**CS60 Chapter 03. The Relational Database Model**

**Contents**

[Definitions—some old and some new 61](#_Toc269211652)

[Functions (operators) of relational algebra 64](#_Toc269211653)

[Restrict operator (a row picker) 65](#_Toc269211654)

[Project operator (a column picker) 66](#_Toc269211655)

[Union operator 68](#_Toc269211656)

[Intersection operator 69](#_Toc269211657)

[Difference operator 70](#_Toc269211658)

[Divide operator 71](#_Toc269211659)

[Another example of the divide operator 73](#_Toc269211660)

[Join operator 74](#_Toc269211661)

[Another example of a natural JOIN, equiJOIN, thetaJOIN, and outer join 77](#_Toc269211662)

[Data dictionary and the system catalog 79](#_Toc269211663)

[The M:N Relationship between two tables and recasting it as two 1:M relationships by using a bridge table 80](#_Toc269211664)

[Justifying some redundancy 85](#_Toc269211665)

[Indexes 86](#_Toc269211666)

[Key words 88](#_Toc269211667)

# Definitions—some old and some new

● A **database** stores raw data and metadata.

● A **database management system** (DBMS) is a set of utility programs that are the interface between the user(s) and the data. A DBMS allows a user to set up the structure of a database and manipulate the data. Types of manipulation include inserting new data, deleting data, updating data, and searching through the data to find and display data or print data. The DBMS may also include programs that allow users to build and use forms, reports, and other graphical interfaces to the data.

● An **entity** is a person, place, event, or thing for which we collect raw data.

● **Attributes** are the characteristics of an entity. They are the names of the columns.

● An **entity set** (also called a table or relation) is the series of rows, each with values for the attributes. A table has both raw data and metadata such as column names and datatypes.

● A **superkey** is an attribute (or combination of attributes) that uniquely identifies each entity in a table. A table may have several superkeys, and a superkey can have more attributes than is necessary to uniquely identify a row. If every row is unique, then certainly all columns could be a superkey.

● A **candidate key** is a superkey that does not have a subset of attributes that uniquely identifies each row. For example, if the candidate key has three columns, it cannot be reduced to two columns and still uniquely identify each row in the table.

As an example of candidate keys, suppose you have the following Employee table:

Employee table

**Employee\_ID**

Lastname

Firstname

Middlename

Social\_Security\_Number

Department

Office\_Phone

Clearly, Employee\_ID and Social\_Security\_Number would be unique, not null, and irreducible to a smaller number of columns that are unique, so each would be candidate key. Maybe the combination of last, first, and middle names would be unique, but not dependably so. If some other combination is unique for existing and “forseeable” data and cannot be reduced to a smaller set, then that would be a candidate key. But certainly the combination of Employee\_ID and Social\_Security\_Number would NOT be a candidate key because you could eliminate either one and still have a candidate key.

Think of candidate keys as candidates to be the primary key. Each is unique, not null, and irreducible to fewer columns if it has several columns. Some other rule (such as simplicity or the values being surrogate) helps to pick one of several candidate keys to be the primary.

● The **primary key** of a table is the unique and not null column(s) used to identify a row (an entity). It must be **unique** and **cannot be NULL**. **A table can have only one primary key**.

It can be one column or several columns. Of all the candidate keys (if a table has more than one candidate key), the primary key is the one selected to uniquely identify the row. If a table has only one candidate key, then it will be the primary key.

Primary keys should be relatively short and not include names such a Boulevard that you may recall as Blvd or Blvd. (with a period) or Avenue or Street. Some organization only use single-column primary keys with surrogate values (made-up values like Student\_IDs at Santa Monica College having up to seven decimal digits). URLs for websites and e-mail addresses may be unique and not null, but far too long and complicated to be practical primary keys.

● A **foreign key** is a column with **values limited to the set of values in the primary key that the foreign key references**, or a null value. A table may no foreign keys, one foreign key, or more than one foreign key.

A foreign key may have one or several columns, but the number of columns and their datatypes in the foreign key will match the number of columns and datatypes in the primary key it references. A one-column foreign key of a certain datatype will reference a one-column primary key with the same datatype. In a multi-column foreign key, the order of the columns matter. The order in the FK must be the same as the order in the PK.



Figure 3.1/Rob 2.1 Foreign keys link to primary keys for the purpose of linking tables in a relational database. Entity integrity and referential integrity are enforced by the DBMS so a link, when required, is possible. The name(s) of a foreign key is usually the same as the name(s) of the primary key in the table it references (points to). Each key here has one column, although multi-column (composite) keys are possible. Only one primary key exists in each table, but several foreign keys can exist in one table when necessary to link a table with several other tables. The center table here has two foreign keys. Each connects to one table.

● A **secondary key** is a column or columns used for data retrieval, but may not be unique. While a customer\_number could be a good primary key, the customer might not remember that number. Although the last name and city often would not be unique, they might serve to retrieve a few rows, from which one row could be selected.

A phone operator who retrieves a phone number when you call Information may ask for a name and city (and maybe also a first name and state). If the list is long, the operator may ask for a first name or a street until the list is shortened and one or a few phone numbers can be retrieved.

● Two attributes (column names) are **synonyms** if **different names are used for the same attribute**. For example, if an attribute is named *Student\_ID* in one table and the same attribute is named *Student\_Number or Student#*  or *StudentID* in another table, the attributes are synonyms. If you are referring to the same characteristic or attribute in several tables, then name it the same everywhere. Don’t use synonyms.

● Two attributes (column names) are **homonyms** if the **same names or similarly sounding names are used for different attributes**. For example, *Last\_Name* in two tables should not be used to refer to a customer's last name and a representative's last name. Homonyms also can be similarly sounding words; the following are homonyms and should not be used: *Last\_Name and LastName* or Student\_Number and Student\_Nmbr and Student\_No and Student#. Although they’re spelled differently, they sound alike if you read (or think) them, and they’re easily confused. A computer distinguishes between these names, but humans can confuse them in speech and thought. **If attributes are different, name them differently so they are neither spelled the same nor sound the same when pronounced. Don’t use homonyms.**

In summary, name the same attributes the same; name different attributes differently enough that they sound differently when you speak the names.

# Functions (operators) of relational algebra

The capability of an RDBMS to manipulate the database (e.g., select rows and columns to display, update values in the tables) is based on a field of mathematicscalled **relational algebra**. Relational algebra was originally used with sets, but E. F. Codd extended relational algebra to include tables (relations) in a relational database. The original relational algebra had to be extended to acknowledge that a table has both metadata and raw data. SQL is the language you will use to define tables, define relationships between tables, and manipulate the data. SQL is also the language behind the tools that graphically set up a database (e.g., using an Entity Relationship Diagram that some DBMSs can convert into SQL statements). SQL is also behind the Fifth Generational Languages used to interface databases by typing or speaking some query in simple English.

