Adaptive Index Structures

Yufei Tao and Dimitris Papadias
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
Motivation

- Traditional indexes allocate a fixed number (usually one) disk page to a node.

- A random disk access involves
  - Seek time $T_{SK}$
  - Data transfer time $T_{TRF}$
  - CPU processing time $T_{CPU}$

- $T_{SK}$ is larger than $T_{TRF}$ and $T_{CPU}$ by 1-2 orders of magnitude. Hence, the number of seek operations dominates the cost.

- Minimizing the number of node accesses, does not necessarily minimize the processing cost.
  - Query $q_1$ visiting 20 continuous pages has cost $T_{SK} + 20 \cdot T_{TRF} + 20 \cdot T_{CPU}$
  - Query $q_2$ visiting 10 random pages has cost $10T_{SK} + 10 \cdot T_{TRF} + 10 \cdot T_{CPU}$
  - Query $q_1$ can be significantly cheaper because it requires much fewer random accesses.
Existing Solutions to Reduce Random Accesses

- De-fragmentation (re-organize the pages to make them continuous)
  - Carried out periodically.
  - Disadvantage 1: Extremely costly
    - Moving a large number of pages
    - Correcting mutual references between pages
  - Disadvantage 2: Low efficiency
    - A good page organization may be destroyed soon by subsequent update operations/overflows.

- Assigning more disk pages to a node
  - Disadvantage: Low adaptability
    - A single node size favors queries with specific selectivity only.
Overview of the Proposed Technique

• We propose adaptive index structures that consider the data and query characteristics for determining the node size.

• Allow different node sizes (in terms of number of allocated pages) in various parts of the tree.

• The node size is continuously optimized according to the characteristics of queries in the data space represented by the node.
An Example

• A conventional B-tree

• An adaptive B-tree

• A query [40, 75], for examples, accesses 5 nodes in the conventional tree, but only 2 in the adaptive counterpart.
Optimal Node Size (for Uniform Data)

\[ P_{OPT}(q_L) = \sqrt{\frac{q_L \cdot N \cdot T_{SK}}{\xi \cdot b_{sp} \left( T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL} \right)}} \]

- \( N \): the dataset cardinality
- \( q_L \): the query length (selectivity)
- \( b_{sp} \): the node capacity
- \( \xi \): the node utilization rate (usually 69%)
- \( T_{SK} \): seek time
- \( T_{TRF} \): page transfer time
- \( T_{EVL} \): evaluation time of one index entry

Note:
- The optimal node size increases with \( N \) and \( q_L \) which increases the number of records retrieved.
- High seek time, fast CPU or data transfer time also increase the optimal size because the I/O accounts for lower percentage.
- The result applies to the leaf level, while for non-leaf level, the optimal size is always 1 (because a range-query on B-trees accesses only one node per non-leaf level).
Optimal Node Size (Non-Uniform Data)

- We maintain a histogram which divides the universe into $num_{bin}$ bins with equal lengths.

- For each bin-$i$ we store
  - $n_i$: the number of records whose keys fall into the range of the bin.
  - $\text{Exp}(q_{Li})$: the average lengths of queries whose ranges intersect that of the bin.

- The optimal node size follows that of the uniform analysis:

  $$p_{OPT_i} = \frac{\sqrt{\text{Exp}(q_{Li}) \cdot n_i \cdot num_{bin} \cdot T_{SK}}}{\sqrt[\xi \cdot b_{sp} \left( T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL} \right)}}$$

- Note that the optimal size changes with both the data and query properties.
Insertion Algorithm

- The algorithm follows the framework of conventional B-trees. First we identify the node that accommodates the changes:
  - If the node does not overflow after inserting the new record, the insertion terminates.
  - Otherwise, handle the overflow.
- When a node $P$ with old size $P.Size_{old}$ overflows, we first compute the new size $P.Size_{new}$ using the equation on the previous slide.
- Different actions are taken depending on the relationship between $P.Size_{old}$ and $P.Size_{new}$:

\[
\begin{align*}
\text{overflow} & \quad P.Size_{new} \in (P.Size_{old}, 2 \cdot P.Size_{old}] \quad \text{node expansion} \\
P.Size_{new} & \in [1, P.Size_{old}] \quad \quad \quad \text{node splitting} \\
P.Size_{new} & \in (2 \cdot P.Size_{old}, +\infty) \quad \text{generates an underflow}
\end{align*}
\]
Insertion (1st case): Node Expansion

- If $P.\text{Size}_{\text{new}} \in (P.\text{Size}_{\text{old}}, 2 \cdot P.\text{Size}_{\text{old}}]$ (the new node size is larger than the old one) we only need to expand node $P$ to its new size, after which the number of entries in $P$ is greater than the minimum node utilization yet smaller than the node capacity).

