Database Design and Implementation

CS 645

Indexes
## High-level overview: indexes

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
<th>salary</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>006</td>
<td>19</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>005</td>
<td>20</td>
<td>55k</td>
<td>...</td>
</tr>
<tr>
<td>004</td>
<td>25</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>007</td>
<td>30</td>
<td>80k</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td>35</td>
<td>75k</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td>35</td>
<td>70k</td>
<td>...</td>
</tr>
<tr>
<td>001</td>
<td>40</td>
<td>65k</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
<th>salary</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>006</td>
<td>19</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>004</td>
<td>25</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>005</td>
<td>20</td>
<td>55k</td>
<td>...</td>
</tr>
<tr>
<td>001</td>
<td>40</td>
<td>65k</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td>35</td>
<td>70k</td>
<td>...</td>
</tr>
<tr>
<td>007</td>
<td>30</td>
<td>80k</td>
<td>...</td>
</tr>
</tbody>
</table>

- **data file = index file**
- **clustered index**

- **index file**
- **unclustered index**
Clustered index

- File is sorted on the index attribute
- Only one per table
Unclustered Index

Several per table

Index File

Data File
Hash-Based Index

Good for point queries but not range queries

H2

h2(age) = 00

18
18
20
22
19

h2(age) = 01

age

H1

h1(sid) = 00

10 21
20 20

h1(sid) = 01

30 18
40 19

h1(sid) = 11

50 22
60 18

unclustered index

clustered index
B+ Trees

Search trees

Idea in B-trees
- Make 1 node = 1 block
- Keep tree balanced in height

Idea in B+ trees
- Make leaves into a linked list: facilitates range queries
B+ Tree Indexes

Leaf pages contain *data entries*:

- Data entries are *sorted* by the search key value
- Leaf pages are chained using prev & next pointers
Non-leaf pages have *index entries*, used **only** to direct searches.
B+ Tree: Most widely used Index

- Height-balanced with arbitrary inserts/deletes

- Fanout (F): number of child pointers of non-leaf node

- Height: $H = \log_F N$
  - $N = \text{number of leaves}$
B+ Tree: Most widely used Index

- Minimum 50% occupancy (except for root)
- Each node contains m entries
  - Can be computed using page size, key size, pointer size.

Diagram:
- Index Entries (Direct search)
- Data Entries ("Sequence set")
B+ Tree: Most widely used Index

- Parameter $d = \text{the order}$
- Each interior node has $d \leq m \leq 2d$ keys (except root)
- Each leaf node has $d \leq m \leq 2d$ keys

Each node also has $m+1$ pointers

Keys $k < 27$
Keys $27 \leq k < 30$
Keys $30 \leq k < 43$
Keys $43 \leq k$

Next leaf

Data records
The search structure (non-leaf part) forms a partitioning of an ordered domain (e.g., integer, string)
Searches in a B+ Tree

Data entries are sorted.
Searches in a B+ Tree

find key 24

24 > 17

24 < 27
Insertions

- Find correct leaf $L$ via a top-down search.
- Put data entry onto $L$.
  - If $L$ has enough space, done!
  - Else, must **split** $L$ (into $L$ and a new node $L_2$)
    - Redistribute entries evenly, **copy up** middle key $k$, insert $(k, \text{pointer to } L_2)$ into parent of $L$.
  - Splitting can happen recursively to non-leaf nodes
    - Redistribute entries evenly, but **push up** middle key.
      - (Contrast with leaf splits.)
- Splits “grow” the tree!
  - First **wider**, then **one level taller** when the root splits.
Example

Inserting 8*
Example

Entry to be inserted in parent node. (Note that 5 is *copied up* and continues to appear in the leaf)

Inserting 8*

```
Entry	to	be	inserted	in	parent	node.
(Note	that	5	is
copied	up	and	continues
to	appear	in	the	leaf)
```
Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index.)
Root has split leading to increased height
Deletions

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is at least half-full, done!
  - If $L$ has only $\lceil n/2 \rceil - 1$ entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling. Must delete index entry (pointing to $L$ or sibling) from parent of $L$.
- Merge could propagate to root, decreasing height.
Example

Delete 19*
Delete 20*
After Deletions

Root

17

5 13

2* 3* 5* 7* 8* 14* 16*

22* 24*

27 30

27* 29*

33* 34* 38* 39*
Example

Delete 24*
… deleting 24*

Must merge nodes.
*Toss* index entry (right)
*Pull down* of index entry (below).
Example of Non-leaf Re-distribution

Tree is shown above *during deletion* of 24*. (What could be a possible initial tree?)

