Designing for Performance

7 Physical design

69. The purpose of physical design. Whenever we build a software system we have two concerns: *correctness* and *efficiency*. Database systems are no exception. So far we have concentrated on the first aspect but the course would not be complete if we did not also discuss the second.
Although the format of our tables has been determined in the previous two design phases there are a number of ways in which we can enhance efficiency. This is what "physical design" is about. In some books this phase is called, appropriately enough, "database tuning".

70. Some implementation details. Implementing a DBMS is a truly major task and we can not attempt to cover this topic even superficially. What we try to attempt in the following, therefore, is only to give you enough insight so that you can understand which problems we address during the physical design phase and why certain tuning techniques actually work.
This is what you should know about the low level details of a DBMS:

- Database records are stored in special files on permanent storage media (such as hard disks). The files are special in that they are typically not read in a linear fashion as in many other applications but the DBMS loads individual **pages** from the file as and when they are needed. Database files are also special in that they are typically *very large*; so large in fact that they do not normally fit into the main memory as a whole.
- The size of a "page" depends on the underlying architecture but could be anything from 2KB to 32KB. The hard disk controller is built in such a way that it can retrieve and store whole pages very efficiently. Although this process is highly tuned it is very slow when compared to the speed of processing inside the processor or even the main memory. To fetch a page from disk takes about 10 milliseconds. Compare this with the time it takes for the processor to deal with the data on one page, which is somewhere around 10 microseconds — a thousand times less. Furthermore, disk access does not become faster very quickly whereas processor speed at this moment in time is still following *Moore's Law*, and doubling about every 18 months.
- Inside main memory the DBMS maintains a **buffer pool** which is a collection of buffers the size of one page. Once a page has been fetched and processed, it is not immediately discarded but kept until the DBMS needs the space for some other purpose. This will obviously improve performance but you can not assume that the number of buffers is large enough to hold your whole database. Instead, the typical situation is that you must share the buffer pool with other users of the system. (The default for PostgreSQL is 64 buffers of size 8KB each. Our installation "`db-teach`" is configured to maintain 1024 pages.)

We draw the following conclusions from this:

a. The number of pages that is needed to store a table should be minimised. This can be achieved by either having records which are small (so that many of them fit onto one page) or by having fewer records overall.

b. An operation which requires the DBMS to look at every record of a large table is expensive.

For more information on how records are stored in a DBMS we recommend chapter 7 of the book by Ramakrishnan and Gehrke.[1]

71. The evaluation of queries. It is also necessary to know a little bit about the evaluation of queries. We start with the selection operation σ_C which picks certain rows and rejects others. In general, this requires the system to look at every row to see whether the selection condition C applies or not. However, if the select condition is very simple, then the system *may know* on which page(s) the relevant records are located. We will discuss the idea of an **index** for this purpose below.
The main operation in DBMSs is the natural join $T_1 \bowtie T_2$. Conceptually, it requires the system to compare every row of T_1 with every row of T_2 and to check whether they agree on the common attribute(s). Since this very

[1] Ramakrishnan & Gehrke: *Database Management Systems.* McGraw-Hill, 2nd edition, 2000.

quickly leads to astronomical numbers of operations, a lot of research has gone into finding ways to speed up this process. The goal is to compare only those rows where there is likely to be a match and to avoid even attempting as many non-matches as possible.
However, optimisation can not take place unless the resulting table is itself reasonably small, for example of size comparable to either T_1 or T_2. We draw the following conclusions:

c. Joining tables together is expensive.

d. Where tables need to be joined it is important that the result table is of a size not exceeding that of T_1 or T_2 by a large factor.

e. The system may need guidance to be able to find matching rows quickly and to avoid non-matching ones.

72. Optimisation technique 1: "Pre-calculating" to avoid queries. Consider a database for a library and suppose that you frequently need to know how many books have been borrowed by some user already (because there is an upper limit on how many books a user may take out). This information can be calculated from the current list of borrowings:

```
SELECT COUNT(*)
FROM   current_borrowings
WHERE  uid = '0123456';
```

However, since we are likely to run this query every time a book is taken out, it makes sense to run it once for *all users* and then to just update it as we go along:

```
CREATE TABLE number_of_borrowed_items(uid      CHAR(7) PRIMARY KEY,
                                      no_items INT     CHECK(no_items >= 0));

INSERT INTO number_of_borrowed_items  SELECT   uid, COUNT(*)
                                      FROM     current_borrowings
                                      GROUP BY uid;
```