But behind SQL is relational algebra to carry out the steps involved. This section describes the operators of relational algebra.

**5th Generational Language**

**interfaces to databases**

**4th Generational Graphical**

**User Interfaces to databases**

**SQL**

****

**Database**

**of raw data and metadata**

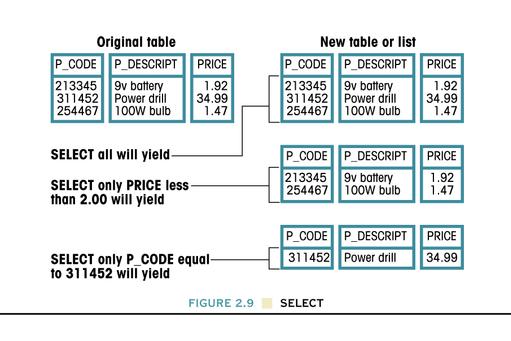
**Figure 3.2**  The functions (the operators) of relational algebra support the SQL statements that you code or the SQL statements that some 5GL or graphical user interface creates.

## Restrict operator (a row picker)

The restrict operator is also called the **select operator**, but it is not the same as the SELECT statement in SQL.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Figure 3.3** The Restrict operator acts on one table and returns another table with only those rows with column values that satisfy some condition. The condition might be that the Last\_Name begin with the letter ***R***. *Restrict* yields values for all attributes (all columns) found in a table. It yields a horizontal subset of a table. Along with the raw data in the yellow rows, the operator also returns column names, datatypes, and other metadata, indicated by the gray.



**Figure 3.4/**Rob Figure 2.9 **Three examples of the Restrict (or Select) operation.**

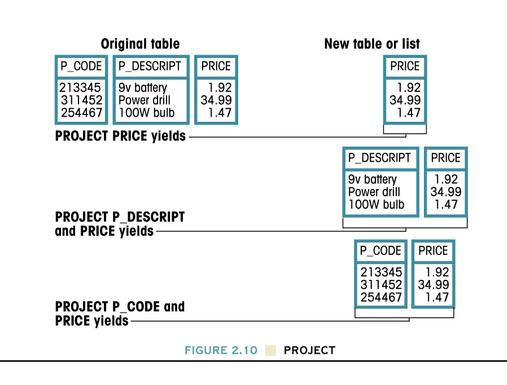
## Project operator (a column picker)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Figure 3.5** The Project operator returns all rows of the original table

but only those columns that are specified. *Project* produces a table

with all values for selected attributes. It yields a vertical subset of a table.



**Figure 3.6/Rob Figure 2.10** The Project operator acts on a table and yields a new table with

fewer columns (or in the limit, a duplicate of the original table).**Product (or Cartesian Product) operator**

|  |  |
| --- | --- |
| **Column\_**  **Letter** | **Column\_**  **Number** |
| A | 1 |
| A | 2 |
| B | 1 |
| B | 2 |
| C | 1 |
| C | 2 |

**yields**

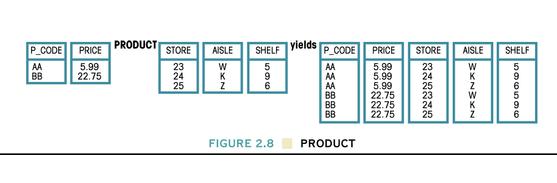
|  |
| --- |
| **Column\_**  **Number** |
| 1 |
| 2 |

###### PRODUCT

|  |
| --- |
| **Column\_**  **Letter** |
| A |
| B |
| C |

**Figure 3.7.** The Product operator acts on two tables and combines them into a new table.

The new table has all possible pairs of rows from the two tables.



**Figure 3.8/Rob Figure 2.8** The Product operator also can combine tables with more than one column each.The Product operator acts on two tables and yields a new table with all columns of both tables. The rows of the new table are composed of all possible combinations (all possible permutations) of each row in the first table with each row of the second table.

For example, the first row of the first table is combined with each row from the second table. Then the 2nd row of the first table is combined with all possible rows of the second table. This process continues until each row of the first table has been combined with each row of the second table.

If the first table has ***m***number of rows and the second table has ***n***number of rows, then the final table (the Cartesian product) has ***m x n***number of rows.

For example, if the1st table has 100,000 rows (or 105 rows)

and a 2nd table has 1,000,000 rows (or 106) rows,

then the product operation of these two tables yields 1011=100,000,000,000 (a hundred billion) rows. If each row has an average of 1000 bytes, then the product has 1014 bytes or 100 terabytes of data.Union operator

The result of a UNION operation with two tables is a table that has the rows in either of the two tables, without duplicates.

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |
| W |
| X |
| Y |
| Z |

|  |
| --- |
| **Column\_1** |
| W |
| X |
| Y |
| Z |

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

**yields**

###### UNION

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |
| W |
| X |
| Y |
| Z |

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |
| Z |

|  |
| --- |
| **Column\_1** |
| A |
| W |
| X |
| Y |
| Z |

**yields**

###### UNION

*Duplicates are eliminated*

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

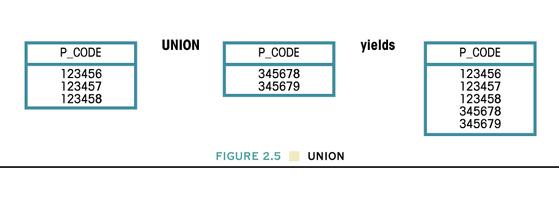
|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

**yields**

###### UNION

*Duplicates are eliminated*

**Figure 3.9** The Union operator combines all rows from two tables into a new table. Duplicate rows are eliminated. The two tables must be union compatible (i.e., the columns used in the union must have the same datatypes and the same domains—the same range of values).



**Figure 3.10/Rob Figure 2.5** The union of two one-column tables produces another table. Except that the data is longer, this example is like the first one above.

