CompSci 516
Database Systems
Lecture 7-8
Index
(B+-Tree and Hash)

Instructor: Sudeepa Roy
Announcements

• HW1 and project proposal deadlines next week:
  – Due on 09/27 (Thurs), 11:55 pm, no late days
  – HW1 submission on gradescope (code on piazza)
  – Proposal submission on sakai (one per group)
  – Project ideas on sakai

• Do not forget to start homeworks early!
  – Especially for the next two HW
Reading Material

- [RG]
  - Storage: Chapters 8.1, 8.2, 8.4, 9.4-9.7
  - Index: 8.3, 8.5
  - Tree-based index: Chapter 10.1-10.7
  - Hash-based index: Chapter 11

Additional reading
- [GUW]
  - Chapters 8.3, 14.1-14.4

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Recap

• Storage:
  – Files -> Records -> Fields
  – Fixed and variable length

• Index
  – Search key k -> Data entry k* -> Record
  – Alternative 1/2/3 for k*
  – Primary/secondary, clustered/unclustered

• Today
  – B+ tree index
  – Hash based index
Tree-based Index
and $B^+$-Tree
Range Searches

• "Find all students with gpa > 3.0"
  – If data is in sorted file, do binary search to find first such student, then scan to find others.
  – Cost of binary search can be quite high.
Index file format

- Simple idea: Create an “index file”
  - `<first-key-on-page, pointer-to-page>`, sorted on keys

Can do binary search on (smaller) index file but may still be expensive: apply this idea repeatedly
Indexed Sequential Access Method (ISAM)

- Leaf-pages contain data entry – also some overflow pages
- DBMS organizes layout of the index – a static structure
- If a number of inserts to the same leaf, a long overflow chain can be created
  - affects the performance

Leaf pages contain data entries.
B+ Tree

- Most Widely Used Index
  - a dynamic structure
- Insert/delete at $\log_F N$ cost = height of the tree (cost = I/O)
  - $F = \text{fanout}$, $N = \text{no. of leaf pages}$
  - tree is maintained height-balanced
- Minimum 50% occupancy
  - Each node contains $d \leq m \leq 2d$ entries
  - Root contains $1 \leq m \leq 2d$ entries
  - The parameter $d$ is called the order of the tree
- Supports equality and range-searches efficiently
B+ Tree Indexes

- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have index entries; only used to direct searches:

```
index entry
```

```
P_0  K_1  P_1  K_2  P_2  \ldots  K_m  P_m
```
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
Example B+ Tree

Note how data entries in leaf level are sorted

Entries < 17

Entries >= 17

- Find
  - 28*?
  - 29*?
  - All > 15* and < 30*
B+ Trees in Practice

• Typical order: $d = 100$. Typical fill-factor: 67%
  – average fanout $F = 133$

• Typical capacities:
  – Height 4: $133^4 = 312,900,700$ records
  – Height 3: $133^3 = 2,352,637$ records

• Can often hold top levels in buffer pool:
  – Level 1 = 1 page = 8 Kbytes
  – Level 2 = 133 pages = 1 Mbyte
  – Level 3 = 17,689 pages = 133 MBytes
Inserting a Data Entry into a B+ Tree

- Find correct leaf L
- Put data entry onto L
  - If L has enough space, done
  - Else, must split L
    - into L and a new node L2
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to L2 into parent of L.

- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key
    - Contrast with leaf splits
- Splits “grow” tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

See this slide later,
First, see examples on the next few slides
Inserting 8* into Example B+ Tree

- **Copy-up**: 5 appears in leaf and the level above
- **Observe how minimum occupancy is guaranteed**

Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)
• Note difference between copy-up and push-up
• What is the reason for this difference?
• All data entries must appear as leaves
  — (for easy range search)
• No such requirement for indexes
  — (so avoid redundancy)

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)
Example B+ Tree After Inserting 8*

Root

- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries (insert 8 to the 2\textsuperscript{nd} leaf node from left and copy it up instead of 13)
  - however, this is usually not done in practice – since need to access 1-2 extra pages always (for two siblings), and average occupancy may remain unaffected as the file grows
Announcements: 9/25

• Private project threads created on piazza
  – Please use these threads (and not emails) for all communications on your project

