Dynamic Indexability and the Optimality of B-trees and Hash Tables

Ke Yi

Hong Kong University of Science and Technology

Dynamic Indexability and Lower Bounds for Dynamic One-Dimensional Range Query Indexes, *PODS ’09*

Dynamic External Hashing: The Limit of Buffering, with Zhewei Wei and Qin Zhang, *SPAA ’09*

+ some latest development
An index is . . .

- An index is a single number calculated from a set of prices
  - Dow Jones, S & P, Hang Seng
An index is . . .

- An index is a single number calculated from a set of prices
  - Dow Jones, S & P, Hang Seng
- An index is a list of keywords and their page numbers in a book
- An index is an exponent
- An index is a finger
- An index is a list of academic publications and their citations
An index is . . .

- An index is a single number calculated from a set of prices
  - Dow Jones, S & P, Hang Seng
- An index is a list of keywords and their page numbers in a book
- An index is an exponent
- An index is a finger
- An index is a list of academic publications and their citations
- An index (search engine) is an inverted list from keywords to web pages
- An index (database) is a (disk-based) data structure that improves the speed of data retrieval operations (queries) on a database table.
Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems
Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems
- B-tree: lookups and range queries; Hash table: lookups
Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems
- B-tree: lookups and range queries; Hash table: lookups

External memory model (I/O model):

- Memory of size $M$
- Each I/O reads/writes a block
- Disk partitioned into blocks of size $B$
The B-tree
The B-tree

A range query in $O(\log_B N + K/B)$ I/Os

$K$: output size
The B-tree

A range query in $O(\log_B N + K/B)$ I/Os

$K$: output size

$\log_B N - \log_B M = \log_B \frac{N}{M}$
The B-tree

A range query in $O(\log_B N + K/B)$ I/Os

$K$: output size

$$\log_B N - \log_B M = \log_B \frac{N}{M}$$

The height of B-tree never goes beyond 5 (e.g., if $B = 100$, then a B-tree with 5 levels stores $n = 10$ billion records). We will assume $\log_B \frac{N}{M} = O(1)$. 

External Hashing

\[ h(x) = \text{last digit of } x \]
External Hashing

\[ h(x) = \text{last digit of } x \]

Ideal hash function assumption: \( h \) maps each object to a hash value uniformly independently at random
**External Hashing**

\[ h(x) = \text{last digit of } x \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>21</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>82</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>34</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

**Ideal hash function assumption:** \( h \) maps each object to a hash value uniformly independently at random

**Expected average** cost of a successful (or unsuccessful) lookup is \( 1 + \frac{1}{2^{\Omega(B)}} \) disk accesses, provided the load factor is less than a constant smaller than 1 [Knuth, 1973]
Exact Numbers Calculated by Knuth

Table 2
AVERAGE ACCESES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

<table>
<thead>
<tr>
<th>Bucket size, b</th>
<th>Load factor, $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>1</td>
<td>1.0048</td>
</tr>
<tr>
<td>2</td>
<td>1.0012</td>
</tr>
<tr>
<td>3</td>
<td>1.0003</td>
</tr>
<tr>
<td>4</td>
<td>1.0001</td>
</tr>
<tr>
<td>5</td>
<td>1.0000</td>
</tr>
<tr>
<td>10</td>
<td>1.0000</td>
</tr>
<tr>
<td>20</td>
<td>1.0000</td>
</tr>
<tr>
<td>50</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Table 3
AVERAGE ACCESES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

<table>
<thead>
<tr>
<th>Bucket size, b</th>
<th>Load factor, $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>1</td>
<td>1.0500</td>
</tr>
<tr>
<td>2</td>
<td>1.0063</td>
</tr>
<tr>
<td>3</td>
<td>1.0010</td>
</tr>
<tr>
<td>4</td>
<td>1.0002</td>
</tr>
<tr>
<td>5</td>
<td>1.0000</td>
</tr>
<tr>
<td>10</td>
<td>1.0000</td>
</tr>
<tr>
<td>20</td>
<td>1.0000</td>
</tr>
<tr>
<td>50</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Exact Numbers Calculated by Knuth


