References

Based on

1. A Study of Index Structures for Main Memory DBMS (VLDB 1986)
2. Query Processing in Main Memory DBMS (SIGMOD 1986)
   both papers by Lehman and Carey

v.s.

3. Cache Conscious Indexing for Decision-Support in Main Memory
   Rao and Ross (VLDB 1999)
MM Data Structures

- Array
- AVL Tree
- B/B+ Tree
- Bucket Hashing/Chains
- Extensible/Linear Hashing
- $T$ Tree
MM Data Structures

AVL Tree Node

AVL Tree

Left Ptr

Right Ptr

Data

Control
MM Data Structures

B Tree Node

![B Tree Node Diagram]

B Tree

![B Tree Diagram]

Figure 1—Tree Structured Indices
Disk Versus Main Memory

Index structures designed for main memory are different from those designed for disk-based systems.

<table>
<thead>
<tr>
<th>Chained Bucket Hashing</th>
<th>Linear Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Chained Bucket Hashing Diagram" /></td>
<td><img src="image2" alt="Linear Hashing Diagram" /></td>
</tr>
</tbody>
</table>

3.1. Updates

- Modified Linear Hashing

![Figure 2 - Hashing-based Indices](image3)

- CS848 Spring 2016
- MM Data Structures
- Laboratories
Extendible Hashing

```
<table>
<thead>
<tr>
<th>Directory Depth</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Depth</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node</td>
<td>00</td>
<td>010</td>
<td>011</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

This diagram illustrates extendible hashing, a technique for managing data structures in memory and on disk. The directory depth is shown at the top, with nodes branching out below it. Each node represents a position in the hierarchy, with the depth indicating the level of the structure. The prefix keys used are shown in the directory depth column, helping to navigate the structure efficiently.
Chained Bucket Hashing [Knut73] is a static structure used both in memory and on disk (Figure 2). It is very fast and uses only the addresses that are actually needed for the data items, the space overhead is very small. This structure is often used when the data size is unknown or cannot be calculated in advance. By using table directory, it can be accessed very fast. A Maximum element of A is the Greatest Lower Bound of A. A Least Upper Bound of A is the Minimum element of A. 

Figure 4 - The Bounds of a T-node

-296--
MM Data Structures: T-Tree

Figure 4 — The Bounds of a T-node
MM Data Structures: T-Tree

1. search — same as in other trees

2. insertion:
   ⇒ if space in the bounding node: put there
   ⇒ if no space: put there after removing min value
     to be recursively inserted to the left child

3. deletion:
   ⇒ find and delete
   ⇒ if underflow: move greatest lower bound here

Balancing the T-tree:
- essentially AVL rotations with redistribution of elements
MM Data Structures: T-Tree

1. search — same as in other trees

2. insertion:
   ⇒ if space in the bounding node: put there
   ⇒ if no space: put there after removing min value
   to be recursively inserted to the left child

3. deletion:
   ⇒ find and delete
   ⇒ if underflow: move greatest lower bound here

Balancing the T-tree:
- essentially AVL rotations with redistribution of elements
MM Data Structures: T-Tree Special Rotation

Regular LR Rotation

Special LR Rotation

(slide elements from B to C)
Experiments

1. insert 30K values
2. search 30k values
3. mix search/insert 60:40
4. range queries
5. scan data
6. delete 15k elements
Results 1

Graph 1 - Index Insertion

Graph 2 - Index Search

Graph 3 - Query Mix of 60% Searches
Results 2

Graph 4 - Range Query

Graph 5 - Scan

Graph 6 - Delete
More complex Operators

Main focus: Joins
- nested loops
- index join w/bucket hash
- index join w/tree
- hash join (builds bucket hash on-line)
- sort merge join (sorts on-line)
- tree merge join (assumes the existence of the T-trees!)

Other operators considered: Projections
- based on sorting
- based on hashing
  ⇒ pretty much expected results
Results 1

JOIN TEST 1 (|R1| = |R2|)

- Hash Join
- Tree Join
- Sort Merge
- Tree Merge

Graph 4 — Vary Cardinality
Results 2

JOIN TEST 2 (Vary |R2|)

Hash Join
Tree Join
Sort Merge
Tree Merge

Seconds

|R2| Percentage of |R1|

Graph 5 — Vary Inner Cardinality
