

— Today's lecture —

- Database efficiency
- Indexing
- Schema tuning

— Database efficiency —

This lecture is on **database efficiency**:

The property that the database uses a (mostly) small amount of computational and storage resources.

Here, “small” is relative to the best use of resources we could hope for:

- We don't want the database to use 100 times more storage or computation time than the best that the computer could be programmed to do.

— RDBMS efficiency is largely automatic —

One of the reasons for the success of RDBMSs is that, to a large extent, they are efficient “automatically”.

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”

This lecture is about some of the things that you, as a database programmer, might have to do to help the RDBMS improve efficiency:

- Define suitable indexes.
- “Tune” the database schema.

— Hard drives —

Relations of large databases are usually stored on hard drives.

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 times slower.
- The rate at which data can be read is on the order of 10 – 100 times slower.
- Data are read/written in large chunks of approximately 50Kbytes

For databases that do not fit in the computer's main memory, the time used for accessing disks is usually the *main performance bottleneck*.

— Using several hard drives —

Many database systems use several hard drives 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, which support various levels of performance improvement and error resilience.

Even in systems with many hard drives, the time used for accessing disks is usually the main performance bottleneck.

Next: Indexing

— Full table scans —

When a DBMS encounters 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 selection query from before:

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

- If we have to report 80% of the tuples in R, it makes sense to do a full table scan (All tuples will be probably be read anyway).
- 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:
 - Is there a way of “skipping over” tuples that will not be selected?

— Non-computer search for information —

The following are well-known examples that there are ways of “going directly to the desired information”:

- A phone book.
- A dictionary.
- A cookbook.

Database systems use similar principles to find information quickly.

Sometimes, the equivalent of a full table scan is needed, e.g. when looking for “recipe combining eggs and mustard at 220 degrees in the oven”.

Point queries

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

```
SELECT *  
FROM R  
WHERE grade = 11
```

This is a so-called **point query**: We report all grades at the “point” 11.

Point queries are often very selective.

Suppose the tuples of R were sorted by grade. Then the DBMS could:

1. Search for the first tuple with grade 11.
2. Report this and all the following tuples with grade 11.

— Range queries —

Consider this kind of selection query with two inequalities in the condition:

```
SELECT *  
FROM R  
WHERE grade > 6 AND grade < 10
```

This is an example of a **range query**, since we report all grades in the range 7 to 9.

Suppose that the tuples of R were sorted according to grade. Then the DBMS could:

1. Search for the first tuple with grade > 6 .
2. Report this and all the following tuples with grade < 10 .

Indexes

To be able to quickly find the first tuple with a specific grade, the DBMS may build an **index** on the grade attribute.

A database index is similar to an index in the back of a book:

1. For every piece of data you might be interested in (e.g., the attribute value 9), the index says where to find it.
2. 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 Oracle the programmer must explicitly specify that the table should be index organized.)
- A primary index is sometimes referred to as a **clustering** or **sparse** index.

— Secondary indexes —

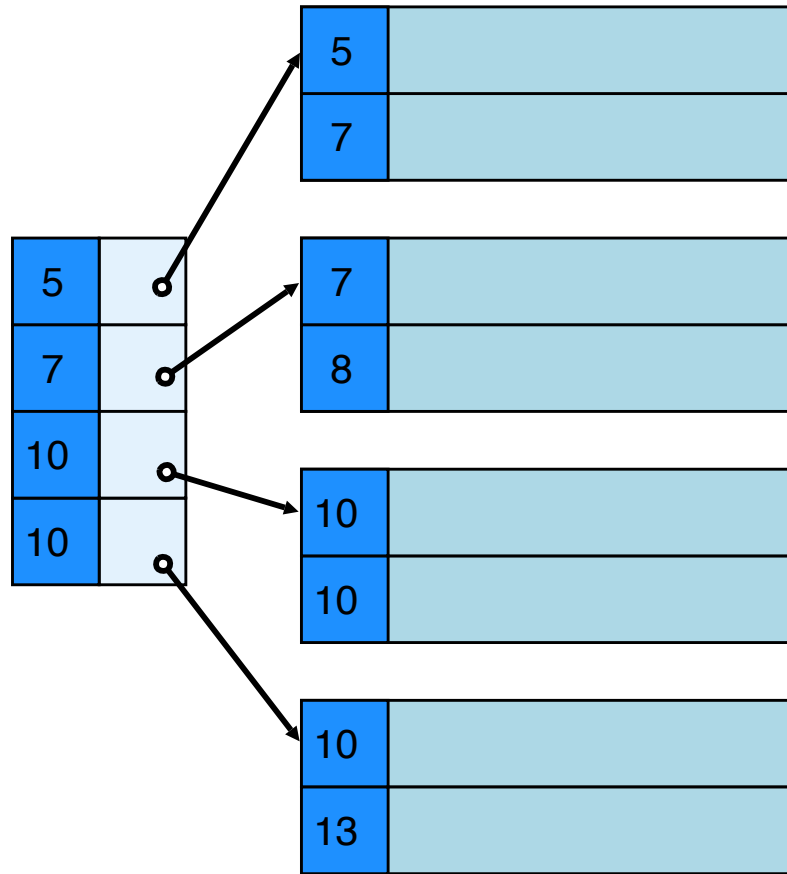
It is possible to have more than one index on a relation. While not part of the SQL standard, additional indexes can usually be created by writing statements such as:

```
CREATE INDEX gradeIndex ON projects (grade);
```