### Table 2: Average Accesses in an Unsuccessful Search by Separate Chaining

<table>
<thead>
<tr>
<th>Bucket size, $b$</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0048</td>
<td>1.0187</td>
<td>1.0408</td>
<td>1.0703</td>
<td>1.1065</td>
<td>1.1488</td>
<td>1.197</td>
<td>1.249</td>
<td>1.307</td>
<td>1.34</td>
</tr>
<tr>
<td>2</td>
<td>1.0012</td>
<td>1.0088</td>
<td>1.0269</td>
<td>1.0581</td>
<td>1.1036</td>
<td>1.1638</td>
<td>1.238</td>
<td>1.327</td>
<td>1.428</td>
<td>1.48</td>
</tr>
<tr>
<td>3</td>
<td>1.0003</td>
<td>1.0038</td>
<td>1.0162</td>
<td>1.0433</td>
<td>1.0898</td>
<td>1.1588</td>
<td>1.252</td>
<td>1.369</td>
<td>1.509</td>
<td>1.59</td>
</tr>
<tr>
<td>4</td>
<td>1.0001</td>
<td>1.0016</td>
<td>1.0095</td>
<td>1.0314</td>
<td>1.0751</td>
<td>1.1476</td>
<td>1.253</td>
<td>1.394</td>
<td>1.571</td>
<td>1.67</td>
</tr>
<tr>
<td>5</td>
<td>1.0000</td>
<td>1.0007</td>
<td>1.0056</td>
<td>1.0225</td>
<td>1.0619</td>
<td>1.1346</td>
<td>1.249</td>
<td>1.410</td>
<td>1.620</td>
<td>1.74</td>
</tr>
<tr>
<td>10</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0004</td>
<td>1.0041</td>
<td>1.0222</td>
<td>1.0773</td>
<td>1.201</td>
<td>1.426</td>
<td>1.773</td>
<td>2.00</td>
</tr>
<tr>
<td>20</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0001</td>
<td>1.0028</td>
<td>1.0234</td>
<td>1.113</td>
<td>1.367</td>
<td>1.898</td>
<td>2.29</td>
</tr>
<tr>
<td>50</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0007</td>
<td>1.018</td>
<td>1.182</td>
<td>1.920</td>
<td>2.70</td>
</tr>
</tbody>
</table>

### Table 3: Average Accesses in a Successful Search by Separate Chaining

<table>
<thead>
<tr>
<th>Bucket size, $b$</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0500</td>
<td>1.1000</td>
<td>1.1500</td>
<td>1.2000</td>
<td>1.2500</td>
<td>1.3000</td>
<td>1.350</td>
<td>1.400</td>
<td>1.450</td>
<td>1.48</td>
</tr>
<tr>
<td>2</td>
<td>1.0063</td>
<td>1.0242</td>
<td>1.0520</td>
<td>1.0883</td>
<td>1.1321</td>
<td>1.1823</td>
<td>1.238</td>
<td>1.299</td>
<td>1.364</td>
<td>1.40</td>
</tr>
<tr>
<td>3</td>
<td>1.0010</td>
<td>1.0071</td>
<td>1.0215</td>
<td>1.0458</td>
<td>1.0806</td>
<td>1.1259</td>
<td>1.181</td>
<td>1.246</td>
<td>1.319</td>
<td>1.36</td>
</tr>
<tr>
<td>4</td>
<td>1.0002</td>
<td>1.0023</td>
<td>1.0097</td>
<td>1.0257</td>
<td>1.0527</td>
<td>1.0922</td>
<td>1.145</td>
<td>1.211</td>
<td>1.290</td>
<td>1.33</td>
</tr>
<tr>
<td>5</td>
<td>1.0000</td>
<td>1.0008</td>
<td>1.0046</td>
<td>1.0151</td>
<td>1.0358</td>
<td>1.0699</td>
<td>1.119</td>
<td>1.186</td>
<td>1.268</td>
<td>1.32</td>
</tr>
<tr>
<td>10</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0002</td>
<td>1.0015</td>
<td>1.0070</td>
<td>1.0226</td>
<td>1.056</td>
<td>1.115</td>
<td>1.206</td>
<td>1.27</td>
</tr>
<tr>
<td>20</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0005</td>
<td>1.0038</td>
<td>1.018</td>
<td>1.059</td>
<td>1.150</td>
<td>1.22</td>
</tr>
<tr>
<td>50</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Extremely close to ideal
Now Let’s Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block.
  - B-tree: Split blocks when necessary.
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80].
Now Let’s Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block.
- B-tree: Split blocks when necessary.
- Hashing: Rebuild the hash table when too full; extensible hashing [Fagin, Nievergelt, Pippenger, Strong, 79]; linear hashing [Litwin, 80].
- These resizing operations only add $O(1/B)$ I/Os amortized per insertion; bottleneck is the first search + insert.
Now Let’s Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
- B-tree: Split blocks when necessary
- Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]
- These resizing operations only add $O(1/B)$ I/Os amortized per insertion; bottleneck is the first search + insert
- Cannot hope for lower than 1 I/O per insertion only if the changes must be committed to disk right away (necessary?)
Now Let’s Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block

