CS252:HACD Fundamentals of Relational Databases

Notes for Section 1: Introduction

1. Cover slide

These notes are adapted from the ones given to Hugh Darwen's students at Warwick University.

Introductory Remarks

In Block 2 of M359, *Relational Databases: Theory and Practice*, we study the theory upon which relational databases are based. Or perhaps it would be better to say, "should be based", or "were intended to be based (by its original proponent)". We also study a database language that is firmly based on this theory. When M359 was developed by the Open University, no implementation of this language was available for students to experiment with and test their TMA solutions. In 2010 former M359 student John Waller decided to make such an implementation as his project on course M450. *Rel359* is that implementation and on successful completion of his project he decided to make *Rel359* available to future generations of M359 students (at their own risk, of course).)

Block 3 of M359 introduces you to the "state of the art" of relational database support in the industry. This takes the form of SQL, the language that was dubbed "intergalactic dataspeak" by database guru Michael Stonebraker in the 1980s (since when its use and its number of implementations have grown enormously). You are strongly encouraged to compare and contrast the theory of Block 2 and the practice of SQL!

Lecture 1: Introduction

Lecture 1 gives a very broad overview of

- what a database is
- what a relational database is
- what a database management system (DBMS) is
- what a DBMS does
- how a relational DBMS does what a DBMS does

We start to familiarise ourselves with terminology and notation used on this course.

We get a brief introduction to each topic that will be considered in more detail in later sections of the course.

2. Some Preliminaries

Note that the word "database" had not been coined when Ted Codd, actually a hardware specialist working on for IBM on its mainframe architecture at the time, did this important work for which he was later a recipient of the Turing Award. Codd died in 2003.

Tutorial D (which is always written that way, in bold face) was invented in the late 1990s by Hugh Darwen and C.J. (Chris) Date for the very purpose that its name suggests—teaching, as opposed to commercial use. It is used for examples and exercises in several books coauthored by them and in particular in the two books by Chris Date that are recommended reading for this course. The official definition of the language is given in their book, "Databases, Types, and The Relational Model: *The Third Manifesto*" (Pearson Education inc., 2007, ISBN 0-321-39942-0), but this book is not suitable for an introductory course of this nature and in any case we will not use all of the language.

An implementation of **Tutorial D**, *Rel,* has been made available to the academic community at large as an Open Source product by its author, **Dave Voorhis** of the University of Derby. John Waller's *Rel359* is an adaptation of *Rel.*

3. What Is a Database?

You will find many definitions of this term if you look around the literature and the Web. Wikipedia offers this: "A structured collection of records or data." I prefer to elaborate a little, as shown on the slide:

> A **database** is an *organised*, machine-readable collection of *symbols*, to be *interpreted* as a *true* account of some *enterprise*. A database is machine-updatable too, and so must also be a collection of *variables*. A database is typically available to a community of users, with possibly varying requirements.

The organised, machine-readable collection of symbols is what you "see" if you "look at" a database at a particular point in time. It is to be interpreted as a true account of the enterprise at that point in time. Of course it might happen to be incorrect, incomplete or inaccurate, so perhaps it is better to say that the account is *believed* to be true.

The alternative view of a database as a collection of variables reflects the fact that the account of the enterprise has to change from time to time, depending on the frequency of change in the details we choose to include in that account.

The suitability of a particular kind of database (such as relational, or object-oriented) might depend to some extent on the requirements of its user(s). When E.F. Codd developed his theory of relational databases (first published in 1969), he sought an approach that would satisfy the widest possible ranges of users and uses. Thus, when designing a relational database we do so without trying to anticipate specific uses to which it might be put, without building in biases that would favour particular applications. That is perhaps *the* distinguishing feature of the relational approach, and you should bear it in mind as we explore some of its ramifications.

4. "Organised Collection of Symbols"

For example, the table shown in the slide shows an organized collection of symbols:

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3

Can you guess what this tabular arrangement of symbols might be trying to tell us? What might it mean, for symbols to appear in the same row? In the same column? In what way might the meaning of the symbols in the very first row differ from the meaning of those below them?

