Advance Indexing

Limock
July 3, 2014

Why Use an On-line Indexing

- Dynamic document collections.
- Process documents indexing while serving queries request.
Off-line indexing

Documents

Main-Memory

List of <term, doc> pairs

Merge pairs with same term

split

merge
On-line Indexing Techniques

Basic Approaches
- Re-build
- In-place
- Re-merge

Vertical Partitioning
- Geometric Partitioning
- Logarithmic Merge

Horizontal Partitioning
- Single-split Indexing
- Multi-split Indexing

Multi-partition
- Single-splat Immediate Merge
- Single-split Multi-partition

Single Partition
- Immediate Merge
Re-build Update

- New index is build from scratch
- Keep existing index for querying
- Discard old index once new index is available
- Simple
- High query performance
- Inefficient index approach
In-place Update

- Append posting terms in in-memory index to terms in on-disk index
- Each term has an over-allocation space
- Relocate terms if over-allocation is exhausted
- Require an effective over-allocation strategy

Figure 1: In-place update strategy. (a) represents the new postings lists of in-memory index. (b) represents state of on-disk inverted index. (c) represents in-place update with relocation for Term2 and no relocation for Term1.
Merge-based – Immediate Merge

- One on-disk inverted index exists.
- Merge existing index and in-memory index to replace the existing index
- High query performance > contiguous posting list
- High index maintenance cost
Each partition contains sub-index
Partition 0 is reserved for in-memory index block
A partition $k$ has index of size 0 or $\left[ r^{(k-1)}S, (r-1)r^{(k-1)}S \right]$ postings

\textbf{Geometric ratio}
Vertical Multi-partition – Logarithmic Merge

- Every sub-index is given unique generation number $g$
- Merge sub-index $g$ to create sub index $g+1$
- $\log_k N$ partition, where $N$ is the number of in-memory blocks created
Proposed Work

- Horizontal partitioning
- Splitting terms into 2: frequent and infrequent
- Maintain frequent-term sub-index with better query performance approach (i.e. active merge)
- Maintain infrequent-term sub-index with better index performance approach (i.e. lazy merge)
Single-split Immediate Search

- Lazy-merge approach for infrequent-term
- Immediate-merge for frequent term
- High query performance
- Poor index maintenance performance
Single-split Multi-partition

- Lazy-merge approach for infrequent-term
- Active-merge for frequent term
- Slightly lower query performance
- Better index maintenance performance
Index Tree

- Binary where each node contain a set of partition and each partition contains a sub-index.
- There are 3 parameter for each node : P, R and T
- Split sub-index into frequent-term(right node) and infrequent-term(left node)
- For left node : decrement value of R and increment value of P
- For right node : increment value of R and decrement value of P
Index Tree

Figure 6: Internal details of a node in Index Tree.

Figure 7: Index Tree with root node initialized to \( R = 3 \) and \( P = 1 \).
Index Tree – Step by Step Construction

(1) = Filled Partition/node  (2) = Filled Partition/node  (3) = Filled Partition/node
(4) = Filled Partition/node  (5) = Filled Partition/node  (6) = Filled Partition/node
(7) = Filled Partition/node  (8) = Filled Partition/node  (9) = Filled Partition/node
(10) = Filled Partition/node  (11) = Filled Partition/node  (12) = Filled Partition/node
Algorithm 1: InsBlock(root, tmpIndex)

1 begin
2 if root = NULL then
3     root ← getTreeNode();
4     root.Partition.sIndex ← tmpIndex;
5     return root;
6 dbPartition ← root.Partition;
7 while dbPartition ≠ NULL do
8     dbIndex ← dbPartition.sIndex;
9     if dbPartition.r > 1 then
10        if EMPTY(dbIndex) then
11           dbIndex ← tmpIndex;
12           dbPartition.r ← dbPartition.r - 1;
13           return root;
14        else
15           dbIndex ← Merge(dbIndex, tmpIndex);
16           dbPartition.r ← dbPartition.r - 1;
17           return root;
18     else if dbPartition.next ≠ NULL then
19           tmpIndex ← Merge(dbIndex, tmpIndex);
20           Reset(dbPartition);
21           dbPartition ← dbPartition.next;
22     else
23           fPart, iPart ← splitMerge(dbIndex, tmpIndex);
24           Reset(dbPartition);
25           root.left ← InsBlock(root.left, fPart);
26           root.right ← InsBlock(root.right, iPart);
27           return root;
28 end
Experiment

- 7 Millions HTML documents
- Average size per document: 12KB
- Compute term frequencies from static AOL query log
- Query log contains over 1.2 million unique query terms
- Maximum in-memory index size is set to 60MB
- 1GB RAM and 240GB hard drive
- Document deletion are not handled
- Parameter P: 3, R: 1, T: 80%-20%
Result – Index and Query Performance

Index Performance Comparison

Query Performance Comparison

Figure 9: Index Maintenance performance for GP, Multi-split, and Single-split approaches. Times represent the total time taken to move in-memory blocks to disk.

