Indexes

Database Systems: The Complete Book
Ch. 13.1-13.3, 14.1-14.2
Hash-Based Indexes
What’s a Hash Function?
Hash Functions

• A hash function is a function that maps a large data value to a small fixed-size value

• Typically is deterministic & pseudorandom

• Used in Checksums, Hash Tables, Partitioning, Bloom Filters, Caching, Cryptography, Password Storage, …

• Examples: MD5, SHA1, SHA2

• MD5() part of OpenSSL (on most OSX / Linux / Unix)

• Can map \( h(k) \) to range \([0,N)\) with \( h(k) \% N \) (modulus)
Hash-based Indexes

- As with trees: request a key $k$ and get record(s) or record id(s) with $k$.
- Hash-based indexes support equality lookups
  - … in constant time (vs $\log(n)$ for tree)
  - … but don’t support range lookups
- Static vs Dynamic Hashing
  - Tradeoffs similar to ISAM vs B+Tree
Static Hashing

Primary Bucket Pages (Contiguous)  Overflow Pages (Linked List)

h(k) % N

k

0

1

2

N-1

...
Static Hashing

• Buckets contain data entries.
• Hash fn maps the search key field of records to one of a finite number of buckets (% N)
• N chosen when the index is created
  • Too small N: Long overflow chains
  • Too big N: Wasted space/Poor IO

What’s to stop us from “just resizing the hashmap?”
Dynamic Solutions: Extendible and Linear Hashing
Extendible Hashing

- **Situation:** A bucket becomes full
  - **Solution:** Double the number of buckets!
  - **Expensive!** (N reads, 2N writes)
- **Idea:** Add one level of indirection
  - A directory of pointers to (noncontiguous) bucket pages.
  - Doubling just the directory is much cheaper.
  - Can we double only the directory?
Extendible Hashing

The directory and data pages have an associated “depth” (global/local).

To look up a value use the last $gd$ bits of the key’s hash value as an index into the dir.
Extendible Hashing

Insert 20 (h(20) = 1100)
(Need to Split Bucket A)

Dir entries not being split point to the same bucket
Extendible Hashing

\( gd = 3 \)

Insert 31 (h(31) = 1001)

Don’t need to double dir when splitting bucket w/ ld < gd

(Need to Split Bucket B)
Extendible Hashing

- Global depth of directory
- **Upper bound** on # of bits required to determine the bucket of an entry.
- Local depth of a bucket
  - **Exact** # of bits required to determine if an entry belongs in this bucket.
- Why use least significant bits (vs MSB)?
Extendible Hashing

• If the entire directory fits in memory, any equality search can be answered in one disk access. (otherwise two)
• Is this true even if the directory spans multiple pages?
• 100 MB file, 100 B/rec = 1m records over 4k pages.
  • Minimum of 25k directory entries.
• Hash table still likely to be < 1M
Extendible Hashing

- Hashing Issues:
  - Need a uniform distribution of hash values.
    - Even a true random function will not provide this
  - What could happen if multiple keys have the same hash value? (A hash ‘collision’)

- Deletions
  - Deleting the last entry in a bucket allows it to be merged with its ‘split image’.
  - Can potentially halve directory if this happens.
Breaking Up Conditions

Boolean formulas can create complex conditions

\[(\text{Officer.Ship} = '1701A' \text{ AND}\text{Officer.Rank} > 2) \text{ OR} \text{Officer.Rank} > 3\]
Breaking Up Conditions

Boolean formulas can create complex conditions

\[(\text{Officer.Ship} = '1701A' \land \text{Officer.Rank} > 2) \lor \text{Officer.Rank} > 3\]

First convert all conditions to **Conjunctive Normal Form** (CNF)

\[(\text{Officer.Ship} = '1701A' \lor \text{Officer.Rank} > 3) \land (\text{Officer.Rank} > 2 \lor \text{Officer.Rank} > 3)\]
Breaking Up Conditions

Boolean formulas can create complex conditions

\[(\text{Officer.Ship} = '1701A' \text{ AND } \text{Officer.Rank} > 2) \text{ OR } \text{Officer.Rank} > 3\]

First convert all conditions to Conjunctive Normal Form (CNF)

\[(\text{Officer.Ship} = '1701A' \text{ OR } \text{Officer.Rank} > 3) \text{ AND } \text{Officer.Rank} > 2\]

