1 Introduction

1.1 Introduction

Chapter 1 of the theory book gives a very broad overview of

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

It introduces you to the terminology and notation used in the remainder of the book and gives a brief introduction to each topic that is covered in more detail in subsequent sections. The first, third and fourth bullet points clearly apply equally well in the SQL world and I have nothing to add on those topics here. However, the second and last bullet points need to be replaced, respectively, by

- what an SQL database is, in particular

and

- how an SQL DBMS does what a DBMS does

Sections 1.2 to 1.5 of the theory book deal with the first bullet point, what a database is, so our comparative study of SQL starts with Section 1.6. (Note, by the way, that I wrote "an SQL", not "a SQL". Some people pronounce the name as "sequel" but the official pronunciation is "ess cue ell".)

Historical Note

The origins of SQL can be traced to an IBM research paper published in 1974, titled *SEQUEL: A Structured English Query Language*, by Donald D. Chamberlin and Raymond F. Boyce. Raymond Boyce, who sadly died in the same year, 1974, was the person whose name is enshrined in *Boyce-Codd Normal Form* (BCNF), described in Chapter 7 of the theory book. This paper actually said nothing about database management but it assumed the existence of relation-like things called tables and defined a notation for expressing queries on tables in the declarative manner that is expected for relational databases.

A prototype implementation of SEQUEL, named SEQUEL-XRM, was produced in 1974–75. Experience with this prototype led in 1976–77 to a revised version of the language, SEQUEL/2, subsequently renamed SQL for legal reasons. Work then began on System R, another, more ambitious prototype, implementing a subset of that revised version. A significant aim of System R was to gainsay those pundits who were sceptical about the possibility of providing a relational DBMS with performance comparable to that obtainable with conventional database technology at the time, and it was widely lauded as having succeeded in that aim. System R became operational in 1977 and was installed at a number of internal sites in IBM and at a few selected customer sites. Experiences with System R resulted in further changes to SQL and soon expectations were high that IBM would eventually develop commercial SQL products—so high that one vendor organization, now named Oracle Corporation, beat IBM to it in 1979 with the first release of ORACLE. IBM's own first SQL product, SQL/DS for the VSE environment, appeared in 1981 and implementations on other platforms followed during the 1980s. One of these was DB2 for OS/MVS in 1983, and much later DB2 became the generic name for all of IBM's implementations. A host of other vendors joined the game over the next two decades, including, to name but a few, Microsoft (with SQL Server, naughtily pronounced "sequel server"!), Sybase, Ingres, and Digital Equipment Corp. (with RDB, now owned by Oracle). Those last two, Ingres and RDB, originally used their own languages, both inspired by Codd's relational calculus, but later acquired an SQL interface.

The first edition of the international (ISO) standard for SQL appeared in 1987 as a ratification of one produced a year earlier by the Database Committee X3H2 of the American National Standards Institute (ANSI). Further development of the standard has taken place since then in ISO—or rather, under the auspices of a "joint technical committee", JTC 1, of ISO and IEC, responsible for all international IT standards. The editions are referred to as SQL:*yyyy,* where *yyyy* is the year of publication.

The so-called "referential integrity" feature, supporting primary keys and foreign keys, appeared in SQL:1989. SQL:1992 contained so many additional features that its language had been referred to at the time as SQL/2. In 1996 additional features known as SQL/PSM (a programming language for stored procedures and functions) and SQL/CLI (a "call level interface") were added. At the same time, the committee was working on further extensions to the language, resulting in the appearance of "SQL/3" as SQL:1999, whose most significant addition was the so-called "object-relational" feature supporting user-defined types. Further revisions appeared in 2003, 2007, and 2011. That last one, SQL:2011 (reference [15]), is notable for its modicum of support for "temporal database" management.

Many of the additional features appearing in post-1989 editions are specified as optional, meaning that they aren't required for an implementation to qualify as standard-conforming.

Historical notes appearing later in this book, concerning specific features of SQL, refer mostly to the pre-2000 editions of the ISO SQL standard: 1987, 1989, 1992, 1996, 1999, though in connection with the notation for queries I can sometimes take you right back to the 1974 SEQUEL paper.