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.
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only ‘direct traffic’; can often compress them.
  - E.g., adjacent index entries with search key values 
    \[ \{Dave Jones, David Smith, Devarakonda Murthy\} \]
  - Can we abbreviate David Smith to Dav?
    - Not correct! Can only compress David Smith to Davi.
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.
Bulk Loading of a B+ Tree

- Already have a large collection of records. Want to create a B+ tree. Doing so by repeatedly inserting records?
- Slow due to repeated traversals and splits.
- Not necessarily the optimal structure. An example?
- Low storage utility. An example?

- **Bulk Loading** can be done much more efficiently!
Bulk Loading Algorithm

**Initialization:**
- Sort all data entries
- Insert pointer to the first (leaf) page in a new (root) page.
Index entries for leaf pages always go into r*, right-most index page just above the leaf level.

When the r* node fills up, it splits.

Split may go up the right-most path to the root.
Multiple Inserts vs. Bulk Loading

**Multiple inserts:**
- Slow due to I/O cost (and locking) overhead.
- No sequential storage of leaf pages.
- Sometimes low storage utility.

**Bulk Loading:**
- Fewer I/Os during build.
- Leaf pages will be stored sequentially (and linked).
- Can control “fill factor” on pages.
The Database Tuning Problem

- We are given a workload description
  - List of queries and their frequencies
  - List of updates and their frequencies
  - Performance goals for each type of query

- Perform physical database design
  - Choice of indexes
  - Tuning the conceptual schema
    - Denormalization, vertical and horizontal partition
  - Query and transaction tuning
The Index Selection Problem

- Given a database schema (tables, attributes)
- Given a “query workload”:
  - Workload = a set of (query, frequency) pairs
  - The queries may be both SELECT and updates
  - Frequency = either a count, or a percentage

- Select a set of indexes that optimizes the workload

In general this is a very hard problem
Index selection decisions

- To index or not to index?
- Which key?
- Multiple keys?
- Clustered or unclustered?
- Hash or trees?
Make some attribute $K$ a search key if the \texttt{WHERE} clause contains:

- An exact match on $K$
- A range predicate on $K$
- A join on $K$
The Index Selection Problem 1

$V(M, N, P)$

Your workload is this

100,000 queries:

```
SELECT * 
FROM V 
WHERE N=?
```

100 queries:

```
SELECT * 
FROM V 
WHERE P=?
```

What indexes?
The Index Selection Problem 1

\[ V(M, N, P) \]

Your workload is this

100,000 queries:

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad V \\
\text{WHERE} & \quad N=? \\
\end{align*}
\]

100 queries:

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad V \\
\text{WHERE} & \quad P=? \\
\end{align*}
\]

A: \( V(N) \) and \( V(P) \) (hash tables or B-trees)
The Index Selection Problem 2

V(M, N, P)

Your workload is this

100,000 queries:

```
SELECT * 
FROM V 
WHERE N>? and N<?
```

100 queries:

```
SELECT * 
FROM V 
WHERE P=?
```

100,000 queries:

```
INSERT INTO V 
VALUES (?, ?, ?)
```

What indexes?
The Index Selection Problem 2

V(M, N, P)

Your workload is this

100,000 queries:

```
SELECT * 
FROM V 
WHERE N>? and N<?
```

100 queries:

```
SELECT * 
FROM V 
WHERE P=?
```

100,000 queries:

```
INSERT INTO V 
VALUES (?, ?, ?)
```

A: definitely V(N) must B-tree; unsure about V(P)
The Index Selection Problem 3

Your workload is this

<table>
<thead>
<tr>
<th>100,000 queries:</th>
<th>1,000,000 queries:</th>
<th>100,000 queries:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> *</td>
<td><strong>SELECT</strong> *</td>
<td><strong>INSERT INTO</strong> V</td>
</tr>
<tr>
<td><strong>FROM</strong> V</td>
<td><strong>FROM</strong> V</td>
<td><strong>VALUES</strong> (?, ?, ?)</td>
</tr>
<tr>
<td><strong>WHERE</strong> N=?</td>
<td><strong>WHERE</strong> N=?</td>
<td><strong>and</strong> P&gt;?</td>
</tr>
</tbody>
</table>

What indexes?