• Project proposal/HW1 deadline
  – Thursday 9/27, 11:55 pm
  – Deadline is strict, submit early
Deleting a Data Entry from a B+ Tree

Each non-root node contains $d \leq m \leq 2d$ entries

- Start at root, find leaf $L$ where entry belongs
- Remove the entry
  - If $L$ is at least half-full, done!
  - If $L$ has only $d-1$ entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$)
    - If re-distribution fails, merge $L$ and sibling
- If merge occurred, must delete entry (pointing to $L$ or sibling) from parent of $L$
- Merge could propagate to root, decreasing height

See this slide later, First, see examples on the next few slides
Example Tree: Delete 19*

- We had inserted 8*
- Now delete 19*
- Easy
Example Tree: Delete 19*

After deleting 19*
Example Tree: Delete 20*

Before deleting 20*
Example Tree: Delete 20*

- < 2 entries in leaf-node
- Redistribute

After deleting 20* - step 1
Example Tree: Delete 20*

- Notice how middle key is copied up

After deleting 20* - step 2

End of Lecture 7
Example Tree: ... And Then Delete 24*

Before deleting 24*
Example Tree: ... And Then Delete 24*

- Once again, imbalance at leaf
- Can we borrow from sibling(s)?
- No – d-1 and d entries (d = 2)
- Need to merge

Root

After deleting 24*
- Step 1
Example Tree: ... And Then Delete 24*

- Imbalance at parent
- Merge again
- But need to “pull down” root index entry

Root

Observe `toss’ of old index entry 27

After deleting 24*
- Step 2

because, three index 5, 13, 30 but five pointers to leaves
Final Example Tree

Root

2* 3* 5* 7* 8* 14* 16* 22* 27* 29* 33* 34* 38* 39*
5 13 17 30
Example of Non-leaf Re-distribution

- An intermediate tree is shown
- In contrast to previous example, can re-distribute entry from left child of root to right child
Intuitively, entries are re-distributed by `pushing through’ the splitting entry in the parent node.

- It suffices to re-distribute index entry with key 20; we’ve re-distributed 17 as well for illustration.
Duplicates

• **First Option:**
  – The basic search algorithm assumes that all entries with the same key value resides on the same leaf page
  – If they do not fit, use overflow pages (like ISAM)

• **Second Option:**
  – Several leaf pages can contain entries with a given key value
  – Search for the left most entry with a key value, and follow the leaf-sequence pointers
  – Need modification in the search algorithm

• **if** $k^* = <k, \text{rid}>$, **several entries have to be searched**
  – Or include rid in $k$ – becomes unique index, no duplicate
  – If $k^* = <k, \text{rid-list}>$, same solution, but if the list is long, again a single entry can span multiple pages
A Note on `Order’

• Order (d)
  – denotes minimum occupancy

• replaced by physical space criterion in practice (`at least half-full’)
  – Index pages can typically hold many more entries than leaf pages
  – Variable sized records and search keys mean different nodes will contain different numbers of entries.
  – Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3))
Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches

- **ISAM is a static structure**
  - Only leaf pages modified; overflow pages needed
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant

- **B+ tree is a dynamic structure**
  - Inserts/deletes leave tree height-balanced; \( \log_F N \) cost
  - High fanout (\( F \)) means depth rarely more than 3 or 4
  - Almost always better than maintaining a sorted file
  - Most widely used index in database management systems because of its versatility.
    - One of the most optimized components of a DBMS

- **Next: Hash-based index**
Hash-based Index
Hash-Based Indexes

• Records are grouped into buckets
  – Bucket = primary page plus zero or more overflow pages

• Hashing function $h$:
  – $h(r) =$ bucket in which (data entry for) record $r$ belongs
  – $h$ looks at the search key fields of $r$
  – No need for “index entries” in this scheme
Example: Hash-based index

Index organized file hashed on AGE, with Auxiliary index on SAL

Employee File hashed on AGE

Alternative 1

File of <SAL, rid> pairs hashed on SAL

Alternative 2
Introduction

• Hash-based indexes are best for equality selections
  – Find all records with name = “Joe”
  – Cannot support range searches
  – But useful in implementing relational operators like join (later)

• Static and dynamic hashing techniques exist
  – trade-offs similar to ISAM vs. B+ trees
Static Hashing

