An Introduction to the DB Relational Database Management System

Ward, J.R.

IIASA Professional Paper
February 1982

An Introduction To The DB Relational Database Management System

J. Robert Ward

February 1982
PP-82-1

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
A-2361 Laxenburg, Austria.

ABSTRACT

This paper is an introductory guide to using the Db programs to maintain and query a relational database on the UNIX† operating system.

In the past decade, increasing interest has been shown in the development of relational database management systems. Db is an attempt to incorporate a flexible and powerful relational database system within the user environment presented by the UNIX operating system.

The family of Db programs is useful for maintaining a database of information that is updated infrequently. A retrieval command to Db is phrased in a language based on the Relational Algebra.

These programs are written in portable C and are currently implemented on a PDP-11/70 and a VAX-11/780‡.

†UNIX is a Trademark of Bell Laboratories.
‡PDP and VAX are Trademarks of Digital Equipment Corporation.

Contents

Users requiring only a basic knowledge of the Db programs may omit the sections marked with a O.

1. Introduction

This paper is an introduction to using the Db programs on a UNIX operating system. It assumes the user has a basic familiarity with UNIX but not necessarily with any database system. Many of the principles of relational databases may be absorbed through a process of "osmosis" while reading this paper and by learning to use Db. A further introduction to the basics of relational database systems is to be found in Sandberg (1981).

This paper is written for newcomers to database systems, as well as those having a specialised knowledge in this field. The Db system is a generalised family of programs that attempts to fulfill many differing requirements. This document is a complete guide to these programs and includes many details that are not necessary to a beginner. Users needing only a basic knowledge may omit the sections marked with a O.

Volume 1 of the IIASA edition of the UNIX Programmer's Manual contains a summary of the Db programs.

1.1. Database Management Systems

The data within a database is maintained exclusively by a set of programs termed a *Database Management System* (DBMS). These programs act as an interface between the user and the data. They ensure that the data is kept in a compact and consistent format, and allow the user to ask a wide range of questions about the data.

A DBMS can be described by the view of the data it presents to the user. For instance, in a hierarchic database, the data is structured in the form of a tree, similar to the structure of a UNIX file system.

The Db programs deal with relational databases. In a relational database, the data is structured as a set of two-dimensional tables. This approach tends to simplify the user's concept of how the data needs to be stored or accessed : the idea of storing data within a table is intuitive. Db can combine such tables together, or restrict output to be a small part of one table.

1.2. The Db Programs

The family of Db programs provides a general tool for maintaining relational databases on the UNIX operating system. This family is comprised of a few individual programs that allow a user to create, maintain and query a relational database. In addition a library of Access Method procedures exists that enables users to write their own specialised programs for data retrieval and manipulation.

Db does not provide any special user environment in which to access a database. For instance, it has no special protection facilities of its own, and there are no commands, say, to delete a relation. Instead it is assumed that these facilities already exist within the framework of UNIX.

Db is useful for maintaining data that in a sense remains static. Although Db allows users to update or replace information, it would be of little use in a transaction system - an airline ticket booking database for instance - where the updates are extremely frequent and far outnumber the queries to the system.

1.3. Terminology

Relational database theory has its own special vocabulary. An explanation of some common phrases is given here.

1.3.1. Query

A *query* is a small command or program given to a database system instructing it how to manipulate some data from a database.

Db has its own language in which a query must be presented. This language has many ideas borrowed from the implementation language C.

1.3.2. Relation

A relational database consists of a set of two-dimensional tables termed *relations*. All the data in the database is contained entirely within such tables. Each relation in the database has a unique name so that it can be identified.

A Db relation is contained entirely within a single binary file. These relation files have a special format that can be read only by the Db programs.

Example. An example of a relation is shown here. It relates a few English and German nouns and includes the corresponding definite articles.

english	article	german
Danube	die	Donau
book	das	Buch
bridge	die	Bruecke
cat	die	Katze
cow	die	Kuh
cupboard	der	Schrank
dog	der	Hund
girl	das	Maedchen
head	der	Kopf
horse	das	Pferd
house	das	Haus
human being	der	Mensch
tree	der	Baum
woman	die	Frau

1.3.3. Domain

A column of data from such a relation is called a *domain*. Thus a domain represents a "vertical slice" of a relation. Each domain in a relation has a unique name. A domain is constrained to hold one particular type of data. For instance, a domain might hold four byte floating point numbers but not any other data type.

Example. Using the example of the above relation, the domain **german** is shown here -

german
Donau
Buch
Bruecke
Katze
Kuh
Schrank
Hund
Maedchen
Kopf
Pferd
Haus
Mensch
Baum
Frau

1.3.4. Attribute

An *attribute* refers to a property of a particular domain, for instance its name. Another attribute is the type of data stored within that domain.

In the Db system, a domain may hold any of the fundamental data types found in the C language as well as character string values. On the VAX implementation, the Db programs also support packed decimal values.

1.3.5. Tuple

A single row from a relation is termed a *tuple*. A tuple represents a "horizontal slice" of a relation.

There is never any information implied by the order of tuples in a relation. The Db programs take advantage of this fact and, generally, add new tuples to a relation at whichever point they consider most appropriate. As a result, Db may print a relation in a seemingly random order, unless instructed to do otherwise.

For reasons beyond the scope of this guide, a relation is usually structured so that each tuple is unique. However Db does not enforce this rule strictly. This issue is discussed in Date (1977).