Now, Sections 1.2 to 1.4 in the theory book describe some concepts that apply to databases in general and therefore equally well to relational databases and SQL databases. For that reason counterparts of those sections are omitted here and the next section is numbered 1.5 to maintain the parallels. Also, for a similar reason, the first two Figures in this chapter are number 1.2 and 1.4.

1.5 "Collection of Variables"

Figure 1.2 in the theory book puts a variable name on a table that is later confirmed as depicting a relation. The same figure serves here to put the same variable name on a picture in tabular form representing an SQL table. To speak of a table being depicted in tabular form isn't as silly as it may seem. There are other ways of visually representing the abstract concept of an SQL table, just as there are other ways of representing relations visually. In fact, it was the suitability of such pictures that led to the terms table, row, and column being advanced as alternatives for relation, tuple, and attribute back in the 1970s.

ENROLMENT

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

Figure 1.2: An SQL table variable, showing its current value.

1.6 What Is an SQL Database?

So, an SQL database is one whose symbols are organized into a collection of *tables*.

Now, Figure 1.2 shows an SQL table as the current value of an SQL variable, `ENROLMENT`, but in that connection there are two important points to be made right away. First, the term *table variable* isn't used in SQL for its counterpart of relation variables. Instead, base relvars are normally referred to as *base tables*, virtual relvars as *views* (or *viewed tables*) and those terms are used in the international standard. Secondly, not every SQL table can be the value of a base table or view. As we shall soon see, table variables and the tables assigned to them are subject to several deviations from relational theory; tables that cannot be assigned to a variable are subject to even more deviations. These deviations, and their consequences, give rise to much of the subject matter in this book.

The theory book's study of relational theory includes

- the "anatomy" of a relation
- **relational algebra:** a set of mathematical operators that operate on relations and yield relations as results
- **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 the book shows "how these, and other constructs, can form the basis of a **database language** (specifically, a *relational* database language)". To parallel the above structure, our study of SQL includes

- the "anatomy" of a table—showing similar components to those of relations but using different terminology
- SQL's operators that operate on tables and yield tables as results
- base tables: their creation and destruction, and operators for updating them
- various special constructs allowing consistency rules to be expressed as constraints on the base tables constituting the database

Historical Note

As already mentioned, during the 1970s the term "table" became widespread as a more intuitive replacement for the mathematical term "relation". For example, it was used in ISBL, whose developers are my dedicatees in the theory book, and also in Business System 12, the relational DBMS produced in 1982 for users of IBM's time-sharing services. Nowadays, however, as those earlier relational DBMSs have left the scene, the term tends to refer specifically to SQL tables and as these differ significantly from relations in several important respects it is perhaps better to revert to E.F. Codd's terminology for relations, as I do in the theory book.

1.7 "Table" Not Equal to "Relation"

The corresponding section in the theory book is titled **"Relation" Not Equal to "Table"**, where the word table was not intended to refer to the SQL construct in particular. Here it does refer to that construct.

The table shown in Figure 1.4 is a copy of its counterpart in the theory book, where it illustrates the point that two tables that differ only in the order of rows or the order of columns represent the same relation.

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

Figure 1.4: Not the same SQL table as Figure 1.2.

In standard SQL, the order of rows still carries no significance but the order of columns does carry significance, so Figures 1.2 and 1.4 do not in fact depict exactly the same SQL table. In the course of our study we will discover how the order of columns affects the SQL operators for operating on tables and those for operating on base tables. (Some SQL implementations support a somewhat unsound operator that delivers the table consisting of the first *n* rows of a given table. The user can specify an ordering for the rows but is not required to do so. The unsoundness arises even in the case where the user does specify an ordering, when there is a tie for any of the first *n* places. The order of the rows involved in the tie is then indeterminate and results are unpredictable.)

Also in the theory book, Figure 1.5 shows a table, copied here, that does not depict any relation. However, it does depict a valid SQL table.

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

Figure 1.5: A possible SQL table.

The single exercise for Chapter 1 of the theory book invited you to list the features seen in this table that mean it cannot possibly represent a relation. One of these—the appearance of two identical column headings—means that this SQL table cannot be the value of a base table, as we shall soon see. We shall also discover how it can happen that the same row appears more than once in an SQL table, like the first and last rows in Figure 1.5, and how something other than a value, or even nothing at all, might appear in a place where a value is expected, as depicted in the second and fourth rows.

