Index tuning:
Special-purpose indexes 2

Rasmus Pagh
Today’s lecture

Part I: Strings
• B⁺-trees and strings
  – String B-trees.
• Inverted indexes.
• Full-text indexing
  – Suffix arrays and suffix trees

Part II: Spatial data
• Multi-dimensional indexes (for points)
  – Range queries
  – Near-neighbor queries
Strings in a B-tree

- Allows searching for strings with a given *prefix*.
- Does not help finding, say, a string containing a given word.
- Even when a B-tree is appropriate, string keys can be problematic:
  - Reduce fan-out, increase depth of tree
  - Prefix compression (i.e., omit part of key shared with the previous key) may help, but not always.
- Motivates looking at special indexes.
String B-tree [Ferragina-Grossi, 95]

- Supports prefix searches.
- Search time is *independent* of the length of the strings stored.
- Special way of guiding the search
  - Verification is needed to check if the search string matches the string found.
  - If no perfect match, finding the nearest match may require additional I/Os.
- Not (yet) in widespread use
  - Main application is in “full-text indexing”
Inverted indexes

- **Motivating task**: In a collection of “documents”, find those containing the words “database” and “tuning”.
- Consider the relation $C(\text{docID}, \text{word})$ that contains $(i, w)$ if and only if document $i$ contains the word $w$.
- The *inverted index* for the document collection is $C$ sorted by $(w, i)$.
- Searching for several words is done by self-joining the inverted index.
**Inverted index limitations**

- Do not allow searches for documents containing a particular substring.
  - Exception: Prefix search.

- Do not allow phrase searches such as “database tuning”
  - Except through a potentially expensive join that considers all occurrences of “database” and “tuning”.

- Updating an inverted index is not I/O efficient unless we can keep a substantial part of C in RAM (exercise).
### Suffix array

[Manber and Myers, 1990]:
Array of pointers to all the suffixes in the text in their lexicographic order.

### Example text:

1  2  3  4  5  6  7  8  9  10 11 12
abra cadabra $

<table>
<thead>
<tr>
<th>pos</th>
<th>sorted suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>$</td>
</tr>
<tr>
<td>11</td>
<td>a$</td>
</tr>
<tr>
<td>8</td>
<td>abra$</td>
</tr>
<tr>
<td>1</td>
<td>abracadabra$</td>
</tr>
<tr>
<td>4</td>
<td>acadabra$</td>
</tr>
<tr>
<td>6</td>
<td>adabra$</td>
</tr>
<tr>
<td>9</td>
<td>bra$</td>
</tr>
<tr>
<td>2</td>
<td>bracadabra$</td>
</tr>
<tr>
<td>5</td>
<td>cadabra$</td>
</tr>
<tr>
<td>7</td>
<td>dabra$</td>
</tr>
<tr>
<td>10</td>
<td>ra$</td>
</tr>
<tr>
<td>3</td>
<td>racadabra$</td>
</tr>
</tbody>
</table>
Suffix arrays and updates

- Suppose there is a (small) change in the text. The suffix array must be updated.

- **Problem**: Most of the suffixes are likely to have changed.

- **Possible solution**:
  - Bound then length of comparison by $k$.
  - Suffixes that are equal in the first $k$ characters may occur in any order (at most $k$ changes needed after update).
  - Use a B-tree to store suffix order.
Full text search II

- Suffix tree: trie of all suffixes of the text  
  [McCreight, 1976]

- Each edge may correspond to several characters.

- More space demanding, but also more powerful than suffix arrays.
Text indexing in DB2

Short story: CREATE INDEX FOR TEXT...

Longer story:
• Some setup is required.
• Index is not automatically built/refreshed. UPDATE INDEX FOR TEXT
• To use, must use special string query language (not just a LIKE operator).

SELECT AUTHOR,TITLE FROM R
WHERE CONTAINS(A,"primary key")=1
Text indexing summary

• Must distinguish between query types:
  – Prefix search (B-tree, String B-tree)
  – Document word search (inverted index)
  – Full-text search (suffix array, suffix tree)

• Indexes for the latter types of searches are generally more expensive to maintain.
  – Major DBMSs require user to update the index manually (except “per-tuple” indexes).
  – I found internals of text index structures to be generally poorly documented.
Part 2: Spatial indexing

• General multi-dimensional indexes:
  – Grid files
  – Generalizations of B-trees

• Range queries
  – Range trees

• Near-neighbor search
  – Space-filling curves
Spatial databases

Examples:

- Geographic Information Systems (GIS)
- Computer-Aided Design (CAD)
- Multimedia databases (*feature vectors*)
- Traffic monitoring

More generally, spatial/multidimensional indexing techniques may be relevant to all queries that contain a range or point condition on more than one attribute.
Spatial data

Two main types:

- Point data (GIS, feature vectors, OLAP)
- Region data: Objects have some spatial extent, e.g. polygons.