## Intersection operator

|  |
| --- |
| **Column\_1** |
| B |
| C |

###### INTERSECT

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

|  |
| --- |
| **Column\_1** |
| B |
| C |
| D |
| E |

**yields**

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

###### INTERSECT

**yields**

|  |
| --- |
| **Column\_1** |

|  |
| --- |
| **Column\_1** |
| W |
| X |
| Y |
| Z |

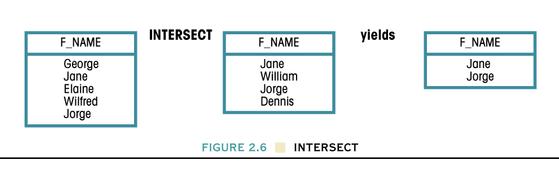
|  |
| --- |
| **Column\_1** |
| A |
| B |
| C |

###### INTERSECT

**yields**

*Null table*

**Figure 3.11 Intersection** produces a table that contains **only the rows that appear in both tables**. The two tables must be union compatible. The above examples formed the intersection with one column in each table, but intersections can involve several columns in each table, such as Last\_Name, First\_Name, Middle\_Name



**Figure 3.12/Rob Figure 2.6** The Intersect operation between two one-column table yields another table. Only *Jane* and *Jorge* appear in both of the original tables, so only those names appear in the result of the intersection. With longer names, this is similar to the first example above.

## Difference operator

|  |
| --- |
| Column\_1 |
| A |

|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

###### DIFFERENCE

|  |
| --- |
| Column\_1 |
| B |
| C |
| D |
| E |

**yields**

**yields**

**yields**

|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

###### DIFFERENCE

|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

|  |
| --- |
| Column\_1 |

*Null table*

|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

|  |
| --- |
| Column\_1 |
| W |
| X |
| Y |
| Z |

|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

###### DIFFERENCE

**Figure 3.13** Difference between two tables yields a new table with **all rows from the first table that are not in the second table.** The columns must be union compatible.

|  |
| --- |
| Column\_1 |
| B |
| C |
| D |
| E |

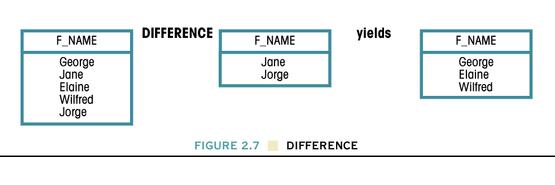
|  |
| --- |
| Column\_1 |
| A |
| B |
| C |

**yields**

###### DIFFERENCE

|  |
| --- |
| Column\_1 |
| D |
| E |

**Figure 3.14** The Difference operation depends on the order of the two original tables. Here I reversed the order of the two tables in the first (the top) and 4th examples above. The final tables differ in the two cases. The DIFFERENCE operation is not commutative, i.e. not independent of the order. However, both INTERSECTION and UNION are commutative.



**Figure 3.15/Rob Figure 2.7** The Difference Operation between two tables yields a table with values from the first table which are not in the second. Five names appear in the first table, but two names are excluded from the final table because they appear in the second table.

## Divide operator

Experience has shown that the divide operator first be studied as a mechanical procedure with the belief that it has practical value. Examples of its use are shown later.

###### Result\_Table

###### Table\_2

**Table\_1**

|  |
| --- |
| CODE |
| A |
| B |

|  |  |
| --- | --- |
| CODE | LOC |
| A | 5 |
| B | 5 |
| C | 6 |
| D | 7 |
| D | 8 |
| E | 8 |
| A | 9 |
| A | 4 |
| B | 3 |

|  |
| --- |
| LOC |
| 5 |

**yields**

###### DIVIDE

**Figure 3.16/Rob Figure 2.16** This example of DIVIDE uses a two-column table and a single-column table to produce a new table. Only the value 5 for column LOC is common to both the values A and B in CODE. At the bottom of Table\_1, the pair A,B appear again, but the corresponding values in column LOC differ (the values are 4 and 3).

###### Result\_Table

###### Table\_2

|  |
| --- |
| LOC |
| 5 |
| 6 |

|  |
| --- |
| CODE |
| A |
| B |

|  |  |
| --- | --- |
| CODE | LOC |
| A | 5 |
| B | 5 |
| C | 6 |
| D | 7 |
| D | 8 |
| E | 8 |
| A | 9 |
| A | 6 |
| B | 6 |

**Table\_1**

**yields**

###### DIVIDE

**Figure 3.17** The LOC value was modified so another pair of rows with CODE A, B have a common value of LOC, 6. This new value now appears in the Result\_Table.

**Figure 3.18** Since the order of rows in a relational database is arbitrary, the result of DIVIDE would be the same if the row order of Table\_1 were rearranged.

|  |  |
| --- | --- |
| CODE | LOC |
| A | 9 |
| A | 6 |
| A | 5 |
| B | 6 |
| B | 5 |
| C | 6 |
| D | 8 |
| D | 7 |
| E | 8 |

###### DIVIDE

**also**

**yields**

|  |
| --- |
| LOC |
| 6 |
| 5 |

|  |
| --- |
| CODE |
| A |
| B |

###### Table\_2

**Table\_1**

Example of using the divide operator, modified from Connolly and Begg[[1]](#footnote-2), page 96:

###### Result\_Table

Suppose a table has been stored that lists Renter\_IDs and Property\_IDs that the renters have viewed. The properties have one or more rooms, and a table has been created that lists the two properties (P2 and P3) with three rooms.

Table: **Property\_with\_3\_Rooms**

|  |
| --- |
| **Property\_ID** |
| P2 |
| P3 |

Table: **Renter\_views\_property**

|  |  |
| --- | --- |
| **Property\_ID** | **Renter\_ID** |
| P1 | R1 |
| P2 | R3 |
| P2 | R1 |
| P1 | R2 |
| P3 | R1 |

Which renters have viewed all properties with three rooms (the two, P2 and P3)? The answer can be found using the divide operator:

|  |
| --- |
| **Renter\_ID** |
| R1 |

**Renter\_views\_property** Divide **Property\_with\_3\_Rooms =**

The result of the divide operation shows that renter R1 has viewed all properties with three rooms. Renter R3 has viewed only one of the two properties with three rooms, and thus has not seen all of them. Renter R2 hasn't viewed either of those 3-room properties.

## Another example of the divide operator

What students have completed the three courses CS60, CS61, and CS65?

Table: Table: Result Table:

**Course\_completed\_by\_student Course Student**

|  |  |
| --- | --- |
| **Student\_ID** | **Course\_**  **Number** |
| 01 | CS60 |
| 01 | CS61 |
| 01 | CS65 |
| 02 | CS60 |
| 02 | CS61 |
| 02 | CS65 |
| 03 | CS60 |
| 04 | CS60 |
| 04 | CS61 |
| 05 | CS60 |
| 05 | CS61 |
| 05 | CS65 |

**Divide**

|  |
| --- |
| **Course\_**  **Number** |
| CS60 |
| CS61 |
| CS65 |

|  |
| --- |
| **Student\_ID** |
| 01 |
| 02 |
| 05 |

These students have taken the three courses CS60, CS61, and CS65.

