6. Efficiency & Scalability
Outline

6.1. Motivation
6.2. Index Construction & Maintenance
6.3. Static Index Pruning
6.4. Document Reordering
6.5. Query Processing
1. Motivation

- Focus in the lecture so far has been on **effectiveness**, i.e., "doing the right things" (e.g., returning useful query results)

- **Efficiency** is about "doing things right", i.e., accomplishing a task using minimal resources (e.g., CPU, memory, disk)

- **Scalability** is about making use of additional resources (e.g., faster/more CPUs, more memory/disk) to accomplish a task
Our focus will be on two major aspects of every IR system:

- **Indexing**: how can we efficiently construct & maintain an inverted index that consumes little space?
- **Query processing**: how can we efficiently identify the top-k results for a given query without having to read posting lists completely?

Other aspects which we will not cover include:

- **Caching** (e.g., posting lists, query results, snippets)
- **Modern hardware** (e.g., GPU query processing, SIMD compression)
Hardware & Software Trends

- CPU speed has increased more than that of disk and memory: faster to read & decompress than to read uncompressed.

- More memory is available; disks have become larger but not faster: now common to keep indexes in (distributed) memory.

- Many (less powerful) instead of few (powerful) machines; platforms for distributed data processing (e.g., MapReduce, Spark).

- More CPU cores instead of faster CPUs; SSDs (fast reads, slow writes, wear out) in addition to HDDs; GPUs and FPGAs.
2. Index Construction & Maintenance

- **Inverted index** as widely used index structure in IR consists of:
  - **dictionary** mapping terms to term identifiers and statistics (e.g., idf)
  - **posting lists** for every term recording details about its occurrences

- How to construct an inverted index from a document collection?
- How to maintain an inverted index as documents are inserted, modified, or deleted?
2.1. Index Construction

- **Observation**: Constructing an inverted index (aka. inversion) can be seen as **sorting a large number of** (term, did, tf) tuples
  - seen in (did)-order when processing documents
  - needed in (term, did)-order for the inverted index

- Typically, the set of all (term, did, tf) tuples does **not fit into the main memory** of a single machine, so that we need to **sort using external memory** (e.g., hard-disk drives)
Lester al. [7] describe the following algorithm by Heinz and Zobel to construct an inverted index on a single machine.

1. Let \( B \) be the number of \((\text{term}, \text{did}, \text{tf})\) tuples that fit into main memory.
2. While not all documents have been processed:
   - Read (up to) \( B \) tuples from the input (documents).
   - Construct in-memory inverted index by grouping & sorting the tuples.
   - Write in-memory inverted index as sorted run of \((\text{term}, \text{did}, \text{tf})\) tuples to disk.
   - Merge on-disk runs to obtain global inverted index.
Index Construction in MapReduce

- **MapReduce** as a platform for **distributed data processing**
  - was developed at **Google**
  - operates on large clusters of **commodity hardware**
  - handles **hard- and software failures** transparently
  - open-source implementations (e.g., **Apache Hadoop**) available
  - programming model operates on **key-value (kv) pairs**
    - **map()** reads input data \((k_1, v_1)\) and emits kv pairs \((k_2, v_2)\)
    - platform groups and sorts kv pairs \((k_2, v_2)\) automatically
    - **reduce()** sees kv pairs \((k_2, \text{list}<v_2>)\) and emits kv pairs \((k_3, v_3)\)
Index Construction in MapReduce

**map**

\[
\text{map}(\text{did, list<term>})
\]

\[
\text{map<term, integer> tfs = new map<term, integer>();}
\]

// determine term frequencies

\[
\text{for each term in list<term>:}
\]

\[
\text{tfs.adjustCount(term, +1);}
\]

// emit postings

\[
\text{for each term in tfs.keys():}
\]

\[
\text{emit (term, (did, tfs.get(term))));
\]

// platform groups & sorts output of map phase by term

**reduce**

\[
\text{reduce}(\text{term, list<(did, tf)>>}
\]

// emit posting list

\[
\text{emit (term, list<(did, tf)>)}
\]
2.2. Index Maintenance

- Document collections are **not static**, but documents are **inserted, modified, or deleted** as time passes; changes to the document collection should **quickly be visible in search results**.