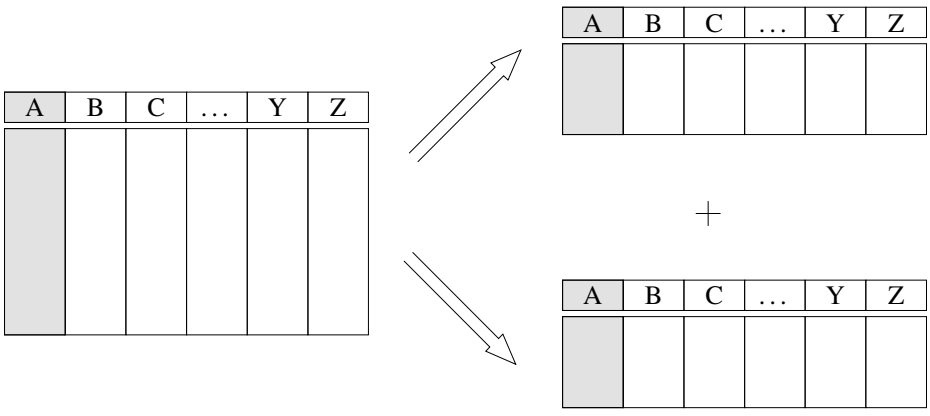
Every time a book is taken out we run the two queries

```
SELECT no_items
FROM   number_of_borrowed_items
WHERE  uid = '0123456';

UPDATE number_of_borrowed_items SET   no_items = no_items + 1
                                WHERE uid = '0123456';
```

which are likely to be faster than the aggregation query we had at the beginning because it touches only a single record of a smaller table.
Note that it does not makes sense to keep values which are *easy* to calculate, for example, because they are derived from the information in a single record. This is because calculation is very cheap when compared to page I/O as we discussed above.

73. Optimisation technique 2: "Splitting" to keep tables small. This can be achieved in two ways. **Vertical splitting**[2] refers to separating out subsets of rows into different tables. Here is a pictorial description:



[2]The use of "vertical" and "horizontal" is not standardised and some authors call horizontal what we might call vertical etc.

The split should be built on a semantic principle:

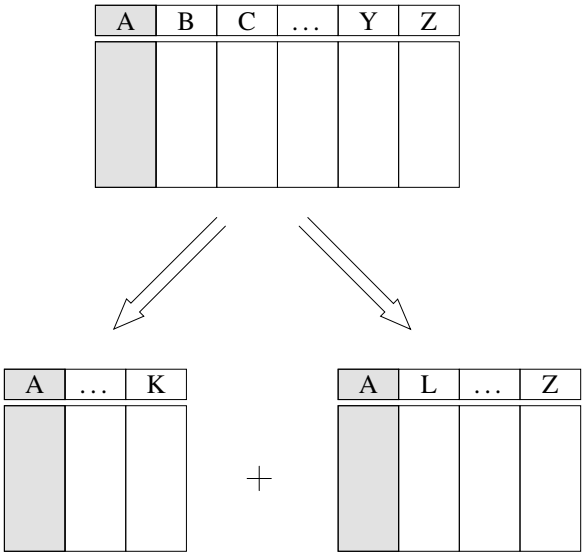
Time Keep last year's sales separate from those that started in the current year.

Status Move all completed sales to an archive table.

Location Have separate tables for customers in different countries.

In all cases it is important that the split agrees with common queries in the sense that they typically only require one to search *one* of the tables. If, as a result of vertical splitting, we have to run queries on several tables in order to collect all data, then the split was counterproductive.

In **horizontal splitting** we attempt to make a table narrower by separating out some attributes into a different table. For example, the University student table has over 50 attributes. Clearly, very few of these are in frequent use, and information about a student's UCAS code and entry qualifications, for example, will rarely be needed once a student has arrived. By duplicating the key attribute, the table can thus be split into one which holds all the current information and one which holds data of more historical nature. As a picture:



As with vertical splitting, here too it is important that a use-case analysis has taken place which supports the assumption that there will be relatively few queries which require the joining together of the two tables.

74. Optimisation technique 3: "Indexes" to avoid linear search. You will remember from your "Algorithms" lectures that searching for a particular entry in a sorted array is much much faster than search in an unsorted file. In the first case we can use binary search which takes $\log(n)$ many steps whereas in the second case the complexity is linear. If the array contains 10 million entries then binary search takes no more than 24 steps whereas linear search takes about 5 million steps on average.

However, a list of database records can only be kept in order according to one attribute, not several ones. This is the same as with telephone books: they are sorted alphabetically according to the last name of customers but not according to telephone number. If you use 1471 on your phone to find out who phoned you last, you get a phone number, not a name, and the telephone book is of no help for finding out the name of the caller.

The way to overcome this restriction is to create an **index** into the data which allows fast access according to an alternative attribute (or a combination of attributes). To give another example, the index at the back of a mail-order catalogue is sorted alphabetically, whereas the detailed descriptions of merchandise typically appear grouped according to similarity in the catalogue itself. The index does not repeat the full information; it just has a pointer to the data in the form of the page number. Database indexes work in exactly the same way; they are contained in separate files and only contain the attribute values according to which we want fast access and pointers to the actual database records.