Do you intuitively guess that the symbols below the first row in the first column are all student identifiers, those in the second column names of students, and those in the third course identifiers? Do you guess that student S1's name is Anne? And that Anne is enrolled on courses C1 and C2? And that Cindy is enrolled on neither of those two courses? If so, what features of the organisation of the symbols led you to those guesses?

Remember those features. In an informal way they form the foundation of relational theory. Each of them has a formal counterpart in relational theory, and those formal counterparts are the only constituents of the organized structure that is a relational database.

5. "To Be Interpreted as a True Account"

For example, from the table just shown:

StudentId	Name	CourseId
S1	Anne	C1

Perhaps those green symbols, organised as they are with respect to the blue ones, are to be understood to mean:

"Student S1, named Anne, is enrolled on course C1."

In that case, your guesses were correct.

An important thing to note here is that only certain symbols from the sentence in quotes appear in the table—S1, C1, and Anne. None of the other words appear in the table. The symbols in the top row of the table (column headings?) might help us to guess "student", "named", and "course", but nothing in the table hints at "enrolled". And even if those assumed column headings had been A, B and C, or X, Y and Z, the given interpretation might still be the intended one.

Another important point is that the interpretation takes the form of a *declarative sentence* of which it can be said, "that is *true*" or "that is *false*". Such declarative sentences are what logicians call *propositions*.

Now, we can take the sentence "Student S1, named Anne, is enrolled on course C1" and replace each of S1, Anne, and C1 by the corresponding symbols taken from some other row in the table, such as S2, Boris, and C1. In so doing, we are applying exactly the same mode of interpretation to each row. If that is indeed how the table is meant to be interpreted, then we can conclude that the following sentences are all true:

Student S1, named Anne, is enrolled on course C1.

Student S1, named Anne, is enrolled on course C2.

Student S2, named Boris, is enrolled on course C1.

Student S3, named Cindy, is enrolled on course C3.

In my paper "What a Database Really Is: Predicates and Propositions", I show you exactly how such interpretations can be systematically formalized. And in the Block 2 material on relational algebra, you will see how they help us to formulate correct queries to derive useful information from a relational database.

6. "Collection of Variables"

This slide gives a name, ENROLMENT, to a table very similar to the one shown in slide 4. The only changes are the additions of the name, ENROLMENT, above the table and an extra row:

ENROLMENT

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Perhaps the table shown in slide 4 was once the value of this same variable, which has since been "updated". Our interpretation of the table in Slide 4 now has to be revised to include the sentence represented by that additional row:

Student S1, named Anne, is enrolled on course C1.

Student S1, named Anne, is enrolled on course C2.

Student S2, named Boris, is enrolled on course C1.

Student S3, named Cindy, is enrolled on course C3.

Student S4, named Devinder, is enrolled on course C1.

Notice that in English we can join all these sentences together to form a single sentence, using conjunctions like "and", "or", "because" and so on. If we join them using "and" in particular, we get a single sentence that is logically equivalent to the given set of sentences in the sense that it is true if *each* one of them is true (and false if *any* one of them is false). A database, then, can be thought of as a representation of an account of the enterprise expressed as a single sentence! (But it's more usual to think in terms of a collection of individual sentences.)

We might also be able to conclude that the following sentences (for example) are false:

Student S2, named Boris, is enrolled on course C2.

Student S2, named Beth, is enrolled on course C1.

Whenever the variable is updated, the set of true sentences represented by its value changes in some way. Updates usually reflect perceived changes in the enterprise, affecting our beliefs about it and therefore our account of it.

7. What Is a Relational Database?

A relational database is one whose symbols are organised into a collection of *relations*. Slide 7 confirms that the examples we have already seen are in fact relations, depicted in tabular form.

Happily, this visual (tabular) representation we have been using thus far is suited particularly well to relational databases: so much so that many people use the word *table* as an alternative to *relation*. The language SQL in particular uses that term, so in the context of relational theory it is convenient and judicious to stick with relation for the theoretical construct, allowing SQL's deviations from relational theory to be noted as differences between tables and relations.

Relation is a formal term in mathematics—in particular, in the logical foundation of mathematics. It appeals to the notion of relationships between things. Most mathematical texts focus on relations involving things taken in pairs but our example shows a relation involving things taken three at a time and, as we shall see, relations in general can relate any number of things (and, as we shall see, the number in question can even be less than two, making the term relation seem somewhat inappropriate).