- **Typical approach**: Collect changes in main memory
  - **deletion list** of deleted documents
  - **in-memory delta inverted index** of inserted and modified documents
  - **process queries over both** the on-disk global and in-memory delta inverted index and **filter out** result documents from the deletion list

- **What if the available main memory has been exhausted?**
Rebuild

- Rebuild the on-disk global index from scratch
  - in a separate location; switch over to new index once completed
  - attractive for small document collections
  - attractive when document deletions are common
  - requires re-processing of entire document collection
  - easy to implement
Merge

- **Merge** the on-disk global index with the in-memory delta index
  - in a *separate location*; switch over to new index once completed
  - for each term, **read** posting lists from on-disk global index and in-memory delta index, **merge** them, **filter out** deleted documents, and **write** the merged posting list to disk
  - requires **reading entire on-disk global index**

- **Analysis:** Let $B$ be capacity of the in-memory delta index (in terms of postings) and $N$ be the total number of postings
  - $N / B$ merge operations each having cost $O(N)$
  - total cost is in $O(N^2)$
Geometric Merge

- Lester et al. [5] propose to **partition the inverted index** into index partitions of geometrically increasing sizes
  - tunable by parameter \( r \)
  - index partition \( P_0 \) is in **main memory** and contains up to \( B \) postings
  - index partitions \( P_1, P_2, \ldots \) are on disk with capacity invariants
    - partition \( P_j \) contains at most \( (r-1) r^{(j-1)} B \) postings
    - partition \( P_j \) is either empty or contains at least \( r^{(j-1)} B \) postings
  - whenever \( P_0 \) **overflows**, a **merge** is triggered

- **Query processing** has to access all (non-empty) partitions \( P_i \), leading to **higher cost** due to required disk seeks
Geometric Merge

<table>
<thead>
<tr>
<th>Partition 3</th>
<th>Partition 2</th>
<th>Partition 1</th>
<th>Partition 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1, 2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$r=3$

Geometric Merge

- **Analysis**: Let $B$ be the capacity of the in-memory partition $P_0$ and $N$ be the total number of postings.
  - There are at most $1 + \lceil \log_r(N/B) \rceil$ partitions.
  - Each posting merged at most once into each partition.
  - Total cost is $O(N \log N/B)$. 
Logarithmic Merge

- **Logarithmic merge** is a simplified variant of geometric merge
  - partition $P_0$ is in **main memory** and contains $B$ postings
  - partition $P_1$ is on disk and contains up to $2B$ postings
  - partition $P_2$ is on disk and contains up to $4B$ postings
  - partition $P_j$ is on disk and contains up to $2^jB$ postings
  - whenever $P_0$ overflows, a cascade of merges is triggered

- **Log-structured merge tree** (LSM-Tree) prominent in database systems (e.g., to manage logs) is based on the same principle

- Wu et al. [9] use the same idea in their **log-structured inverted index** to support high update rates when indexing **social media**
3. Static Index Pruning

- **Static index pruning** is a form of **lossy compression** that
  - removes postings from the inverted index
  - allows for **control of index size** to make it fit, for instance, into main memory or on low-capacity device (e.g., smartphone)

<table>
<thead>
<tr>
<th></th>
<th>a: [d_1, 2, d_3, 5, d_7, 2, d_9, 1, d_{11}, 3, d_{13}, 2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b: [d_5, 3, d_7, 2, d_8, 9, d_{11}, 4, d_{15}, 2]</td>
</tr>
<tr>
<td></td>
<td>c: [d_5, 3, d_8, 1, d_{11}, 7, d_{15}, 2]</td>
</tr>
</tbody>
</table>