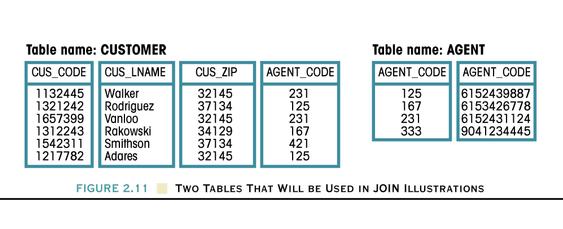
If the table on the left had millions of rows, you would wish to have the divide operator available to you!

These rows are nicely organized so you can see which students have taken the three courses, but since the row order and column order are arbitrary in a relational table, the rows could be shuffled and the two columns could be swapped and have no effect on the result table (even there the row order is arbitrary).

## Join operator

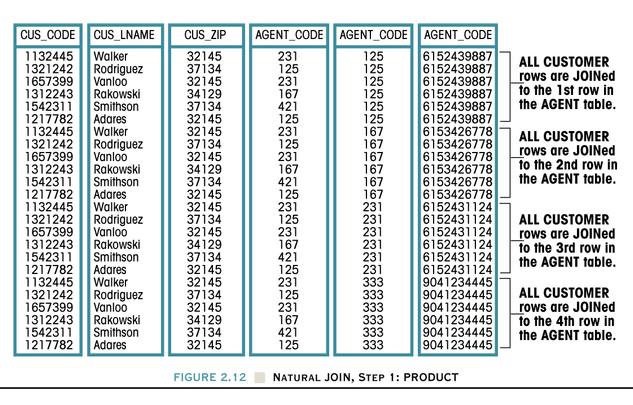
**Join** allows us to combine information from two or more tables. JOIN is a powerful and useful operator that allows you to retrieve table from several tables that have been linked together by foreign keys referencing primary keys.

The JOIN operation is composed of three other, more basic, operations: PRODUCT, SELECT, and PROJECT. Furthermore, several kinds of joins are possible.



**AGENT\_PHONE**

**Figure 3.19/Rob Figure 2.11** The original tables, CUSTOMER and AGENT, used in the examples of natural join and the outer join

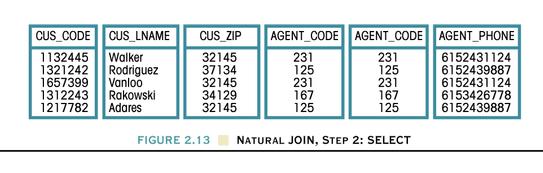


**AGENT\_PHONE**

###### Natural Join

**Figure 3.20/Rob Figure 2.12** Step 1 of a natural JOIN:

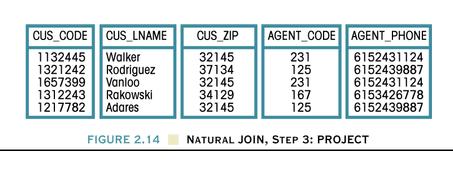
The PRODUCT operation combines Customer and Agent into a new table with all possible pairs of rows from the original two tables.



**Figure 3.21/Rob Figure 2.13** Step 2 of a **natural JOIN**:

The RESTRICT (SELECT) operation restricts the rows to those

with the same values in the AGENT\_CODE columns.



**Figure 3.22/Rob Figure 2.14** Step 3 of a **natural JOIN**:

The PROJECT operation eliminates the duplicate AGENT\_CODE column. The differences between a natural join and an equi-join is this last step to eliminate duplicate columns.

**EquiJOIN** links tables based on the same equality condition between columns as the natural join, except the EquiJOIN does not eliminate duplicate columns and the names of the columns that are matched need not be the same.

**Theta JOIN** is a join that compares specified columns of each table using any of the comparison operators (also called relational operators):

=

<

>

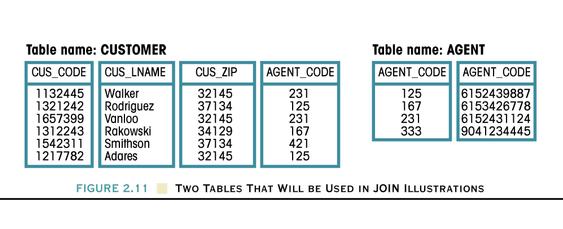
<=

>=

<>

Simple comparisons of two values, A and B, using these operators all take the form A θ B where θ is any one of the six relational operators (=, <, >, <=, >=, <>). Such a simple comparison is a theta comparison, and any join condition of the form **A θ B** is a theta join. When an equals sign is used, a theta-join becomes an equi-join.

If you look at the earlier Rob Figure 2.11 (from a couple pages back and repeated below), notice that AGENT\_CODE 421 in the Customer table does not appear in the Agent table, and AGENT\_CODE 333 in the Agent table does not appear in the Customer table. These are unmatched rows.

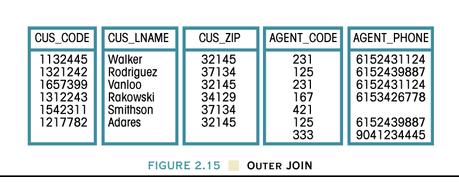


**AGENT\_PHONE**

**Figure 3.19/Rob Figure 2.11 (repeated)** Note that two values of AGENT\_CODE

are not matched in the other table.

In an **outer JOIN,** the unmatched pairs are retained and the values for the unmatched other tables are left blank or null. The result of a (full) outer JOIN is shown below.



**Figure 3.23/Rob Figure 2.15** The two tables of Figure 2.11 form this table

after an **outer join**. This is more precisely called a **full outer join** or **two-sided outer join** because the unmatched rows from both tables are kept.

## Another example of a natural JOIN, equiJOIN, thetaJOIN, and outer join

With the two tables below, we will now form three different joins. Except for simpler raw data and more readable column names, the database is the same as Rob Figure Q2.5. Notice that some of the PROFESSOR\_CODE in the left table are null (shown here as empty cells). Students 8 and 4 have no advisors. Since the professor with code 3 doesn’t appear in the Student table, that prof has no students to advise.

