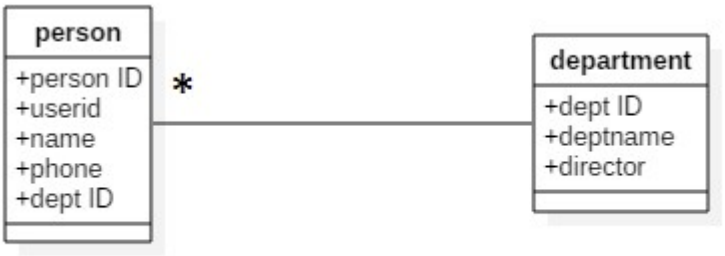
The following is from the upcoming 2nd Edition of Enterprise Delphi Databases.

Sharding and De-Normalization for Speed and Freedom

Usually databases should be normalized, which generally means storing different but related pieces of information in different logical tables called relations.

For example, a Person table might express your userid, name, phone number, etc. A relation would express a different fact which might relate you to a department, a shared office building, etc.



Un-normalized databases are ones that have not been normalized yet, usually due to inexperience.

De-normalizing is different from un-normalized tables, it is a more advanced strategy of optimizing performance of frequent read operations at the expense of less frequent operations. This is accomplished by partially and purposefully abandoning the normalized model: adding redundant copies or grouping data differently.

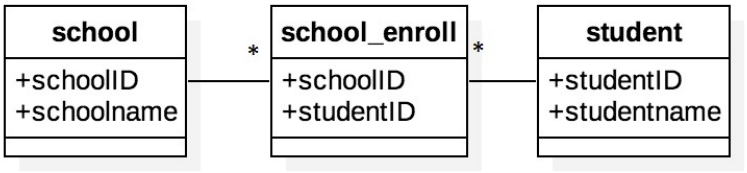
We will discuss several common examples.

Storing Meta-Information

A common use is to store meta-information, such as the count of sub-entities under some form of umbrella. Consider a school board system with tables about schools.

It is probably handy to know the student count at each school in everyday operations. Since some students take courses at multiple different schools (night classes, or enrichment programs), the model

looks like:

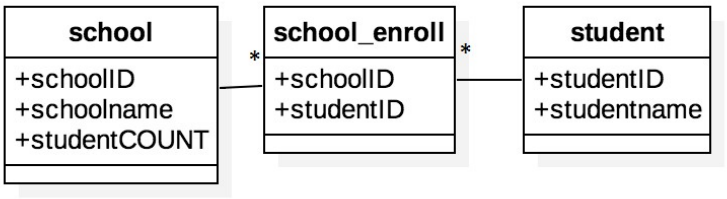


And that quantity can be calculated with :

```
SELECT COUNT(*)
FROM
        school_enroll
        NATURAL JOIN school
WHERE schoolname = 'PCVS' ;
```

Now suppose the web page displays the schools and their enrollments 500 times per hour. That's a lot of database querying.

It would be more efficient to add a studentCOUNT integer to the schools



and just display:

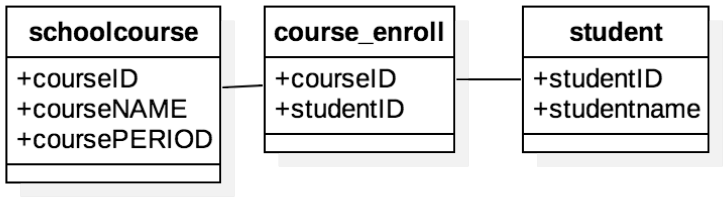
```
SELECT
        studentCOUNT
FROM
        school_enroll
WHERE
        schoolname = 'PCVS'
```

Then whenever we add a new student, something we do every few days, we update schoolCOUNT to have the correct new count.

This significantly reduces the number disk requests for a very common read operation, and comes only at some slight expense when we add, delete or change enrollment.

Storing One-To-Many Relations

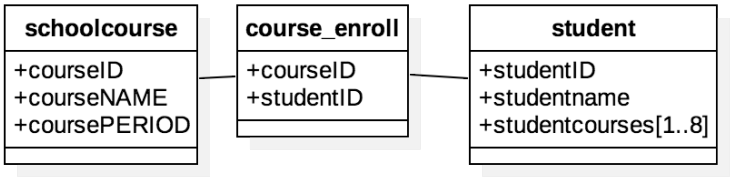
Suppose your school board mobile application will display each student's timetable at every class change. Each student can take up to 8 courses, and you have 1,000 students at each of 50 schools. That's 50,000 queries per inter-class gap occurring about 8 times per day.



Each user's query is something like this for student 2301 at period 1:

```
SELECT
        courseNAME
FROM
        student
        NATURAL JOIN course_enroll
        NATURAL JOIN schoolcourse
WHERE
        studentID = 2301
        and coursePERIOD = 1
```

What if you cached an array of courseIDs[1..8] in the student record, your database would not have to rescan all those tables;



So we would only have to query:

```
SELECT
        studentcourses
WHERE
        studentID = 2301
```

And select the first element out of the array, where the array has:COURSE.