Figure 11: Query performance for GP, Multi-split, and Single-split approaches. Times (in ms) represent average time taken to evaluate a query after moving ‘n’ in-memory index blocks to disk.
Result – Query vs Index Comparison

Query Index Performance

- Geometric Partition
- Multi-split (Index Tree)
- Single-split multi-partition
- Single-split Immediate
Papers


General Information

• Cost of query evaluation depends on:
  - Number of query terms
  - Length of their posting lists

• Precomputation of common sub-queries help reduce this cost

• Replace multiple query terms with one composite term

• Stored in the index as regular posting lists

• Constraint: Memory
Proposed Work

- Using *bitmaps* to encode term co-occurrence
- Require $k$ bits for encoding $k$ different terms with the posting list in a list with bitmaps
- Able to resolve any queries involving any of the $2^k$ combination of the chosen terms
- Constraint: Posting lists are longer compared to precomputed list

Figure 1: An example index with bitmaps for terms York and Hall and a precomputed list for New York (left) and example query workload (right).
Evaluation Time – Cost Function

- Cost Function:
  \[
  F(q) = |L_1| \sum_{i=2}^{i=n} G(|L_i|)
  \]

- \( G(x) = C_1 + C_2 \log(x) \)

- Cost Function – recalculated:
  \[
  F(q) = |L_1| \sum_{i=2}^{i=n} (12 + \log |L_i|)
  \]
Index Construction - Bitmaps

- Let $B$ the association matrix where $b_{ij}$ if there is a bit for term $t_j$ in list $L_i$'s bitmaps.

- minimize : $\sum_{q \in Q} F(B, q)$

- subject to : $\sum_i b_{ij} |L_i| \leq S$  \hspace{1cm} $b_{ij} \in \{0, 1\}$

- Compute benefit ratio : $\lambda_{ij} = \frac{\sum_{q \in Q} F(B \cup \{b_{ij}\}, q) - F(B, q)}{|L_i|}$
Index Construction – Precomputed Lists

• Let $P$ a set of precomputed lists where $p_{ij}$ is the indicator variable representing the results of query $t_i t_j$

• minimize: $\sum_{q \in Q} F(P, q)$

• subject to: $\sum_{ij} p_{ij} |L_i \cap L_j| \leq S$ \hspace{1em} $p_{ij} \in \{0, 1\}$

• Compute benefit ratio: $\lambda_{ij}' = \sum_{q \in Q} \frac{F(P \cup \{p_{ij}\}, q) - F(P, q)}{|L_i \cap L_j|}$
Index Construction - Hybrid

- Hybrid algorithm that at each step select precomputed list $p_{ij}$ or a bitmap $b_{ij}$ that maximizes marginal benefit given by Equation 1 or 2
Algorithm 2 Query rewrite algorithm using bitmaps

1: Assume that $|L_1| \leq |L_2| \leq \ldots \leq |L_n|$
2: Unmark terms $t_1, t_2, \ldots, t_n$
3: $L \leftarrow \emptyset$
4: for $i \leftarrow 1$ to $n$ do
5: \hspace{1em} if $t_i$ is unmarked then
6: \hspace{2em} $L \leftarrow L \cup \{L_i\}$
7: \hspace{2em} Mark $t_i$
8: \hspace{1em} for $j \leftarrow i + 1$ to $n$ do
9: \hspace{2em} \hspace{1em} if $b_{ij} = 1$ then
10: \hspace{2em} \hspace{2em} Mark $t_j$
11: return $L$
Algorithm 3 Query rewrite algorithm using precomputed lists

1: Assume that $|L_1| \leq |L_2| \leq \ldots \leq |L_n|$
2: Unmark terms $t_1, t_2, \ldots, t_n$
3: $L \leftarrow \emptyset$
4: for $i \leftarrow 1$ to $n$ do
5: if $t_i$ is unmarked then
6: $L \leftarrow L_i$
7: Mark $t_i$
8: for $j \leftarrow i + 1$ to $n$ do
9: if $p_{ij} = 1 \wedge t_j$ is unmarked then
10: $L \leftarrow L_{ij}$
11: Mark $t_j$
12: break
13: $L \leftarrow L \cup \{L\}$
14: return $L$
Query Evaluation - Hybrid

- First invoke Algorithm 3 to identify precomputed lists
- Then invoke Algorithm 2 for removing some of the lists that are covered by bitmaps in shorter lists.
- This is to minimize $|L_1|$
Experiment

- Data: TREC WT01g corpus
- 1.68 million web pages
- Extract textual content and discard HTML tags
- Each post in index contain 4 byte of docID and variable size payload (0-32 bits) containing bitmaps
- Index size: 1.5GB, max 6GB
- AOL query log for query workload
- 21M queries for training set
- 50K queries from remaining 2.6M queries for testing set
Result – Query Latency Improvement

Figure 3: Query latency reduction with different precomputation techniques. Precomputation budget is 25% the size of the original index size.

Figure 4: Query latency reduction as a function of memory sharing between bitmaps and precomputed lists. Total precomputation budget is 25% the size of the original index size.
Figure 6: Query latency reduction with increasing pre-computation budget (in percents of the original index size).

Figure 7: Query latency reduction of long tail queries compared to all queries. Precomputation budget is 25% of the original index size.
Thank you