Relational database theory is built around the concept of a relation. Our study of the theory will include:

- The "anatomy" of a relation.
- **Relational algebra:** a set of mathematical operators that operate on relations.
- **Relation variables:** their creation and destruction, and operators for updating them.
- **Relational comparison operators,** allowing **consistency** rules to be expressed as **constraints** (commonly called **integrity constraints**) on the variables constituting the database.

And we will see how these, and other constructs, can form the basis of a **database language** (specifically, a *relational* database language).

8. Relation ≠ Table

The title of this slide is trying to say that the terms "relation" and "table" are not synonymous. The following table (from the slide) is different from the one we have just been looking at, but represents the same relation:

Name	StudentId	CourseId
Devinder	S4	C1
Cindy	S3	C3
Anne	S1	C1
Boris	S2	C1
Anne	S1	C2

For one thing, although every relation can be depicted as a table, not every table is a representation of (i.e., *denotes*) some relation. For another, several different tables can all represent the same relation.

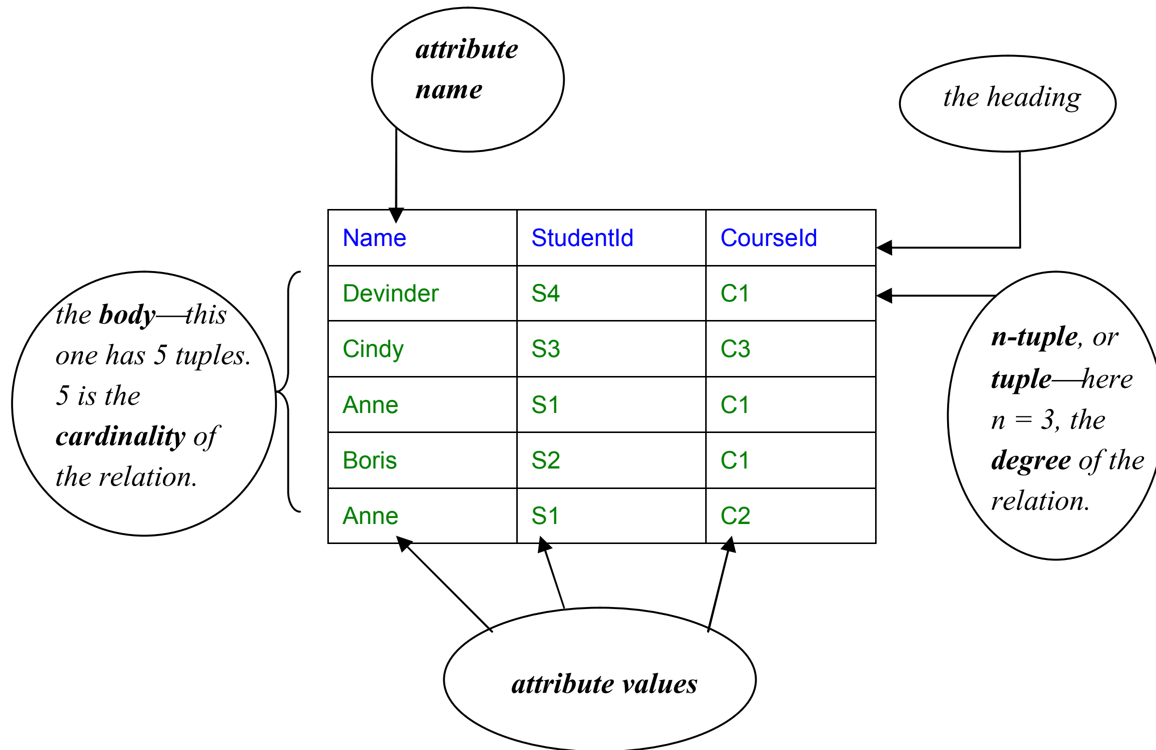
A table that does not depict any relation is shown in the EXERCISE given in the last slide of this lecture.

Several different tables can all denote the same relation, because we can simply change the left-to-right order in which the columns are shown and/or the top-to-bottom order in which the rows are shown and yet still be depicting the same relation.