- B-tree: Split blocks when necessary

- Hashing: Rebuild the hash table when too full; extensible hashing [Fagin, Nievergelt, Pippenger, Strong, 79]; linear hashing [Litwin, 80]

- These resizing operations only add $O(1/B)$ I/Os amortized per insertion; bottleneck is the first search + insert

- Cannot hope for lower than 1 I/O per insertion only if the changes must be committed to disk right away (necessary?)

- Otherwise we probably can lower the amortized insertion cost by buffering, like numerous problems in external memory, e.g. stack, priority queue,… All of them support an insertion in $O(1/B)$ I/Os — the best possible
Dynamic B-trees

Dynamic Hash Tables
Dynamic B-trees for Fast Insertions

Dynamic B-trees for Fast Insertions

- Insertion: $O\left(\frac{\ell}{B} \log_\ell \frac{N}{M}\right)$
- Query: $O\left(\log_\ell \frac{N}{M}\right)$ (omit the $\frac{K}{B}$ output term)
Dynamic B-trees for Fast Insertions

  - Insertion: $O\left(\frac{\ell}{B} \log_\ell \frac{N}{M}\right)$
  - Query: $O\left(\log_\ell \frac{N}{M}\right)$ (omit the $\frac{K}{B}$ output term)

- **Stepped merge tree** [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, VLDB’97]: variant of LSM-tree
  - Insertion: $O\left(\frac{1}{B} \log_\ell \frac{N}{M}\right)$
  - Query: $O\left(\ell \log_\ell \frac{N}{M}\right)$
Dynamic B-trees for Fast Insertions

  - Insertion: $O\left(\frac{\ell}{B} \log_\ell \frac{N}{M}\right)$
  - Query: $O\left(\log_\ell \frac{N}{M}\right)$ (omit the $\frac{K}{B}$ output term)

- **Stepped merge tree** [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, VLDB’97]: variant of LSM-tree
  - Insertion: $O\left(\frac{1}{B} \log_\ell \frac{N}{M}\right)$
  - Query: $O(\ell \log_\ell \frac{N}{M})$

Usually $\ell$ is set to be a constant, then they both have $O\left(\frac{1}{B} \log \frac{N}{M}\right)$ insertion and $O(\log \frac{N}{M})$ query
More Dynamic B-trees

- **Buffer-tree (buffered-repository tree)** [Arge, WADS’95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA’00]

- **Streaming B-tree** [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA’07]

- **Y-tree** [Jermaine, Datta, Omiecinski, VLDB’99]
More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS’95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA’00]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA’07]
- Y-tree [Jermaine, Datta, Omiecinski, VLDB’99]
More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS’95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA’00]

- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA’07]

- Y-tree [Jermaine, Datta, Omiecinski, VLDB’99]

\[
\begin{array}{c|c}
q & u \\
\frac{1}{\log B} & \frac{1}{B} \log B \\
1 & \frac{1}{B} B^\epsilon \\
B^\epsilon & \frac{1}{B}
\end{array}
\]

- Deletions? Standard trick: inserting “delete signals”
More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS’95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA’00]

- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA’07]

- Y-tree [Jermaine, Datta, Omiecinski, VLDB’99]

- Cache-oblivious model [Demaine, Fineman, Iacono, Langerman, Munro, SODA’10]

<table>
<thead>
<tr>
<th>$q$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{\log B}$</td>
<td>$\frac{1}{B} \log B$</td>
</tr>
<tr>
<td>$1$</td>
<td>$\frac{1}{B} B^\epsilon$</td>
</tr>
<tr>
<td>$B^\epsilon$</td>
<td>$\frac{1}{B}$</td>
</tr>
</tbody>
</table>

- Deletions? Standard trick: inserting “delete signals”
- No better solutions known ...
Compare with the rich results in RAM!

