Index tuning: Basics

Rasmus Pagh
Some simple SQL queries

1. SELECT *
   FROM Movie
   WHERE studioName = 'Disney'
       AND year = 1990;

2. SELECT *
   FROM (SELECT *
           FROM Movie
           WHERE studioName = 'Disney') M
   WHERE year = 1990;
Memory access cost

• Basic facts about hardware:
  – The speed of a CPU instruction is 2-3 orders of magnitude higher than the latency of a RAM access.
  – The latency of RAM access is 5-6 orders of magnitude lower than the latency of hard drive access.
  – New SSD drives improve latency by roughly 1 order of magnitude – still a large gap.

• RAM vs disk analogy: Go to Australia to borrow a cup of sugar if your neighbor is not home!
Cache-efficiency

• To amortize the latency, modern storage transfers *blocks* of memory (aka. page [I/O], cache-line [cache miss]).

• This means that sequential access is much faster than random access.

• Recently accessed blocks are kept in a cache (aka. buffer).

• Back to the sugar example: Consider loading the suitcases with sugar (and other things from Australia) instead of just bringing home one cup.
What is this?
Access paths

- For many database queries and updates, only a small fraction of the data needs to be accessed.
- Extreme examples are looking at or updating the single tuple with a given key value.
- General question: "How do we access the relevant data, and not (much) more?"
- **Term.**: Need efficient access paths.
Heap file

- Standard organization of a relation in most DBMSs.
- Tuples are stored in **any order** (typically insertion order).
- Block-conscious: When accessing a tuple we load a (nearly) full block of tuples into the cache.
- Mechanism for finding a particular tuple: Read the whole relation (**full table scan**).
Sorted relations

• Suppose a relation with N tuples is sorted by attribute A.
• It can be searched for an A-value using binary search (cost < log₂N < 40 I/Os).
• This can be a huge improvement!
• Also works if we are interested in a range of A-values.
• Problems:
  – If we want to search sometimes for A, sometimes for B, how should we sort?
  – How to maintain sorted order?
Introducing redundancy

• If several attributes are relevant for search, one possibility is to have several copies of the relation, sorted in different ways.
  – This is an example of a covering index.

• Phone books have info sorted by:
  – Name of business
  – Business of business
  – Other?
Redundancy, cont.

• Redundancy may be too expensive in terms of space and update cost:
  – Space can be reduced by using pointers.
• The cost of maintaining different copies of data puts a hard limit on redundancy
  – Must keep up with database update rate.
• A structure with pointers to the tuples of a relation is called a secondary index.
Primary indexes

• If the tuples of a relation are stored sorted according to some attribute, an index on this attribute is called **primary**.
  – Primary indexes make point and range queries on the key *very efficient*.

• Many DBMSs automatically build a primary index on the primary key of each relation.

• A primary index is roughly equivalent to a **clustering** or **sparse** index.
Secondary indexes

• Secondary index creation examples:
  CREATE INDEX myIndex ON myRel(A);
  CREATE INDEX i2 ON myRel(B,C,A);

• The index contains the mentioned attributes (key of the index), sorted lexicographically.

• Secondary indexes make:
  – most point queries on the key (or a prefix of the key) more efficient.
  – some range queries on the first component of the key more efficient.
Adding a directory

• A sorted relation may be searched more quickly if we have a directory:
  - A sorted list with a representative key from each block (e.g., smallest key).
  - A pointer to the block of each key.

• But how do we search the directory?
  - Not so important if it fits in RAM.
  - Otherwise, it seems that this problem is the same as the original problem...
  - This observation leads to B-trees (many variants exist, B⁺-trees most common).
Choosing to use an index

• The choice of whether to use an index is made by the DBMS for every instance of a query
  – May depend on query parameters (different selectivity for different values)
  – Don’t have to take indexes into account when writing queries

• Estimating selectivity is done using statistics
  – More on estimation in two weeks.
Next: Deriving the “best index”

- Focuses on select on a single table using a single index.
  - Possibly the select is followed by a join.
- Assumes indexing is done with B-trees.
- Identifies two candidates, A and B
  - Further investigation may be needed to choose among them.
The three stars

(B-tree) indexes may be evaluated on what “stars” they have, relative to a particular query.

- Star 1: The columns on which there is an equality comparison with a constant are the first columns of the index.
- Star 2: Results of every query is stored in the order given by ORDER BY (if any).
- Star 3: Includes all columns mentioned in the SELECT (i.e., is covering).
Choose columns of the index like this:

- First, the columns with an equality condition, e.g., \( A=42 \). (Star 1)
- Then, the most selective column with a range condition \((B \text{ BETWEEN } 1 \text{ AND } 10)\).
- Then, remaining columns of the query. Let any remaining columns of an ORDER BY come first, in proper order.
  - Specify DESC for decreasing order attributes.

Has Star 1 and 3, but may not have 2.
Candidate B

Choose columns of the index like this:

- First, the columns with an equality condition, e.g., $A=42$.
- Then, remaining columns of the query. Let any remaining columns of an ORDER BY come first, in proper order.
  - Specify DESC for decreasing order attributes.

Has all stars, but does not make use of range conditions.
Problem session

• Consider the following queries:
  1. SELECT id,fname FROM R
     WHERE lname=:v1 AND city=:v2
     ORDER BY fname
  2. SELECT id,fname FROM R
     WHERE lname BETWEEN :v1 AND :v2
        AND city=:v2
     ORDER BY fname

• What are the Candidate A and B indexes? What stars do they qualify for?
More simple SQL queries

3. SELECT *
   FROM Movie, MovieExec
   WHERE Movie.title = 'Star Wars' AND
   Movie.prod = MovieExec.cert;

4. SELECT name
   FROM Movie
   WHERE studioName LIKE 'D%' AND
   year>1980 AND
   year<1990;
Other impact of indexes

The DBMS may use indexes in other situations than a simple point or range query.

• Some joins can be executed using a modest number of index lookups
  – May be faster than looking at all data

• Some queries may be executed by only looking at the information in the index
  – Index only query execution plan
  – May need to read much less data.
Horizontal partitioning, hashing

- Split the relation into several partitions according to some attribute value A.
  - Partitions are not necessarily sorted.
- Range partitioning: Each piece corresponds to a range of A-values.
- Hash partitioning: Split according to a hash of A-values (destroy order)
  - If we split into many partitions, so that each part is around 1 block, we get a hash index.
  - Some DBs have this as a native index type.
Index types

Common:
• B-trees (point queries, range queries)
• Hash tables (only point queries, but somewhat faster)

More exotic:
• Bitmap indexes
• Full text indexes (substring searches)
• Spatial indexes (proximity search, 2D range search,...)
• ... and thousands more
Next two lectures

• In the lectures on the 4\textsuperscript{th} and 9\textsuperscript{th} we will look at several index types for special purposes.
  – High update rates.
  – Low update rates (precomputation).
  – Small cardinality attributes.
  – Text searching.
  – Spatial and high-dimensional indexing.

• Assignment on indexing coming soon!