Example. An example of a single tuple from the relation above is shown here -

english	article	german
cupboard	der	Schrank

1.3.6. Field

A single atomic item of data is termed a *field*. A field is represented by the intersection of a specific tuple with a specific domain.

Example. Here is an example of a single field from the above relation -

Katze

1.3.7. Cardinality And Degree

The number of tuples in a relation is termed its *cardinality*. The number of domains is termed its *degree*.

Example. The sample relation shown above has a cardinality of fourteen and a degree of three.

1.3.8. Key

Usually a relation will be formed so that the fields from one or more domains are unique. Any tuple could then be uniquely identified by specifying a single field from this domain. Such a field is said to be a *key* and the corresponding domain is said to be a *key domain*.

Example. In the sample relation, such a key domain could be **english.** In this domain the fields are all different and so specifying an English noun would determine an entire tuple. In the same way, domain **german** could also be a key domain. The domain **article,** however, would not be a suitable key domain since its fields take on only one of three possible values and are not unique.

1.4. The Various Programs

Dbcreate reads ascii data and converts it to the special file format required by the Db programs. This is generally the first step in setting up a database.

Dbappend is used to place additional ascii data into a relation.

Dbls gives the user information about the data stored in a database. It is the Db equivalent of the standard UNIX *ls* command.

Db is the main query processor. Db will list data from a database, create new items of data, or append data to an existing database. Finally, Db will combine or transform existing data from a database.

Dbedx invokes a screen editor, allowing the user to edit a relation interactively.

Dbmodify "cleans up" a relation. It frees unused disc space and generally tidies up a relation.

1.5. Constructing A Database

Learning how to use a database system and learning how to construct an actual database are two different problems. This paper addresses the former problem, that is, how to use the Db programs to manage a database. When setting up a database, there are various principles to be followed in deciding how to exploit any dependencies present in the data. A good introduction to databases in general, and this problem in particular, is found in Date (1977).

1.6. Current Implementations

Db is currently implemented on a PDP-11/70 and a VAX-11/780, both running under the UNIX operating system (Ritchie/Thompson 1974). It is written almost entirely in portable C (Kernighan/Ritchie 1978), with the exception of approximately thirty lines of assembler code in the VAX version. Moreover the query language implemented for Db has a syntax strongly reminiscent of C.

1.7. Access Methods Library

There is a library of routines that one may use to access data from a database. These procedures can be called from a C or Fortran program. They allow the user to write his own specialised programs to retrieve and process data. A full description of these routines is in the IIASA edition of the UNIX Programmer's

Manual. These pages also describe how Db relations are formatted internally.

1.8. Examples Used In This Paper

This paper includes examples of Db queries and the output that they produce when applied to a very simple database. Most examples refer to a small hypothetical database that might be used in a book lending library. This library system is composed of three relations **books**, **loans** and **people** that are introduced in sections 2.5, 5.5.4 and 5.7.4.

2. Dbcreate - Creating A New Relation From Ascii Data

The Dbcreate program is used to convert ascii data into the particular file format known to the Db programs. Dbcreate is generally the first step in setting up a database.

2.1. Arguments To Dbcreate

Dbcreate reads ascii data from the standard input and forms a new relation file containing this data. The first argument to Dbcreate is the name of the new relation.

The remaining arguments specify the names of the domains and their data types. Each remaining argument consists of a name, followed by a slash (/), followed by a single character specifying the type of data for that domain. Here is a list of the available type characters and their corresponding data types. (This table is a subset of the one shown in appendix I.)

Type Character	**Data Type**
c	Char Integer
s or i	Short Integer
u	Unsigned Short Integer
l	Long Integer
f	Floating Point
d	Double Precision Floating Point
S	Character String
p	15 Digit Packed Decimal
P	31 Digit Packed Decimal

2.2. Input Data To Dbcreate

The input data to Dbcreate is assumed to be in free format with each input tuple on a separate line. All fields within a tuple are usually delimited by white space ; that is, by spaces or new-line characters.

If Dbcreate encounters invalid data - for instance it might possibly read some alphabetic characters when it expected a number - it will complain and skip the rest of the input line. It will then continue to read the rest of the input data.

It is possible to delimit input fields with characters other than white space. This is most useful when Dbcreate is formatting a relation containing string fields. Since, by default, Dbcreate considers a space to mean the end of a field, this makes it impossible to create a relation containing string fields consisting of several words. The solution is to delimit each item of data in the input by some character other than a space. For instance, the fields could be surrounded by slashes or periods. Use the **-df** option to inform Dbcreate of another delimiting character.

Even the last field in a tuple must be terminated by the delimiting character. If Dbcreate reads a field without a delimiter, it complains and skips to the next line of input data.

Special characters, such as a line-feed or even the current delimiting character itself, can be incorporated into a string field using the backslash convention employed by both the C language and Db (see section 5.4.5).

It is perfectly acceptable to have null string fields. However, Dbcreate does not permit null numeric fields. Each numeric field must be given an explicit value in the input data.

When reading input data from a terminal, Dbcreate prompts for each new tuple with a ">". The input data is then terminated by a control-D on a line by itself. When reading from a file, Dbcreate reads until the end of the file.

Example. The C Shell command

```
dbcreate books bookno/s author/S title/S < infile
```

creates a new relation in a file called **books.** This new relation has three domains. The first is called **bookno** and holds *short* integer values. The other two domains are called **title** and **author** respectively. These hold character *string* values.