- **Dynamic index pruning**, in contrast, refers to query processing methods (e.g., WAND or NRA) that avoid reading the entire index
3. Static Index Pruning

- **Static index pruning** is a form of **lossy compression** that
  - removes postings from the inverted index
  - allows for **control of index size** to make it fit, for instance, into main memory or on low-capacity device (e.g., smartphone)

\[
\begin{array}{cccc}
\text{a} & d_3, 5 & d_{11}, 3 \\
\text{b} & d_5, 3 & d_8, 9 & d_{11}, 4 \\
\text{c} & d_5, 3 & d_{11}, 7 \\
\end{array}
\]

- **Dynamic index pruning**, in contrast, refers to query processing methods (e.g., WAND or NRA) that avoid reading the entire index
3.1. Term-Centric Index Pruning

- Carmel et al. [4] propose **term-centric** static index pruning

- **Idea**: Remove postings from posting list for term \( v \) that are **unlikely to contribute** to top-k result of query including \( v \)

- **Algorithm**: For each term \( v \)
  - determine **k-th highest score** \( z_v \) of any posting in posting list for \( v \)
  - remove all postings having a score less than \( \varepsilon \cdot z_v \)

- Despite its simplicity the method guarantees for any query \( q \) consisting of \( |q| < 1 / \varepsilon \) terms a “close enough” top-k result
3.2. Document-Centric Index Pruning


Idea: Remove postings for document $d$ corresponding to non-important terms for which it is unlikely to be in the query result.

Importance of term $v$ for document $d$ is measured using its contribution to the KL divergence from background model $D$.

$$P[v | \theta_d] \log \left( \frac{P[v | \theta_d]}{P[v | \theta_D]} \right)$$

$\text{DCP}_{\text{Const}}$ selects constant number $k$ of postings per document.

$\text{DCP}_{\text{Rel}}$ selects a percentage $\lambda$ of postings per document.
Term-Centric vs. Document-Centric

- Büttcher and Clarke [3] compare term-centric (TCP) and document-centric (DCP) index pruning on TREC Terabyte
- **Okapi BM25** as baseline retrieval model
- **on-disk inverted index**: 12.9 GBytes, 190 ms response time
- **pruned in-memory inverted index**: 1 GByte, 18 ms response time

### [ TREC 2004 Terabyte queries (topics 701-750) ]

<table>
<thead>
<tr>
<th></th>
<th>BM25 Baseline</th>
<th>DCP_{\lambda=0.062}^{\text{Rel}}</th>
<th>DCP_{\text{Const}}^{(k=21)}</th>
<th>TCP_{(n=16000)}^{(k=24500)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@5</td>
<td>0.5224</td>
<td>0.5020</td>
<td>0.4735</td>
<td>0.4490*</td>
</tr>
<tr>
<td>P@10</td>
<td>0.5347</td>
<td>0.4837</td>
<td>0.4755</td>
<td>0.4347*</td>
</tr>
<tr>
<td>P@20</td>
<td>0.4959</td>
<td>0.4490</td>
<td>0.4224</td>
<td>0.4163</td>
</tr>
<tr>
<td>MAP</td>
<td>0.2575</td>
<td>0.1963</td>
<td>0.1621**</td>
<td>0.1808</td>
</tr>
</tbody>
</table>

### [ TREC 2005 Terabyte queries (topics 751-800) ]

<table>
<thead>
<tr>
<th></th>
<th>BM25 Baseline</th>
<th>DCP_{\lambda=0.062}^{\text{Rel}}</th>
<th>DCP_{\text{Const}}^{(k=21)}</th>
<th>TCP_{(n=16000)}^{(k=24500)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@5</td>
<td>0.6840</td>
<td>0.6760</td>
<td>0.6000**</td>
<td>0.5640**</td>
</tr>
<tr>
<td>P@10</td>
<td>0.6400</td>
<td>0.5980</td>
<td>0.5300*</td>
<td>0.5380**</td>
</tr>
<tr>
<td>P@20</td>
<td>0.5660</td>
<td>0.5310</td>
<td>0.4560**</td>
<td>0.4630**</td>
</tr>
<tr>
<td>MAP</td>
<td>0.3346</td>
<td>0.2465</td>
<td>0.1923**</td>
<td>0.2364</td>
</tr>
</tbody>
</table>
4. Document Reordering

