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**

![Diagram of B+ Tree](image)
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  |  *  *  *  | 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

- **Find**
  - 28*?
  - 29*?
  - All > 15* and < 30*

Note how data entries in leaf level are sorted
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.)
Inserting 8* into Example B+ Tree

• 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)

Need to split parent

STEP-2

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*

- 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
Deleting a Data Entry from a B+ Tree

- 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

Each non-root node contains \( d \leq m \leq 2d \) entries
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
Example Tree: ... And Then Delete 24*

Before deleting 24*
• Once again, imbalance at leaf
• Can we borrow from sibling(s)?
• No – d-1 and d entries (d = 2)
• Need to merge
Example Tree: ... And Then Delete 24*

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

After deleting 24* - Step 2

Observe ‘toss’ of old index entry 27

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
After Re-distribution

• 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.
• **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 \( k^* = \langle k, \text{rid} \rangle \), several entries have to be searched
  – Or include rid in \( k \) – becomes unique index, no duplicate
  – If \( k^* = \langle k, \text{rid-list} \rangle \), some 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

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*

![Diagram showing hashing with primary bucket pages and overflow pages]

h(key) mod N

key

h

Primary bucket pages

Overflow pages

N-1

0

2

...
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](image)
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
  - 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.

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).

Bucket A
- LOCAL DEPTH: 2
- GLOBAL DEPTH: 32*16

Bucket B
- LOCAL DEPTH: 2
- GLOBAL DEPTH: 1* 5* 21*13

Bucket C
- LOCAL DEPTH: 2
- GLOBAL DEPTH: 10*

Bucket D
- LOCAL DEPTH: 2
- GLOBAL DEPTH: 15* 7* 19*

Bucket A2
- LOCAL DEPTH: 2
- GLOBAL DEPTH: 4* 12* 20* ("split image" of Bucket A)

Bucket A2 (new "split image" of Bucket A)
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 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 \( N_{\text{Level}} \)

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

Next - 1

Next

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.

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

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

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

- In the middle of a round **Level** – originally 0 to $N_{\text{Level}}$

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

Next to $N_{\text{Level}}$ yet to be split

Round ends when all $N_R$ initial (for round $R$) buckets are 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
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, \( N_0 = 4 = 2^{d_0} \), \( d_0=2 \)

- 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>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 44 36*</td>
<td>000 00</td>
</tr>
</tbody>
</table>
| 1 | 9 25 5* | 001 01 | Data entry \( r \) with \( h(r)=5 \)
| 00 | 14 18 10 30* | 010 10 | Primary bucket page |
| 01 | 31 35 7 11* | 011 11 | (The actual contents of the linear hashed file) |
| 10 | 100 00 | 44 36* | |

- **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 = $N_{level-1}$ and a split?
  • Start a new round
  • Increment Level
  • Next reset to 0
Example of Linear Hashing

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

<table>
<thead>
<tr>
<th>Level=0, $N_0 = 4 = 2^{d_0}$, $d_0=2$</th>
<th>Level=0, $N_0 = 4 = 2^{d_0}$, $d_0=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
</tr>
</tbody>
</table>
Example of Linear Hashing

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

\[
\begin{array}{c|c|c|c|c}
\text{h} & \text{h} & \text{PRIMARY} & \text{OVERFLOW} \\
1 & 0 & \text{PAGES} & \text{PAGES} \\
000 & 00 & 32* & \ \\
001 & 01 & 9* 25* 5* 37* & \ \\
010 & 10 & 14* 18* 10* 30* & \ \\
011 & 11 & 31* 35* 7* 11* & 43* \\
100 & 00 & 44* 36* & \ \\
\end{array}
\]

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

insert 29* = 11101

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

After inserting 22*, 66*, 34* (check yourself)

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

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

Insert 50* = 110010

Next=3

Next=0

Duke CS, Fall 2018

CompSci 516: Database Systems
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