COMP 430
Intro. to Database Systems
Indexing
How does DB find records quickly?

- Various forms of *indexing*
- An index is automatically created for primary key.
- SQL gives us some control, so we should understand the options.
  - Concerned with user-visible effects, not underlying implementation.

```sql
CREATE INDEX index_city_salary
ON Employees (city, salary);

CREATE UNIQUE CLUSTERED INDEX idx
ON MyTable (attr1 DESC, attr2 ASC);
```

Options vary.
Phone book model

Data is stored with search key. *Clustered* index.

Organized by **one search key:**

- Last name, first name.
- Searching by any other key is slow.
Library card catalog model

Index stores pointers to data.

*Non-clustered index.*

Organized by search key(s).

- Author last name, first name.
- Title
- Subject
Evaluating an indexing scheme

• Access type flexibility
  • Specific key value – e.g., “John”, “Smith”
  • Key value range – e.g., salary between $50K and $60K

Advantage:
• Access time

Disadvantages:
• Update time
• Space overhead

Creating an index can slow down the system!
Ordered indices

Sorted, as in previous human-oriented examples
Types of indices we’ll see

• Dense vs. sparse
• Primary vs. secondary
• Unique vs. not-unique
• Single- vs. multi-level

These ideas can be combined in various ways.
Typically unique index also, since primary search key typically same as primary key.

**Sparse**

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>80</td>
<td>30</td>
</tr>
<tr>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>140</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Dense**

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

What were primary search keys in phone book & library card catalog?
Primary index – trade-off dense vs. sparse

- Dense – faster access
- Sparse – less update time, less space overhead

Good trade-off: Sparse, but link to first key of each file block.
Secondary index – a non-clustering index

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>140</td>
</tr>
</tbody>
</table>

Needs to be dense, otherwise we can’t find all search keys efficiently.

What were secondary search keys in phone book & library card catalog?
Secondary index typically not unique

<table>
<thead>
<tr>
<th>Index</th>
<th>Buckets</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

...
Index size

- Dense index – search key + pointer per record
- Sparse index – search key + pointer per file block (typically)

Many records, but want index to fit in memory

Solution: Multi-level index
Multi-level primary index

Sparse Index

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>250</td>
</tr>
</tbody>
</table>

(Semi-)dense Index

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>130</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
</tr>
<tr>
<td>190</td>
</tr>
<tr>
<td>220</td>
</tr>
<tr>
<td>250</td>
</tr>
</tbody>
</table>

Data

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
<tr>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

...
Multi-level secondary index

Sparse Index

| 10 | 50 | 90 | 130 |

Dense Index

| 10  | 20  | 30  | 40  |
| 50  | 60  | 70  | 80  |
| 90  |     |     |     |

Data

| 70 | 40 | 30 | 100 |
| 10 | 90 |    | 60  |
|    | 20 | 80 | 50  |
Multi-level index summary

• Access time now very locality-dependent
• Update time increased – must update multiple indices
• Total space overhead increased
Updating database & indices – inserting

Moving all records is too expensive.
Have same issue when updating indices.
Updating database & indices – deleting

Moving all records is too expensive.
Have same issue when updating indices.
Updating file leads to fragmentation

Performance degrades with time.
Need to periodically reorganize data & indices.

Solution: B+-trees
B+-tree indices
B+-trees

- Balanced search trees optimized for disk-based access
  - Reorganizes a little on every update
  - Shallow & wide (high fan-out)
  - Typically, node size = disk block

- Slight differences from more commonly-known B-trees
  - All data in leaf nodes
  - Leaf nodes sequentially linked

CREATE INDEX … ON … USING BTREE;

Or, is the default in many DBMSs.
B+-tree example

... Data records ...
B+-tree performance

• Very similar to multi-level index structure

  – Slightly higher per-operation access & update time

+ No degradation over time
+ No periodic reorganization

B+-tree widely used in relational DBMSs
What about non-unique search keys?

• Allow duplicates in tree. Maintain $\leq$ order instead of $<$.  
  • Slightly complicates tree operations.
• Make unique by adding record-ID.  
  • Extra storage. But, record-ID useful for other purposes, too.

• List of duplicate records for each key.  
  • Trivial to get all duplicates.  
  • Inefficient when lists get long.
Indexing on VARCHAR keys

- Key size variable, so number of keys that fit into a node also varies.

- Techniques to maximize fan-out:
  - Key values at internal nodes can be prefixes of full key. E.g., “Johnson” and “Jones” can be separated by “Jon”.

- Key values at leaf nodes can be compressed by sharing common prefixes. E.g., “Johnson” and “Jones” can be stored as “Jo” + “hnson”/”nes”
Hash indices
Basic idea of a hash function

$h()$ distributes values uniformly among the buckets.