What does it mean to say that the order of columns and the order of rows doesn't matter? We will find out the answer to this question when we later study the typical *operators* that are defined for operating on relations (e.g., to compute results of queries against the database) and relation variables (e.g., to update the database). None of these operators will depend on the notion of some row or some column being the first or last, or immediately before or after some other column or row.

9. Anatomy of a Relation

Here is the picture shown on the slide:



Because of the distinction I have noted between the terms "relation" and "table", we prefer not to use the terminology of tables for the anatomical parts of a relation. We use instead the terms proposed by E.F. Codd, the researcher who first proposed relational theory as a basis for database technology, in 1969.

Try to get used to these terms. You might not find them very intuitive. Their counterparts in the tabular representation might help:

relation	:	table
(n-)tuple	:	row
attribute	:	column

Also (repeating what is shown in the slide):

The **degree** is the number of attributes.

The **cardinality** is the number of tuples.

The **heading** is the *set* of attributes (note **set**, because the attributes are not ordered in any way and no attribute appears more than once).

The **body** is the *set* of tuples (again, note **set**—the tuples are not ordered and no tuple appears more than once).

An attribute has an **attribute name**,[1] and no two have the same name.

Each attribute has an **attribute value** in each tuple.

[1] An attribute also has a type, for which M359 uses the term "domain".

10. What Is a DBMS?

A database management system (DBMS) is exactly what its name suggests—a piece of software for managing databases and providing access to them. But be warned!—in the industry the term database is commonly used to refer to a DBMS, especially in promotional literature. You are strongly discouraged from adopting such sloppy practice (if such a system is a database, what are the things it manages?)

Before looking at the components we expect to find in a relational DBMS, we need to briefly review what is expected of a DBMS in general.

A DBMS responds to *commands*[2] given by *application programs*, custom-written or general-purpose, executing on behalf of users. Commands are written in the *database language* of the DBMS (e.g., SQL). Responses include completion codes, messages and results of *queries*.

In order to support multiple concurrent users a DBMS normally operates as a *server*. Its immediate users are thus those application programs, running as *clients* of this server, typically (though not necessarily) on behalf of *end users*. Thus, some kind of *communication protocol* is needed for the transmission of commands and responses between client and server. Before submitting commands to the server a client application program must first establish a *connection* to it, thus initiating a *session*, which typically lasts until the client explicitly asks for it to be terminated. That is all you need to know about *client-server architecture* as far as M359 is concerned.

The term *data sublanguage* is sometimes used instead of database language. The "sub-" prefix refers to the fact that application programs are sometimes written in some more general-purpose programming language (the "host" language), in which the database language commands are embedded in some prescribed style. Sometimes the embedding style is such that the embedded statements are unrecognised by the host language compiler or interpreter, and some special *preprocessor* is used to replace the embedded statements by, for example, CALL statements in the host language.

A **query** is an expression that, when evaluated, yields some result derived from the database. Queries are what make databases useful. Note that a query is not a command. The DBMS might support some kind of command to evaluate a given query and make the result available for access, also using DBMS commands, by the application program. The application program might execute such commands in order to display a result in tabular form in a window.

11. What Does a DBMS Do?

In response to requests from application programs, we expect a DBMS to be able to:

- create and destroy variables in the database
- take note of integrity rules (*constraints*)
- take note of *authorisations* (who is allowed to do what, to what)
- update variables (honouring constraints and authorisations)
- provide results of *queries*

To amplify some of those terms:

The **requests** take the form of commands written in the database language supported by the DBMS.

[2] The less appropriate term *statement* is very commonly used instead of command, especially for certain kinds of command; I use that term quite often myself, as you will see. The term *imperative* is also occasionally used.

The **variables** are the constituents of the database, like the ENROLMENT variable we looked at earlier. (*Note:* M359 doesn't actually use the term variable for the constituents of a database. In Block 2 it uses the same term, *relation,* both for the variables and for the values assigned to or derived from those variables. In Block 3, it similarly puts the SQL term *table* to both purposes.)