Results 3

JOIN TEST 3 (Vary |R1|)

- Hash Join
- Tree Join
- Sort Merge
- Tree Merge

Seconds

|R1| Percentage of |R2|

Graph 6 — Vary Outer Cardinality
Conclusions Drawn (1st two papers)

Observations:

- actual data/data structure overhead for LH/EH (larger nodes—not so good)
- arrays good for lookups, bad otherwise (sparse arrays?)
- trees good both in performance and storage utilization
  ⇒ T-trees WIN (:-)
- general queries:
  ⇒ T-trees best for ordered data  ⇒ LH for unordered data
- sort-merge only for very high-output joins
Cache Conscious Indexing

Observation: modern hardware–caching data is important

Paper discusses

- hash
- interpolation search (arrays)
- AVL trees
- B-trees
- T-trees
- CSS trees

mainly read/search workload from OLAP/Decision Support Apps.
Cache Conscious Indexing

Observation: modern hardware–caching data is important

Paper discusses

1. hash
2. interpolation search (arrays)
3. AVL trees
4. B-trees
5. T-trees
6. CSS trees

mainly read/search workload from OLAP/Decision Support Apps.
Figure 3: Layout of a full CSS-tree

CSS trees

Full CSS-Tree (m=4)

65*4 elements in the array
CSS trees

⇒ pretty much B+tree with 100% fill factor adjusted to cache lines
Theoretical Analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>branching factor</th>
<th># of levels (l)</th>
<th>comparisons per internal node (nComp)</th>
<th>comparisons per leaf node (A_child)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>2</td>
<td>log_2 n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T-trees</td>
<td>2</td>
<td>log_2 (n/m) - 1</td>
<td>1</td>
<td>log_2 m</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>( m/2 )</td>
<td>log_2 m/n</td>
<td>(1 + ( 2/(m+1) )) log_2 m</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>( m + 1 )</td>
<td>log_2 m/n</td>
<td>log_2 m</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>( m )</td>
<td>log_2 m/n</td>
<td>log_2 m</td>
<td>log_2 m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Total comparisons</th>
<th>Moving across Level</th>
<th>Cache Misses ( mK_c ) (&lt;= 1 )</th>
<th>Cache Misses ( mK_c ) (&gt; 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>log_2 n</td>
<td>log_2 n ( A_b )</td>
<td>log_2 n</td>
<td>log_2 n</td>
</tr>
<tr>
<td>T-trees</td>
<td>log_2 n</td>
<td>log_2 n ( D )</td>
<td>log_2 n</td>
<td>log_2 n</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>log_2 n</td>
<td>log_2 m/n ( D )</td>
<td>log_2 m</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>( m+3 )/( m+1 ) log_2 n</td>
<td>log_2 m/n ( A_{fcss} )</td>
<td>log_2 m</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>log_2 n</td>
<td>log_2 m/n ( A_{lcss} )</td>
<td>log_2 m</td>
<td>log_2 m</td>
</tr>
</tbody>
</table>

Table 1: Time analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>Space (indirect)</th>
<th>Typical Value</th>
<th>Space (direct)</th>
<th>Typical Value</th>
<th>RID-Ordered Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>0</td>
<td>0 MB</td>
<td>0</td>
<td>0 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>( 2^k + 2^k )</td>
<td>1.5 MB</td>
<td>( 2^k + 2^k )</td>
<td>1.5 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>( 2^k + 2^k )</td>
<td>2.8 MB</td>
<td>( 2^k + 2^k )</td>
<td>2.8 MB</td>
<td>Y</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>( 2^k + 2^k )</td>
<td>2.7 MB</td>
<td>( 2^k + 2^k )</td>
<td>2.7 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Hash table</td>
<td>( 2^k + 2^k )</td>
<td>5.7 MB</td>
<td>( 2^k + 2^k )</td>
<td>5.7 MB</td>
<td>Y</td>
</tr>
<tr>
<td>T-tree</td>
<td>( (h - 1) nR )</td>
<td>8 MB</td>
<td>( (h - 1) nR )</td>
<td>8 MB</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>( 2^k + 2^k )</td>
<td>11.4 MB</td>
<td>( 2^k + 2^k )</td>
<td>11.4 MB</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 2: Space analysis
Theoretical Analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>branching factor</th>
<th># of levels (l)</th>
<th>comparisons per internal node (nComp)</th>
<th>comparisons per leaf node (A_{child})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>2</td>
<td>log_2 n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T-trees</td>