- Range reporting
  - $O(\sqrt{\log N / \log \log N})$ insertion and query [Andersson, Thorup, JACM’07]
  - $O(\log N / \log \log N)$ insertion and $O(\log \log N)$ query [Mortensen, Pagh, Pătraşcu, STOC’05]
- Other results that depend on the word size $w$

- Predecessor
  - $\Theta(\sqrt{\log N / \log \log N})$ insertion and query [Andersson, Thorup, JACM’07]

- Partial-sum
  - $\Theta(\log N)$ insertion query [Pătraşcu, Demaine, SODA’04]
Are the EM and DB people just dumb?
Our Main Result

For any dynamic range query index with a query cost of $q$ and an amortized insertion cost of $u$, the following tradeoff holds

$$\begin{align*}
q \cdot \log (uB/q) &= \Omega(\log B), & \text{for } q < \alpha \log B, \alpha \text{ is any constant;} \\
uB \cdot \log q &= \Omega(\log B), & \text{for all } q.
\end{align*}$$
Our Main Result

For any dynamic range query index with a query cost of $q$ and an amortized insertion cost of $u$, the following tradeoff holds

\[
\begin{align*}
q \cdot \log(uB/q) &= \Omega(\log B), & \text{for } q < \alpha \log B, \alpha \text{ is any constant}; \\
ub \cdot \log q &= \Omega(\log B), & \text{for all } q.
\end{align*}
\]

Assuming $\log_B \frac{N}{M} = O(1)$, all the bounds are tight!

Current upper bounds:

<table>
<thead>
<tr>
<th>$q$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log B$</td>
<td>$\frac{1}{B} \log B$</td>
</tr>
<tr>
<td>$1$</td>
<td>$\frac{1}{B} B^e$</td>
</tr>
<tr>
<td>$B^e$</td>
<td>$\frac{1}{B}$</td>
</tr>
</tbody>
</table>
Our Main Result

For any dynamic range query index with a query cost of $q$ and an amortized insertion cost of $u$, the following tradeoff holds

$$
\begin{align*}
q \cdot \log(uB/q) &= \Omega(\log B), & \text{for } q < \alpha \log B, \alpha \text{ is any constant;} \\
uB \cdot \log q &= \Omega(\log B), & \text{for all } q.
\end{align*}
$$

Assuming $\log_B \frac{N}{M} = O(1)$, all the bounds are tight!

Current upper bounds:

<table>
<thead>
<tr>
<th>$\log \frac{N}{M}$</th>
<th>$\frac{q}{\log B}$</th>
<th>$\frac{u}{\frac{1}{B} \log B}$</th>
<th>$\frac{1}{B} \log \frac{N}{M}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{B} B^\epsilon$</td>
<td>$\frac{1}{B} B^\epsilon$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can’t be true for $B = o(\sqrt{\log n \log \log n})$, since the exponential tree achieves $u = q = O(\sqrt{\log n / \log \log n})$ [Andersson, Thorup, JACM’07]. $(n = N/M)$
The real question

How large does $B$ need to be for buffer-tree to be optimal for range reporting?

