

# Indexing

KBL 9.1-9.4, 9.5 until p. 354 (incl), 9.6.1

Rasmus Pagh



# Database efficiency

*One of the great dividends of investing in an RDBMS is that you don't have to think too much about the computer's inner life. You're the programmer and say what kinds of data you want. The computer's job is to fetch it and you don't really care how.*

Philip Greenspun in "SQL for Web Nerds"

Next: Why you should care about how the DBMS uses *indexes*.



# Disk crash course

- Relations of large databases are usually stored on hard drives (or SSDs).
- Hard drives can store large amounts of data, but work rather slowly compared to the memory of a modern computer:
  - The time to access a specific piece of data is on the order of  $10^6$  ( $10^4$ ) times slower.
  - The **rate** at which data can be read is on the order of 100 (10) times slower.
- Time for accessing disk may be the main performance bottleneck!



# Multiple disks

- Many database systems use several disks to
  - Enable several pieces of data to be fetched in parallel.
  - Increase the total rate of data from disk.
- Systems of several disks are often arranged in so-called RAID systems, with various levels of error resilience.
- Even in systems with many disks, the time used for accessing them is usually the performance bottleneck.



# Full table scans

When a DBMS sees a query of the form

```
SELECT *  
FROM R  
WHERE <condition>
```

the obvious thing to do is read through the tuples of R and report those tuples that satisfy the condition.

This is called a **full table scan**.



# Selective queries

Consider the query from before

- If we have to report 80% of the tuples in R, it makes sense to do a full table scan.
- On the other hand, if the query is very **selective**, and returns just a small percentage of the tuples, we might hope to do better.



# Point queries

- Consider a selection query with a single equality in the condition:

```
SELECT *  
FROM actorInfo  
WHERE year=1975
```

- This is a point query: We look for a single value of "year".
- Point queries are easy if data is sorted by the right attribute.



# Range queries

- Consider a selection query of the form:  

```
SELECT *  
FROM actorInfo  
WHERE year>1975 and year<1994
```
- This is a **range query**: We look for a range of values of "year".
- Range queries are **also** easy if data is sorted by the right attribute.
  - But often not be as selective as point queries.



# Indexes

- To speed up queries the DBMS may build an **index** on the year attribute.
- A database index is similar to an index in the back of a book:
  - For every piece of data you might be interested in (e.g., the attribute value 1975), the index says where to find it.
  - The index itself is organized such that one can quickly do the lookup.
- Looking for information in a relation with the help of an index is called an **index scan**.



# 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.
  - In MySQL this depends on the storage engine: InnoDB builds an index on the primary key, MyISAM does not.
- A primary index is sometimes referred to as a **clustering** or **sparse** index.



# Secondary indexes

- It is possible to create further indexes on a relation. Typical syntax:

```
CREATE INDEX myIndex ON involved(actorId);
```

- The non-primary indexes are called **secondary** indexes (sometimes **non-clustering** or **dense** indexes)
  - Secondary indexes make *most* point queries on the key more efficient.
  - Secondary indexes make *some* range queries on the key more efficient.



# Multi-attribute indexes

Defining an index on several attributes:

```
CREATE INDEX myIndex  
ON actorInfo (lastname,firstname,year);
```

Speeds up point queries such as:

```
SELECT *  
FROM actorInfo  
WHERE firstname='Tom' and lastname='Cruise'
```

An index on several attributes usually gives index for any **prefix** of these attributes, due to lexicographic sorting.



# Problem session

- What kinds of point and range queries are "easy" when the relation is stored as in the previous example:
  1. A range query on firstname?
  2. A range query on lastname?
  3. A point query on lastname?
  4. A point query on lastname combined with a range query on firstname?
  5. A point query on firstname combined with a range query on lastname?



# Index scan vs full table scan

Point and range queries on the attribute(s) of the **primary index** are almost always best performed using an index scan.

**Secondary indexes** should be used with **high selectivity queries**:

As a rule of thumb, a secondary index scan is faster than a full table scan for queries returning less than 10% of a relation.



# 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
  - Don't have to take indexes into account when writing queries
- Estimating selectivity is done using statistics
  - In MySQL, statistics is gathered by executing statements such as `ANALYZE TABLE` involved



# What speaks against indexing?

- Space usage:
  - Small for primary index
  - Similar to data size for secondary index
- Time usage for keeping indexes updated under data insertion/change:
  - Small to medium for primary index
  - High for secondary index (but has been going down...)



# 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 ("covering index").
  - May need to read much less data.



# Index types

Common:

- B-trees (point queries, range queries)
- Hash tables (only point queries, but somewhat faster)
- Bitmap indexes

More exotic:

- Full text indexes (substring searches)
- Spatial indexes (proximity search, 2D range search,...)
- ... and thousands more



# Conclusion

- Large databases need to be equipped with suitable indexes.
  - Need understanding of what indexes might help a given set of queries.
  - Important distinction: Primary vs secondary.
  - A detailed understanding of various index types is beyond the scope of this course.



## Related course goal

Students should be able to:

- decide if a given index is likely to improve performance for a given query.