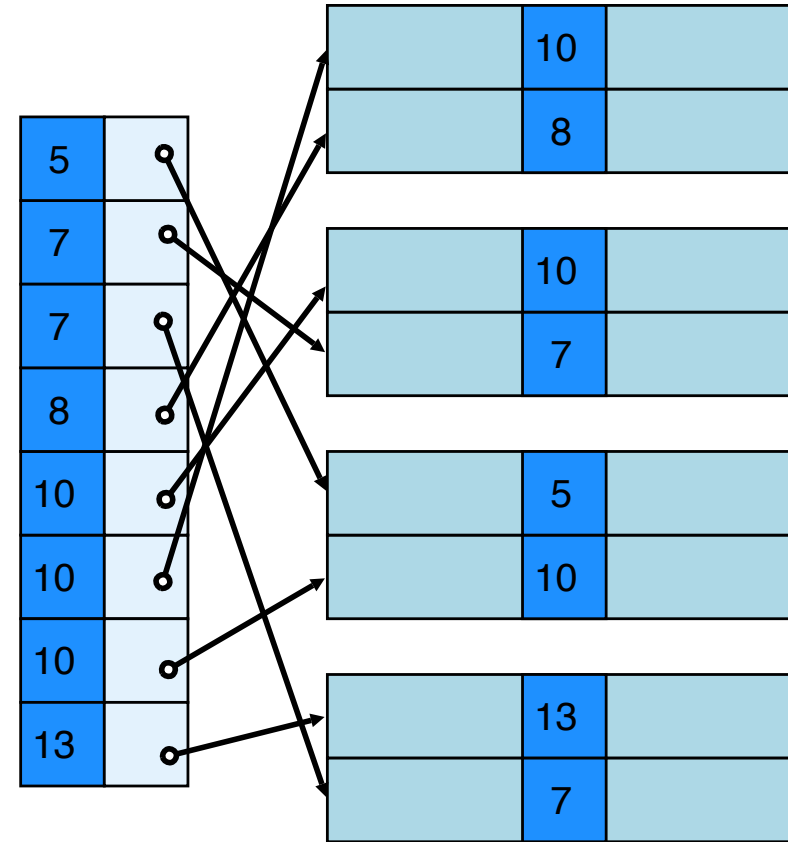
The non-primary indexes are called **secondary indexes**.

- Secondary indexes make *most* point queries on the key more efficient.
- Secondary indexes make *some* range queries on the key more efficient.
- A secondary index is sometimes referred to as **non-clustering** or **dense** index.

Primary and Secondary indexes



A Primary Index



A Secondary Index

— Index scan versus full 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 10-20% of a relation.

— Choosing whether to use the index —

Even if an index scan is possible, a good DBMS sometimes chooses to do a full table scan.

The decision is usually based on *statistics* on the data in the relation, which allows the selectivity of the query to be estimated.