Known: somewhere between $\Omega(\sqrt{\log n \log \log n})$ and $O(n^c)$
Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]
Lower Bound Model: Dynamic Indexability

- **Indexability**: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]

- Objects are stored in disk blocks of size up to $B$, possibly with redundancy.
Lower Bound Model: Dynamic Indexability

- **Indexability**: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]

  a query reports \{2,3,4,5\}

\[
\begin{array}{ccccccc}
4 & 7 & 9 & 1 & 2 & 4 & 358
\end{array}
\begin{array}{ccccccc}
2 & 6 & 7 & 1 & 8 & 9 & 45
\end{array}
\]

- Objects are stored in disk blocks of size up to $B$, possibly with redundancy.
Lower Bound Model: Dynamic Indexability

- **Indexability**: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]

A query reports \{2,3,4,5\}

\[
\begin{array}{ccc}
4 & 7 & 9 \\
1 & 2 & 4 \\
3 & 5 & 8 \\
2 & 6 & 7 \\
1 & 8 & 9 \\
4 & 5 \\
\end{array}
\]

Cost = 2

- Objects are stored in disk blocks of size up to \(B\), possibly with redundancy.
- The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).
Lower Bound Model: Dynamic Indexability

- **Indexability**: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]

A query reports \(\{2,3,4,5\}\) and the cost is 2.

Objects are stored in disk blocks of size up to \(B\), possibly with redundancy.

The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).

Similar in spirit to popular lower bound models: cell probe model, semigroup model.
Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model.
  - Tradeoff between space ($s$) and query time ($q$).
Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
  - Tradeoff between space ($s$) and query time ($q$)
  - 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$  
    [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
- Tradeoff between space \((s)\) and query time \((q)\)
- 2D range queries: \(s/N \cdot \log q = \Omega(\log(N/B))\) [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
- 2D stabbing queries: \(q \cdot \log(s/N) = \Omega(\log(N/B))\) [Arge, Samoladas, Yi, ESA'04, Algorithmica'99]
Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
- Tradeoff between space ($s$) and query time ($q$)
- 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$ [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
- 2D stabbing queries: $q \cdot \log(s/N) = \Omega(\log(N/B))$ [Arge, Samoladas, Yi, ESA’04, Algorithmica’99]
- 1D range queries: $s = N, q = 1$ trivially
Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
  - Tradeoff between space \((s)\) and query time \((q)\)

- 2D range queries: \(s/N \cdot \log q = \Omega(\log(N/B))\) [Hellerstein, Koutsoupias, Papadimitriou, PODS’97], [Koutsoupias, Taylor, PODS’98], [Arge, Samoladas, Vitter, PODS’99]

- 2D stabbing queries: \(q \cdot \log(s/N) = \Omega(\log(N/B))\) [Arge, Samoladas, Yi, ESA’04, Algorithmica’99]

- 1D range queries: \(s = N, q = 1\) trivially
  - Adding dynamization makes it much more interesting!
Dynamic Indexability

- Still consider only insertions
Dynamic Indexability

- Still consider only insertions

<table>
<thead>
<tr>
<th>Memory of size $M$</th>
<th>Blocks of size $B = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>time $t$: 1 2 7</td>
<td>4 7 9 4 5</td>
</tr>
</tbody>
</table>

← snapshot
Dynamic Indexability

- Still consider only insertions

<table>
<thead>
<tr>
<th>Memory of size $M$</th>
<th>Blocks of size $B = 3$</th>
<th>snapshot</th>
<th>6 inserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>time $t$: 1 2 7</td>
<td>4 7 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time $t + 1$: 1 2 6 7</td>
<td>4 7 9</td>
<td>4 5</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Indexability

- Still consider only insertions

<table>
<thead>
<tr>
<th>Memory of size $M$</th>
<th>Blocks of size $B = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>time $t$: 1 2 7</td>
<td>4 7 9 4 5 ← snapshot</td>
</tr>
<tr>
<td>time $t + 1$: 1 2 6 7</td>
<td>4 7 9 4 5 6 inserted</td>
</tr>
<tr>
<td>time $t + 2$:</td>
<td>4 7 9 1 2 5 6 8 8 inserted</td>
</tr>
</tbody>
</table>
Dynamic Indexability

- Still consider only insertions

<table>
<thead>
<tr>
<th>Memory of size $M$</th>
<th>Blocks of size $B = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>time $t$: 1 2 7</td>
<td>4 7 9</td>
</tr>
<tr>
<td></td>
<td>4 5</td>
</tr>
<tr>
<td>time $t + 1$: 1 2 6 7</td>
<td>4 7 9</td>
</tr>
<tr>
<td></td>
<td>4 5</td>
</tr>
<tr>
<td>time $t + 2$:</td>
<td>4 7 9</td>
</tr>
<tr>
<td></td>
<td>1 2 5</td>
</tr>
<tr>
<td></td>
<td>6 8</td>
</tr>
</tbody>
</table>

