CS252: Why Performance Is Irrelevant

Hugh Darwen

CS252 students doing database design exercises sometimes imagine that they must address issues of "performance". I explain here why this is not required, even though performance considerations are likely to be extremely important when you design databases for use out there in the industry.

One good reason is that in CS252 we do not teach you anything related to performance issues. Therefore it would be unreasonable of us to assess you on your understanding of such issues. It would also be unreasonable to reward a student who brings to the module a previously acquired good understanding of these issues and tries to put that understanding into practice in the exercises we set.

Now, you might be asking *why* we do not teach anything related to performance issues. So I'll try to answer that question. (But if you aren't asking it and you accept the reason given in my second paragraph above, then you can stop reading now and skip the rest of this document.)

The short answer is that to do justice to such issues in 14 lectures would leave precious little time for teaching the theory plus a couple of contrasting languages (**Tutorial D** and SQL) that illustrate that theory and try to put it into practice. But I'd like to say a little more than that on the subject of performance.

A relational database design is an example of what is commonly referred to as a *logical schema*—logical because it is derived from the stated requirements on purely logical grounds. And because relational theory is so firmly and immediately rooted in logic, the term is particularly appropriate in connection with *relational* databases. A logical design maps the various kinds of statements about the enterprise, that we wish to represent in the database, to *relation variables*. And it includes logical expressions, known as *constraints,* that govern the combinations of values that can validly be assigned to those variables at any time.

We have seen that various different logical schemas can be devised to meet exactly the same requirements and we have encountered some reasons for preferring one schema over another. But the reasons we have encountered in CS252 have nothing to do with performance!

Take Henry VIII's wives, for example. At the end of lecture HACD.9 I concluded that the 6NF decomposition was not a good idea. The main reason was to do with all those constraints that need to be declared in the 6NF design in order to make sure the database is always correctly "glued" together. This reason is a *psychological* one: the more constraints we have to write down, the more likely we are to make mistakes, for example. The 5NF design, in which all the relvars concerning Henry's wives are joined together, very conveniently *implies* all of those constraints.

Another reason for spurning 6NF is that the DBMS being used might not support any kind of "multiple assignment" operator that is needed if updates to such a database are to be supported at all. That reason is a *pragmatic* one, to do with whatever particular DBMS we are using. For example, although **Tutorial D** fully supports multiple assignment, Rel does not (yet) and nor do SQL systems in general (though I learned recently that the latest version of Oracle supports some kind of simultaneous insertion into two or more tables, so perhaps the industry is at last moving on that front).

The students-and-courses example illustrates a further criterion for sometimes preferring one logical schema over another. My original ENROLMENT relvar has nothing *logically* wrong with it, so long as we include in the design a constraint to the effect that the same student identifier is always paired with the same name, however many times that student identifier appears in the current value of ENROLMENT at any time. However, the 5NF design derived by decomposing ENROLMENT into IS_CALLED and IS_ENROLLED_ON solves the problem a lot more neatly by making sure that every student's name is recorded no more than once—recorded as far as can be seen via the logical schema, that is! And that brings me to perhaps the most important point I wish to make here.

The question arises as to whether our choice of logical schema can be guided by performance considerations in addition to psychological and pragmatic ones. For example, might we decide to prefer the non-5NF ENROLMENT over the 5NF IS_CALLED/IS_ENROLLED_ON because for some reason we would expect things to go faster that way?

Well, first of all, there are many different kinds of "things" that the users of the database might wish to do, so we would have to decide *which* kinds of things are going to be done most often. Secondly—and this is the point that is so often overlooked—having decided which operations we are going to optimise for, we have to know how the DBMS actually stores the data and executes those operations. In CS252 we do not teach you how Oracle stores data and executes SQL operations; nor how Rel stores data and executes **Tutorial D** operations. That's because our purpose is to teach you SQL and **Tutorial D**, not Oracle and Rel! Besides, why should we teach you how one particular DBMS works in 2006, when that might be significantly different from how it will work in 2020 and even more significantly different from how several other implementations of the same language currently work?

But people sometimes make assumptions about how a DBMS works, without realising how naïve and possibly incorrect those assumptions might be. Let me give you a couple of examples.

My first example is the common assumption that "joins slow queries down".

With our students-and-courses example, this assumption means that the query `ENROLMENT WHERE StudentId = 'S1'` (against the non-5NF design) goes faster than `IS_ENROLLED_ON JOIN ( IS_CALLED WHERE StudentId = 'S1' )`, and the same applies to lots of similar queries. Under this assumption the two-relvar 5NF design might be rejected in favour of the single-relvar one.