$h()$ distributes typical subsets of values uniformly among the buckets.
Using hash function for indexing

CREATE INDEX ... ON ... USING HASH;


Use hash function to find bucket. Search/insert/delete from bucket. Bucket items possibly sorted.
Buckets can overflow

- Overflow some buckets – skewed usage
- Overflow many buckets – not enough space reserved

Solutions:
- Chain additional buckets – degrades to linear search
- Stop and reorganize
- Dynamic hashing (extensible or linear hashing) – techniques that allow the number of buckets to grow without reheashing existing data
Advantages & disadvantages of hash indexing

+ One hash function vs. multiple levels of indexing or B+-tree

- Simple hashing degrades poorly when data is skewed relative to the hash function.
- Can’t easily access a data range.

- Storage about the same.
  + Don’t need multiple levels.
  - But, need buckets about half empty for good performance.
Multiple indices & Multiple keys
Using indices for multiple attributes

What if queries like the following are common?

```
SELECT ... 
FROM Employee 
WHERE job_code = 2 AND performance_rating = 5;
```
Strategy 1 – Index one attribute

CREATE INDEX idx_job_code on Employee (job_code);
SELECT ... FROM Employee WHERE job_code = 2 AND performance_rating = 5;

Internal strategy:
1. Use index to find Employees with job_code = 2.
2. Linear search of those to check performance_rating = 5.
Strategy 2 – Index both attributes

```
CREATE INDEX idx_job_code on Employee (job_code);
CREATE INDEX idx_perf_rating on Employee (performance_rating);

SELECT ...
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;
```

Internal strategy – chooses between
• Use job_code index, then linear search on performance_rating.
• Use performance_rating index, then linear search on job_code.
• Use both indices, then intersect resulting sets of pointers.
Strategy 3 – Index attribute set

CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT ...
FROM Employee
WHERE job_code = 2 AND performance_rating = 5;

Attribute sets ordered lexicographically:

(jc1, pr1) < (jc2, pr2) iff either
• jc1 < jc2
• jc1 = jc2 and pr1 < pr2

Note that this prioritizes job_code over performance_rating!

This strategy typically uses ordered index, not hashing.
Strategy 3 – Index attribute set

Efficient

CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT ... FROM Employee WHERE job_code = 2 AND performance_rating = 5;

SELECT ... FROM Employee WHERE job_code = 2 AND performance_rating < 5;

Inefficient

CREATE INDEX idx_job_perf on Employee (job_code, performance_rating);

SELECT ... FROM Employee WHERE job_code < 2 AND performance_rating = 5;

SELECT ... FROM Employee WHERE job_code = 2 OR performance_rating = 5;
Strategy 4 – Grid indexing

- $n$ attributes viewed as being in $n$-dimensional grid
- Numerous implementations
  - Grid file, K-D tree, R-tree, ...
- Mainly used for spatial data
Strategy 5 – Bitmap indices

• Coming next...
Bitmap indices
Basic idea of bitmap indices

• Assume records numbered 0, 1, ..., and easy to find record \(i\).

<table>
<thead>
<tr>
<th>Record #</th>
<th>id</th>
<th>gender</th>
<th>income_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51351</td>
<td>M</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>73864</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>13428</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>53718</td>
<td>M</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>83923</td>
<td>F</td>
<td>3</td>
</tr>
</tbody>
</table>

Create Bitmap Index ...

Bitmaps

M F 1 2 3 4 5
1 0 1 0 0 0 0
0 1 0 1 0 0 0
0 1 1 0 0 0 0
1 0 0 0 0 1 0
0 1 0 0 1 0 0

F’s bitmap: 01101
Queries use standard bitmap operations

```
SELECT ... FROM ... WHERE gender = 'F' AND income_level = 1;
F's bitmap 01101 & 1's bitmap 10100 = 00100
```

```
SELECT ... FROM ... WHERE gender = 'F' OR income_level = 1;
F's bitmap 01101 | 1's bitmap 10100 = 11101
```

```
SELECT ... FROM ... WHERE income_level <> 1;
1's bitmap ~ 10100 = 01011
```
Space overhead

• Normal: \#records \times (#values + 1) bits, if nullable
  • Typically used when \# of values for attribute is low.

• Encoded: \#records \times \log(#values)
  • But need to use more bitmaps

• Compressed: \sim 50\% of normal
  • Can do bitmap operations on compressed form.
A couple details

• When deleting records, either ...  
  • Delete bit from every bitmap for that table, or  
  • Have an *existence bitmap* indicating whether that row exists.

• Can use idea in B+-tree leaves.
## Summary of index types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-level</td>
<td>Index generally too large to fit in memory</td>
</tr>
<tr>
<td>Multi-level</td>
<td>Degrades due to fragmentation</td>
</tr>
<tr>
<td>B+-tree</td>
<td>General-purpose choice</td>
</tr>
<tr>
<td>Spatial</td>
<td>Good for spatial coordinates</td>
</tr>
<tr>
<td>Hash</td>
<td>Good for equality tests; potentially degrades due to skew &amp; overflow</td>
</tr>
<tr>
<td>Bitmap</td>
<td>Good for multiple attributes each with few values</td>
</tr>
</tbody>
</table>