1.8 Anatomy of a Table

Figure 1.6 shows the terminology used in SQL to refer to parts of the structure of a table.

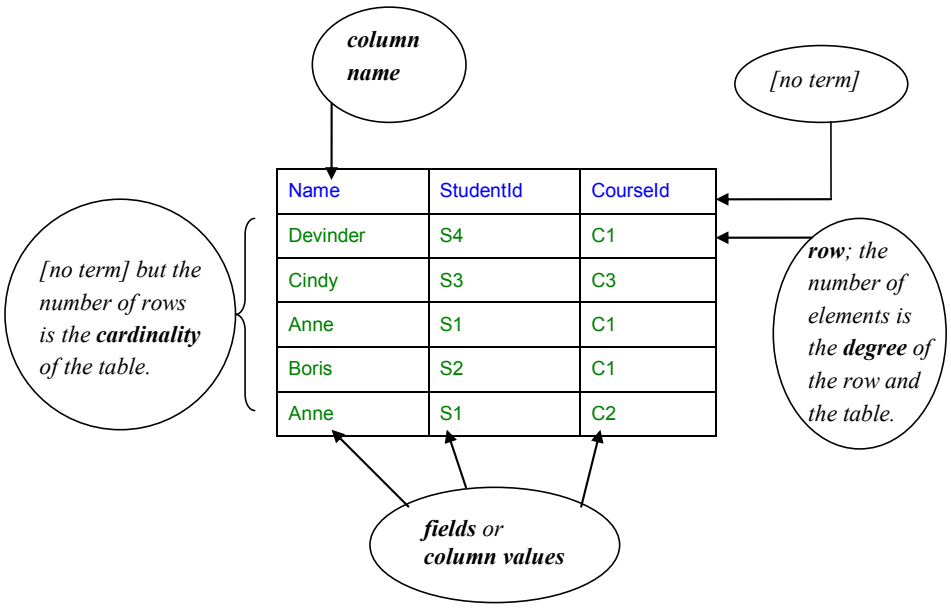

Figure 1.6: Anatomy of an SQL table.

As you can see, SQL has no official terms for its counterparts of the heading and body of a relation, but for convenience I do use those terms occasionally in the present book as applying to tables. SQL's counterpart of relational *attribute* is *column*. The figure shows the term *field* being used for SQL's counterpart of *attribute value* but please note that the international standard uses this term not only for a component of a row, as shown, but also for the corresponding component of the relevant *row type*.

1.9 What Is a DBMS?

The answer to this question is of course given in Section 1.9 of the theory book.

This book is concerned with SQL DBMSs and SQL databases in particular. Soon we will be looking at some of the features we might expect to find in an SQL DBMS—"might" because not all the standard features we describe are found in all SQL implementations.

1.10 SQL Is a Database Language

Section 1.10 in the theory book begins

> [The] commands given to a DBMS by an application are written in the database language of the DBMS. 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 unrecognized 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.

Those remarks apply to SQL, as does the remainder of that section. Moreover, Part 10 of the international standard for SQL defines classes for use in object-oriented languages, in particular, Java, thus allowing method invocations to be used in place of old-fashioned CALL statements.

1.11 What Does an SQL DBMS Do?

In response to requests from application programs, an SQL DBMS can, for example,

- create and destroy base tables
- take note of integrity rules expressed as *constraints*
- take note of *authorisations* (who is allowed to do what, to what)
- update variables (honouring constraints and authorisations)
- provide results of *queries*

—in other words, all of the things mentioned in the corresponding list of bullet points in Section 1.11 of the theory book.

In the remaining sections of this chapter you will see examples of how an SQL DBMS does these things (via commands—usually called statements—written in SQL, of course).

1.12 Creating and Destroying Base Tables

Example 1.1 shows an SQL command to create the base table counterpart of the `ENROLMENT` variable shown in Figure 1.2.

Example 1.1: Creating a base table.

```
CREATE TABLE ENROLMENT
            ( StudentId  SID ,
              Name       VARCHAR(30),
              CourseId   CID ,
              PRIMARY KEY ( StudentId, CourseId )
            ) ;
```

(Example 1.1 and the other examples in this chapter all end in semicolons. Actually, a semicolon is required in SQL only in scripts consisting of more than one statement. A script consisting of a single statement must not end in a semicolon.)