- Pages containing data = a collection of buckets
  - each bucket has one primary page, also possibly overflow pages
  - buckets contain data entries $k^*$

```
h(key) mod N
```

```
key
```

```
h
```

```
Primary bucket pages
```

```
Overflow pages
```

Duke CS, Fall 2018
CompSci 516: Database Systems
Static Hashing

• # primary pages fixed
  – allocated sequentially, never de-allocated, overflow pages if needed.
• $h(k) \mod N = \text{bucket to which data entry with key } k \text{ belongs}$
  – $N = \# \text{ of buckets}$

![Diagram of static hashing with bucket allocation and overflow pages.](attachment:static_hashing_diagram.png)
Static Hashing

• Hash function works on search key field of record r
  – Must distribute values over range 0 ... N-1
  – $h(\text{key}) = (a \times \text{key} + b)$ usually works well
    • bucket = $h(\text{key}) \mod N$
  – $a$ and $b$ are constants – chosen to tune $h$

• Advantage:
  – #buckets known – pages can be allocated sequentially
  – search needs 1 I/O (if no overflow page)
  – insert/delete needs 2 I/O (if no overflow page) (why 2?)

• Disadvantage:
  – Long overflow chains can develop if file grows and degrade performance (data skew)
  – Or waste of space if file shrinks

• Solutions:
  – keep some pages say 80% full initially
  – Periodically rehash if overflow pages (can be expensive)
  – or use Dynamic Hashing
Dynamic Hashing Techniques

- Extendible Hashing
- Linear Hashing
Extendible Hashing

• Consider static hashing
• Bucket (primary page) becomes full

• Why not re-organize file by doubling # of buckets?
  – Reading and writing (double #pages) all pages is expensive

• Idea: Use directory of pointers to buckets
  – double # of buckets by doubling the directory, splitting just the bucket that overflowed
  – Directory much smaller than file, so doubling it is much cheaper
  – Only one page of data entries is split
  – No overflow page (new bucket, no new overflow page)
  – Trick lies in how hash function is adjusted
Example

- Directory is array of size 4
  - each element points to a bucket
  - #bits to represent = \( \log 4 = 2 \) = global depth

- To find bucket for search key \( r \)
  - take last global depth # bits of \( h(r) \)
  - assume \( h(r) = r \)
  - If \( h(r) = 5 \) = binary 101
  - it is in bucket pointed to by 01
Example

Insert:
- If bucket is full, split it
- allocate new page
- re-distribute

Suppose inserting 13*
- binary = 1101
- bucket 01
- Has space, insert
Example

Insert:
- If bucket is full, split it
- allocate new page
- re-distribute

Suppose inserting 20*
- binary = 10100
- bucket 00
- Already full
- To split, consider last three bits of 10100
- Last two bits the same 00 – the data entry will belong to one of these buckets
- Third bit to distinguish them
Example

Global depth: Max # of bits needed to tell which bucket an entry belongs to to

Local depth: # of bits used to determine if an entry belongs to this bucket
- also denotes whether a directory doubling is needed while splitting
- no directory doubling needed when $9^* = 1001$ is inserted (LD< GD)

![Diagram showing bucket distribution and depth]
When does bucket split cause directory doubling?

• Before insert, local depth of bucket = global depth
• Insert causes local depth to become > global depth
• directory is doubled by copying it over and `fixing’ pointer to split image page
Comments on Extendible Hashing

• If directory fits in memory, equality search answered with one disk access (to access the bucket); else two.
  – 100MB file, 100 bytes/rec, 4KB page size, contains $10^6$ records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  – Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large
  – Multiple entries with same hash value cause problems

• Delete:
  – If removal of data entry makes bucket empty, can be merged with `split image`  
  – If each directory element points to same bucket as its split image, can halve directory.
Linear Hashing

- This is another dynamic hashing scheme
  - an alternative to Extendible Hashing
- LH handles the problem of long overflow chains
  - without using a directory
  - handles duplicates and collisions
  - very flexible w.r.t. timing of bucket splits
Linear Hashing: Basic Idea