$V(M, N, P)$
The Index Selection Problem 3

$V(M, N, P)$

Your workload is this

100,000 queries: 

```
SELECT * 
FROM V 
WHERE N=?
```

1,000,000 queries: 

```
SELECT * 
FROM V 
WHERE N=?
and P>?
```

100,000 queries: 

```
INSERT INTO V 
VALUES (?, ?, ?)
```

A: $V(N, P)$
The Index Selection Problem 2

V(M, N, P)

Your workload is this

1,000 queries:

```
SELECT * 
FROM V 
WHERE N>? and N<?
```

100,000 queries:

```
SELECT * 
FROM V 
WHERE P>? and P<?
```
The Index Selection Problem 2

\[ V(M, N, P) \]

Your workload is this

1,000 queries:

```sql
SELECT * FROM V WHERE N>? and N<?
```

100,000 queries:

```sql
SELECT * FROM V WHERE P>? and P<?
```

A: \( V(N) \) unclustered; \( V(P) \) clustered
Basic Index Selection Guidelines

- Consider queries in workload in order of importance
- Consider relations accessed by query
  - No point indexing other relations
- Look at WHERE clause for possible search key
- Try to choose indexes that speed-up multiple queries
- And then consider the following...
Consider creating a multi-attribute key on K1, K2, if …
WHERE clause has matches on K1, K2, …
But also consider separate indexes
SELECT clause contains only K1, K2, ..
A covering index is one that can be used exclusively to answer a query, e.g. index R(K1,K2) covers the query:

Can be answered with an index-only plan

```
SELECT K2
FROM R
WHERE K1=55
```
To Cluster or Not to Cluster?

- Range queries benefit mostly from clustering
- Covering indexes do *not* need to be clustered

Why?
Percentage of tuples retrieved

Cost

Sequential scan

Unclustered index

Clustered index

SELECT *
FROM R
WHERE K>? AND K<?
Updates

- Indexes speed up queries
  - SELECT FROM WHERE

- But they usually slow down updates:
  - INSERT, DELETE, UPDATE

- However some updates benefit from indexes

```
UPDATE  R
SET     A = 7
WHERE   K = 55
```
hash indexes
Hash Table v.s. B+ tree

Rule 1: always use a B+ tree 😊

Rule 2: use a Hash table on K when:
- There is a very important selection query on equality (WHERE $K=?$), and no range queries
- You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation (you will understand that in a few lectures)
Hash Indexes

- **Hash-based** indexes are best for *equality selections*. **Cannot** support range searches.
- E.g., retrieve a student with id ‘123’ or all students at age=20.

- Static and dynamic hashing techniques exist.

- As for any index, 3 alternatives for data entries $k^*$:
  - $\langle k, \text{data record with key value } k \rangle$
  - $\langle k, \text{rid of data record with search key value } k \rangle$
  - $\langle k, \text{list of rids of data records with search key } k \rangle$
**Static Hashing**

$h(k) \mod N$ = bucket to which data entry with key $k$ belongs. $k_1 \neq k_2$ can lead to the same bucket.

**Static structure:** # buckets (N) fixed

- **Primary pages:** allocated sequentially, never de-allocated;
- **Overflow pages:** allocated/de-allocated if needed.
Static Hashing

- Hash function on the search key distributes values over \([0 \ldots N-1]\).
  - \(h(key) \mod N = (a \times key + b) \mod N\)
  - \(a\) and \(b\) are constants; a lot is known about how to tune \(h\)

- Buckets contain data entries in a chain of pages.
  - Long overflow chains degrade performance.
  - Dynamic techniques fix this problem.
Extendible Hashing

When bucket (primary page) becomes full, why not re-organize file by *doubling* num. of buckets?

Reading and writing all pages is expensive!

**Idea**: use a *directory of buckets*. When bucket is full:

1) *double the directory*,
2) *split just the bucket that overflowed*.

Directory much smaller than file, so doubling is cheap.

Only one page of data entries is split. *No overflow page*!

Trick lies in how hash function is adjusted.
Example

- Directory is array of size $N=4$, **global depth** $D = 2$.

- To find bucket for key:
  1) get $h(key)$,
  2) take last **global depth** # bits of $h(key)$, i.e., mod $2^D$.
  - If $h(key) = 5 = \text{binary 101}$,
  - Take last 2 bits, go to bucket pointed to by 01.

- Each bucket has **local depth** $L$ ($L \leq D$) for splits!
If bucket is full, *split* it:
- Allocate new page,
- Re-distribute,
- If needed, *double* directory.

Double the directory if *global depth D = local depth L*
- Split if D = L.
- Otherwise, don’t.

Insert *k* with \( h(k) = 20 \)?
Insert $h(k) = 20$ (Causes Doubling)
Points to note

20 = binary 10100. Last 3 bits needed to distinguish A, A2.