Constraints (sometimes called **integrity constraints**) are rules governing permissible values and permissible combinations of values, of the variables. For example, it might be possible to tell the DBMS that no student's assessment score can be less than zero. A database that **violates** a constraint is false, by definition. A database that **satisfies** all its constraints is said to be **consistent**, even though it cannot in general be guaranteed to be true.

Authorisations are for **security** what constraints are for integrity**.** Some of the data in a database might represent sensitive information whose accessibility is restricted to certain **privileged** users only. Similarly, it might be desired to allow some users to access certain parts of the database without also being able to update those parts.

Note the three parts of an authorisation: who, what, and to what. "Who" is a user of the database; "what" is one of the operations that are defined for operating on the variables in the database; "to what" is one of those variables.

12. Create and Destroy Variables

Two *Rel359* commands are shown on Slide 12, one to create a variable in the database, the other to destroy a variable. Here, again, is the first one—

```
relation ENROLMENT
          StudentId:  CHAR
          Name:       CHAR
          CourseId:   CHAR
       primary key ( StudentId, CourseId ) ;
```

—explained as follows:

- **relation** is a key word indicating that a relation variable is being created.
- **ENROLMENT** is the variable's name.
- Lines 3-5 define the attributes of ENROLMENT, giving the name and domain (or type) for each one. CHAR is the type name used in *Rel359* for character strings.
- The **primary key** specification indicates that the variable is subject to a certain kind of **constraint**, in this case declaring that no two tuples in the relation assigned to ENROLMENT can ever have the same combination of attribute values for StudentId and CourseId (i.e., we cannot enrol the same student on the same course more than once, so to speak).
- You will learn more about constraints in general and key constraints in particular later in the course.

The second command shown on the slide—

```
drop ENROLMENT ;
```

—shows how a variable can be destroyed. After execution of this command the variable no longer exists and any attempt to reference it is in error.

13. Take Note of Integrity Rules

Here is the constraint declaration shown on the slide—

```
constraint NamePresent
    ( select ENROLMENT where Name = '' )
    is empty ;
```

—and here is its explanation:

> **constraint** is a key word indicating that a constraint is being declared.
>
> **NamePresent** is the name of the constraint.
>
> **select** ENROLMENT **where** Name = '' is a *Rel359* expression denoting the relation consisting of all the tuples of ENROLMENT whose Name attribute value is the empty string=.
>
> **(select** ENROLMENT **where** Name = '') **is empty** is a truth-valued expression yielding *true* if the relation denoted by the expression in parentheses is indeed empty (i.e., no tuple in ENROLMENT satisfies the condition Name = ''), otherwise yielding *false*.

The declaration tells the DBMS that the database is *inconsistent* if the value of NamePresent is ever *false*, and that the DBMS is therefore to reject any attempt to update the database that, if accepted, would bring about that situation.

14. Take Note of Authorisations

Tutorial D and *Rel359* don't include any commands for creating and destroying permissions, because security and authorisation, though important, is not specifically a *relational* database issue. In SQL they look like this, as shown on the slide.

```
GRANT SELECT ON ENROLMENT TO User9 ;
GRANT UPDATE ON ENROLMENT TO User8 ;
```

The first GRANT statement confers permission to "select", authorising the user User9 to read the data in ENROLMENT but not to update it. The second GRANT statement authorises the user User8 to use SQL UPDATE commands on ENROLMENT. SQL uses the key word REVOKE for withdrawing a previously given permission. You learn all about authorisation in SQL in Block 3.

15. Updates Variables

The usual way of updating a variable in computer languages is by *assignment.* For example, if X is an integer variable, the assignment X := X + 1 updates X such that its value immediately after execution of the assignment is one more than its value was immediately beforehand. The expression on the right of **:** = denotes the *source* for the assignment and the variable name on the left denotes the *target*.

When the target is a relation variable—as it always is when it is part of a relational database—the source must be a relation. You will learn how to write such expressions that denote relations in Block 2, but in any case assignment, though it *should* be available (it isn't in SQL), is not the usual way of applying updates to a relational database. This is because there is very often only a small amount of difference, in a manner of speaking, between the "old" value and the "new" value and it is usually much more convenient to be able to express the update in terms of that small difference.