- Use a family of hash functions $h_0$, $h_1$, $h_2$, ...
  - $h_i(key) = h(key) \mod(2^iN)$
  - $N =$ initial # buckets
  - $h$ is some hash function (range is not 0 to $N-1$)
  - If $N = 2^{d_0}$, for some $d_0$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d_0 + i$
    - Note: $h_i(key) = h(key) \mod(2^{d_0+i})$
  - $h_{i+1}$ doubles the range of $h_i$
    - if $h_i$ maps to $M$ buckets, $h_{i+1}$ maps to $2M$ buckets
      - similar to directory doubling
  - Suppose $N = 32$, $d_0 = 5$
    - $h_0 = h \mod 32$ (last 5 bits)
    - $h_1 = h \mod 64$ (last 6 bits)
    - $h_2 = h \mod 128$ (last 7 bits) etc.
Linear Hashing: Rounds

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin
- During round $\text{Level}$, only $h_{\text{Level}}$ and $h_{\text{Level}+1}$ are in use
- The buckets from start to last are split sequentially
  - this doubles the no. of buckets
- Therefore, at any point in a round, we have
  - buckets that have been split
  - buckets that are yet to be split
  - buckets created by splits in this round
Overview of LH File

• In the middle of a round Level – originally 0 to $N_{Level}$

Buckets that existed at the beginning of this round:
this is the range of $h_{Level}$

Next-1

Next

Buckets that will be split:

Next

Buckets split in this round:
if $h_{Level}(r)$ is in this range, must use $h_{Level+1}(r)$ to decide if entry is in `split image' bucket.

if $h_{Level}(r)$ is in this range, no need

`split image' buckets:
created (through splitting of other buckets) in this round

• Buckets 0 to Next-1 have been split
• Next to $N_{Level}$ yet to be split
• Round ends when all $N_{Level}$ initial (for round Level) buckets are split
Overview of LH File

• In the middle of a round Level – originally 0 to \(N_{\text{Level}}\)

Bucket to be split

Next to \(N_{\text{Level}}\) yet to be split

Round ends when all \(N_R\) initial (for round R) buckets are split

• Buckets 0 to Next-1 have been split

Search: To find bucket for data entry \(r\), find \(h_{\text{Level}}(r)\):
• If \(h_{\text{Level}}(r)\) in range ‘Next to \(N_{\text{Level}}\)’, \(r\) belongs here.
• Else, \(r\) could belong to bucket \(h_{\text{Level}}(r)\) or \(h_{\text{Level}}(r)+N_R\)
• Apply \(h_{\text{Level}+1}(r)\) to find out

Buckets that existed at the beginning of this round:
this is the range of \(h_{\text{Level}}\)

\(N_{\text{Level}}\)

Next

Next - 1

Bucket to be split

\(0\)

Buckets split in this round:
if \(h_{\text{Level}}(r)\) is in this range, must use \(h_{\text{Level}+1}(r)\) to decide if entry is in
’split image’ bucket.

if \(h_{\text{Level}}(r)\)
is in this range, no need

’split image’ buckets: created (through splitting of other buckets) in this round
Linear Hashing: Insert

• **Insert:** Find bucket by applying $h_{\text{Level}} / h_{\text{Level+1}}$:
  – If bucket to insert into is full:
    1. Add overflow page and insert data entry
    2. Split Next bucket and increment Next

• **Note:** We are going to assume that a split is `triggered’ whenever an insert causes the creation of an overflow page, but in general, we could impose additional conditions for better space utilization ([RG], p.380)
## Example of Linear Hashing

### Level 0

- $N_0 = 4 = 2^{d_0}$, $d_0 = 2$

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Next = 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>000</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>32<em>44</em>36*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>001</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>9<em>25</em>5*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>010</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>14<em>18</em>10<em>30</em></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>011</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>31<em>35</em>7<em>11</em></td>
<td></td>
</tr>
</tbody>
</table>

*Data entry $r$ with $h(r) = 5$*

- Insert $43^* = 101011$
- $h_0(43) = 11$
- Full
- Insert in an overflow page
- Need a split at Next (=0)
- Entries in 00 is distributed to 000 and 100

(This info is for illustration only!)

(The actual contents of the linear hashed file)
Example of Linear Hashing

Level=0, \( N_0 = 4 = 2^{d_0} \), \( d_0=2 \)

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>32<em>44</em>36*</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>(The actual contents of the linear hashed file)</td>
</tr>
</tbody>
</table>

(This info is for illustration only!)