**Database name: CH2\_QUESTIONS**

**Table: STUDENT Table: PROFESSOR**

Stores student and student’s advisor (if any) Stores advisor and advisor’s department

|  |  |
| --- | --- |
| **PROFESSOR\_**  **CODE** | **DEPARTMENT\_**  **CODE** |
| 1 | 2 |
| 2 | 6 |
| 3 | 6 |
| 4 | 4 |

|  |  |
| --- | --- |
| **STUDENT\_**  **CODE** | **PROFESSOR\_**  **CODE** |
| 8 |  |
| 9 | 2 |
| 2 | 4 |
| 5 | 2 |
| 4 |  |
| 7 | 1 |

**Step 1 of a JOIN:**

Form the **PRODUCT** of the original tables

|  |  |  |  |
| --- | --- | --- | --- |
| **STUDENT\_**  **CODE** | **PROFESSOR\_**  **CODE** | **PROFESSOR\_CODE** | **DEPARTMENT\_**  **CODE** |
| 8 |  | 1 | 2 |
| 9 | 2 | 1 | 2 |
| 2 | 4 | 1 | 2 |
| 5 | 2 | 1 | 2 |
| 4 |  | 1 | 2 |
| 7 | **1** | **1** | 2 |
| 8 |  | 2 | 6 |
| 9 | **2** | **2** | 6 |
| 2 | 4 | 2 | 6 |
| 5 | **2** | **2** | 6 |
| 4 |  | 2 | 6 |
| 7 | 1 | 2 | 6 |
| 8 |  | 3 | 6 |
| 9 | 2 | 3 | 6 |
| 2 | 4 | 3 | 6 |
| 5 | 2 | 3 | 6 |
| 4 |  | 3 | 6 |
| 7 | 1 | 3 | 6 |
| 8 |  | 4 | 4 |
| 9 | 2 | 4 | 4 |
| 2 | **4** | **4** | 4 |
| 5 | 2 | 4 | 4 |
| 4 |  | 4 | 4 |
| 7 | 1 | 4 | 4 |

It's at this point where (1) the natural JOIN and the equiJOIN and (2) the thetaJOIN differ.

In the next step 2 that restricts what rows will appear

• for a natural join or equi-join, only rows are selected where these two columns are **equal**.

• for a theta-join, the condition could use any of the six relational conditions (=, <, >, <=, >=, <>) in comparing these two columns. If an = sign is selected, the theta-join reduces to an equi-join.

**Step 2 of a natural JOIN or equiJOIN:** Perform a SELECT to yield only the rows for which the value for the PROFESSOR\_CODE from the STUDENT table are matched by (equal to) that code from the PROFESSOR table.

|  |  |  |  |
| --- | --- | --- | --- |
| **STUDENT\_**  **CODE** | **PROFESSOR\_**  **CODE** | **PROFESSOR\_**  **CODE** | **DEPARTMENT\_**  **CODE** |
| 7 | 1 | 1 | 2 |
| 9 | 2 | 2 | 6 |
| 5 | 2 | 2 | 6 |
| 2 | 4 | 4 | 4 |

*Values in these columns are equal in either a natural join or equi-join.*

**from from**

**STUDENT PROFESSOR**

**Step 2 of a thetaJOIN:**

The condition used to restrict rows uses **any one** of the six comparison operators:

*Boolean operators:*

*= equal to*

*< less than*

*> greater than*

*<= less than or equal to*

*>= greater than or equal to*

*<> not equal to*

Sometimes in mathematics the operators =, <, >, <=, >=, and <> are referred to with the Greek letter θ (theta). Thus if any one of the six is used in the join, it is a theta-join. An equi-join uses the equals sign (=) and is thus a special case of a θ-join.

|  |  |  |  |
| --- | --- | --- | --- |
| **STUDENT\_**  **CODE** | **PROFESSOR\_**  **CODE** | **PROFESSOR\_CODE** | **DEPARTMENT\_**  **CODE** |
| 9 | 2 | 1 | 2 |
| 2 | 4 | 1 | 2 |
| 5 | 2 | 1 | 2 |
| 2 | 4 | 2 | 6 |
| 2 | 4 | 3 | 6 |

Suppose you restrict the rows to those where

**PROFESSOR\_CODE (from STUDENT)**

> (greater than)

**PROFESSOR\_CODE (from PROFESSOR)**

Note: While this shows how a θ-join differs from other joins, restricting rows based on PROFESSOR\_CODE does not have any meaning (Why would you use this type of join here?)

**Step 3 of a natural join:** Perform a PROJECT to eliminate duplicate columns for PROFESSOR\_CODE.

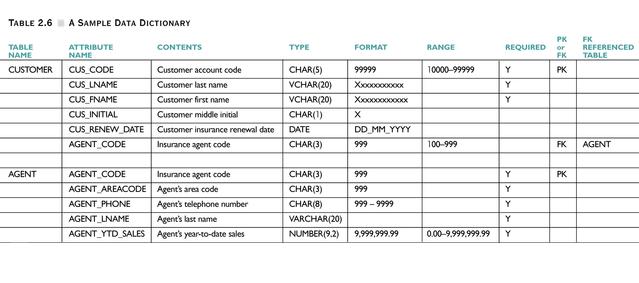
**Step 4 of a full outer join** includes all the rows of the inner join or equijoin but also the two students who have no advisor and the professor who advises no students.

|  |  |  |
| --- | --- | --- |
| **STUDENT\_**  **CODE** | **PROFESSOR\_**  **CODE** | **DEPARTMENT\_**  **CODE** |
| 7 | 1 | 2 |
| 9 | 2 | 6 |
| 5 | 2 | 6 |
| 2 | 4 | 4 |
| 8 |  |  |
| 4 |  |  |
|  | 3 | 6 |

# Data dictionary and the system catalog

**The data dictionary is a table within the database that lists metadata for all tables within the database.** Besides names, descriptive comments, and datatypes, a data dictionary could also store information about any constraints on columns. **Constraints** are any restrictions on the raw data such as (a) the unique and (b) NOT NULL constraints of a primary key, (c) restrictions on the values of foreign keys, unique constraints, NOT NULL constraints that require that a value be entered, and constraints that the value must be one of those listed in a set or lie in some range such as 50-100 (a type of constraint often coded as a CHECK constraint).

A data dictionary also can be used as a **planning tool** by a database designer, maybe handwritten or typed with a word processor or spreadsheet program. An example follows, but rather than being a system table in a relational database, it must be a report generated to display data or be a plan. You can tell because most values in the Table\_Name column are empty or null.

****

**System catalog is a very detailed system data dictionary that describes all objects within the database.**

Sometimes the data dictionary is itself not stored by the DBMS, but a larger set of metadata called the **system catalog—**described below**—**is stored, and the data dictionary can be extracted from it.