- Sequences of non-decreasing integers (here: document identifiers) in posting lists are compressed using:
  - **delta encoding** representing elements as difference to predecessor
    \[ \langle 1, 7, 11, 21, 42, 66 \rangle \rightarrow \langle 1, 6, 4, 10, 21, 24 \rangle \]
  - **bit-wise or byte-wise integer encoding** (e.g., 7-bit encoding or Gamma encoding) representing smaller integers with fewer bits
    \[ 314 = 00000000 \, 00000000 \, 00000001 \, 00111010 \rightarrow 00000010 \, 10111010 \]

- Document reordering methods seek to improve compression effectiveness by assigning document identifiers so as to obtain small gaps
4.1. Content-Based Document Reordering

- Silvestri et al. [10] develop methods for the scenario when **only document contents** are available but no meta-data (e.g., URL)

- **Intuition:** Similar documents, having many terms in common, should be assigned **numerically close document identifiers**

- Documents are modeled as **sets** (not bags) of terms

- Document similarity is measured using the **Jaccard coefficient**

\[ J(d_i, d_j) = \frac{|d_i \cap d_j|}{|d_i \cup d_j|} \]
Top-Down Bisecting

- **Algorithm:** TDAssign(document collection \( D \))
  
  // split \( D \) into equal-sized partitions \( D_L \) and \( D_R \)
  
  pick representatives \( d_L \) and \( d_R \) (e.g., randomly)

  - **if** \(|D_L| \geq |D| / 2\) \( \lor \) \(|D_R| \geq |D| / 2\)
    
    assign \( d \) to smaller partition
  
  - **else if** \( J(d, d_L) < J(d, d_R) \)
    
    assign \( d \) to \( D_L \)
  
  - **else**
    
    assign \( d \) to \( D_R \)

  **return** TDAssign(\( D_L \)) \( \oplus \) TDAssign(\( D_R \))

- TDAssign has **time complexity** in \( O(|D| \log |D|) \)
**Algorithm:** kScan(document collection D)

// split D into k equal-sized partitions D<sub>i</sub>

n = |D|

for i = 1 ... k

pick longest document d<sub>i</sub> from D

assign n/k documents with highest similarity J(d, d<sub>i</sub>) to D<sub>i</sub>

D = D \ D<sub>i</sub>

return < d from D<sub>1</sub> > ⊕ ... ⊕ < d from D<sub>k</sub> >

- **kScan** has **time complexity** in O(k |D|)

- **kScan** **outperforms** TDAAssign **in terms of compression effectiveness** (bits per posting) in experiments on collections of web documents
4.2. URL-Based Document Reordering


- **Intuition**: Documents with lexicographically close URLs tend to have similar contents (e.g., www.x.com/a and www.x.com/b).

- **Algorithm**:
  - sort documents lexicographically according to their URL
  - assign consecutive document identifiers (1 … |D|)
Silvestri [11] reports experiments conducted on a large-scale crawl of the Brazilian Web (about 6 million documents).