Explanation 1.1:

`CREATE TABLE` is SQL's counterpart of the key words **`VAR, BASE,`** and **`RELATION`** as used in **Tutorial D**.

`ENROLMENT` is the *table name* for the base table.

The text from the opening parenthesis to the end of the fourth line specifies *the declared type* of the table, meaning that every table ever assigned to `ENROLMENT` must be a table of that type.

The declared type of `ENROLMENT` is a *table type*, indicated by the key word `TABLE` and a comma-separated list (commalist, for short) of *column definitions*. A column definition consists of a column name followed by a type specification. Thus, each column of the table also has a declared type. Two or more columns can have the same type but *not* the same name. The type names `SID` and `CID` (for student ids and course ids) refer either to user-defined *types* or to user-defined *domains*. User-defined types and domains have to be defined by some user of the DBMS before they can be referred to. The type name `VARCHAR(30)` (character strings of length 30 or less), by contrast, is a *predefined* type: it is provided by the DBMS itself, is available to all users, and cannot be destroyed.

Chapter 2, "Values, Types, Variables, Operators", deals with types in more detail, and shows you how to define types and domains.

`PRIMARY KEY` indicates that the table is subject to a key constraint, in this case declaring that no two rows in the table assigned to `ENROLMENT` can ever have the same combination of `StudentId` and `CourseId` fields (i.e., we cannot enrol the same student on the same course more than once, so to speak). We will learn more about SQL constraints in general and SQL's key constraints in particular in Chapter 6.

Destruction of `ENROLMENT` is the simple matter shown in Example 1.2,

Example 1.2: Destroying a base table.

```
DROP TABLE ENROLMENT ;
```

After execution of this command the base table no longer exists and any attempt to reference it is in error.

Historical Note

The first SQL products, and indeed, the first edition (1987) of the international standard, had no support for key constraints. The `PRIMARY KEY` construct, along with the other features of the so-called "referential integrity" enhancements, appeared in SQL:1989. Because key constraints could not previously be expressed at all, by the time the feature was added there were plenty of existing base table definitions that lacked them and in such base tables it was even permitted for the same row to appear more than once (with what significance? you may well ask—good question!). As a consequence, the declaration of a key constraint had to be made optional in the interests of backwards compatibility. This is just one of many dubious features of SQL that arise from misjudgments made in the original language that have been impossible to correct for that reason.

1.13 Taking Note of Integrity Rules

For example, suppose the university has a rule to the effect that there can never be more than 20,000 enrolments altogether. Example 1.3 shows how to declare the corresponding constraint in standard SQL, but please note that not many SQL implementations actually support this optional feature at the time of writing (2012).

Example 1.3: Declaring an integrity constraint.

```
CREATE ASSERTION MAX_ENROLMENTS
    CHECK ( ( SELECT COUNT(*) FROM ENROLMENT ) <= 20000 ) ;
```

Explanation 1.3:

- **`ASSERTION`** is the key word indicating that a constraint is being declared.
- **`MAX_ENROLMENTS`** is the name of the constraint.
- **`( SELECT COUNT(*) FROM ENROLMENT )`** is an SQL expression yielding the cardinality of (number of rows in) the current value of ENROLMENT. Note that the enclosing parentheses are actually required: without them, the expression would denote a certain table rather than a numerical value. See Chapter 5, Section **5.3, Aggregate Operators,** for more details on this construct.
- **`( SELECT COUNT(*) FROM ENROLMENT ) <= 20000`** is a truth-valued expression—a *condition*, yielding *true* if the cardinality is less than or equal to 20000, otherwise yielding *false*. As we shall see later, not all conditions are such that they always yield either *true* or *false*—most of them aren't—but this is a special case that does happen to adhere to the usual rule of logic.
- Enclosing the condition in parentheses preceded by the key word `CHECK` is necessitated because there are other things that can be optionally specified in connection with a constraint.

The declaration tells the DBMS that the database is *inconsistent* if the condition of `MAX_ENROLMENTS` 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.

Example 1.4 shows how to retract a constraint that ceases to be applicable.