Level=0, \( N_0 = 4 = 2^{d_0} \), \( d_0=2 \)

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>32*</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>43*</td>
</tr>
</tbody>
</table>

Next is incremented after split
• Note the difference between overflow page of 11 and split image of 00 (000 and 100)
Example of Linear Hashing

• Search for 18* = 10010
  • between Next (=1) and 4
  • this bucket has not been split
    • 18 should be here

• Search for 32* = 100000 or 44* = 101100
• Between 0 and Next-1
  • Need \( h_1 \)

• Not all insertion triggers split
  • Insert 37* = 100101
  • Has space

• Splitting at Next?
  • No overflow bucket needed
  • Just copy at the image/original

• Next = \( N_{\text{level}-1} \) and a split?
  • Start a new round
  • Increment Level
  • Next reset to 0

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
h & h & \text{PRIMARY PAGES} & \text{OVERFLOW PAGES} \\
\hline
1 & 0 & 32^* & 32^* \\
000 & 00 & 9^* 25^* 5^* \\
001 & 01 & 14^* 18^* 10^* 30^* \\
010 & 10 & 31^* 35^* 7^* 11^* \\
011 & 11 & \\
100 & 00 & 44^* 36^* 43^* \\
\end{array}
\]
Example of Linear Hashing

- Not all insertion triggers split
- Insert 37* = 100101
  - Has space

Level=0, \( N_0 = 4 = 2^{d_0} \), \( d_0=2 \)
Example of Linear Hashing

- Splitting at Next?
  - No overflow bucket needed
  - Just copy at the image/original

\[ \text{Level}=0, \quad N_0 = 4 = 2^{d_0}, \quad d_0 = 2 \]

\[ \begin{array}{c|l}
  h & h \\
  \hline
  1 & 0 \\
  000 & 00 \\
  001 & 01 \\
  010 & 10 \\
  011 & 11 \\
  100 & 00 \\
  101 & 01 \\
\end{array} \]

\[ \begin{array}{c|l}
  \text{PRIMARY PAGES} & \text{OVERFLOW PAGES} \\
  \hline
  32* & \\
  9* 25* 5* 37* & \\
  14* 18* 10* 30* & \\
  31* 35* 7* 11* & 43* \\
  44* 36* & \\
\end{array} \]

\[ \begin{array}{c|l}
  \text{Level}=0, \quad N_0 = 4 = 2^{d_0}, \quad d_0 = 2 \\
  h & h \\
  \hline
  1 & 0 \\
  000 & 00 \\
  001 & 01 \\
  010 & 10 \\
  011 & 11 \\
  100 & 00 \\
  101 & 01 \\
\end{array} \]

\[ \begin{array}{c|l}
  \text{PRIMARY PAGES} & \text{OVERFLOW PAGES} \\
  \hline
  32* & \\
  9* 25* & \\
  14* 18* 10* 30* & \\
  31* 35* 7* 11* & 43* \\
  44* 36* & \\
  5* 37* 29* & \\
\end{array} \]

\[ \text{insert } 29^* = 11101 \]
Example: End of a Round

insert 50* = 110010

(after inserting 22*, 66*, 34* - check yourself)

Level=0, \( N_0 = 4 = 2^{d_0} \), \( d_0 = 2 \)

Level=1, \( N_1 = 8 = 2^{d_1} \), \( d_1 = 3 \)

(Please refer to the diagram for a visual explanation.)
LH vs. EH

They are very similar
- $h_i$ to $h_{i+1}$ is like doubling the directory
- LH: avoid the explicit directory, clever choice of split
- EH: always split – higher bucket occupancy

Uniform distribution: LH has lower average cost
- No directory level

Skewed distribution
- Many empty/nearly empty buckets in LH
- EH may be better
Summary

• Hash-based indexes: best for equality searches, cannot support range searches.
• Static Hashing can lead to long overflow chains.
• Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it
  – Duplicates may still require overflow pages
  – Directory to keep track of buckets, doubles periodically
  – Can get large with skewed data; additional I/O if this does not fit in main memory
Summary

• Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages
  – Overflow pages not likely to be long
  – Duplicates handled easily

• For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed
  – bad