Some suitable input data, read from the file **infile**, might be as follows -

```
1 Austen Persuasion
2 Shaw Pygmalion
3 Zola Nana
4 Austen Emma
```

Example. If, however, it is necessary to include titles of more than one word in the relation, the following command and data are appropriate. Here the **-df/** option in the argument list informs Dbcreate to expect each input field to be delimited by a slash. Note that even the last field on each line is delimited.

```
dbcreate books bookno/s author/S title/S -df/
```

```
1/Austen/Persuasion/
2/Shaw/Pygmalion/
3/Zola/Nana/
4/Austen/Emma/
5/Austen/Pride And Prejudice/
6/Hardy/Tess Of The D'Urbervilles/
7/Hardy/Jude The Obscure/
8/Hardy/Far From The Madding Crowd/
9/Eliot/The Mill On The Floss/
10/Eliot/Silas Marner/
11/Hardy/The Mayor Of Casterbridge/
```

The new relation **books** may now be viewed by typing the command

```
db books
```

(In general, the command **db relation** will list the entire contents of **relation** to the standard output.) The following output will appear on the terminal -

```
=========================================
bookno  author  title
-----------------------------------------
     1  Austen  Persuasion
     2  Shaw    Pygmalion
     3  Zola    Nana
     4  Austen  Emma
     5  Austen  Pride And Prejudice
     6  Hardy   Tess Of The D'Urbervilles
     7  Hardy   Jude The Obscure
     8  Hardy   Far From The Madding Crowd
     9  Eliot   The Mill On The Floss
    10  Eliot   Silas Marner
    11  Hardy   The Mayor Of Casterbridge
=========================================
```

2.3. The Various Data Types

When creating a large relation it is wise to choose an appropriate data type for each domain. The Db programs recognise several types of numeric data so that accurate information can be preserved without wasting unnecessary disc space. A table summarising the available data types is shown in appendix I.

The Db programs support four different data types that can hold integral numbers, and two data types for floating point numbers.

Character string fields are stored sensibly and do not take up more space than is needed for each field. Any character, with the exception of a null (binary zero) byte can be stored.

On the VAX-11/780 implementation, the Db programs also accomodate packed decimal values. These have no equivalent in C, nor in Fortran. They permit large integer numbers to be stored away in a relation. Two ranges of packed decimal values are used by the Db programs. One range holds numbers to a 15 digit accuracy and the other to 31 digit accuracy.

2.4. Restrictions On Domain Names

Domain names can be composed of any sequence of characters but they are limited to a length of twenty. Each domain within a single relation must have a unique name.

2.5. Dbcreate Sorts Data Into Order

Unless told otherwise, Dbcreate sorts the input data into order before creating a new relation. By default, a new relation is sorted in ascending order of the first domain.

It also removes duplicate tuples from the new relation. (Two tuples are considered to be duplicates if their corresponding fields have identical values.) The number of tuples actually placed into the new relation will be reported if one uses the **-p** option to Dbcreate (see section 8.6).

Example. The command

dbcreate books author/S bookno/s title/S -df /

with the data

```
Austen/1/Persuasion/
Shaw/2/Pygmalion/
Zola/3/Nana/
Austen/4/Emma/
Austen/5/Pride And Prejudice/
Austen/5/Pride And Prejudice/
Austen/5/Pride And Prejudice/
Hardy/6/Tess Of The D'Urbervilles/
Hardy/7/Jude The Obscure/
Hardy/8/Far From The Madding Crowd/
Hardy/8/Far From The Madding Crowd/
Hardy/8/Far From The Madding Crowd/
Eliot/9/The Mill On The Floss/
Eliot/10/Silas Marner/
Hardy/11/The Mayor Of Casterbridge/
```

would produce the following output when listed by the command -

db books

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Shaw	2	Pygmalion
Zola	3	Nana

Notice that Dbcreate has sorted the relation by the first domain, in this case by **author.** The tuples "Pride And Prejudice" and "Far From The Madding Crowd" that appeared more than once in the input data, now occur only once in the new relation.

2.6. Keeping Duplicate Tuples

It is possible to prevent the removal of duplicate tuples if necessary by a **-k,** for "keep", option in the argument list to Dbcreate.

Furthermore, the original order of the input data may be preserved in the new relation by the **-heap** option to Dbcreate. This will speed up the action of Dbcreate since the tuples no longer have to be sorted. As a side effect this flag implies that duplicate tuples are to be kept ; that is, the **-heap** option automatically sets the **-k** option. As may be expected, there are good reasons why Dbcreate should sort the input data into order. These are discussed below (see section 7.3).

2.7. Creating One Relation Similar To Another ○

Once a relation exists, it is possible to create a new relation with identical attributes without respecifying the domain names and data types. The **-like** option to Dbcreate indicates that a new relation should be created "like" an old one. Only the domain names and their associated data types are copied from the original relation to the new relation. The actual data is not copied.

Example. The command

dbcreate newbooks -like books -df#

creates a new relation **newbooks** with the same domains and domain types as those of the relation **books.** The delimiter **-df** option must still be specified to Dbcreate if required. In this example, input fields are to be delimited by sharp signs. Data is read from the standard input.

2.8. Specifying The Maximum Length Of String Domains ○

Dbcreate has a facility which allows a limited form of data checking. It may be instructed to reject character string fields that exceed a certain length. One may specify an integer after an **/S** in the list of domain names and types given to Dbcreate. Any input field longer than this value causes Dbcreate to complain and reject the input tuple.