This reduced disk reading comes at expense when students add or drop courses, usually two actions per school semester.

mORMot easily handles dynamic arrays, and that's one way to do it, but I like the sharding alternative.

```
TSQLstudent = class ( TSQLRecord )
Private
 Fstudentcourses : variant;
Published
 Studentcourses : variant read Fstudentcourses
        write Fstudentcourses;
End;

// set your courses
Student.studentcourses := _ObjFast($['course1',
        'ENG233$]$,␣'course2',␣'SCI203',␣...␣$]$);

//␣get␣period␣1
coursename␣:=␣JSONGet(␣student.studentcourses,
␣␣␣␣␣␣␣␣'course'␣+␣IntToStr(␣1␣)␣);
```

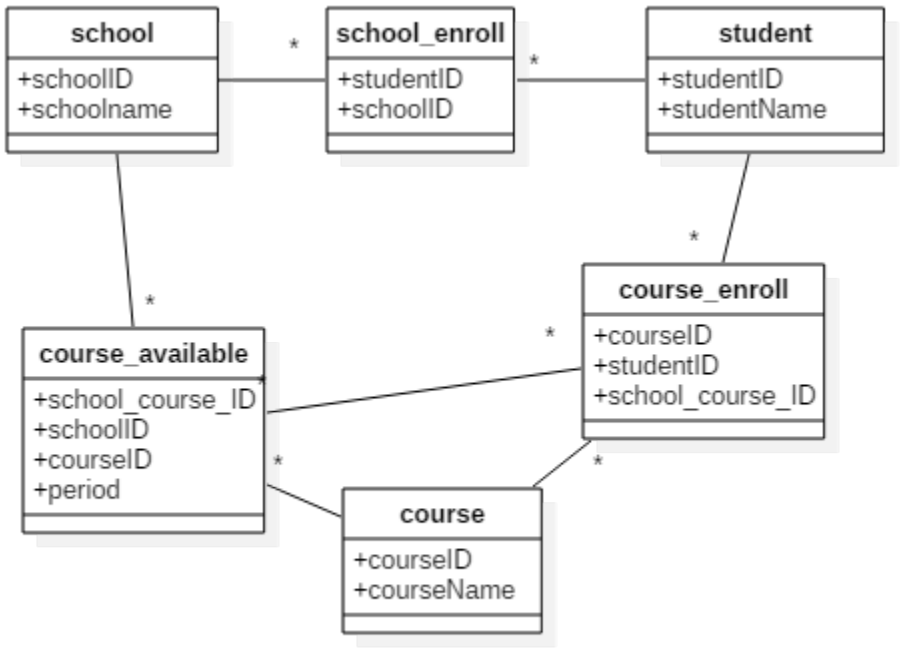
This will speed up performance incredibly over the normalized version as fewer disk accesses are needed.

In a more realistic model, the school board will likely offer the same course at multiple schools and some students will be cross-enrolled at other schools for certain courses like remedial math, extended languages, etc.

Now to show the course for period 1 for student 234

```
SELECT
        schoolname , couseName
FROM
        student stu
        JOIN course_enroll ce
        JOIN course_available ca
        JOIN course c
        JOIN school s ON stu.studentID = ce.studentID,
                ce.school_course_ID = ce.school_course_ID,
                ca.courseID = ce.courseID,
                s.schoolID = ca.school
WHERE
        studentID = 234 AND period = 1;
```

And how many times per second will this query be done?



Sharding can greatly relieve the database if we store an array of *schoolName* : *courseName* in the student record.

```
Student.studentcourses = \_ObjFast($[$'course1','PCVS:ENG233$]$,
'course2','Eastwood:SCI203',\ldots␣$]$);
```

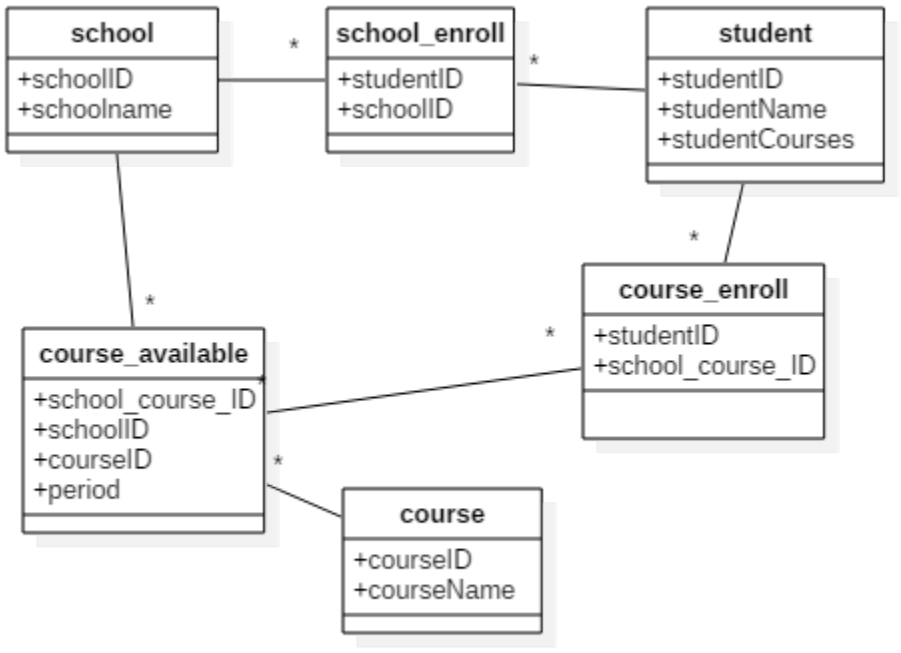
Now we can just read the student record for studentID = 234 and grab the first element of the studentCourses array.