The **system catalog** is a detailed system-created database whose tables store the database characteristics. The **system catalog describes all objects in the database**: tables, indexes, virtual tables called views, stored programs written by the user to manipulate the data, and others. The **system catalog also stores** the object creator, authorized users of that object, and access privileges. Sometimes the system catalog **is** the data dictionary. If a DBMS has both a system catalog and a data dictionary, the system catalog will store more than shown above, and the data dictionary will be derived from it so the metadata isn't redundantly stored.

**The system catalog** can be queried just like any other table if you have sufficient privileges to look at this table or tables.

# The M:N Relationship between two tables and recasting it as two 1:M relationships by using a bridge table

The list below shows student names and the courses that each student takes. Each student takes three courses (and attends a particular section of each course) and each section of the course has two students. This is a very small college with only two students, three courses, and one section of each course. This is a many-to-many relationship between STUDENT and SECTION.

**STUDENT'S LAST NAME COURSE NAME AND NUMBER AND SECTION NUMBER**

Lee Database Concepts and Applications, CS60, section 1001

Analytic Geometry and Calculus, Math 7, section 2002

Technical Communication, ENGL 70, section 3003

Smith Database Concepts and Applications, CS60, section 1001

Analytic Geometry and Calculus, Math 7, section 2002

Technical Communication, ENGL 70, section 3003

This data and other details about the particular section of the course could be stored in two tables, but note in advance that the following two tables are misdesigned.

(1) a **STUDENT** table (with attributes of the student, including what section of each course the student is enrolled)

|  |  |  |
| --- | --- | --- |
| **STUDENT\_ID** | **STUDENT\_LASTNAME** | **SECTION\_**  **NUMBER** |
| 70001 | Lee | 1001 |
| 70001 | Lee | 2002 |
| 70001 | Lee | 3003 |
| 70002 | Smith | 1001 |
| 70002 | Smith | 2002 |
| 70002 | Smith | 3003 |

*Data here is repeated as many times as a student is enrolled in different sections.* A more complete table would have many more attributes (first name, middle name, street address, city, state, ZIP, major, etc.), and these other attributes would also be repeated many times. This table is not designed right.

(2) and a **SECTION** table (with attributes for each section of the course)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SECTION\_**  **NUMBER** | **STUDENT\_ID** | **COURSE\_**  **NUMBER** | **CLASS\_**  **MEETING\_TIME** | **CLASS\_**  **ROOM** | **PROFESSOR\_ID** |
| 1001 | 70001 | CS 60 | Monday 6:45p | BUS 207 | 10000 |
| 1001 | 70002 | CS 60 | Monday 6:45p | BUS 207 | 10000 |
| 2002 | 70001 | MATH 7 | Tuesday 1:00p | MATH 32 | 20000 |
| 2002 | 70002 | MATH 7 | Tuesday 1:00p | MATH 32 | 20000 |
| 3003 | 70001 | ENGL 70 | Wednesday 6:45p | T 204 | 30000 |
| 3003 | 70002 | ENGL 70 | Wednesday 6:45p | T 204 | 30000 |

The data here is repeated as many times as the section has students enrolled. A section with 45 students would repeat this data 45 times. Other columns would store the course title (such as Database Concepts and Applications) and the catalog description of the course. This table is not designed right.

A many-to-many relationship exists between STUDENT and SECTION (each student can be enrolled in many sections and each section has many students enrolled. The Entity Relationship Diagram (ERD) would appear as:

**STUDENT**

**SECTION**

**(of one course)**

**takes**

**M**

**N**

**Figure 3.24** The E-R diagram for the two tables with their M:N relationship.

This two-table database has three problems:

(1) The Class\_Meeting\_Time is not atomic (not composed of a single piece of data, but rather is composed of meeting day such as Monday (or days such as Monday, Wednesday) and a period of time (6:45p-9:50p)).

(2) The tables have many repeated entries, not all of which would be redundant. The redundancies

● take more time, labor, and computers to enter and change

● can lead to inconsistent data (the insert, delete, and update anomalies we've talked about before)

● increase the amount of data stored on disk and in RAM during data manipulations, and

● slow down data retrieval and other manipulations of the data.

(3) A many-to-many relationship exists. A many-to-many relationship between tables is not "wrong" in the conceptual or logical model (a stage in designing a database). To be implemented in a relational database, it needs to be converted into one-to-many relationships.

The many-to-many relationship can be converted into two one-to-many relationships with the aid of a new **bridge table** and some reorganization of the STUDENT and SECTION tables. A bridge table is also called a **composite table** or a **linking table**. In an ERD, a bridge table is shown as a diamond within a rectangle.

**1**

**M**

**1**

**M**

**SECTION**

(of one course)

**STUDENT**

**ENROLL**

*Will be changed*

*Will be changed*

*Bridge table*

**Figure 3.25** The revised E-R diagram composed of a bridge table, ENROLL, with

two 1:M relationships. One foreign key in ENROLL links to the primary key in STUDENT. Another foreign key in ENROLL connects to the primary key in SECTION. In an Entity-Relationship Diagram, a bridge table is shown as a rectangle with a diamond inside.

Furthermore, the STUDENT and SECTION tables are altered. These three tables are shown below.

**STUDENT** table lists attributes of each student once

|  |  |
| --- | --- |
| **STUDENT\_ID** | **STUDENT\_LASTNAME** |
| 70001 | Lee |
| 70002 | Smith |

**Primary key**

**ENROLL** bridge table lists all student IDs and the sections that they're enrolled in.

|  |  |
| --- | --- |
| **STUDENT\_ID** | **SECTION\_**  **NUMBER** |
| 70001 | 1001 |
| 70001 | 2002 |
| 70001 | 3003 |
| 70002 | 1001 |
| 70002 | 2002 |
| 70002 | 3003 |

**Foreign key 1 Foreign key 2**

**Bridge table**

**Primary key** of ENROLL table includes both columns. This is one primary key with two columns. The combination of values in the two columns must be unique, and no values in those columns can be null.