This maximum length is remembered in the relation file and may be subsequently copied to another new relation by the **-like** option. This length restriction will also apply when appending new tuples to the relation by the programs Dbappend and Dbedx.

Example.
dbcreate payroll surname/S20 address/S40 comments/S salary/I

This creates a new relation **payroll**. The domains **surname** and **address** are constrained to hold strings no greater than twenty and forty characters respectively. There is no restriction on the length of fields in domain **comments**. (Actually this is not strictly true since no tuple can exceed a certain maximum size. This limit, though, is fairly generous. See section 8.1 below.)

2.9. Miscellaneous Options To Dbcreate

The **-p** option in an argument list makes Dbcreate report the number of tuples placed into the new relation. This figure may be less than the number of input data tuples if some of the input tuples are duplicates.

The **-s** option forces Dbcreate to output a relation to the standard output. There must still be a relation name in the argument list, although it need not be the name of the file where the output is finally sent. This option is useful for sending a new relation through a UNIX pipe to another Db program for further processing. Thus, Dbcreate may be used as a filter in a chain of piped processes. Its purpose would then be to convert ascii data to a relational format readable by the Db programs.

The **-s** option suppresses any sorting or removal of duplicate tuples. That is, the **-s** option also sets the **-heap** and **-k** options.

Example. The C Shell command

dbcreate stdout -like payroll -s -p > outfile

creates a new relation in **outfile.** This new relation has similar attributes to a relation called **payroll.**

Example. The command

dbcreate stdout -s name/S20 initial/S2 age/u | db ......

reads data from the standard input and converts it to Db format. The output from Dbcreate is piped to Db for further processing.

3. Dbappend - Adding More Ascii Data To A Relation

The program Dbappend is used to append more tuples to a relation. Dbappend reads ascii data from the standard input, just as Dbcreate does, and adds the tuples to the relation.

The input data for Dbappend has the same format as it does for Dbcreate. The delimiting character for each input field may be specified by a **-df** option.

Example. The command

```
dbappend books -df+
```

with the input data

```
Lawrence+12+Women In Love+
Lawrence+13+The Virgin And The Gypsy+
Lawrence+14+Sons And Lovers+
Hemingway+15+For Whom The Bell Tolls+
```

places these new tuples into **books**. The entire relation may now be listed by the command

```
db books
```

producing the following output -

```
author        bookno  title
Austen             1  Persuasion
Austen             4  Emma
Austen             5  Pride And Prejudice
Eliot              9  The Mill On The Floss
Eliot             10  Silas Marner
Hardy              6  Tess Of The D'Urbervilles
Hardy              7  Jude The Obscure
Hardy              8  Far From The Madding Crowd
Hardy             11  The Mayor Of Casterbridge
Shaw               2  Pygmalion
Zola               3  Nana
Lawrence          12  Women In Love
Lawrence          13  The Virgin And The Gypsy
Lawrence          14  Sons And Lovers
Hemingway         15  For Whom The Bell Tolls
```

This relation will serve as the basis for many future examples.

3.1. Appending Duplicate Tuples

Dbappend normally refuses to append a tuple if it is a duplicate of some tuple already present in the relation. (This does not apply if the relation was orginally created with the **-heap** option of Dbcreate.) By specifying a **-k** option in the argument list, Dbappend will insert new tuples regardless of whether they are already present in the relation. This speeds up Dbappend, since it no longer needs to check for duplicated tuples.

As shown by the example above, placing new tuples into a relation destroys the original sorting order. In practice, the order is not disrupted excessively and new tuples are inserted close to where they should be according to the sorting order. If it is really necessary to restore the order of sorted tuples, use the program Dbmodify (see section 7.1).

4. Dbls - Listing Information About Relations

The program Dbls is used to list information about a set of relations. It is the Db equivalent of the standard UNIX *ls* program. Its arguments are the names of relations to be examined. Arguments may also be the names of directories in the file system hierarchy : Dbls then examines the files under the named directories. If there are no arguments Dbls examines every file in the current directory.

If the first argument is a **-l** or **-v** option, then Dbls becomes "verbose" and lists even more information than it would do usually.

By default, Dbls complains if it finds a file that is not a Db relation or a file that cannot be opened. For instance, one might not have the required permissions to read a relation file. These complaints may be suppressed by a **-w** option. Thus, the command

dbls -w

lists information concerning any relations found in the current directory. It would ignore any files that are not relations or that could not be examined because of permission restrictions.

Example. The command

dbls books

lists basic information about **books**. The output might be -

```
        Mode   Tuples   Deg   Pages   Domains
books   Sort       15     3    1+ 0   author/S bookno/s title/S
```

The output reports the number of tuples, the degree of the relation, as well as the names and types of each domain (described by the same type character as for Dbcreate). A Mode of **Sort** indicates that the relation has been sorted. The page numbers are explained below (see section 7.6).

Example. The command

dbls -v

lists all information about relations in the current directory. For the relation **books** the output might be -

```
       Mode Tuples Deg Pages Fix Max  Fl Modified            Size
books  Sort     15   3 1+ 0   12  50 VU Dec  1 11:37 1981    2048
               Domain Ty Key  Fl Print Offs Fix Max Smallest Largest
               author  S   1 VA     9    2        9
               bookno  s   2  A     6    6    2           1      15
                title  S   3 VA    26    8       26
```

In addition to the basic information, Dbls reports the modification date of the relation and the size of the file in bytes. Dbls lists the name, data type and the smallest and largest valued fields currently held in each domain. (This information is not kept for string or packed decimal domains.)