- Snapshot

- 6 inserted

- Transition cost = 2

- 8 inserted
Dynamic Indexability

- Still consider only insertions

<table>
<thead>
<tr>
<th>Memory of size $M$</th>
<th>Blocks of size $B = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>time $t$: 1 2 7</td>
<td>4 7 9 4 5 ← snapshot</td>
</tr>
<tr>
<td>time $t+1$: 1 2 6 7</td>
<td>4 7 9 4 5 6 inserted</td>
</tr>
<tr>
<td>time $t+2$:</td>
<td>4 7 9 1 2 5 6 8 8 inserted</td>
</tr>
</tbody>
</table>

- Transition cost = 2

- Update cost: $u = \text{amortized transition cost per insertion}$
The Ball-Shufflimg Problem

$B$ balls → $q$ bins
The Ball-Shuffling Problem

\[ \begin{align*}
B \text{ balls} & \quad \rightarrow \quad q \text{ bins} \\
\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet & \quad \rightarrow \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\
\text{cost} = 1
\end{align*} \]
The Ball-Shuffling Problem

$B$ balls \rightarrow $q$ bins

- \hspace{1cm} \text{cost} = 1
- \hspace{1cm} \text{cost} = 2

\text{cost of putting the ball directly into a bin} = \# \text{ balls in the bin} + 1
The Ball-Shuffling Problem

\( B \) balls \( \rightarrow \) \( q \) bins
The Ball-Shuffling Problem

\( B \) balls \rightarrow \ \text{Shuffle:} \ \rightarrow \ q \ bins

\text{cost} = 5
The Ball-Shuffling Problem

$B$ balls \quad \rightarrow \quad q$ bins

Shuffle:

Cost of shuffling $= \# \text{ balls in the involved bins}$
The Ball-Shuffling Problem

$B$ balls $\rightarrow$ $q$ bins

Cost of shuffling = $\#$ balls in the involved bins

Putting a ball directly into a bin is a special shuffle

Shuffle: $\rightarrow$ cost = 5
The Ball-Shuffling Problem

\[ B \text{ balls} \quad \rightarrow \quad q \text{ bins} \]

Cost of shuffling = \# balls in the involved bins

Putting a ball directly into a bin is a special shuffle

Goal: Accommodating all \( B \) balls using \( q \) bins with minimum cost
The Workload Construction

round 1: • • • • • • •
The Workload Construction

Round 1:

Round 2:
The Workload Construction

round 1:
•
•
•
•
•
round 2:
•
•
•
•
•
round 3:
•
•
•
•
•
...

round $B$:
•
•
•
•
•

keys

time
The Workload Construction

round 1: ● ● ● ● ● ● ● ●
round 2: ● ● ● ● ● ● ● ●
round 3: ● ● ● ● ● ● ● ● ...
round $B$: [●●●●●●●●●●]

Queries that we require the index to cover with $q$ blocks

$\# \text{ queries} \geq 2MB$
The Workload Construction

Queries that we require the index to cover with $q$ blocks 

$\# \text{ queries} \geq 2MB$

Snapshots of the dynamic index considered
There exists a query such that

- The $\leq B$ objects of the query reside in $\leq q$ blocks in all snapshots
- All of its objects are on disk in all $B$ snapshots (we have $\geq MB$ queries)
- The index moves its objects $uB^2$ times in total
The Reduction

An index with update cost \( u \) and query \( A \) gives us a solution to the ball-shuffling game with cost \( uB^2 \) for \( B \) balls and \( q \) bins.
The Reduction

An index with update cost $u$ and query $A$ gives us a solution to the ball-shuffling game with cost $uB^2$ for $B$ balls and $q$ bins.

Lower bound on the ball-shuffling problem:

**Theorem:** The cost of any solution for the ball-shuffling problem is at least

\[
\begin{cases} 
\Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\
\Omega(B \log_q B), & \text{for any } q.
\end{cases}
\]
The Reduction

An index with update cost $u$ and query $A$ gives us a solution to the ball-shuffling game with cost $uB^2$ for $B$ balls and $q$ bins.