<table>
<thead>
<tr>
<th></th>
<th>VByte</th>
<th>Gamma</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>11.40</td>
<td>12.72</td>
<td>12.71</td>
</tr>
<tr>
<td>URL</td>
<td>9.72</td>
<td>7.72</td>
<td>7.69</td>
</tr>
<tr>
<td>kScan</td>
<td>9.81</td>
<td>8.82</td>
<td>8.80</td>
</tr>
</tbody>
</table>

URL-based document ordering outperforms content-based document ordering (kScan), requiring fewer bits per posting on average.
5. Query Processing

- **Query processing methods** operate on inverted index
  - **holistic query processing methods** determine the full query results (e.g., document-at-a-time and term-at-a-time)
  - **top-k query processing methods** (aka. dynamic index pruning) determine only the top-k query result and avoid reading posting lists completely
    - Fagin’s **TA** and **NRA** for score-ordered posting lists
    - **WAND** and **Block-Max WAND** for document-ordered posting lists
Broder et al. [2] describe **WAND** (weak AND) as a **top-k query processing method** for document-ordered posting lists:

- **DAAT-style traversal** of posting lists in parallel.
- Assumes that the **maximum score** \( \max(i) \) per posting list is known.
- **Pivoted cursor movement** based on current top-k result:
  - Let \( \text{min}_k \) denote the worst score in the current top-k result (1).
  - Sort cursors for posting lists based on their current document identifier \( \text{cdid}(i) \) (2).
  - Pivot document identifier \( p \) is the smallest \( \text{cdid}(j) \) such that (3):
    \[
    \text{min}_k < \sum_{i \leq j} \max(i)
    \]
  - Move all cursors \( i \) with \( \text{cdid}(i) < p \) up to pivot \( p \).
Example: Pivoted cursor movement based on top-1 result

It is safe to move the cursor for posting lists \(a\) and \(b\) forward to \(d_9\).
Example: Pivoted cursor movement based on top-1 result

### Top-1

<table>
<thead>
<tr>
<th></th>
<th>d₁, 2</th>
<th>...</th>
<th>d₃, 1</th>
<th>...</th>
<th>max(a) = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\min_k = 8
\]
Example: Pivoted cursor movement based on top-1 result

- Top-1
  - $d_1: 8$
  - $a \quad d_1, 2 \quad ... \quad d_3, 1 \quad ... \quad \max(a) = 3$

- $b \quad d_1, 3 \quad ... \quad d_2, 3 \quad ... \quad \max(b) = 3$

- $c \quad d_1, 3 \quad ... \quad d_9, 3 \quad ... \quad \max(c) = 3$

It is safe to move the cursor for posting lists $a$ and $b$ forward to $d_9$.

<table>
<thead>
<tr>
<th>cdid</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_2, 3$</td>
<td>3</td>
</tr>
<tr>
<td>$d_3, 1$</td>
<td>6</td>
</tr>
<tr>
<td>$d_9, 3$</td>
<td>9</td>
</tr>
</tbody>
</table>

$\min_k = 8$
Example: Pivoted cursor movement based on top-1 result

Top-1

\[ a \quad d_1, 2 \quad \ldots \quad d_3, 1 \quad \ldots \quad \max(a) = 3 \]

\[ b \quad d_1, 3 \quad \ldots \quad d_2, 3 \quad \ldots \quad \max(b) = 3 \]

\[ c \quad d_1, 3 \quad \ldots \quad d_9, 3 \quad \ldots \quad \max(c) = 3 \]

It is safe to move the cursor for posting lists \(a\) and \(b\) forward to \(d_9\)

\[ \min_k = 8 \]

\(c_{did}\) \(\Sigma\)

\[ d_2, 3 \quad 3 \]

\[ d_3, 1 \quad 6 \]

\[ d_9, 3 \quad 9 \]

\(p = d_9\)

(1) \quad (2) \quad (3)
4.2. Block-Max WAND

- Ding and Suel [5] propose the **block-max inverted index**
  - store posting list as sequence of **compressed posting blocks**
  - each block contains a **fixed number of postings** (e.g., 64)
  - keep **minimum document identifier** and **maximum score** per block

\[
\begin{align*}
(1, 5) & \quad (7, 2) & \quad (11, 3) \\
\text{a} & \quad d_1, 2 & \quad d_3, 5 & \quad d_7, 2 & \quad d_9, 1 & \quad d_{11}, 3 & \quad d_{13}, 2 \\
\text{max}(a) &= 5
\end{align*}
\]