There are two types of indexes, tree-based and hash-based. The first variety is an optimised form of the binary search idea. The most common data structure used is that of the **B+ tree**. It is not important for us at this point to understand how exactly it works but you should remember that it not only makes searching for a particular entry easy but also searching for a *range* of entries, as in

```
SELECT firstname, lastname
FROM staff
WHERE office >= 100 AND office < 200;
```

The syntax for creating an index is not standardised. In PostgreSQL we would say:

```
CREATE INDEX some_name ON staff USING BTREE (office);
```

You will not need the name of the index unless you want to get rid of it again:

```
DROP INDEX some_name;
```

Please note that by default PostgreSQL builds and maintains a B+ tree index on the primary key of each table. This explains the return message you get when you create a table:

```
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
         index 'test_pkey' for table 'test'
```

The other type of index you can build is based on **hashing**. A hash function computes a numeric value from some arbitrary input data (usually a string). That number describes the place where more information about the corresponding record can be found. Hashing is very fast because the functions used are not very sophisticated, and therefore a hash index is more efficient than a B+ tree. However, hash functions attempt to spread the input data uniformly across so-called "buckets" and therefore all information about similarity is lost (i.e., similar inputs do not lead to similar hash values). Therefore a hash index is only helpful for finding individual records ("equality queries"), not for range queries.

A typical application for a hash index is a table which has alternative keys. Since only one can be declared to be the primary key, we should tell the system to build indexes for the other ones. An example for this is the alternative key attribute "bc" ("banner code") in the "courses" table.

As is the case with indexes in real life, their creation and maintenance causes some effort: every time a new record is entered or an existing record is deleted, the indexes on the table concerned need to be adjusted. For a table which is likely to see a lot of "traffic" it is therefore questionable whether additional indexes (over and above the one triggered by the primary key declaration) are really useful. It may be a difficult decision to take and some experimentation may be necessary. This is not problematic as an index can always be created or dropped after the system has been set up.

75. Optimisation technique 4: "De-normalisation" to keep information together and to avoid joins. As we said above, the natural join operation is the most costly one in a database system, and as you know from the SQL queries you have written, it is also a common operation (which is triggered whenever more than one table is mentioned in the "FROM" part of a query). Apart from trying out alternative ways to express queries, so as to avoid join operations, the only other way to increase efficiency of query evaluation is to **de-normalise** tables.

De-normalisation joins together tables which previous design steps have told us to keep separate in order to avoid redundancy and anomalies. So it is clear that we are "playing with fire" here, and we must build safeguards against the anomalies arising from unnormalised tables into our database application program.

Let's look at the example from the last section, the tables that arose from breaking up "lecturing", and which we called

course_instances = (cid, year, numbers)
taught_by = (cid, year, sid).

The first is a weak entity dependent on "courses", which has additional information about the course such as "name", "level", "semester", and "bc". So, to get the full information about the course being taught, we would often need to do a natural join of "courses" and "course_instances".

To avoid this join, we might consider putting the "courses" information straight into the course_instances table, particularly if we are not intending to keep too many years represented in the database. We would thereby introduce some mistakes, such as pretending that certain courses were actually taught in the current year when in fact they weren't. But at least we know that we would not have to repeat course information *within a year*. So, we would avoid the aggregation anomaly with the original (bad) design. The information on whether a course was actually taught in a particular year could still be gleaned from the taught_by table (or it can be indicated by adding a new Boolean attribute). Also, we are unlikely to run any aggregate queries over the additional attributes "name", "level", "semester", and "bc". If this optimisation is adopted, then we will save one natural join every time we seek to link (the name of) a lecturer with (the name of) a course.

It is interesting to note that by normalisation and subsequent de-normalisation we have **not** returned to the original (and faulty) table design. While the new design also has some redundancy contained in it, this fact is now clearly understood (and hopefully documented) and can be addressed explicitly in the application program which forms the interface between the database and the user. The original redundancy was *not known* before the search for functional dependencies uncovered it. Although this is perhaps a matter of taste, the numerical redundancy in the original tables appears to be rather uglier than the repetition of general course information in each yearly description of courses taught.

76. Summary. Let's quickly come back to the goals listed at the beginning of this section and check which optimisation technique can be used to support them:

a. Minimising the number of pages needed to hold a table: This is addressed by the two splitting techniques.

b. Avoiding linear scans of tables: This is addressed by additional indexes which allow the DBMS to fetch certain records very quickly. Keeping additional tables with pre-calculated values also addresses this goal.

c. Avoiding natural joins: This is addressed by de-normalisation. Another way, not discussed above, is to run separate queries on the two tables and join the results together in the application program. For example, if we are seeking information about a single person that is distributed among several tables, then it is not necessary to run a join query but the data can be fetched from each table individually using the person identifier as the search key.

d. Avoiding giant intermediary tables in queries: Rewrite queries appropriately and make sure that the joining condition involves a key attribute of at least one of the tables involved. (See also item 81 in Section 11, and the discussion of losslessness there.)

e. Speeding up unavoidable joins: This is addressed by creating additional indexes for the attributes involved in the join operation.

The other point to remember is that there is no optimisation technique which does not incur costs in some other way:

- Keeping additional table with pre-calculated values slows down insertions, deletions, and updates, and in addition takes up storage space.
- Splitting may require the duplication of queries in order to collect all information.
- Additional indexes need to be maintained after insertion and deletion. They also take up space in memory.
- De-normalised tables will contain redundancies which must be controlled by the interface program.

Finally, please keep in mind that some of the optimisations we have discussed here are *in conflict with each other*, for example, splitting horizontally makes tables narrower whereas de-normalisation makes them wider.