Lower bound on the ball-shuffling problem:

**Theorem:** The cost of any solution for the ball-shuffling problem is at least

\[
\begin{cases} 
    \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant}; \\
    \Omega(B \log_q B), & \text{for any } q.
\end{cases}
\]

\[
\begin{align*}
    q \cdot \log(uB/q) &= \Omega(\log B), & \text{for } q < \alpha \log B, \alpha \text{ is any constant}; \\
    uB \cdot \log q &= \Omega(\log B), & \text{for all } q.
\end{align*}
\]
The cost of any solution for the ball-shuffling problem is at least

\[
\begin{align*}
\Omega(q \cdot B^{1+\Omega(1/q)}), & \quad \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\
\Omega(B \log_q B), & \quad \text{for any } q.
\end{align*}
\]
Theorem: The cost of any solution for the ball-shuffling problem is at least
\[
\begin{cases}
\Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant}; \\
\Omega(B \log_q B), & \text{for any } q.
\end{cases}
\]
**Theorem:** The cost of any solution for the ball-shuffling problem is at least

\[
\begin{cases}
\Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\
\Omega(B \log_q B), & \text{for any } q.
\end{cases}
\]

![Diagram](image-url)
Ball-Shuffling Lower Bounds

**Theorem:** The cost of any solution for the ball-shuffling problem is at least

\[
\begin{align*}
&\Omega\left(q \cdot B^{1+\Omega(1/q)}\right), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\
&\Omega\left(B \log_q B\right), & \text{for any } q.
\end{align*}
\]

Cost lower bound

- $B^2$
- $B^{4/3}$
- $B \log B$
Ball-Shuffling Lower Bounds

**Theorem:** The cost of any solution for the ball-shuffling problem is at least

\[
\begin{align*}
\Omega(q \cdot B^{1+\Omega(1/q)}), & \quad \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant}; \\
\Omega(B \log_q B), & \quad \text{for any } q.
\end{align*}
\]

cost lower bound

\[
\begin{array}{c}
B^2 \\
B^{4/3} \\
B \log B \\
B
\end{array}
\]
Dynamic Hash Tables

- B-tree query I/O: $O(\log_B \frac{N}{M})$
- Hash table query I/O: $1 + 1/2^{\Omega(B)}$; insertion the same
Dynamic Hash Tables

- B-tree query I/O: \( O\left(\log_B \frac{N}{M}\right) \)
- Hash table query I/O: \( 1 + 1/2^{\Omega(B)} \); insertion the same

A long-time conjecture in the external memory community:
The insertion cost must be \( \Omega(1) \) I/Os if the query cost is required to be \( O(1) \) I/Os.
Dynamic Hash Tables

- B-tree query I/O: $O(\log_B \frac{N}{M})$
- Hash table query I/O: $1 + 1/2^{\Omega(B)}$; insertion the same

A long-time conjecture in the external memory community:
The insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

Buffering is useless?
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)

Memory:

- $m$
- $2m$
- $4m$
- $8m$
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

memory

- $m$
- $2m$
- $4m$
- $8m$
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c$, $c \leq 1$
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$
- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c$, $c \leq 1$
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c$, $c \leq 1$

\[
x \quad \frac{x}{\beta} \quad 2x
\]

\[
\begin{array}{c}
m \\ 2m \\ 4m \\ 8m
\end{array}
\]

Memory allocation:

\[
\frac{m}{2m} = \frac{1}{2}
\]

\[
\frac{m}{4m} = \frac{1}{4}
\]

\[
\frac{m}{8m} = \frac{1}{8}
\]
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O\left(B^{c-1}\right)$

For some parameter $\beta = B^c$, $c \leq 1$
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O\left(B^{c-1}\right)$
  - Query: $1 + O\left(1/B^{c}\right)$

For some parameter $\beta = B^{c}, \ c \leq 1$

<table>
<thead>
<tr>
<th>Memory</th>
<th>$m$</th>
<th>$2m$</th>
<th>$4m$</th>
<th>$8m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x$</td>
<td>$x/\beta$</td>
<td>$2x$</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O\left(\frac{1}{B} \log \frac{N}{M}\right)$
  - Expected average query: $O(1)$

- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O(B^{c-1})$
  - Query: $1 + O(1/B^c)$
  - Still far from the target $1 + 1/\Omega(2^B)$

For some parameter $\beta = B^c$, $c \leq 1$
Query-Insertion Tradeoff for Successful queries

[Wei, Yi, Zhang, SPAA’09]

Insertion

standard hashing

\[ 1 + 1/2^{\Omega(B)} \]

\[ 1 - O(1/B^{(c-1)/4}) \]

\[ O(1) \]

\[ \Omega(1) \]

\[ O(B^{c-1}) \]

\[ \Omega(B^{c-1}) \]

upper bounds

lower bounds

Query
Indexability Too Strong!

- Naïve solution: For every $B$ items, write to a block.
- Query cost is 1, insertion is $1/B$
Indexability Too Strong!

- Naïve solution: For every $B$ items, write to a block.
  - Query cost is 1, insertion is $1/B$

Too many possible mappings!
Indexability Too Strong!

- Naïve solution: For every $B$ items, write to a block.
  - Query cost is 1, insertion is $1/B$

- Indexability + information-theoretical argument
  
  If with only the information in memory, the hash table cannot locate the item, then querying it takes at least 2 I/Os.
The Abstraction

- Consider the layout of a hash table at any snapshot. Denote all the blocks on disk by $B_1, B_2, \ldots, B_d$. Let $f : U \rightarrow \{1, \ldots, d\}$ be any function computable within memory.

  We divide items inserted into 3 zones with respect to $f$.

  - **Memory zone** $M$: set of items stored in memory. $t_q = 0$.
  - **Fast zone** $F$: set of items $x$ such that $x \in B_{f(x)}$. $t_q = 1$.
  - **Slow zone** $S$: The rest of items. $t_q = 2$. 
The Key

The hash table can employ a family $\mathcal{F}$ of at most $2^M$ distinct $f$’s.

Note that the current $f$ adopted by the hash table is dependent upon the already inserted items, but the family $\mathcal{F}$ has to be fixed beforehand.
How about All Queries? (Latest results)

- We are essentially talking about the membership problem
  - Can’t use indexability model
  - Have to use cell probe model
All queries (the membership problem)

(The cell probe model)

Query

truncated buffer tree

buffer tree

hashing

$1 + 1/2^\Omega(B)$

$0 \quad 1/B \log M/B n \quad 1/B \log n \quad B^\epsilon/B \log n \quad 1$

Insert

$\epsilon$

$\ell \log \ell n, \log \ell n$

$\ell B \log \ell n, \log \ell n$

$B \epsilon \log \ell n,$

$\log \ell n$, 

$\log \ell n,$

$\log \ell n,$
All queries (the membership problem)

(The cell probe model)

Query

truncated buffer tree

buffer tree

hashing

$1 + 1/2^\Omega(B)$

$[\text{Yi, Zhang, SODA'10}]$

$n^\epsilon$

$\frac{\ell}{B} \log \ell n, \log \ell n$

$0, B \log \log n$
All queries (the membership problem)

(The cell probe model)

Query

truncated buffer tree

buffer tree

hashing

[Verbin, Zhang, STOC'10]

[Yi, Zhang, SODA'10]

$1 + \frac{1}{2^{\Omega(B)}}$

$\log_B \log n$
THE BIG BOLD CONJECTURE

All these fundamental data structure problems have the same query-update tradeoff in external memory when $u = o(1)$, for sufficiently large $B$.

Partial-sum: all $B$; Range reporting: $B > n^\epsilon$; Predecessor: unknown.
THE BIG BOLD CONJECTURE

All these fundamental data structure problems have the same query-update tradeoff in external memory when $u = o(1)$, for sufficiently large $B$.

Partial-sum: all $B$; Range reporting: $B > n^\epsilon$; Predecessor: unknown.

Strong implication: The buffer tree (and many of the log method based structures) is simple, practical, versatile, and optimal!
The End

THANK YOU

Q and A