- We will focus on point data, but some of the techniques we will talk about also work for region data.
- We will talk mostly about 2D, but all ideas extend (with some cost) to higher dimensions.
Spatial queries

Examples:

• (Orthogonal) range queries:
  – Select all points with coordinates in given ranges.

• Nearest neighbor queries:
  – Find the nearest point to a given query point.

• Spatial join:
  – Join with spatial condition, e.g. “are closer than 1 km”. Not discussed today.
Motivating example

• An orthogonal 2D range query:

```
SELECT x, y FROM map
WHERE (x BETWEEN 1000 AND 2000)
AND (y BETWEEN 2000 AND 3000)
```

• Assume there are \( n \) rows in \( \text{map} \), and that we have covering B-tree indexes on \((x, y)\) and \((y, x)\).

• What strategies may the DBMS use to perform the query?
Grid file, in a picture
Grid file properties

• Simple implementation
  – Suppose points have the form (id,x,y)
  – Grid cell size G
  – Make index on (x div G, y div G, id)

• Weak point: The number of points in a cell may vary a lot when points are not uniformly distributed.
  – Sometimes need 1 I/O to retrieve few points.
  – Sometimes need many I/Os to retrieve the points in a single cell.
Grid files in DB2

- Works on columns with a spatial data type.
- Specify up to three different grid sizes
  - Each data item will be inserted in the "most suitable" grid file.
  - Searches need to go to all three grids.
- Syntax example:
  ```sql
  CREATE INDEX idx ON R(a)
  EXTEND USING db2gse.spatial_index (1.0, 10.0, 100.0)
  ```
2D external search trees

Several ways to generalize 1D search, based on how the space is split

- **kd-tree**: Alternates between vertical and horizontal split, | — | — | —

- **quad-tree**: Horizontal and vertical split, ++++++

- **R-tree**: Split into any collection of rectangles, ❋ ❋ ❋ ...
kd-tree in a picture
R-tree in a picture
Spatial search tree properties

• In contrast to B-trees, a search may need to inspect a large number of points that do not match the query.
  – This is particularly true in higher dimensions, and when points are not evenly distributed.
  – Some search structures offer a worst-case guarantee on the cost of a 2D range query of around $\sqrt{n}$, for $n$ points.
Range trees

• *Range trees* provide fast multi-dimensional range queries at the cost of higher space usage.
  – Performance acceptable only in low dimensions.

• Next, we will see a simple way of implementing the same idea using a collection of standard B-trees!
  – Focus on 2D: We have a set of points (id, x, y), where x and y are integers.
Ranges via prefixes

• Covering ranges by prefixes:
  – Suppose \( a \) and \( b \) are \( w \)-bit integers.
  – Any range \([a;b]\) can be split into at most \(2^w\) intervals where each interval consists of all integers with a particular prefix.

• **Thus**: Enough to solve the case where a prefix is specified in one dimension.

• **Redundancy**: For *each* prefix length \( i \), create an index for \((\text{prefix}_i(x),y,id)\).
  – Supports lookup of a prefix of \(x\)-coordinate, and a range of \(y\)-coordinates.
Range tree summary

• 2D search broken into many 1D searches
  – at most twice the number of bits in coords.
• Each 1D search can be handled efficiently (using B-trees)
  – Query essentially only reads points that are in the range.
  – Can quickly get a count of the number of points.
• Has better worst case performance than the other index types we saw.
Space-filling curves

**Idea:** Create 1-to-1 correspondence between points in 2D and 1D that "preserves locality".

Image licensed under the Creative Commons Attribution-Share Alike 1.0 Generic license
Z-ordering

- Simplest space-filling curve
- Consider point given by binary coordinates: 
  \((00101110, 01101011)\)
- Mapped to the number formed by interleaving: 
  \(0001110011101101\).
- Mapping a 2D range query: Determine the smallest interval containing range.
  - Z-order: Top-left and bottom right corners determine the extremes.
Weak points of space-filling curves

• *Some* points that are close in 2D will be far apart when mapping to 1D.

• Chance of running into this problem can be minimized by adding a random shift to all coordinates.
  – Alternatively, consider a number of space-filling curves slightly shifted along both coordinates.
Approximate nearest neighbor

- Exact near neighbor queries are difficult, especially
  - when data changes, and
  - there may be many points at almost minimal distance to the query point.

- Often: Enough to find a neighbor that is not much further away than the nearest neighbor.
  - Allows much more efficient solutions.
  - The ratio between distances can be guaranteed.
Approximate NN picture
Spatial indexing summary

• Many different indexes, with different strengths and weaknesses.

• Distinguishing features include:
  – Linear space, or larger overhead?
  – Good for any point distribution?
  – Support for queries: Range q., near neighbor q., stabbing q., intersection q.,...?
  – Exact or approximate results?
  – Fast updates, or meant for static use?

• Most common in practice: R-trees, kd/quad-trees, (space-filling curves).