Example 1.4: Retracting an integrity constraint.

```
DROP ASSERTION MAX_ENROLMENTS ;
```

Historical Note

`CREATE ASSERTION` first appeared in SQL:1992 but it remains (in SQL:2011) an optional conformance feature and its uptake by SQL vendors has been negligible. With it, SQL's support for database integrity would be almost complete in the sense described in the theory book ("almost", because it still has no counterparts of those relations of degree zero, `TABLE_DEE` and `TABLE_DUM`); without it that cannot be so, as explained in Chapter 6.

1.14 Taking Note of Authorisations

As relational theory is silent on the issue of authorisation, it offers nothing with which SQL's vast edifice in support of what it calls *privileges* can be compared. Example 1.5 is a very simple case showing how the corresponding example in the theory book could be done in SQL.

Example 1.5: Granting privileges

```
GRANT SELECT, INSERT, DELETE, UPDATE ON ENROLMENT TO User9 ;
```

Explanation 1.5:

- **GRANT** is the key word indicating that privileges are being granted.
- **SELECT, INSERT, DELETE, UPDATE** identify operators that can be applied to a base table. The first, SELECT, is used here to indicate that permission to access the current value is being granted. The others are all update operators with very much the same effects as their counterparts in **Tutorial D**. If DELETE were omitted, for example, then User9 would not be allowed to use DELETE commands on ENROLMENT.
- **ON ENROLMENT** identifies the base table on which those privileges are being granted.
- **TO** is the key word required to precede the commalist of user names, officially termed *authorization identifiers*, denoting the users on whom the specified privileges are to be conferred.

So, our example actually grants four distinct privileges, all on the same base table. Example 1.6 shows how, for example, just two of these can be withdrawn (*revoked* is the official term):

Example 1.6: Revoking privileges

```
REVOKE DELETE, UPDATE ON ENROLMENT FROM User9 ;
```

Note that SQL does not use names for privileges, as suggested in the theory book's Example 1.5. If it did, then the syntax for *revoking* privileges might be simpler, but then we would need to provide a name for each of the four distinct privileges being granted in Example 1.5 and use two of those names in Example 1.6. The syntax for assigning names would surely be rather cumbersome—and consider that in real life the list of user names following TO might be very large.

These two simple examples hardly scratch the surface of the "vast edifice" I referred to. Furthermore, it is doubtful whether any university course in SQL would cover the whole of that edifice as defined in the international standard. Mercifully, we shall have nothing more to say about it in this book.

1.15 Updating Variables

For *assignment,* SQL uses the key word SET, as in SET X = X + 1 (read as "set X equal to X+1") rather than X := X + 1 as found in many computer languages.

When the target is a base table, direct assignment is not available in SQL. Only differential update operators such as the usual INSERT, DELETE, and UPDATE are available. Example 1.8 illustrates SQL's DELETE.

Example 1.8: Updating by deletion

```
DELETE FROM ENROLMENT WHERE StudentId = SID ( 'S4' ) ;
```

As you can see, this differs from Example 1.8 in the theory book only by the addition of the noise word FROM. The expression `SID ( 'S4' )` assumes the existence of a user-defined operator whose invocation here yields the value of user-defined type SID that represents the student id S4.

Next, in Example 1.9, we look at UPDATE.

Example 1.9: Updating by replacement

```
UPDATE ENROLMENT SET Name = 'Ann'
       WHERE StudentId = SID ( 'S1' ) ;
```

Note the use of SET, as already noted in connection with direct assignment to variables other than base tables. Otherwise the syntax is very similar to that used in **Tutorial D**, as are the semantics.

Finally, Example 1.10 illustrates the use of INSERT.

Example 1.10: Updating by insertion

```
INSERT INTO ENROLMENT
    VALUES ( SID ( 'S4' ) ,
             'Devinder' ,
             CID ( 'C1' ) ) ;
```

Here we see our first consequence of the difference between relations and SQL tables. The key word VALUES is SQL's counterpart of RELATION as used in relation selector invocations in **Tutorial D** (see Example 1.10 in the theory book). Most importantly, note that the expression denoting the operand of VALUES specifies fields (column values) without giving their column names, raising the question, how does SQL know which value goes into which column? The answer, as I'm sure you've guessed, is that the first value, `SID ( 'S4' )`, goes in the first column of ENROLMENT, the second in the second, and so on. That in turn raises the question, how do we know which is the first column of ENROLMENT, which the second, and so on, considering that in relational theory there is no ordering to the attributes of a relation? The answer, as I'm sure you've guessed again, lies in the order in which the column definitions are listed in the invocation of CREATE TABLE that brought ENROLMENT into existence (Example 1.1). However, SQL does make provision in case you remember the column names but forget the order: you could write the names in parentheses, `( StudentId, Name, CourseId )`, the order corresponding to that of the VALUES entries, after the table name ENROLMENT.