these are available without having to decompress the block
Block-Max WAND

- Pivoted cursor movement considering per-block maximum scores
  - determine pivot $p$ according to WAND
  - perform shallow cursor movement for all cursors $i$ with $\text{cdid}(i) < p$ (i.e., do not decompress if a new posting block is reached)
  - if any document from current blocks can make it into top-$k$, i.e.:
    $$\min_k < \sum_{i: \text{cdid}(i) \leq p} \text{block\_max}(i)$$
    perform deep cursor movement (i.e., decompress posting blocks) and continue as in WAND
  - else move cursor with minimal $\text{cdid}(i)$ to
    $$\min \left( \min_{i: \text{cdid}(i) \leq p} \text{next\_block\_mdid}(i), \text{cdid}(p + 1) \right)$$
Example: Pivoted cursor movement based on top-1 result

**Top-1**

<table>
<thead>
<tr>
<th></th>
<th>d₁, 2</th>
<th>...</th>
<th>d₃, 1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d₁, 2</td>
<td>...</td>
<td>d₃, 1</td>
<td>...</td>
</tr>
<tr>
<td>b</td>
<td>d₁, 3</td>
<td>...</td>
<td>d₂, 3</td>
<td>...</td>
</tr>
<tr>
<td>c</td>
<td>d₁, 3</td>
<td>...</td>
<td>d₉, 3</td>
<td>...</td>
</tr>
<tr>
<td>d</td>
<td>d₂, 3</td>
<td>...</td>
<td>d₁₁, 3</td>
<td>...</td>
</tr>
</tbody>
</table>

max(a) = 3

max(b) = 3

max(c) = 3

max(d) = 3
Block-Max WAND

Example: Pivoted cursor movement based on top-1 result

<table>
<thead>
<tr>
<th>Top-1</th>
<th>a</th>
<th>d₁, 2</th>
<th>...</th>
<th>d₃, 1</th>
<th>...</th>
<th>max(a) = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d₁ : 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>d₁, 3</th>
<th>...</th>
<th>d₂, 3</th>
<th>...</th>
<th>max(b) = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₁</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>d₁, 3</th>
<th>...</th>
<th>d₉, 3</th>
<th>...</th>
<th>max(c) = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₁</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d</th>
<th>d₂, 3</th>
<th>...</th>
<th>d₁₁, 3</th>
<th>...</th>
<th>max(d) = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₂</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

max(d₁) = 3
max(d₂) = 3
max(d₃) = 3
shallow
Example: Pivoted cursor movement based on top-1 result

\[
\begin{align*}
\textbf{Top-1} & \quad \begin{array}{c}
\textbf{a} & d_1, 2 \quad \ldots \quad d_3, 1 \quad \ldots \quad \max(a) = 3
\end{array} \\
\quad \begin{array}{c}
\textbf{b} & d_1, 3 \quad \ldots \quad d_2, 3 \quad \ldots \quad \max(b) = 3
\end{array} \\
\quad \begin{array}{c}
\textbf{c} & d_1, 3 \quad \ldots \quad d_9, 3 \quad \ldots \quad \max(c) = 3
\end{array} \\
\quad \begin{array}{c}
\textbf{d} & d_2, 3 \quad \ldots \quad d_{11}, 3 \quad \ldots \quad \max(d) = 3
\end{array}
\end{align*}
\]

\(\text{shallow}\)
Summary

- **Inverted indexes** can be **efficiently constructed** offline by using external memory sort or MapReduce.

- **Inverted indexes** can be **efficiently maintained** by using logarithmic/geometric partitioning.

- **Static index pruning methods** reduce index size by systematically **removing postings**.

- **Document reordering methods** reduce index size by assigning document identifiers so as to yield **smaller gaps**.

- **Query processing** on document-ordered inverted indexes can be greatly sped up by **pivoted cursor movement** as part of WAND and Block-Max WAND.
References


References