The flags **V** and **U** in the output indicate that the tuples are of variable size and that the relation has been updated since it was created or cleaned up by Dbmodify. Flag **V** in the attributes information says that those fields are of variable length. Flag **A** indicates that the domain is sorted on ascending values. A **D** flag would indicate that the domain is sorted on descending values.

Other values are printed mainly for system debugging purposes and indicate how a relation is structured internally.

5. Db - Retrieving Data From A Database

Db is used to retrieve data from a set of relation files. The user presents a query command to Db, informing it how existing relations are to be combined, or how new data is to be calculated from the information in a database. Db can also create new relations or append tuples to existing ones.

Any output produced by Db is itself in a tabular or relational format. This also applies to any intermediate results. A temporary result may become the input to another command. That is, an input relation can be processed in one way and the result passed on to a later command for further processing.

A query is formulated in a very high-level language. Some of the commands bear a superficial resemblance to the familiar arithmetic operators. However, they do not deal with single numbers but instead instruct Db to manipulate whole relations of data.

5.1. Presenting Commands To Db

There are three ways in which a query can be presented to Db. For a short, simple query the easiest way is to type the query in the argument list to Db, that is, on the same line as the **db** command itself.

This method has the advantage that the C Shell will remember previous commands (if its **history** variable is set) and allows one to repeat or make small changes to a Db query. It has the disadvantage that it tends to be clumsy to use for more complicated queries. Moreover, many characters used in Db queries have a special meaning to the C Shell and so, in general, the command to Db must be surrounded by single quotes. The various functions of the C Shell and how it interprets special characters are described in Joy (1980).

The second way is to let Db read commands from the standard input. This is usually a terminal, although the input may be redirected from a file by the C Shell. If reading from a terminal, the command input is terminated by a control-D on a line by itself. Reading commands from the standard input has the advantage that a permanent record of the query can be kept in a file. Another advantage is that the C Shell cannot impose its own interpretation on various characters from the input.

The third way is to use the **-F** option to Db. This option is followed immediately by the name of a file from which the commands are read. This option is useful if relational data is being piped to Db. In this case, Db would be reading data from the standard input and so could not read commands from the standard input at the same time. Only one **-F** option is allowed.

Whether Db is reading its commands from a terminal, from a file or from an argument list, the input is always in free format. White space is sometimes necessary to separate adjacent words. Apart from this, spaces and new-lines can be inserted anywhere.

Example. The C Shell command

```
db payroll
```

prints the entire contents of the relation **payroll.**

Example. This command

```
db < query
```

makes Db read its commands from the file **query.**

Example. The command

```
db
```

by itself causes Db to wait for commands to be typed on the terminal. Input is terminated by a control-D.

Example. The command

db -Fcommands.db

makes Db read from the file **commands.db.**

Example. Finally, the command

dbcreate stdout -like payroll -s | db -Fcommands.db

invokes both the Dbcreate and Db programs simultaneously. The data produced by Dbcreate becomes the input to Db. The data fed to Db is processed according to the query commands in the file **commands.db**

5.2. Listing A Relation

As demonstrated above, simply typing the name of a relation by itself causes all its data to be printed. The usual output is preceded by a header showing the domain names. The listed fields are normally justified, that is, they are formatted properly.

By default, the headers are repeated once per screen when output is going to a terminal. They are repeated every sixty lines when output is sent to a file, so that the file can be printed in a reasonable format on a line-printer. By using the **-h** option of Db, the frequency of these headers may be changed, or the headers can be suppressed altogether. If an option **-h20,** say, is present in the argument list to Db, the headers are repeated once every twenty lines in the output. If there is no number after this option, the headers are not listed at all.

The **-df** option can also be used with Db, just as with Dbcreate or Dbappend. When given to Db, this option specifies a delimiting character which is listed after each field of output. The default character is a space. This option is useful if the output is to be further processed by programs such as the standard UNIX utilities, *awk* or *sed*.

There is also a **-dt** option that can be used to specify a character to be output after each complete tuple. The default is a new-line character. Use of delimiter characters other than the defaults will switch off justification.

Example. The command

db books

prints the entire contents of the relation **books** to the standard output. By default, the output includes headers, and the printed fields are justified. Assuming **books** to be the same example of a relation used above, the output is -

```
author      bookno  title
Austen           1  Persuasion
Austen           4  Emma
Austen           5  Pride And Prejudice
Eliot            9  The Mill On The Floss
Eliot           10  Silas Marner
Hardy            6  Tess Of The D'Urbervilles
Hardy            7  Jude The Obscure
Hardy            8  Far From The Madding Crowd
Hardy           11  The Mayor Of Casterbridge
Shaw             2  Pygmalion
Zola             3  Nana
Lawrence        12  Women In Love
Lawrence        13  The Virgin And The Gypsy
Lawrence        14  Sons And Lovers
Hemingway       15  For Whom The Bell Tolls
```

Example. This command -

```
db -h -df@ books
```

produces -

```
Austen@1@Persuasion@
Austen@4@Emma@
Austen@5@Pride And Prejudice@
Eliot@9@The Mill On The Floss@
Eliot@10@Silas Marner@
Hardy@6@Tess Of The D'Urbervilles@
Hardy@7@Jude The Obscure@
Hardy@8@Far From The Madding Crowd@
Hardy@11@The Mayor Of Casterbridge@
Shaw@2@Pygmalion@
Zola@3@Nana@
Lawrence@12@Women In Love@
Lawrence@13@The Virgin And The Gypsy@
Lawrence@14@Sons And Lovers@
Hemingway@15@For Whom The Bell Tolls@
```

5.3. Processing A Single Relation

The operators introduced below all act upon a single relation. The *selection* and *projection* operators restrict output to the desired information. For instance, one can specify precisely which tuples are to be listed from a large relation, or which domains are to be printed.