Often computation of the statistics is controlled manually. In Oracle, statistics can be computed as follows:

```
ANALYZE TABLE <relation> COMPUTE STATISTICS;
```

— Indexes on several attributes —

An index can be defined on one or more attributes, e.g.

```
CREATE INDEX myIndex ON projects (grade,start,exam);
```

Such an index speeds up point queries such as:

```
SELECT * FROM projects  
WHERE grade = 13 AND start='2003-11-24' AND exam='2004-01-28';
```

Due to the way indexes are (usually) implemented (as a **B-tree**), an index on several attributes automatically gives index for any **prefix** of these attributes.

Example:

myIndex also gives an index for (grade,start) and for (grade).

— Indexes on several attributes, intuition —

An index on the attributes (*grade*, *start*, *exam*) has a *similar effect* as storing the tuples sorted first according to *grade*, secondly according to *start*, and thirdly sorted according to *exam*.

<i>grade</i>	<i>start</i>	<i>exam</i>
...
11	2003-11-24	2004-01-28
11	2004-11-27	2005-01-07
13	2003-11-24	2004-01-28
13	2003-11-24	2004-01-28
13	2003-11-24	2005-01-07
13	2004-11-27	2005-01-07
13	2004-11-27	2005-01-07

— Problem session (5 minutes) —

What kinds of point and range queries are “easy” when the relation is stored as in the previous example:

- A range query on the first attribute?
- A range query on the second attribute?
- A point query on the second attribute?
- A point query on the first attribute combined with a range query on the second attribute?
- A point query on the second attribute combined with a range query on the first attribute?

— Other uses of indexes —

Indexes are used by the DBMS to speed up other operations than point and range queries.

- **Join operations** are often considerably faster when the join attributes are indexed.
- Indexes are used by DBMSs to efficiently check **referential integrity constraints** (such indexes are usually automatically created).

— Time usage of indexes —

Based on what you have seen until now, a natural question would be:

Why not make indexes for all possible sets of attributes?

The main reason is that indexes need to be updated when the relation changes, and index updates take time.

Thus we have the following trade-off for every index:

- It speeds up certain database queries, but
- slows down every update to the relation.

Whether an index is a good idea is thus a matter of weighting the time saved on queries against the additional time spent on updates.

— Space usage of indexes —

In addition to taking time to update, an index has a space cost.

- **Primary indexes** usually have modest space requirements, i.e., considerably less than the space for the relation itself.
- **Secondary indexes** use space similar to that required for the attributes indexed.

For space reasons, one should therefore be careful with creating secondary indexes on large amounts of data.

— Other types of indexes —

Indexing is a science of its own:

- There are special index types such as *bitmap indexes* and *hash indexes* that are more efficient than B-trees for some types of data.
- There are *join indexes* that speed up join operations (but are expensive to update).
- There are indexes for geometric and multidimensional data.
- There are indexes for textual data.
- ...

MDM has a few short descriptions, but a deeper understanding of indexes is beyond the scope of this course.

— Join “Advanced database technology” —

If you have taken an introductory course in algorithms (at ITU or elsewhere), or will take the course in the fall semester, there is a possibility to learn about the inner workings of indexes, and other things touched upon by MDM 6. (Whether this course will exist after the new study reform is not settled yet)

Advanced database technology Taught by Srinivasa Rao and Rasmus Pagh

The course contains explanation of disk based algorithms, indexes, query optimization, and many other things to make you a database expert!

Understand performance rules of thumb and know when they apply!

Next: Schema tuning

— Use the “smallest” possible data type —

DBMSs often offer many different data types. Among other things, this is to allow programmers to choose a data type with the smallest possible space usage.

Space may have an impact on time, e.g., if tuples are twice as long as needed, a full table scan takes twice as long as needed.

In Oracle, space can often be saved by replacing fixed length strings such as `CHAR(20)` with variable length strings such as `VARCHAR(20)`.

— Use of a code table —

For non-standard data types with few possible values (e.g. “types of wood”), space can be saved by introducing a relation with short integer codes for each possible value (encoded in a less space efficient data type).

The drawback of this approach is that a join operation is necessary to combine the information in the two relations.

— Normalization and efficiency —

The good: Normalization can be used to eliminate redundancy in a database design, and thus avoid all sorts of problems. Also, less redundancy means that e.g. full table scans are faster.

The bad: Normalization implies that more join operations must be performed when answering database queries. Performance may not be adequate.