- Special care must be taken for allocating continuous pages. An example:
  - The size of $P$ needs to be enlarged from 2 to 4 pages.
  - The mutual references among pages must be fixed.
Insertion (2\textsuperscript{nd} case): Node Splitting

- If $P.\text{Size}_{\text{new}} \neq P.\text{Size}_{\text{old}}$, an overflowing node $P$ is split into several (2) nodes by distributing the entries evenly.
- There are multiple ways to decide the number $NM_{\text{SPLT}}$ of resulting nodes so that the number of entries in each node is within the range $[\frac{1}{2} \cdot P.\text{Size}_{\text{new}}, P.\text{Size}_{\text{new}}]$.
- Entries in the original node are evenly divided into $NM_{\text{split}}$ nodes, where $NM_{\text{SPLT}}$ is determined by the following equation, which minimizes the number of nodes:

$$NM_{\text{SPLT}} = \left\lceil \frac{(b_{sp} \cdot P.\text{Size}_{\text{old}} + 1)}{(b_{sp} \cdot P.\text{Size}_{\text{new}})} \right\rceil$$
Deletion Algorithm

- The algorithm also follows the framework of conventional B-trees: First we identify the node that contains the entry to be deleted.
  - If the node does not incur underflows then the deletion terminates.
  - Otherwise, handle the underflow.
- As with overflows, when a node $P$ with old size $P.\text{Size}_{old}$ underflows, we first compute the new size $P.\text{Size}_{new}$, and adopt different actions based on its comparison of $P.\text{Size}_{old}$:

  \[
  \begin{align*}
  P.\text{Size}_{new} & \in (\frac{1}{2} \cdot P.\text{Size}_{old}, P.\text{Size}_{old}] & \text{node contraction} \\
  P.\text{Size}_{new} & \in [P.\text{Size}_{old}, +\infty) & \text{node merging} \\
  P.\text{Size}_{new} & \in [1, \frac{1}{2} \cdot P.\text{Size}_{old}) & \text{generates an overflow}
  \end{align*}
  \]

- **Node contraction** simply reduces the size of a node to its new value, by freeing the “tailing pages” originally assigned.
- **Merging** is performed as with conventional merging algorithms, except that the underflowing node may be merged with several sibling nodes.
Performance of Adaptive B-Trees

• Asymptotical optimality:
  – Given \( N \) records, an adaptive B-tree consumes \( O(N/b) \) disk pages, and answers a range query with \( O(\log_b N + K/b) \) I/Os, where \( K \) is the number of records retrieved.

• Cost model:
  \[
  \text{TIME}_{QAB}(q_L) = (h-1) \cdot \left( T_{SK} + T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL} \right)
  + \left( \frac{n_i \cdot \text{num}_{bin} \cdot q_L}{\xi \cdot b_{sp} \cdot P_i} + 1 \right) \left( T_{SK} + P_i \cdot T_{TRF} + \xi \cdot b_{sp} \cdot P_i \cdot T_{EVL} \right)
  \]

• For high cardinality and large query length, the speed up over conventional B-trees converges to:
  \[
  \text{Speedup} \rightarrow \frac{T_{SK} + T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL}}{T_{TRF} + \xi \cdot b_{sp} \cdot T_{EVL}}
  \]
Generalization of the Technique

- The proposed technique can be applied to other structures to obtain adaptive versions.
  - First decide the optimal node size as a function of data and query parameters.
  - Then modify the original update algorithms with the principle: whenever a node is created or incurs over/under-flows, its size is re-computed using the current statistical information.
- An example of adaptive R-trees:
Experimental Settings

- $T_{SK}=10\text{ms}$, $T_{TRF}=1\text{ms/Kbyte}$, $T_{EVL}=1\mu\text{s per entry}$
- Node size=$1\text{K bytes}$, resulting in node capacities of 125 and 50 entries for B-, R-trees respectively.
- Relational datasets
  - Cardinality 100K-2M
  - Uniform or gaussian distributions
- Spatial dataset
  - Use a real dataset (560K) containing road segments (density map shown on the previous slide).
- Workloads of 500 selection queries
Experiment 1: Speedup VS Query Length

- Uniform data and query distributions

![Graph showing speedup vs query length selectivity for dataset cardinality = 1M](image)

- Estimated speedup vs dataset cardinality (M)

- Query length - selectivity = 1%
Experiment 2: Non-Uniform Queries

- We use a histogram with 50 bins
- Uniform dataset (1M records)
- The query lengths follow a Gaussian distribution:
  - (i) queries at the center of the data space have the largest length (2% range),
  - (ii) queries at the edges of the data space have length 0 (i.e., equality selection).

**NODE SIZE PER BIN**

![Node size per bin graph](image)

**QUERY COST PER BIN**

![Query cost per bin graph](image)
Experiment 3: Speedup VS Update Frequency

- We tested with workloads that mix query and updates with certain frequency, and measure the cost per operation.

- Adaptive B-trees are faster than conventional B-trees except for extremely frequent updates (close to 100%).
Experiment 4: Bulkloading

• We create a dataset with 500K uniform records, and bulkload a B- and an adaptive B-tree.
• Then, we perform another 500K insertions.
• The diagram shows the query cost of the two structures as a function of the number of insertions (5 means 50K insertions are performed and so on).

Before 150K insertions both trees have similar performance because most accesses are sequential. After that, the B-tree starts to incur node splits that break the sibling adjacency, and its performance deteriorates very quickly.
Experiments 5: Application to R-Trees

- We measure the speedup as a function of query size, and query locations.

```
window length (10% length corresponds to 1% of the area)
```
Conclusion

• We introduce the concept of adaptive index structures, which dynamically adapt their node sizes to minimize the query cost.

• We also propose a general framework for converting traditional structures to adaptive versions, through a set of update algorithms.
  – The only requirement for our methods is the existence of analytical models that estimate the number of node accesses. Such models have been proposed for most popular structures rendering our framework directly applicable to them.

• Analytical and experimental evaluation confirms that adaptive indexes outperform conventional counterparts significantly in a wide range of scenarios.

• Future work:
  – Consider also cache?
  – What if corresponding models do not exist?