The *sort* operators are useful to order tuples in a listed output.

5.3.1. Selection - Retrieving Specific Tuples

The selection operator (::) tells Db to select only certain tuples from a relation.

One can regard the :: operator as meaning "such that" or "where". Its main purpose is to reduce or narrow down the range of printed tuples. This selection operator is used to form a horizontal subset of a relation.

The name of a relation from which tuples are to be extracted appears on the left hand side of this operator. On the right hand side is an expression. Only tuples from the relation which match the expression or qualification on the right hand side are selected.

The qualification on the right hand side of the :: operator can be any general scalar expression, that is, an expression which can eventually be calculated and reduced to a single value. The syntax of the qualification is based on the syntax of the C language itself. The following paragraphs describe how the selection operator works.

A name appearing in the expression on the right hand side is taken to refer to a domain from the source relation. By specifying domain names, fields can be compared to specific values or with one another, and a selection made on this basis.

Integer and floating point numbers are allowed and indeed are generally needed. (Db also recognises octal or hexadecimal integers should they be required. Octal integers start with **0**, **0o** or **0O** and hexadecimal integers commence with **0x** or **0X**.) Character string constants are permitted : they are surrounded by double quotes.

Domain names and constants are combined with the following operators - **+, -, *, /, %, ==, !=, <, <=, >, >=, &&, ||** and **!**. These operators have direct equivalents in the C language. A summary of these operators is in appendix II. There are others, used to manipulate bit fields, that are discussed later (see section 5.4.7). A qualification may also call upon any of the functions incorporated into Db (see section 5.5).

For every tuple from the source relation, Db evaluates the qualification to a single number. If this value is not zero, the tuple is printed. Otherwise the tuple is discarded. In this respect the selection operator has a strong similarity to the **if** (.....) or **while** (.....) statements of the C language.

Example. The query command

books :: bookno == 9

produces the following output when given to Db -

```
author   bookno   title
Eliot         9   The Mill On The Floss
```

The English equivalent of this query is "List tuples from relation **books** such that fields from domain **bookno** have the value 9". In this example there is only one such tuple.

Example. The query

books :: bookno <= 5

produces the following output when given to Db -

```
author   bookno   title
Austen        1   Persuasion
Austen        4   Emma
Austen        5   Pride And Prejudice
Shaw          2   Pygmalion
Zola          3   Nana
```

In this case five tuples are listed. The English version of this query is "List tuples from relation **books** where fields from domain **bookno** are less than or equal to 5".

Note that if this query were presented to Db on the command line and not from a file or the terminal, then it would have to be enclosed in single quotes. Otherwise the C Shell would interpret the command to mean that Db should read input from a file named = , which is not the intended meaning. For example, this C Shell command is suitable -

db 'books :: bookno <= 5'

Another way of presenting this query to Db would be to escape the < with a backslash -

db books :: bookno \<= 5

Example. The query

books :: author == "Austen"

produces this output -

```
author   bookno   title
Austen        1   Persuasion
Austen        4   Emma
Austen        5   Pride And Prejudice
```

In this example we have asked for "tuples where the **author** field is 'Austen'".

Note the double quotes around the word "Austen". Without the quotes, Db would interpret the query as meaning "List tuples where the **author** field has the same value as the **Austen** field." It would then complain because there is no domain named **Austen** in this relation.

Example. The query

books :: author == title

produces the output -

```
author   bookno   title
```

Db prints the headings, but because there are no tuples with identical **author** and **title** fields, none are printed.

Example. The following query lists all tuples where the **author** fields are alphabetically less than 'Hardy'.

books :: author < "Hardy"

The output would be -

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner

This example shows how an inequality can be used to compare string fields. The output consists of those tuples where the **author** field contains names up to, but not including, 'Hardy'.

Example. The query

books :: bookno + 3 > 10

produces the output -

author	bookno	title
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls

In this example, tuples having a **bookno** value greater than seven are listed.

Example. Finally, the query

books :: author == "Austen" || author == "Zola"

lists those tuples where the **author** field is either 'Austen' or 'Zola'. The output would be -

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Zola	3	Nana

5.3.2. Projection - Specifying Certain Domains

The *projection* operator (**%%**) requests Db to list only certain columns from a relation.

The projection operator forms a vertical subset of a relation, in the same way that a selection operator forms a horizontal subset. Only those domains that the user requests are listed after a projection.

The name of the source relation from which data is read appears on the left hand side of the projection operator. On the right side, the user specifies a list of those domains in which he is interested. The domains are printed in the same order in which their names appear in the list.

