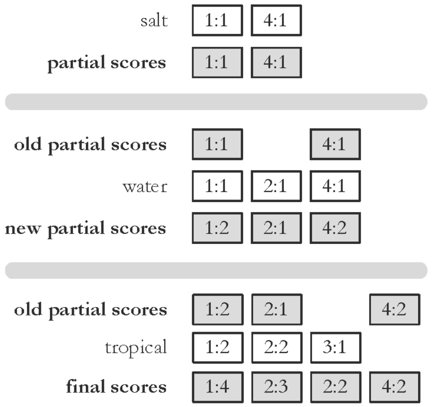


salt	1:1			4:1
water	1:1	2:1		4:1
tropical	1:2	2:2	3:1	
score	1:4	2:3	3:1	4:2

Document-At-A-Time

```
procedure DocumentAtATimeRetrieval($Q$, $I$, $f$, $g$, $k$)
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue($k$)
    for all terms $w_i$ in $Q$ do
        $l_i \leftarrow$ InvertedList($w_i$, $I$)
        $L$.add( $l_i$ )
    end for
    for all documents $d \in I$ do
        for all inverted lists $l_i$ in $L$ do
            if $l_i$ points to $d$ then
                $s_D \leftarrow s_D + g_i(Q)f_i(l_i)$          ▷ Update the document score
                $l_i$.movePastDocument( $d$ )
            end if
        end for
        $R$.add( $s_D, D$ )
    end for
    return the top $k$ results from $R$
end procedure
```

Term-At-A-Time



salt	1:1	4:1		
partial scores	1:1	4:1		
old partial scores	1:1		4:1	
water	1:1	2:1	4:1	
new partial scores	1:2	2:1	4:2	
old partial scores	1:2	2:1		4:2
tropical	1:2	2:2	3:1	
final scores	1:4	2:3	2:2	4:2

Term-At-A-Time

```
procedure TermAtATimeRetrieval($Q$, $I$, $f$, $g$ $k$)
    $A \leftarrow$ HashTable()
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue($k$)
    for all terms $w_i$ in $Q$ do
        $l_i \leftarrow$ InvertedList($w_i$, $I$)
        $L$.add( $l_i$ )
    end for
    for all lists $l_i \in L$ do
        while $l_i$ is not finished do
            $d \leftarrow l_i$.getCurrentDocument()
            $A_d \leftarrow A_d + g_i(Q)f(l_i)$
            $l_i$.moveToNextDocument()
        end while
    end for
    for all accumulators $A_d$ in $A$ do
        $s_D \leftarrow A_d$                ▷ 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
- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists
    - better for simple feature functions
  - Calculate scores for fewer documents
    - e.g., conjunctive processing
    - better for complex feature functions

Conjunctive Term-at-a-Time

```
procedure TermAtATimeRetrieval($Q$, $I$, $f$, $g$, $k$)
    $A \leftarrow$ HashTable()
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue($k$)
    for all terms $w_i$ in $Q$ do
        $l_i \leftarrow$ InvertedList($w_i$, $I$)
        $L$.add( $l_i$ )
    end for
    for all lists $l_i \in L$ do
        while $l_i$ is not finished do
            if $i = 0$ then
                $d \leftarrow l_i$.getCurrentDocument()
                $A_d \leftarrow A_d + g_i(Q)f(l_i)$
            else
                $d \leftarrow l_i$.getCurrentDocument()
                $d \leftarrow A$.getNextDocumentAfter($d$)
                $l_i$.skipForwardTo($d$)
                if $l_i$.getCurrentDocument() $= d$ then
                    $A_d \leftarrow A_d + g_i(Q)f(l_i)$
                else
                    $A$.remove($d$)
                end if
            end if
        end while
    end for
    for all accumulators $A_d$ in $A$ do
        $s_D \leftarrow A_d$                ▷ Accumulator contains the document score
        $R$.add( $s_D, D$ )
    end for
    return the top $k$ results from $R$
end procedure
```

Conjunctive Document-at-a-Time

```
procedure DocumentAtATimeRetrieval($Q$, $I$, $f$, $g$, $k$)
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue($k$)
    for all terms $w_i$ in $Q$ do
        $l_i \leftarrow$ InvertedList($w_i$, $I$)
        $L$.add( $l_i$ )
    end for
    while all lists in $L$ are not finished do
        for all inverted lists $l_i$ in $L$ do
            if $l_i$.getCurrentDocument() $> d$ then
                $d \leftarrow l_i$.getCurrentDocument()
            end if
        end for
        for all inverted lists $l_i$ in $L$ do $l_i$.skipForwardToDocument($d$)
            if $l_i$ points to $d$ then
                $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$          ▷ Update the document score
                $l_i$.movePastDocument( $d$ )
            else
                break
            end if
        end for
        $R$.add( $s_d, d$ )
    end while
    return the top $k$ results from $R$
end procedure
```

Threshold Methods

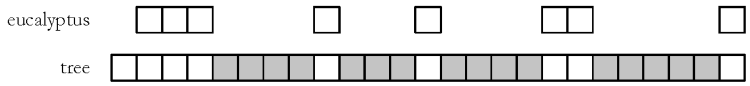
- Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small
- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
  - score of the *k*th-highest scoring document
  - gives *threshold* τ
  - 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'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table
- *MaxScore* method compares the maximum score that remaining documents could have to τ'
  - *safe* optimization in that ranking will be the same without optimization

---

MaxScore Example



- 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

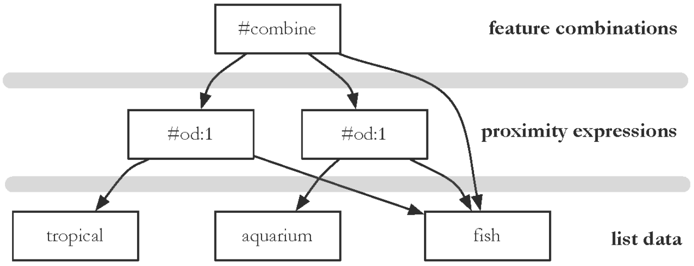
- Early termination of query processing
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization
- List ordering
  - order inverted lists by quality metric (e.g., PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to produce good documents

Structured Queries

- *Query language* can support specification of complex features
  - similar to SQL for database systems
  - *query translator* converts the user’s input into the structured query representation
  - Galago query language is the example used here
  - e.g., Galago query:

`#combine(#od:1(tropical fish) #od:1(aquarium fish) fish)`

Evaluation Tree for Structured Query



Distributed Evaluation

- 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
  - Document distribution
    - by far the most popular
  - Term distribution

Distributed 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 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

Caching

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