<td>2</td>
<td>log_2 \frac{n}{m} - 1</td>
<td>1</td>
<td>log_2 m</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>\frac{m}{2}</td>
<td>log_2 \frac{m}{n}</td>
<td>1</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>m + 1</td>
<td>log_{m+1} \frac{n}{m}</td>
<td>(1 + \frac{2}{m+1}) log_2 m</td>
<td>log_2 m</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>m</td>
<td>log_{m} \frac{n}{m}</td>
<td>(1 + \frac{2}{m+1}) log_2 m</td>
<td>log_2 m</td>
</tr>
</tbody>
</table>

Method | Total comparisons | Moving across Level Level | Cache Misses $\frac{mK}{c} \leq 1$ | Cache Misses $\frac{mK}{c} > 1$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>log_2 n</td>
<td>log_2 n * $A_b$</td>
<td>log_2 n</td>
<td>log_2 n</td>
</tr>
<tr>
<td>T-trees</td>
<td>log_2 n</td>
<td>log_2 n * $D$</td>
<td>log_2 n</td>
<td>log_2 n</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>log_2 n</td>
<td>log_{m+1} \frac{n}{m} * $D$</td>
<td>log_{m+1} \frac{n}{m} * $A_{fcss}$</td>
<td>log_{m+1} \frac{n}{m} * $A_{lcss}$</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>\frac{m+3}{m+1} log_{m+1} \frac{m}{m} log_2 n</td>
<td>log_{m+1} \frac{n}{m} * $A_{fcss}$</td>
<td>log_{m+1} \frac{n}{m} * $A_{lcss}$</td>
<td>log_{m+1} \frac{n}{m} * $A_{lcss}$</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>log_2 n</td>
<td>log_{m} \frac{n}{m} * $A_{lcss}$</td>
<td>log_{m} \frac{n}{m} * $A_{lcss}$</td>
<td>log_{m} \frac{n}{m} * $A_{lcss}$</td>
</tr>
</tbody>
</table>

Table 1: Time analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>Space (indirect)</th>
<th>Typical Value</th>
<th>Space (direct)</th>
<th>Typical Value</th>
<th>RID-Ordered Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>$\frac{nK^2}{sc}$</td>
<td>0 MB</td>
<td>$\frac{nK^2}{sc}$</td>
<td>0 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Full CSS-trees</td>
<td>$\frac{nK^2}{sc-K}$</td>
<td>2.5 MB</td>
<td>$\frac{nK^2}{sc-K}$</td>
<td>2.5 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Level CSS-trees</td>
<td>$\frac{nK(P+K)}{sc-P-K}$</td>
<td>2.7 MB</td>
<td>$\frac{nK(P+K)}{sc-P-K}$</td>
<td>2.7 MB</td>
<td>Y</td>
</tr>
<tr>
<td>enhanced B+-trees</td>
<td>$\frac{(h-1)nR}{2nP(K+R)}$</td>
<td>5.7 MB</td>
<td>$\frac{(h-1)nR}{2nP(K+R)}$</td>
<td>5.7 MB</td>
<td>Y</td>
</tr>
<tr>
<td>Hash table</td>
<td>$\frac{hnR}{2P}$</td>
<td>8 MB</td>
<td>$\frac{hnR}{2P}$</td>
<td>48 MB</td>
<td>N</td>
</tr>
<tr>
<td>T-trees</td>
<td>$\frac{2nP(K+R)}{sc-2P}$</td>
<td>11.4 MB</td>
<td>$\frac{2nP(K+R)}{sc-2P}$ + $nR$</td>
<td>51.4 MB</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 2: Space analysis
How does this pan-out in Practice?

(a) Total number of key accesses

(b) Cache misses in secondary level cache

Figure 6: Ultra Sparc II, 16 integers per node
How does this pan-out in Practice?

Figure 5: Varying array size, Ultra Sparc II

(a) 8 integers per node  
(b) 16 integers per node

Figure 5: Varying array size, Ultra Sparc II
Conclusions (paper #3)

T-Trees aren’t as good as B+ trees

*if one pays enough attention to cache (CSS-tree)*

Similar results reported in

Cache Conscious Trees–How Do They Perform on Contemporary Commodity Microprocessors (ICCSA 2007)

![Graphs showing search performance comparison between T-Trees, B+-Trees, CSB+-Trees, and CST-Trees](image)

**Fig. 5.** Search performances in machine-A (1CPU, dual-cores, and shared L2 cache)
Conclusions (paper #3)

T-Trees aren’t as good as B+ trees

*if one pays enough attention to cache (CSS-tree)*

Similar results reported in

*Cache Conscious Trees—How Do They Perform on Contemporary Commodity Microprocessors (ICCSA 2007)*

---

**Fig. 5.** Search performances in machine-A (1CPU, dual-cores, and shared L2 cache)
Meta Conclusions

How is such a discrepancy possible???

1. different hardware: VAX 11/750 vs SPARC II
   ⇒ cache behaves like a very very fast disk (but still a disk)

2. more careful analysis (and coding) of algorithms pays off
How is such a discrepancy possible???

1. different *hardware*: VAX 11/750 vs SPARC II
   ⇒ cache behaves like a very very fast disk (but still a disk)

2. more careful analysis (and coding) of algorithms pays off