Sharding to denormalize adds extra write operations. For this example it means we have to update the student record every time we add or delete a course for the student.

Even though we have denormalized the table, other purposes the normalized full structure be kept intact as though we had not denormalized.

For example, suppose we realize we could eliminate the course_enroll table in this example because we have the data repeated in the studentCourses array. But then if we wanted to create a class list, we would have to iterate through every single student in the database to find those enrolled in the course at a particular school.

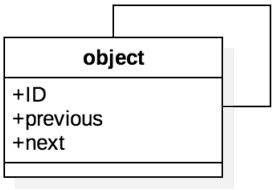
Also note, sharding in this implementation will not speed up unrelated queries like displaying a class list. To accomplish fast class lists that we would have to add an array of students to course_available.



Storing Linked Lists

There are many situations where you wish to set ordering of data, but then you typically have to maintain an index of all the row entries. That slows down when you need to insert or delete an element. If your list is very large, and additions or deletions are frequent, your performance is lost just trying to maintain list order.

A standard computer science solution is to use linked list. And we can do that with mORMot.




```
TSQLobject = class ( TSQLRecord )
Private
 Fprev, Fnext : TID; Published
 prev : TID read Fprev write Fprev; next : TID read Fnext write FNext;
End;
```

Sparse or Unexpected Fields

Sometimes you could use a gazillion rarely used fields, but you will rarely search on them, or maybe you cannot anticipate all of them at design time.

For example, an asset management database may solve all your problems, but suddenly someone asks you to store whether the monitor is dual-voltage capable. Do you rewrite all your access screens and update the database schema for every request like this?

Sharding can be your salvation in these situations. Since it uses variants, it can store any imaginable field name and value or values if you want an array.

Sharding has very small additional cost to performance or space unless you wish to interpret the data. The BLOB format is an efficient use of space and there is very little processing involved managing it. When you create, read, update or delete the sharded data there is a small CPU penalty compared to other fields, but it is orders of magnitude smaller than separate disk accesses typically required for normalized tables.

On ORM in mORMot

If you choose to use the ORM in mORMot (and avoid SQL entirely), think of your system as an object repository and totally de-normalize your database.

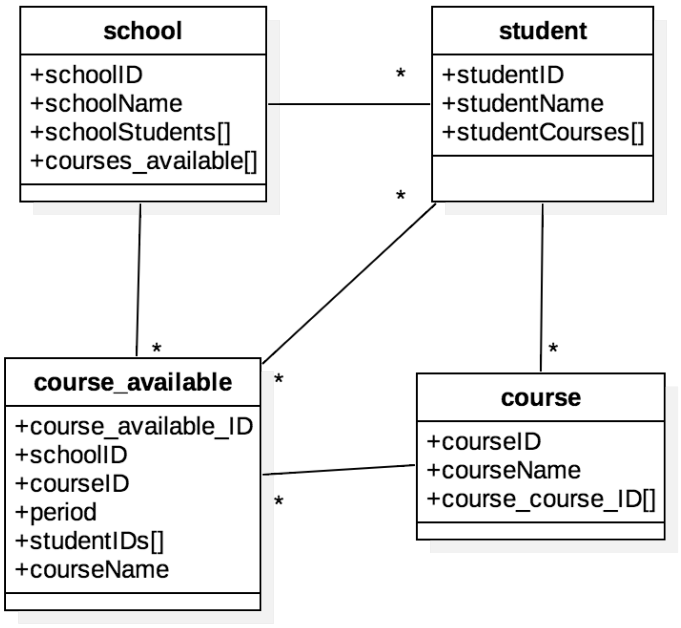
For example, here is how we could re-envision the multiple students attending multiple schools to attend classes based on common course names.

Most developers are surprised to see that one table might contain a list of, say, 1,500 students. But an array of 1,500 TIDS is just a single efficient 12 kB BLOB. In modern systems that is a small amount of memory.

Final Thoughts

mORMot supports normalized joined tables through its SQL interface, it also supports limited normalized tables.

mORMot's ORM recommends de-normalized objects arrays and sharding rather than pure relational tables. The performance is often optimized by reducing disk reads.


school
+schoolID
+schoolName
+schoolStudents[]
+courses_available[]
student
+studentID
+studentName
+studentCourses[]
course_available
+course_available_ID
+schoolID
+courseID
+period
+studentIDs[]
+courseName
course
+courseID
+courseName
+course_course_ID[]
*
*
*
*
*
*