Example. The Db query

books %% title author

lists only the domains **title** and **author** from relation **books.** The output is -

```
title                           author
Persuasion                      Austen
Emma                            Austen
Pride And Prejudice             Austen
The Mill On The Floss           Eliot
Silas Marner                    Eliot
Tess Of The D'Urbervilles       Hardy
Jude The Obscure                Hardy
Far From The Madding Crowd      Hardy
The Mayor Of Casterbridge       Hardy
Pygmalion                       Shaw
Nana                            Zola
Women In Love                   Lawrence
The Virgin And The Gypsy        Lawrence
Sons And Lovers                 Lawrence
For Whom The Bell Tolls         Hemingway
```

Example. The Db query

books %% author

lists only the **author** domain from the relation. The output is -

```
author
Austen
Austen
Austen
Eliot
Eliot
Hardy
Hardy
Hardy
Hardy
Shaw
Zola
Lawrence
Lawrence
Lawrence
Hemingway
```

In this example, Db has created several duplicated tuples. One might wish to form a list of authors, with each author appearing only once in the output.

The solution to this problem is given below (see section 5.3.5).

Example. If one were to pose the query

books %% badcolumn

then Db would rightly complain with the diagnostic -

```
db - Line 1 - Invalid domain name badcolumn of relation books
```

because there is no domain named **badcolumn** in this relation.

Example. Similarly the query

books %% title title

is also in error, since domain names must be always unique. Db would produce the diagnostic -

```
db - Relation (Project) - Domain name title not unique
```

5.3.3. Combining Both Selection And Projection

Both the selection and projection operators produce output which is still in tabular format. In other words, the output from either operator is itself a relation.

Since a selection gives a horizontal slice of a relation and a projection gives a vertical slice, the two may be combined to narrow down a large relation to the information desired.

The following example shows how relational operators can be combined in Db. Every relational operator in Db produces another relation as its output. So, the result of any operator can become the input to another.

Example. To answer the question "Which books in the library are written by Austen ?", one uses the following query -

books :: author == "Austen" %% title

The output is -

title
Persuasion
Emma
Pride And Prejudice

5.3.4. Sorting A Relation

Db possesses a powerful *sort* operator (##). On the left of this operator is the relation to be sorted. On the right hand side is a list of domain names upon which the sort takes place.

Numeric fields are sorted according to their value and character string fields according to their lexographic order.

The sort takes place using the first domain in the list, and then on the second domain and so on. If any name in the domain list is preceded by a hyphen the order of the sort is reversed for that domain.

Example. The query to sort the entire **books** relation on the domain **title** is -

books ## title

The output is -

author	bookno	title
Austen	4	Emma
Hardy	8	Far From The Madding Crowd
Hemingway	15	For Whom The Bell Tolls
Hardy	7	Jude The Obscure
Zola	3	Nana
Austen	1	Persuasion
Austen	5	Pride And Prejudice
Shaw	2	Pygmalion
Eliot	10	Silas Marner
Lawrence	14	Sons And Lovers
Hardy	6	Tess Of The D'Urbervilles
Hardy	11	The Mayor Of Casterbridge
Eliot	9	The Mill On The Floss
Lawrence	13	The Virgin And The Gypsy
Lawrence	12	Women In Love

Example. To sort the relation by **author** and then backwards on the domain **title** the query is -

books ## author -title

with the output -

author	bookno	title
Austen	5	Pride And Prejudice
Austen	1	Persuasion
Austen	4	Emma
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	11	The Mayor Of Casterbridge
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hemingway	15	For Whom The Bell Tolls
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Shaw	2	Pygmalion
Zola	3	Nana

5.3.5. Removing Duplicate Tuples

There is a variation of the sort operator that performs the same function except that it removes duplicate tuples in the sorting process. This operator is typed # , as opposed to the two sharps for the previous operator.

Sometimes it is desirable to remove duplicate tuples, sometimes they must be retained. In practice, duplicate tuples arise when a projection operator has removed one or more domains.

Example. To produce a list of authors, with each author appearing only once in the output, the query is -

books %% author # author

The output is -

author
Austen
Eliot
Hardy
Hemingway
Lawrence
Shaw
Zola

5.3.6. Creating New Domains With The Projection Operator

The projection operator is used to remove unwanted domains from a listed relation. It can also be used to form additional domains, based on data from a source relation.

On the right hand side of the projection operator appears a list of domain names, as described above. Alternatively, instead of a single name appearing in the list, an assignment may be made to a new domain. The syntax of an assignment is : the name of the new domain, followed by a = operator, followed by an expression. The expression is evaluated for each source tuple and the result placed into the new domain. Any arbitrary expression can be used on the right hand side of the = sign.

Thus the projection operator can be used, say, to add the fields in two domains together on a tuple by tuple basis.

