Index tuning: Special-purpose indexes

Rasmus Pagh
Today

• Fresh motivation, from The Economist.

• Index types for special situations.
  – High update rates: Buffered indexes.
  – Small cardinality attributes: Bitmap indexes.
  – Low update rates: Materialized views.
Aside: Proactive vs reactive

Two kinds of tuning:

• **Proactive**: Analyze, and try to anticipate need for indexes, partitioning, and other tuning choices.

• **Reactive**: Use experiments and measurements to identify performance bottlenecks in a running system, and try to remove them.

The aim of this course is to equip you to do both kinds of tuning.
Buffering in B-trees

• Relatively new technique to speed up updates in search trees.
  – Explicit in [Brodal and Fagerberg 2003]
  – Survey by [Graefe 2007]
  – Latest result in [Brodal et al., 2010]

• We will consider a simple approach for tables that are not too much larger than “fast memory” (RAM).
Simple approach

• Suppose R is frequently updated and, say, 2-20x larger than available RAM.

• Create a “buffer” table $R_U$ that records recent updates.
  – Table and index stays in RAM.
  – A view is made available that gives access to the table with the updates performed.

• When the size of $R_U$ becomes close to the amount of RAM available, transfer all the updates to R (on disk).
  – Each I/O makes many updates.
Using a standard B-tree

• Suppose that we only care about fast *insertions* in R, and don’t want to introduce new relations.
• Give R a new boolean attribute \( b \) that indicates if a tuple is in the buffer. The buffered tuples should fit in RAM.
• New tuples inserted with \( b=1 \) (trigger).
• Introduce \( b \) as the first attribute of the index (tuples in buffer stored together).
• Periodically retrieve all tuples with \( b=1 \) and insert them with \( b=0 \) (in order).
The general picture

- The same approach works with small use of internal memory if we accept to increase the depth of the B-tree.
  - Trade-off between search time and insertion time.
- Also, deletions and updates can be buffered.
- However, it seems that no major DBMS supports memory efficient indexes with fast updates / slower searches (yet).
Buffering hash tables?

- It turns out that it is essentially only possible to buffer hash tables using the memory-demanding “simple” approach [Wei, Yi, Zhang 2009].
- However, the buffering idea can be used to decrease the cost of overflows to virtually zero.
  - 1 I/O for almost all accesses
Buffering in hash tables

• Same trick as in buffered B-trees: Don’t do updates right away, but put them in a buffer.

• Advantage: Several keys moved to the overflow block at once.
• Disadvantage: Buffer takes space.
• Details in [JensenPagh07].
Aside: Two-choice hashing

• **Idea:**
  - Use two hash functions, $h_1$ and $h_2$.
  - $x$ is stored in either block $h_1(x)$ or $h_2(x)$, use two I/Os for lookup.
  - When inserting $x$, choose the *least loaded* block among $h_1(x)$ and $h_2(x)$.

• Can be shown: Overflow probabilities are much smaller than with one function, especially when “B is small”.

• If two storage units are available, the 2 I/Os can be done in parallel.

• Also works with 3 of 5, etc. (Robustness)
Data analysis query

- We will use this query as an example in the following:

```
SELECT SUM(S.sales)
FROM Sales S, Times T, Locations L
WHERE S.tid=T.id AND S.lid=L.id AND T.year=2010 AND L.ctry='DK'
GROUP BY T.year, L.state
```
Indexing low cardinality attributes

• Suppose there are only 4 different locations in our previous example.
• Then we may represent the locations of N tuples using only 2N bits.
• However, a (secondary) index on location seems to require at least N pointers.
• Can we get by reading less data?
Basic bitmap index

• For each possible value of \textit{lid} and each tuple, store a bit that is 1 iff the tuple contains the given value.
• Store these bits \textbf{ordered by column} (with no RID).
• Use to efficiently find the tuples with particular value(s) of \textit{lid}.
• Combine with information on tuples with matching \textit{tid} value (RID intersect).
Gain of bitmap indexes

- How much can **at most** be gained by using bitmaps, compared to using a space-efficient B-tree index?
  - Theoretically 1 bit/tuple vs \( \log N \) bits/tuple.
  - Typically 1 bit/tuple vs 32-64 bits/tuple.

- Consider **star joins** similar to our previous example.