Sometimes a database designer may choose to **denormalize** a database schema, i.e., join several relation schemas into one.

- This may make some queries run faster.
- However, updates must then be handled more carefully, and the space usage may rise.

Example of Denormalization

Normalized 1

nameid	name	email
1	Jakob Hansen	jh@itu.dk
2	Peter Jensen	pj@itu.dk
3	Sofie Madsen	soma@itu.dk

Normalized 2

nameid	course	grade
1	ADS	10
1	ADBT	11
1	IADS	10
2	ADS	9
2	DBT	7
2	IADS	6
2	OOP	03
3	ADS	13

Denormalization of 1 and 2

course	name	email	grade
ADBT	Jakob Hansen	jh@itu.dk	11
ADBT	Peter Jensen	pj@itu.dk	7
DBS	Jakob Hansen	jh@itu.dk	10
DBS	Peter Jensen	pj@itu.dk	9
DBS	Sofie Madsen	soma@itu.dk	13
IADS	Jakob Hansen	jh@itu.dk	10
IADS	Peter Jensen	pj@itu.dk	6
OOP	Peter Jensen	pj@itu.dk	03

— Denormalization —

Typical denormalization scenarios:

- **1-1 relationship.** Joining the relations of the two entity types will result in no (or little) redundancy, and will usually still be normalized.
- **1-M relationship.** Joining in this case will not result in a much larger relation if an entity instance on the “one” side is typically related to few instances on the “many” side, or if the total size of the attributes on the “one” side is small.

Some dangers of denormalization:

- Destroys the advantages of a normalized design. Requires more programming and is more error-prone.
- May speed up some queries, but slow others down.

Recent research results on join processing suggest that in future database systems the effect of denormalization will diminish.

— Denormalization vs materialized views —

An alternative to denormalizing the database schema is to create a materialized view that maintains the join of some of the relations.

The good:

- Accessing the materialized view is the same as accessing the corresponding relation in a denormalized design.
- Updates can be done in the normalized schema (no anomalies).

The bad:

- Space usage is even larger than in denormalized design.
- The DBMS may not handle updates as efficiently as updates in a hand-coded denormalized design.

— Vertical and horizontal partitioning —

Related to, but independent of, denormalization, the DB programmer can instruct the DBMS to use a particular physical layout of a relation.

- **Vertical partitioning.** If some (frequent) queries deal only with some attributes of a relation, it can speed up processing to physically split the relation into two parts – one with those attributes, and one with the rest.
- **Horizontal partitioning.** If some (frequent) queries deal with tuples having a particular value on some attribute, it can speed up processing to physically split the relation into parts depending on the value of that attribute. This is a “light” alternative to having a primary index.

The cost of horizontal partitioning is that it takes more time to retrieve an entire tuple. The cost of vertical partitioning is that insertions of new tuples take more time.

Partitioning Example

Vertical Partitioning

Important information

nameid	course	grade
⋮	⋮	⋮

Less important information

examinator	ensor	examdate	room
⋮	⋮	⋮	⋮

Horizontal Partitioning

Students enrolled after 2003

name	course	grade	examinator	ensor	examdate	room
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Students enrolled before 2003

name	course	grade	examinator	ensor	examdate	room
⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Problem session 5 min —

For each of the commands suppose that they are executed very often on the table exam. Will denormalization, horizontal partitioning or vertical partitioning help to speed up the execution of the command?

1. `SELECT * FROM exam WHERE year > 2004`
2. `SELECT name, course, grade FROM exam`
3. `SELECT * FROM exam NATURAL JOIN s NATURAL JOIN t`
4. `INSERT (...) INTO exam`
5. `DELETE FROM exam WHERE grade < 6`

The table “exam”

name	course	grade	examinator	ensor	year	room
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Bonus Problem: Can primary or secondary indexes be used to achieve a speedup?

— Most important points in this lecture —

As a minimum, you should after this week:

- Remember that disk accesses are the performance bottleneck of most large databases.
- Be able to estimate the value of a particular index based on:
 - The selectivity of typical queries.
 - The frequency of queries and updates.
- Understand the basic techniques of schema tuning.

In this lecture some details were given that you will not find in the course book or supplementary material: You are expected to know this to the detail given by the slides.