- **Global depth D of directory**: Max # of bits needed to tell which bucket an entry belongs to.
- **Local depth L of a bucket**: Actual # of bits needed to determine if an entry belongs to this bucket.

Bucket split causes directory doubling if before insertion, $L$ of bucket = $D$ of directory.
Deletes

- Remove a data entry from bucket
  - If bucket is empty, can be merged with `split image`.
  - If each directory entry points to same bucket as its split image, can halve directory.
  - If assume more inserts than deletes, do nothing...
Comments on Extendible Hashing

- **Access cost**: If directory fits in memory, equality search takes one I/O to access the bucket; else two.

- **Skews**: If the distribution of hash values is skewed, directory can grow large. An example?

- **Duplicates**: Entries with same key value need overflow pages!
Linear Hashing

- Dynamic hashing, an alternative to Extendible Hashing.
- Advantage: adjusts to inserts/deletes \textit{without a directory}.

\textbf{Idea}: Use a family of hash functions $h_0, h_1, h_2, \ldots$

\begin{itemize}
  \item $h_i(key) = h(key) \mod (2^i \times N)$; \(N = \text{initial \# buckets}\)
  \item $h$ is some hash function (range is \textit{not} 0 to $N-1$)
  \item $h_0 = h(key) \mod N$
  \item $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
  \item If $N = 2^{d_0}$, for some $d_0$, $h_i$ consists of applying $h$ and looking at the last $d_i = d_0 + i \text{ bits}$.
\end{itemize}
Linear Hashing (Contd.)

LH avoids directory by
1) using *temporary* overflow pages, and
2) choosing bucket to split in a *round-robin* fashion.

Splitting proceeds in `rounds`. Round $L$, $N_L = 2^L N$ buckets:

- *Next* bucket to be split: Buckets $[0, \text{Next}-1]$ have been split; $[\text{Next}, N_L]$ yet to be split.
- Round $L$ ends when all $N_L = 2^L N$ initial buckets have been split.
L<sup>th</sup> Round of splitting

Bucket to be split Next

2<sup>L</sup> N buckets at the beginning of this round. This is the range of h<sub>L</sub>.

`Split image' buckets: created (through splitting buckets) in this round

Buckets split in this round
**Searches**

**Search:** To find bucket for data entry $k^*$, apply $h_L(k)$:
- If $h_L(k)$ in range $[\text{Next}, N_L]$, $k^*$ belongs here.
- Else, apply $h_{L+1}(r)$ to choose between bucket $h_L(k)$ and its split image bucket $h_L(k) + N_L$.

Bucket to be split $\text{Next}$

2$^L$N buckets at the beginning of this round. This is the range of $h_L$.

'Bucket split in this round'

'Split image' buckets: created (through splitting buckets) in this round
**Inserts**

- **Insert**: Find bucket $B$ by applying $h_L / h_{L+1}$. If $B$ is full:
  - Add overflow page, insert data entry $k^*$.
  - *(Maybe)* split bucket $\text{Next}$ (often $B \neq \text{Next}$) using $h_{L+1}$, increment $\text{Next}$.
  - Can choose any criterion to `trigger` split.
- Since buckets are split round-robin, long overflow chains don’t develop!

- Compared to *Extendible Hashing*:
  - Switching of hash functions is *implicit* in how the # of bits examined is increased.
  - No need to *physically double the directory*!
Example of linear hashing

On split, $h_{L+1}$ is used to re-distribute entries.
Example: end of a round

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>66<em>18</em>10<em>34</em></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em>43*</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>00</td>
<td>44<em>36</em></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>01</td>
<td>5<em>37</em>29*</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>10</td>
<td>14<em>30</em>22*</td>
<td></td>
</tr>
</tbody>
</table>

Next = 3

Round $L=0$, $N_L=4$, $h_0$: mod $2^2$ / $h_1$: mod $2^3$

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>66<em>18</em>10<em>34</em></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em>43*</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>00</td>
<td>44<em>36</em></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>01</td>
<td>5<em>37</em>29*</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>10</td>
<td>14<em>30</em>22*</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>11</td>
<td>31<em>7</em></td>
<td></td>
</tr>
</tbody>
</table>

Next = 0

Round $L=1$, $N_L=8$, $h_1$: mod $2^3$

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>66<em>18</em>10<em>34</em></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>43<em>35</em>11*</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>00</td>
<td>44<em>36</em></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>01</td>
<td>5<em>37</em>29*</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>10</td>
<td>14<em>30</em>22*</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>11</td>
<td>31<em>7</em></td>
<td></td>
</tr>
</tbody>
</table>
Summary of Dynamic Hashing

- Linear Hashing (LH) avoids directory by splitting buckets round-robin, and using overflow pages.
- Overflow pages not likely to be long.
- Duplicates handled easily.
- Space utilization can be lower than Extendible Hashing (EH), since splits not concentrated on ‘dense’ data areas.