**SECTION** table lists attributes of each section of a course

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SECTION\_**  **NUMBER** | **COURSE\_**  **NUMBER** | **CLASS\_**  **MEETING\_TIME** | **CLASS\_**  **ROOM** | **PROFESSOR\_ID** |
| 1001 | CS 60 |  | BUS 207 | 10000 |
| 2002 | MATH 7 |  | MATH 32 | 20000 |
| 3003 | ENGL 70 |  | T 204 | 30000 |

**Primary Foreign key**

Where should the course title (such as Database Concepts and Applications) and course description appear? If included in the SECTION table, they would be repeated for each section if a course had several sections. They also would depend only on the COURSE\_NUMBER. If a course can have multiple sections, include those attributes in a **COURSE** table:

|  |  |  |  |
| --- | --- | --- | --- |
| **COURSE\_**  **NUMBER** | **COURSE\_NAME** | **COURSE\_**  **CREDIT** | **COURSE\_**  **DESCRIPTION** |
| CS 60 | Database Concepts and Applications | 3 | This course ... |
| MATH 7 | Analytic Geometry and Calculus | 5 | Topics include ... |
| ENGL 70 | Technical Communication | 3 | This course ... |

**Primary**

**COURSE**

**1**

**M**

**SECTION**

**(of one course)**

**STUDENT**

**M**

**1**

**1**

**M**

**ENROLL**

*Bridge table*

**Figure 3.26** An **Entity Relationship Diagram** (ERD) that also connects

the SECTION table to the COURSE table. A 1:M relationship exists between the COURSE and SECTION tables (one course can have many sections)

1

|  |
| --- |
| **STUDENT** |
| **STUDENT\_ID** |
| STUDENT\_LAST\_NAME |
| STUDENT\_FIRST\_NAME |
| STUDENT\_STREET |
| STUDENT\_CITY |
| STUDENT\_STATE |
| STUDENT\_ZIP |
| STUDENT\_DOB |
| STUDENT\_PHONE |
| DEPARTMENT\_CODE |
| PROFESSOR\_ID |

∞

1

|  |
| --- |
| **COURSE** |
| **COURSE\_NUMBER** |
| COURSE\_NAME |
| COURSE\_CREDIT |
| COURSE\_DESCRIPTION |

|  |
| --- |
| **ENROLL** |
| **STUDENT\_ID** |
| **SECTION\_NUMBER** |

1

|  |
| --- |
| **SECTION** |
| **SECTION\_NUMBER** |
| COURSE\_NUMBER |
| CLASS\_MEETING\_TIME |
| PROFESSOR\_ID |

∞

∞

**Figure 3.27** The **relational schema** that corresponds to the Entity Relational Diagram in Figure 2.26. In each table of a relational schema, the table name is first listed. The primary key (one or more columns) is bolded. A line connects each foreign key to the primary key that it references. If a foreign key is composite (has two or more columns), then a line connects each part of the foreign key to its corresponding part of the primary key. Like the ERD, a relational schema displays entities (tables), the attributes (column names) in each table, and the connectivity (whether 1:1, 1:M, or M:N) except that a relational schema uses *infinity* (∞)for *many*.

At least one feature remains that deserves criticism: the CLASS\_MEETING\_TIME column in the Section table. We need to store the classday(s) and either (a) beginning classtime and ending classtime, or (b) a beginning time and a duration. What columns would you set up?

# Justifying some redundancy

This database illustrates a case when data is redundant at one instant of time, but not redundant over a longer time. It also shows how to record an invoice that lists several things that are being recorded on one invoice. The database has four tables:

Table: **CUSTOMER**

Primary key: **CUS\_CODE**

Foreign key: **none**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CUS\_CODE** | **CUS\_LNAME** | **CUS\_FNAME** | **CUS\_INITIAL** | **CUS\_AREACODE** | **CUS\_PHONE** |
| 10 | Ramas |  |  |  |  |
| 11 | Dunne |  |  |  |  |
| 12 | Smith |  |  |  |  |
| 13 | Olowski |  |  |  |  |
| 14 | Orlando |  |  |  |  |
| 15 | O'Brian |  |  |  |  |

Table: **INVOICE**

Primary key: **INV\_NUMBER**

Foreign key: **CUS\_CODE references CUS\_CODE in Customer**

|  |  |  |
| --- | --- | --- |
| **INV\_NUMBER** | CUS\_CODE | INV\_DATE |
| 1001 | 14 | 01-JAN-2006 |
| 1002 | 11 | 01-JAN-2007 |
| 1003 | 12 | 01-JAN-2008 |
| 1004 | 11 | 16-MAR-2009 |

Table: **LINE** Note: This table stores individual lines numbers of items on an invoice

Primary key: **INV\_NUMBER+LINE\_NUMBER** (A two-column primary key)

Foreign keys: **INV\_NUMBER** and **PROD\_CODE** reference different tables

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INV\_NUMBER** | **LINE\_NUMBER** | **PROD\_CODE** | **LINE\_UNITS** | **LINE\_PRICE** |
| 1001 | 1 | BBB-BBBBB | 1 | 189.99 |
| 1001 | 2 | DDD-DDDDD | 3 | 2.99 |
| 1002 | 1 | CCC-CCCCC | 2 | 18.63 |
| 1003 | 1 | EEE-EEEEE | 1 | 6.79 |
| 1003 | 2 | DDD-DDDDD | 1 | 2.99 |
| 1003 | 3 | AAA-AAAAA | 1 | 12.95 |
| 1004 | 1 | AAA-AAAAA | 1 | 12.95 |
| 1004 | 2 | DDD-DDDDD | 2 | 2.99 |
| 1004 | 3 | BBB-BBBBB | 1 | 169.99 |

*Copied from PRODUCT table (below) to maintain historical accuracy. The PROD\_PRICE in the table below may change. Note the different names: LINE\_PRICE and PROD\_PRICE.*

*Perhaps auto-generated. Maintains a consistent order.*

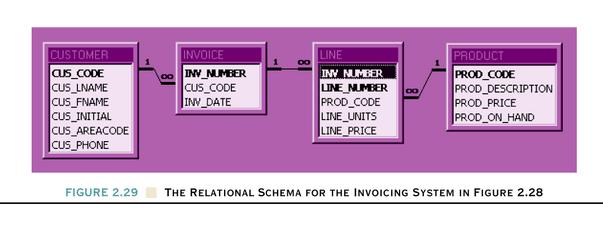
Table:  **PRODUCT**

Primary key: **PROD\_CODE**

Foreign key: **none**

|  |  |  |  |
| --- | --- | --- | --- |
| **PROD\_CODE** | **PROD\_DESCRIPTION** | **PROD\_PRICE** | **PROD\_ON\_HAND** |
| AAA-AAAAA | Claw hammer | 12.95 | 23 |
| BBB-BBBBB | Houselite chain saw, 16-in. bar | 169.99 | 4 |
| CCC-CCCCC | Sledge hammer, 16-lb. head | 18.63 | 6 |
| DDD-DDDDD | Rat-tail file | 2.99 | 15 |
| EEE-EEEEE | Steel tape, 12-ft. length | 6.79 | 8 |

The price for the Houselite chain saw is now $169.99, but on January 1, 2006 it was sold for $189.99 on invoice 1001.