Block 2 is not concerned with updating, so you don't actually learn how to update until you come to the SQL commands for that purpose in Block 3. *Rel359* therefore uses the **Tutorial D** commands.

The differential update operators expected in a relational DBMS are usually called **insert, update,** and **delete,** and those are the names used in **Tutorial D** and *Rel359* (also in SQL). Take a look at **delete** first:

```
delete ENROLMENT where StudentId = 'S4' ;
```

Informally, this deletes all the tuples for student S4 and means “student S4 is not enrolled on any courses”. More formally, it assigns to the variable ENROLMENT the relation resulting from the removal, from the current value of ENROLMENT, of every tuple in which the value of the StudentId attribute is the student identifier S4, and the retention of every other tuple in the current value of ENROLMENT. In other words, it assigns to ENROLMENT the value of **select** ENROLMENT **where not** (StudentId = 'S4').

Next, we look at **update**:

```
update ENROLMENT where StudentId = 'S1'
                 ( Name := 'Ann' ) ;
```

Note that **update** uses a **where** clause, just like **delete**. The **where** clause is followed by a list of assignments—in the example, just one assignment—but these are assignments to attributes, not assignments to variables. Informally, this updates each tuple for student S1, changing its Name value to 'Ann'. More formally, it assigns to the variable ENROLMENT the relation that is identical to the current value in all respects except for the value for the attribute Name in the tuples whose StudentId value is 'S1', which is the string 'Ann' in each case. (I would have written “except possibly” had I not known that the existing Name value in those tuples is 'Anne' in each case, allowing for the fact that in some circumstances no change takes place as a result of executing this command.)

The final example on the slide illustrates the use of **insert**:

```
insert ENROLMENT
    relation {
         tuple { StudentId 'S4' ,
                 Name 'Devinder' ,
                 CourseId 'C1' } }  ;
```

Informally, this adds a tuple to ENROLMENT indicating that student S4, still called Devinder, is now enrolled on course C1. More formally, it assigns to the variable ENROLMENT the relation consisting of every tuple in the current value of ENROLMENT and every tuple (there is only one) in the relation denoted by the expression following the word ENROLMENT. That expression is a relation literal in **Tutorial D**. M359 does not give any notation for relation literals, so *Rel359* uses the **Tutorial D** notation as shown.

The first example has no effect on the database in the case where the current value of ENROLMENT has no tuples for student S4.

The second example has no effect on the database in the case where the current value of ENROLMENT has no tuples for student S1.

The third example has no effect on the database in the case where the current value of ENROLMENT already contains the tuple representing the enrolment of student S4, named Devinder, on course C1.

Note that any attempt to insert a tuple with the empty string, '', for the Name value, would be rejected by the DBMS if the constraint NamePresent shown on Slide 13 is in effect. Similarly, an **update** command specifying Name := '' would also be rejected.

16. Provides Results of Queries

Expressing queries in relational algebra is dealt with in detail in Block 2. Here I present just a simple example to give you the flavour of things to come in that Block. The example is a query expressing the question, who is enrolled on course C1?

```
project ( select ENROLMENT where CourseId = 'C1'
over StudentId, Name
```

Note carefully that this is not a command. It is just an expression, denoting a value—in this case, a relation. In a relational database language the result of a query is always another relation! Here is the relation that is the result of this query, shown as usual in tabular form:

CourseId	No_of_students
C1	3
C2	1
C3	1

And here is an explanation of the query:

select[3] is the key word identifying the *Rel359* operator of that name. This operator operates on a given relation and yields the relation consisting of every tuple in the given relation that satisfies the **where** condition. The operators, including this one, that operate on relations and yield relations constitute the *relational algebra*.

CourseId = 'C1' is the **where** condition, specifying that just the tuples for course C1 are required.

project … over StudentId, Name specifies that from the result of the previous operation (**select**) just the StudentId and Name attributes are required.

17. EXERCISE

Consider this table:

A	B	A
1	2	3
4		5
6	7	8
9	9	?
1	2	3

Give three reasons why it cannot possibly represent a relation.By the way, this table is supported by SQL, and the three reasons represent some of SQL's serious and far-reaching deviations from relational theory.

End of Notes

[3] Advance warning on SQL: SQL also uses the key word SELECT but with a completely different meaning!