But the assumption might not be correct! And even if it *is* correct, the assumption that a query such as `ENROLMENT { StudentId, Name }` will be required significantly less often than `ENROLMENT` is also questionable. And in proposing that point as an argument in favour of the 5NF design I am assuming that the projection will always run slower than the simple relvar reference, `IS_CALLED`. But even *that* assumption cannot be relied upon unless we actually know how the particular system we are using works.

Here are the two designs under consideration:

```
1.  VAR ENROLMENT BASE RELATION
        { StudentId CHAR, Name CHAR, CourseId CHAR }
    KEY { StudentId, CourseId } ;
```

```
2.  VAR IS_CALLED BASE RELATION
        { StudentId CHAR, Name CHAR }
    KEY { StudentId } ;

    VAR IS_ENROLLED_ON BASE RELATION
        { StudentId CHAR, CourseId CHAR }
    KEY { StudentId, CourseId } ;

    CONSTRAINT All_named_are_enrolled_and_vice_versa
      IS_ENROLLED_ON { StudentId } =
      IS_CALLED { StudentId } ;
```

The constraint in Design 2 is implied in Design 1 because each tuple in ENROLMENT gives both a name and a course for the student in question. I didn't include this constraint when I discussed decomposing ENROLMENT because one of the possible advantages of decomposing is that it enables students who are not yet enrolled on anything to at least have their names recorded. But here we are considering two *equivalent* designs, meaning that if Design 1 can be considered at all, then the whole of Design 2 is needed to meet the same requirements. In other words, if the constraint in Design 2 is not required, then Design 1 cannot be considered as an alternative.

Now, if the DBMS adopts a simple method of storage in which elements of the logical schema map in a very direct and obvious way to elements of the *storage schema*, then we might expect each ENROLMENT tuple to be stored on disk as a single record consisting of three contiguous fields (possibly very long fields, considering that the CHAR specification imposes no length restriction on the corresponding attribute values!). Similar expectations would apply to IS_ENROLLED_ON tuples and IS_CALLED tuples.

But DBMSs typically do *not* use such simple storage algorithms. Some, for example, would actually store a wife of Henry VIII (as in lecture HACD.8) in three separate records even when the logical schema uses the preferred single-relvar design. Conversely, it is quite reasonable to expect a relational DBMS to implement the 6NF design, noting the constraints, by storing just one record for each wife.

It was very much part of the relational vision of the 1970s that the database designer would be able to ignore performance considerations when formulating a logical schema. The storage schema and its mapping to the logical schema would either be determined automatically by the DBMS, perhaps according to the kind of business the product in question is intended for (e.g., "data warehouse" or "on-line transaction processing"), or, failing that, provide some storage language to enable it to be specified explicitly by the database designer as an *implementation* of the logical schema. In the latter case we can even imagine that logical schema design and storage schema design, being quite different skills, are undertaken by different professions.

I have one final example that I hope will drive the point home. It arises from my solution to the Lecture HACD.9 exercise, where I use a single attribute for the postal address of a customer. Postal addresses are quite long, and it is easily possible for several customers (perhaps members of the same family) to have the same postal address. Wouldn't it save space to store the postal address just once, even in those cases? We could achieve that by assigning a unique id to each address and have an AddressId attribute in Customer in place of Address. The AddressId would be a

foreign key referencing a separate relvar, keyed on AddressId, pairing address ids with actual addresses. But if long character strings are actually stored by the DBMS in the way I am about to describe—a method that I once devised and implemented myself for a real industrial-strength DBMS—then that idea could actually be extremely *counter*productive!

The method I have just mentioned used what we called a *literal pool* (a term taken from compiler technology). In the physical record for storing a customer's address, a fairly short, fixed-length field was allocated. In this field were stored the first few characters of the address and a pointer to the place in the literal pool where the rest was stored. Suppose "first few" was actually 5, and 8 bytes were used for the pointers, and some address common to three people was 100 characters long. Then the total storage for the addresses of those three customers was 95+(3*13) = 134 instead of the 300 you might have been expecting. With the AddressId approach, assuming address ids occupy 6 bytes, the storage used would be 6+5+8+95 = 114 for the Address tuple and 3*6 = 18 for the AddressId fields in the records for the Customer tuples. So the saving on storage is pretty marginal, though it does get better in this particular example as the number of people living together increases. However, the AddressId design entails more pointer-chasing than the simple design and, furthermore, it loses the advantage of storing the first few characters in the Customer record. It is working *against* the system! (Having the first few characters to hand, so to speak, often saves the system from having to access all of a string to evaluate some comparison.)

Do not spend too much time questioning whether that literal pool idea was really a good one. For one thing I haven't given all the details of how it worked, but my main point is that the actual algorithms used by a DBMS might be quite different from what you are assuming and in that case it doesn't matter whether those algorithms are good or bad.

End