**Rob Figure 2.29** The Relational Schema for the Invoicing System shown above

A similar problem could occur at Santa Monica College (or any school) with course numbers. As the faculty recognize the need for new courses, they are proposed and (sometimes) approved by the respective department and the curriculum committee. Some courses are offered for decades, but may become obsolete. After many years of disuse, a course number may be used again. To avoid confusion, a student’s records would need to reflect the course name and credit hours when the student enrolled, as well as the course number (like CS60). This is analogous to the earlier example: the price for an item be the price when the product was purchased, not the current price.

# Indexes

The arrangement of rows stored in a relational database has no special significance. However, we may want to display or print a list of names in alphabetical order. An **index** can be thought of as a special "table" (actually a special object) set up to improve performance (e.g., shorten the time to query a database) in which the order of entries is significant. However to think that a balanced-tree index, a common index in relational databases, is some sorted, truncated table is an oversimplification.

While an index maintains information in alphabetical or numerical order (and maybe in reverse order such as Z to A), an index does not redundantly store the table it indexes. Once the index is created on a table, the DBMS automatically updates the index to reflect any changes to that table. An index is an **object** different from a table, although sometimes the table IS the index. To sound confusing, databases such as SQL Server, Oracle, and other DBMSs have indexes that replace a table and include all columns of the original table.

Suppose for simplicity you have a table with 12 rows with columns Record\_Number, Last\_Name, and Salary:

###### Index for the Table Index for the Table

###### based on the Last\_Name, based on the Salary,

###### created, stored, revised created, stored, revised

Structure to navigate the index fast

Structure to navigate the index fast

**Employee table**

|  |  |
| --- | --- |
| **Salary** | **Record\_**  **Number** |
| 1000 | 5 |
| 2000 | 11 |
| 3000 | 8 |
| 4000 | 1 |
| 5000 | 2 |
| 6000 | 3 |
| 7000 | 10 |
| 8000 | 7 |
| 9000 | 12 |
| 10000 | 4 |
| 11000 | 9 |
| 12000 | 6 |

|  |  |  |
| --- | --- | --- |
| Record\_Number | Last\_Name | Salary |
| 1 | K | 4000 |
| 2 | F | 5000 |
| 3 | I | 6000 |
| 4 | B | 10000 |
| 5 | J | 1000 |
| 6 | E | 12000 |
| 7 | L | 8000 |
| 8 | A | 3000 |
| 9 | H | 11000 |
| 10 | C | 7000 |
| 11 | D | 2000 |
| 12 | G | 9000 |

|  |  |
| --- | --- |
| Last\_Name | Record\_Number |
| A | 8 |
| B | 4 |
| C | 10 |
| D | 11 |
| E | 6 |
| F | 2 |
| G | 12 |
| H | 9 |
| I | 3 |
| J | 5 |
| K | 1 |
| L | 7 |

*Index key*

*Pointers to the Table*

*Pointers to the Table*

*Index key*

These indexes look very much like a sorted table with the indexed column(s) and pointers. But a **B-tree index,** commonly used with databases, is more than merely a sorted table. A B-tree index looks like an upside-down tree (or the truck of a tree with its roots) as a structure that the DBMS can search through quickly. The database management system includes software to use that structure.

Other indexes:

● a **library's catalog** (computerized or on cards) indexed by the book’s Dewey Decimal Number, title, topic, and author. These are separate indexes. The book is stored systematically by the Dewey Decimal Number in the bookshelves where you can find it. When searching for the book by its title or author or topic, those indexes will speed retrieval of the data.

● the **index of a book** where you look up a topic alphabetically and find the page number. In the Table of Contents, you look up a page number or Chapter number and find the title of the corresponding section.

● **an index** of advertisers in a magazine. In an alphabetized list, you look up the name of an advertiser and find the page where their ad appears in the magazine.

● Rob and Coronel's example (modified) of a table of paintings with the corresponding painter. Rather than searching through the entire column of Painter\_Number in the Painting Table to get a list of all paintings by one painter, create an index by Painter\_Number and Painting\_Number. The paintings by one painter are grouped together and already sorted.

###### Painting Table Index

|  |  |  |
| --- | --- | --- |
| **Painter\_**  **Number** | **Painting\_**  **Number** | Pointers |
| 123 | 1 |  |
| 123 | 2 |  |
| 123 | 4 |  |
| 126 | 3 |  |
| 126 | 5 |  |

|  |  |
| --- | --- |
| **Painting\_**  **Number** | **Painter\_**  **Number** |
| 1 | 123 |
| 2 | 123 |
| 3 | 126 |
| 4 | 123 |
| 5 | 126 |

*Indexed columns*

*Pointers to the Painting Table*

Components of an Index

Primary keys and other columns with unique values are often automatically indexed by the DBMS.

When the values of the indexed column are changed or new values inserted, the DBMS automatically changes the index if necessary. Maintaining an index requires computer resources. Unnecessary indexing places an unnecessary load on the CPU. If you have 50 indexes on a table and you insert one row into the table, the table **and** the 50 indexes will all be changed. While the maintenance on the indexes is automatic, it still absorbs computer resources.

Indexing is an important tool for the DBA (Database Administrator) to "tune" the database for optimum performance. A book or course on **performance tuning** describes indexing in detail.

In summary, a relational DBMS uses indexes to improve performance (reduce time to access the data or sort the data). One column of a table can be indexed (e.g., Last\_Name) or several columns can form one index (e.g., Last\_Name, First\_Name, Middle\_Name). A table can have several indexes, but normally an index does not include all columns of a table. Instead, certain columns are indexed in ascending or descending order and stored along with a pointer that allows you to access or retrieve the non-indexed columns. That arrangement is stored and used by the DBMS wherever possible rather than repeatedly sorting the same data or searching completely through an unsorted list to find a name, number, or other value.

# Key words

attributes

database

data dictionary or system catalog

database management system (DBMS)

entity

entity set

homonyms and synonyms

index

keys

superkey

candidate key

primary key

foreign key

secondary key

operators of relational algebra

restrict

project

product (or Cartesian Product)

union

intersection

difference

divide

join

natural join

outer join

equi-join

theta join

relationship M:N recast as two 1:M relationships with a bridge table

1. Reference: Thomas Connolly and Carolyn Begg, Database Systems—A Practical Approach to Design, Implementation, and Management, 2nd Edition, Addison Wesley (1999). [↑](#footnote-ref-2)