Intersection in Integer Inverted Indices

Peter Sanders, Frederik Transier
### Inverted Index

- **Goals:**
  - Locating words in a huge number of documents.
  - Supported operators: AND, OR, phrase, fuzzy, …

- **Unique IDs for:**
  - documents
  - terms

- **A document becomes a list of term IDs.**

- **Inverted Index:**
  For each term we store a list of document IDs.

![Structure of an Inverted Index](image)

- **Dictionary:**
  - `hello` → `7 35 51 75 103`

- **Inverted Lists:**
  - `world` → `1 22 75 99`
  - `bye` → `7 35 51 103 120 151`
  - `…`
The AND query is an intersection of the corresponding inverted lists:

Query: "hello" AND "world"

1. hello: 7, 35, 51, 75, 103
2. world: 1, 22, 75, 99
3. bye: 7, 35, 51, 103, 120, 151
Example of an *Inverted List*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>18</td>
<td>105</td>
<td>117</td>
<td>132</td>
</tr>
</tbody>
</table>

- An uncompressed list requires $5 \times 32$ bits = 160 bits.

- But $5 \times \lceil \log(132) \rceil = 40$ bits would be enough.

- $\Delta$-encoding: Store differences to the predecessors rather than values.

- $5 \times \lceil \log(87) \rceil = 35$ bits.

Example of an $\Delta$-encoded *Inverted List*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>11</td>
<td>87</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
### Compression: Variable Bit Length Encoding

The sum of all used bits is 22.

But the $\Delta$–encoded list requires 35 bits.

#### Different encodings of an Inverted List

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>uncompressed</td>
<td>7</td>
<td>18</td>
<td>105</td>
<td>117</td>
<td>132</td>
</tr>
<tr>
<td>$\Delta$-encoded</td>
<td>7</td>
<td>11</td>
<td>87</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>$\Delta$-encoded (binary)</td>
<td>0000111</td>
<td>0001011</td>
<td>1010111</td>
<td>0001110</td>
<td>0001111</td>
</tr>
<tr>
<td>escaping (binary)</td>
<td>01111</td>
<td>10111</td>
<td>01010</td>
<td>01111</td>
<td>11111</td>
</tr>
</tbody>
</table>

**Escaping:**
- Split values into blocks.
- Use only as many blocks as necessary.

Now, the list can be encoded in 30 bits.
Intersection Algorithms: Zipper

- **Zipper**: Scan both lists as in a binary merge operation.
- needs time $O(n+m)$
- supports linear compressing schemes
- very cache efficient
- best if $n \approx m$
- in main memory: the decompression overhead outweighs the gains in memory access time.
The simplest binary search algorithm: Locate each element of the shorter list in the longer one.

*Exponential search:* Find an upper-bound by exponential probing.

Binary-search using *divide-and-conquer (Baeza-Yates)*:
- Locate the middle element of the shorter list in the longer one.
- Proceed recursively on the left and right parts.

Binary-search algorithms are incompatible with linear encodings. (including Δ-encoding)
Intersection Algorithms: Two-Level Representations

- A two level representation is used to overcome this limitation.

Two level representation of an inverted list

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>105</th>
<th>132</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

0111 1 1011 1 0101 0 0111 1 1110 1 1111 1

- Binary search algorithms are executed on the top-level.
- Followed by a linear search in the compressed list.
- **Skipper**: A two level variant of Zipper.
Randomized Inverted Index:
document IDs are assigned randomly.
(e.g. using pseudo-random permutation)

Two-level data structure:
- Split the range of document IDs into buckets based on their most significant bits.
- **Lookup-table**: direct access to the first value of a bucket

**Inverted List with lookup table**

(lookup table size: $\lceil \log_2(n/B) \rceil$)
**Intersection Algorithms: Randomized Inverted Indices**

**Lookup** is an intersection algorithm that runs on this data structure:

```plaintext
Function lookup(M, N)
  O := {} // output
  i := -1 // current bucket key (now a dummy)
  foreach d ∈ M do // unpack M
    h := d >> k_N // bucket key
    l := d & (2^{k_N}-1) // least significant bits
    if h > i then // a new bucket
      i := h // set current bucket
      j := t[i] // get start position
      e := t[i+1] // get end position
      while j < e do // bucket not exhausted
        l' := N[j] // unpack if necessary
        if l ≤ l' then
          if l = l' then O := O ∪ {d}
          break while
          j++
    return O
```

Lookup runs in expected time $O(m + \min\{n, Bm\})$.
Experiments: Test System

- Implementation using C++ and the STL
- Intel Xeon, 3.2 GHz (4 GB main memory, 512 KByte L2 cache)
- SuSE Linux Enterprise Server 9 (kernel 2.6.5)
- gcc 3.3.3 (-O3)
- timing with `clock_gettime`
Experiments: Test Data

Three real world instances:

<table>
<thead>
<tr>
<th></th>
<th>WT2g</th>
<th>WT2g.s</th>
<th>BibTex</th>
</tr>
</thead>
<tbody>
<tr>
<td># documents</td>
<td>239</td>
<td>492</td>
<td>1 195 624</td>
</tr>
<tr>
<td>volume [KB]</td>
<td>1 483</td>
<td>926</td>
<td>70 783</td>
</tr>
<tr>
<td>max list length</td>
<td>212</td>
<td>974</td>
<td>387 794</td>
</tr>
<tr>
<td># terms</td>
<td>1 532</td>
<td>125</td>
<td>164 345</td>
</tr>
</tbody>
</table>

Generated pair-wise queries:

- 100 subintervals in the range [0.001, 1] of length ratios.
- 10 pair-wise queries for each interval were generated.
- So that lists as big as possible were involved.
Experiments: Space Consumption (WT2g)

- No significant differences between ∆-bitc. and ∆-escaped for the rand. representation.
- Escaping can exploit nonuniformity of input.
- ∆-bitc. does not work well for the det. representation.
Experiments: Performance of *Lookup* (WT2g)

- Large buckets are bad for small ratios...
- ...but good for nearly equal lengths.
- Bucket size 8 seems to be a good compromise.
- bit-compressed Δ-encoding
Experiments: Performance of *Skipper* (WT2g)

Bucket size 32 seems to be the best choice. (for other algorithms, too)
Zipper, skipper and lookup are all very good for lists of similar lengths.

Lookup is best up to ratio close to one.
Experiments: Space-time Tradeoff of Encodings (WT2g.s)

Performance loss of Δ-encoding is negligible low for rand. representation.

Escaping requires perceivable more time.

Clearly different running times for det. representation.
Randomization gives theoretical performance guarantees, but in practice deterministic data often outperforms randomization.

*Lookup* is also a good heuristics for non-randomized data.
Experiments: Performance on Non-compressed data (WT2g)

Strong negative practical performance impact of randomization on non-compressed data.

Break even with Zipper.
Results

- *Zipper* is best for nearly equal lengths.
- But the benefits are not so big for compressed data.
- Break even with Skipper and Lookup $\approx 1/20$.
- *Lookup* algorithm is among the best over the entire spectrum.
- Especially for compressed indices and small $m/n$.
- *Skipper* is simple and has also a good performance.
- But cannot compete with lookup or B.-Y. for very small $m/n$.
- *Baeza-Yates* is good for very small $m/n$.
- But cannot compete with lookup.
Future Work

- Even better compression?
- Highly modular implementation.
- Study integration of
  - updates,
  - text output,
  - phrase queries,
  - …
Thank You for Your attention!