- *Skewed* data distributions: hash values of data entries are not uniformly distributed! Cause problems with EH and LH.
Summary of Hashing

- **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. *(But duplicates may 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.
Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
- Overflow pages not likely to be long.
- Duplicates handled easily.
- Space utilization could be lower than Extendible Hashing, since splits not concentrated on ‘dense’ data areas.
  - Especially true with skewed data.
  - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash indexes, a skewed data distribution means hash values of data entries are not uniformly distributed! Cause problems with EH and LH.
spatial data
Types of Spatial Data

Point Data
- Points in a multidimensional space
- E.g., satellite imagery, feature vectors

Region Data
- Object have spatial extend with location and boundary
- DB typically uses geometric approximations
Spatial Queries

- **Range queries**
  - E.g., find all cities within 50 miles from UMass
  - Query has associated region
  - Answer includes overlapping or contained regions

- **Nearest neighbor queries**
  - Find the 10 cities closest to UMass
  - Results order by proximity

- **Spatial joins**
  - Find all cities near a lake
  - Expensive. Join condition involves regions and proximity
Applications

- Geographic Information Systems (GIS)
  - Geospatial information
  - All classes of queries are common
- Computer-aided design / manufacturing
  - Spatial objects (e.g., plane fuselage)
  - Range queries and spatial joins
- Multimedia databases
  - High dimensional objects in feature vector form
  - Nearest neighbor queries
Single-dimensional Indexes

- B+ trees

- When we use composite keys, we effectively linearize a higher dimensional space
Multi-dimensional Indexes

- Cluster entries to exploit near-ness in multi-dimensional space
- Keeping track of entries and maintaining a balanced structure is a challenge

Spatial clusters
Example queries

- Find hotels in a 5 mile radius of the conference venue
- Find all cities that lie on the Nile
- Given a face, find the five most similar faces
- Multi-dimensional range queries
  - $50 \lhd \text{age} \lhd 55 \text{ and } 80k \lhd \text{salary} \lhd 90k$
We need an index
- One-dimensional indexes do not support multidimensional search efficiency
- Hash indexes only support point queries
- Graceful inserts and deletes

Why?
The R-Tree

- An adaptation of the B+ tree

- Each key stored in a leaf is intuitively a box, or collection of intervals (one per dimension)
R-tree properties

- Leaf entry = \( \langle n\text{-dimensional box, rid} \rangle \)
  - Alternative 2, with the box as the key value
  - Box is the tightest bounding box for a data object

- Non-leaf entry = \( \langle n\text{-dim. box, ptr to child} \rangle \)
  - Box covers all boxes in subtree
  - All leaves at equal distance from the root
  - Nodes 50% full (except root)
  - Or can pick any parameter \( m \)
Example

leaf
index
spatial object
Example
Search for overlapping box

Start at root

- If node is non-leaf, for each entry \(<E,\text{ptr}>\), if box \(E\) overlaps \(Q\) search subtree at \(\text{ptr}\)
- If current node is leaf for each entry \(<E,\text{ptr}>\), if box \(E\) overlaps \(Q\), rid has an object that may overlap

May have to search several subtrees
Improving search using constraints

- Boxes can be represented compactly
- Convex polygons would be more accurate
  - Less overlap between nodes. May have to fetch fewer nodes
  - Cost of overlap test is higher
Insertions

- Start at root
- Go to best-fit leaf L
  - Go to child whose box needs the least enlargement
  - Resolves ties by picking the smallest area child
- If L has space, insert entry and stop, otherwise split L
- Propagate splits accordingly
Splitting a node

- Entries must be distributed between L1 and L2
- Reduce likelihood that both L1 and L2 will be searched in subsequent queries
- Redistribute to minimize their total area

bad

good
R-tree variants

- **R***
  - Forced inserts to reduce overlap
  - Re-insert some portion of the entries

- **R+**
  - Insert into multiple leaves
  - Single path search, but redundancy
GiST

- Abstracts the “tree” nature of a class of indexes including B+ and R-tree variants
- The similarities between insert/delete/search make it possible to provide templates
- B+ trees are very important in a DBMS, so they are implemented separately
- GiST provides an alternative for implementing other tree indexes