- Main case where there is no gain:
  - A single dimension is very selective.
  - (Usually only the case for high cardinality attributes.)
Compressed bitmap indexes

- If there are many possible values for an attribute (it has “high cardinality”), basic bitmap indexing is not space efficient (nor time efficient).
- **Observation:** A column will have few 1s, on average. It should be possible to “compress” long sequences of 0s.
- **How to compress?** Usual compression algorithms consume too much computation time. Need simpler approach.
Word-aligned hybrid (WAH) coding

• In a nutshell: [WOS04]
  – Split the bitmap B into pieces of 31 bits.
  – A 32-bit word in the encoding contains one of the following, depending on the value of its first bit:
    • A number specifying the length of an interval of bits where all bits of B are zeros.
    • A piece of B (31 bits).
  – The conjunction (“AND”) or disjunction (“OR”) of two compressed bitmaps can be computed by a simple scan.
WAH analysis

• Let N be the number of rows of the indexed relation, and c the cardinality of the indexed attribute.

• At most N WAH words will encode a piece of the bitmap.

• Reasonable assumption:
  – All (or most) gaps between consecutive 1s can be encoded using 31 bits.
  – Thus, at most N+c gaps.

• Total space usage: 2N+c words.

• Compares favorably to B-trees.
Creating bitmap indexes

• No support in DB2 (but bitmaps are used internally).

• Oracle:
  – CREATE BITMAP INDEX ON R(A)
  – Internal representation is another compressed bitmap format (BBC).
  – Documentation recommends use mainly for low-cardinality attributes, and systems with low concurrency (crude locking mechanism?).
**Bitmap join indexing**

- Similar to defining a join!
  - A join index is an index on a join result.

- **Example**: (for Oracle)
  A bitmap join index that allows us to find the sales in a given state:
  ```sql
  CREATE BITMAP INDEX ON sales(locations.ctry)
  FROM sales, locations
  WHERE sales.lid=locations.id
  ```

- Can even index multiple attributes in a multi-way join.
Low update rates

• An SQL view is similar to a macro. E.g.
  
  ```sql
  CREATE VIEW MyView AS
  SELECT *
  FROM Sales S, Times T, Locations L
  WHERE S.tid=T.id AND S.lid=L.id
  ```

• A query on `MyView` is transformed into a query that performs the join of Sales, Times, and Locations.

• In contrast, a **materialized view** *physically* stores the query result.
  - Additionally: can be indexed!
  - DB2 term: materialized query table.
“Refreshing” a materialized view

• Any change to the underlying tables may give rise to a change in the materialized view. There are at least three options:
  – Update for every change ("REFRESH IMMEDIATE")
  – Update only on request ("REFRESH DEFERRED")
  – Update when the view is accessed ("lazy")
Using a materialized view (DB2)

1. Materialized view is created:
   ```sql
   CREATE TABLE SalaryByLocation AS
   (SELECT location_id,country_id,SUM(salary) AS s
    FROM Employees NATURAL JOIN Departments
    NATURAL JOIN Locations
    GROUP BY location_id, country_id)
   DATA INITIALLY DEFERRED REFRESH DEFERRED
   ```

2. Can be used like any table:
   ```sql
   SELECT country_id, SUM(salary) AS salary
   FROM SalaryByLocation
   GROUP BY country_id;
   ```
   ```sql
   CREATE INDEX idx ON SalaryByLocation(country_id)
   ```

3. Manual refreshing:
   ```sql
   REFRESH TABLE SalaryByLocation
   ```
Automatically using mat. views

• Suppose a user does not know about the materialized view and writes directly

```sql
SELECT location_id, country_id, SUM(salary) AS s
FROM Employees NATURAL JOIN Departments
    NATURAL JOIN Locations
GROUP BY country_id
```

• A smart DBMS will realize that this can be rewritten to a query on the materialized view.

• Rewrite capability is a key technique in relational OLAP systems.
Conclusion

• We have seen three techniques:
  – Buffering to speed up updates (and how to get this effect in a normal B-tree).
  – Bitmap indexing, “compressed pointers” (especially good for intersecting many index results).
  – Materialized views (tables that store precomputed results enable fast answers to queries, requires low update environment).

• Next week:
  – High-dimensional indexing. Text indexing.
  – Guest lecture on partitioning and DLCM.