A few words are needed to say what happens when rows to be deleted or updated do not exist and rows to be inserted already exist.

Example 1.8 has no effect on the database in the case where the current value of ENROLMENT has no rows for student S4, but the DBMS might give a warning message.

Example 1.9 has no effect on the database in the case where the current value of ENROLMENT has no rows for student S1, but again the DBMS might give a warning message.

Regarding Example 1.10, in the case where the current value of ENROLMENT already contains the row `( SID ( 'S4' ) , 'Devinder' , CID ( 'C1' ) )`, the effect is to add another copy of that row anyway, unless to do so would cause some explicitly declared constraint (perhaps a key constraint) to be violated. By the way, the outer parentheses enclosing the three expressions denoting that row are required only when the row contains more than one field. Somewhat counterintuitively, the expression `VALUES 1, 2` denotes a table having two rows and just one column, as does `VALUES (1), (2)`, whereas `VALUES (1, 2)` denotes a table with one row and two columns.

Historical Notes

`INSERT, DELETE,` and UPDATE, including the use of `SET` *`name = expression`* for specifying column assignments, all appeared in the first SQL products. Support for local variables arrived in SQL:1996 with SQL's computationally complete programming language, SQL/PSM. It was natural, then, to use the same syntax for assignment to variables. Had the need for a programming language been realized at the outset, then perhaps a more usual syntax for assignment might have been adopted and copied in column assignments in UPDATE commands. That said, it can be noted that Basic uses similar syntax to SQL, replacing SET by LET.

The curious syntax for VALUES expressions is explained by the fact that they were originally restricted to just one row and could appear only as the source operand for INSERT. For imagined convenience, the parentheses surrounding the list of field expressions could be omitted when the row consisted of a single field. In SQL:1992 the syntax was extended to allow more than one row to be specified, using commas to separate the individual, possibly parenthesized, row expressions. However, that extension was specified as an optional conformance feature and it remains optional in SQL:2011. Use of a VALUES expression other than as an operand of INSERT is also an optional conformance feature.

1.16 Providing Results of Queries

Expressing queries in SQL is the (big) subject of Chapters 4 and 5. Here I present just a simple example to give you the flavour of things to come in those chapters. Example 1.11, as in the theory book, is a query expressing the question, who is enrolled on course C1?

Example 1.11: A query in SQL

```
SELECT DISTINCT StudentId, Name
FROM   ENROLMENT
WHERE  CourseId = CID ( 'C1' )
```

Note carefully that Example 1.11 is not a command (hence the absence of a semicolon). It is just an expression, denoting a value—in this case, a table. In SQL the result of a query is always another table. Figure 1.7 shows the result of Example 1.11 in the usual tabular form.

StudentId	Name
S1	Anne
S2	Boris
S4	Devinder

Figure 1.7: Result of query in Example 1.11.

Explanation 1.11:

- **`SELECT DISTINCT StudentId, Name`** specifies that the result of the `WHERE` invocation is to be projected over `StudentId` and `Name`.
- **`FROM ENROLMENT`** specifies the table operand for the invocation of `WHERE`.
- **`WHERE CourseId = CID ( 'C1' )`** specifies that just the rows for course C1 are required.

Historical Note

The syntax illustrated here is very similar to that defined for SEQUEL back in 1974. A table expression is required to begin with a `SELECT` invocation. That invocation or, to give its official term, *clause*, is followed by a sequence of clauses, each denoting a table that is the single table operand of the clause that follows it. The table denoted by the last clause is the table operand of the `SELECT` clause. The second clause must always appear and must always be a `FROM` clause. If a `WHERE` clause appears at all, it must be the third one. We shall meet other clauses, and the rules governing their possible appearance, later. These rules define the *structure* that inspired the name of the language when SEQUEL morphed into SQL, though in the title of the SEQUEL paper it referred to the ability to nest such expressions inside other such expressions. Officially the name is just SQL, not standing for anything—neither "structured query language" nor, as some would have it, "standard query language").