Simplification may be possible
Indexing

- Indexes are typically built over one (key) field \( k \)
- Index stores mappings from key \( k \) to:
  - \( k \rightarrow \) The full tuple with key value \( k \)
  - \( k \rightarrow \) Record ID for Tuple with key value \( k \)
  - \( k \rightarrow \) List of Record/RecordIDs with key value \( k \)
- The choice of data to store is orthogonal to the choice of how to map key to value.
Multi-Attribute Indexes

We can create an ordering on \(<A,B>\):
\(<A_1, B_1>\) is less than \(<A_2, B_2>\)
whenever
• \(A_1\) is less than \(A_2\)
• \(A_1 = A_2\) and \(B_1\) is less than \(B_2\)

Can we use this sort order to find all \(<A,B>\) where…

All \(A < 3\)?
All \(A = 3\) and \(B = 2\)?
All \(A = 3\) and \(B < 2\)?
All \(A < 3\) and \(B = 2\)?
Access Paths and Join Algorithms

*Database Systems: The Complete Book*
Ch. 15.4-15.6
Example

```
SELECT COUNT(*)
FROM Students S,
     CourseRegs R
WHERE S.Name = 'Alice'
    AND S.Id = R.StudentId
    AND R.Grade > 90
    AND R.Grade < 100
```

What is the Equivalent Relational Algebra Expression?
Example

```
SELECT COUNT(*)
FROM Students S,
     CourseRegs R
WHERE S.Name = 'Alice'
  AND S.Id = R.StudentId
  AND R.Grade > 90
  AND R.Grade < 100
```

How Do We Optimize This Expression?
Example

What Indexes Might be Helpful?

When?
Indexes

Clustered Index

Unclustered Index (Secondary Index)
Indexes

How the Data is Organized

- ISAM
- B+Tree
- Other Tree-Based
- Hash Table
- Other Hash-Based
- Other...

How the Data is Laid Out

- In the Index
  - Clustered
- Outside of the Index
  - Sorted
  - Heap
Multiple Indexes

Can we have multiple indexes over one table?

How does this affect our design considerations?
Access Paths

How do I read from the data

Originally

\[ \sigma_c \]

| \[ R \] |

“File Scan + Select”

Now

How do we pick?

IndexScan\((R, c, \text{Index#})\)

New Index Scan Operator
Joins

• Two General Classes of Joins
  • Equality (Equi-) Joins: \( R.B = S.B \)
  • Inequality (Inequi-) Joins: \( R.B < S.B \)

• How do the outputs of these joins differ?
  Inequi-joins are \( O(N^2) \) (as bad as NLJ)
  We will focus on Equi-joins
Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}
Implementing: Joins

Solution 2 (Block-Nested-Loop)
Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks

2) NLJ on each pair of blocks
Implementing: Joins

Solution 3 (Sort-Merge Join)

Keep iterating on the set with the lowest value. When you hit two that match, emit, then iterate both

A
1
2
3
5

B
1
4
5
6

Done!
Implementing: Joins

Solution 4 (External Hash)

1) Build a hash table on both relations

2) In-Memory Nested-Loop Join on each hash bucket (subdivide buckets using a different hash fn if needed)
Implementing: Joins

Solution 5 (Grace/Hybrid Hash)

Keep the hash table in memory

(A) 1 2 3 5

(B) 4 5 6

(Essentially a more efficient nested loop join)
Implementing: Joins

Solution 6 (Index-Nested-Loop)

Like nested-loop, but use an index to make the inner loop much faster!
What are the tradeoffs of each algorithm?

What properties do we care about?  How do the algorithms compare?
### Implementing: Joins

#### Tradeoffs

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Pipelined?</th>
<th>Memory Requirements?</th>
<th>Predicate Limitation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Loop</td>
<td>1/2</td>
<td>1 Table</td>
<td>No</td>
</tr>
<tr>
<td>Block-Nested Loop</td>
<td>No</td>
<td>2 ‘Blocks’</td>
<td>No</td>
</tr>
<tr>
<td>Index-Nested Loop</td>
<td>1/2</td>
<td>1 Tuple (+Index)</td>
<td>Single Comparison</td>
</tr>
<tr>
<td>Sort-Merge</td>
<td>If Data Sorted</td>
<td>Same as reqs. of Sorting Inputs</td>
<td>Equality Only</td>
</tr>
<tr>
<td>Hash</td>
<td>No</td>
<td>Max of 1 Page per Bucket and All Pages in Any Bucket</td>
<td>Equality Only</td>
</tr>
<tr>
<td>Grace Hash</td>
<td>1/2</td>
<td>Hash Table</td>
<td>Equality Only</td>
</tr>
</tbody>
</table>
Extra Content - External Sort