Example. In these examples, a new relation **data** will be used. The relation was created with the command

dbcreate data event/s data1/f data2/f

and its contents are -

event	data1	data2
1	3.4	4.5
2	-3.6	6.8
3	0.001	3.2
4	2.9	6.7
5	10.1	9.8

One might wish to list the fields of domain **data1** when multiplied by a constant value. Such a query might be -

data %% data1 mult = data1 * 3.75

The output is -

data1	mult
3.4	12.75
-3.6	-13.5
0.001	0.00375
2.9	10.875
10.1	37.875

Example. If one wished to list **data** with two new domains, **sum** which is the addition of **data1** and **data2**, and **diff** which is the difference of the two, one would use the following query -

data %% event data1 data2 sum = data1 + data2 diff = data1 - data2

The listed output is -

event	data1	data2	sum	diff
1	3.4	4.5	7.9	-1.1
2	-3.6	6.8	3.2	-10.4
3	0.001	3.2	3.201	-3.199
4	2.9	6.7	9.6	-3.8
5	10.1	9.8	19.9	0.3

Note that the domains **event data1** and **data2** must appear after the projection if they are to be included in the output. The command

data %% data1 = data1 * data2

is not ambiguous since domain **data1** appears only once in the output. In this case the output is -

data1
15.3
-24.48
0.0032
19.43
98.98

5.3.7. Syntactic Sugar For Assignments

The construction

data %% data1 += data2

is exactly equivalent to

data %% data1 = data1 + data2

There are also similar operators -=, *=, /= and %=. Thus one could use a query such as

data %% event data1 data2 -= data1

5.4. Scalar Expressions

Scalar expressions occur in selection or projection operations. A scalar expression can eventually be calculated and evaluated to a single value.

It is important to understand how scalar expressions are evaluated. Scalar expressions have a similar syntax and semantics to expressions in the C language. This section lists the properties of such expressions.

5.4.1. Types Of Expressions

When a relation is created by Dbcreate every domain has an associated data type, for instance, *long* integer or character *string*. A complete table of these types is shown in appendix I. Most of these types have direct equivalents in both the C language and Fortran. For instance the type *float* of Db corresponds with *real*4* of Fortran.

The data type *unsigned* of Db corresponds to *unsigned short* of the C language. A type of *int* in Db corresponds to *short* in the C language. In Db, *unsigned* and *unsigned short* are synonymous, and so are *int* and *short*.

Db usually permits expressions involving mixed types and will convert values from one type to another if necessary. Using the example of relation **data**, where **event** is a domain of *short* integers and **data1** is a domain of floating point numbers, the following query would be perfectly acceptable to Db -

data :: event > data1

However Db would object if it were presented with this query -

```
books :: title == bookno
```

In this example Db is being asked to compare a *string* of characters with a numeric value. Instead Db would print the message -

```
db - Line 1 - Operands of == are string and short
```

This means that the == operator is being asked to compare a *string* value with a *short* integer value, something Db refuses to do.

The numeric data types can be assigned back and forth and compared with one another. When Db performs arithmetic on two operands of different types, it always chooses a result which retains as much accuracy as possible. For example, adding a *long* integer with a *float* number produces a result which is another *float*.

Strings lie in a class of their own and cannot be compared or converted to numeric fields (but see section 5.5.2).

5.4.2. Packed Decimal Integers

On the VAX implementation, Db also supports packed decimal values. These permit signed integer numbers of greater precision than is possible with *long* integers. There are two types of packed decimal integers : *pack15* numbers are kept to a 15 digit accuracy and *pack31* numbers to a 31 digit accuracy. Packed decimal arithmetic is always carried out to full precision. However a packed decimal value requires at least twice the space of a *long* integer and arithmetic calculations may be slower.

The only exception to the rule that numeric fields are freely interchangeable is in the case of packed decimal values. These can only be compared with one another or with *long* integers. They cannot be compared or converted to either of the floating point types. Similarly, *long* integers can be converted to packed decimal values but the floating point types cannot.

5.4.3. Casts - Converting The Type Of Expressions

It is sometimes necessary to be able to convert the type of an expression. For instance, the way in which arithmetic is performed depends on the data types of the operands. In the case of the **data** relation above, we might wish to find the values of the **event** domain when divided by 2. Such a query might be -

```
data %% event eventdiv2 = event / 2
```

and the output is -

```
event  eventdiv2
    1          0
    2          1
    3          1
    4          2
    5          2
```

Because domain **event** is of type *short* integer, and the division is also by an integer number, the arithmetic is according to the rules of integer division. The results in output domain **eventdiv2** are themselves integers and are therefore truncated.

If one wishes the division to be performed exactly, there are two possible solutions. The first is to use a query such as -

```
data %% event eventdiv2 = event / 2.0
```

In this case the divisor is a floating point number and so the division is performed exactly.

The second solution is to use a cast to convert the type of **event** before the division takes place. Casts in Db have the same syntax as they do in the C language. An expression is preceded by a type name in parentheses : the expression is evaluated and then converted to the required data type. Thus another way of solving this problem would be to type -

data %% event eventdiv2 = (float)event / 2

In this example the domain **event** is cast to a type of *float* before being used in the division. The output would then be -

```
event   eventdiv2
    1         0.5
    2           1
    3         1.5
    4           2
    5         2.5
```

Another reason to use casts is that they can alter the printing width of listed output. Numeric fields are always printed in a width large enough for the biggest number that can be held in a given data type. For instance, *long* integers are printed in a width of eleven characters since that is required to print the widest number that can be held. One could use a cast to convert, say, a domain of *long* integers to one of *unsigned short* numbers. Then the printing width would only be five characters.

A final reason for the existence of casts is when Db is creating a new relation from existing data (see section 5.8). Then it may be important to ensure that the domain types in the new relation are the desired ones to avoid wasting disc space.

5.4.4. The Types Of Constants

Integer constants, including octal and hexadecimal values, are always considered to be *long* integers. Similarly floating point constants are always considered to be *double* precision.

5.4.5. Character Strings

Within a double quoted character string a backslash (\ ) has a special meaning. For instance, the sequence **\n** is interpreted as a new-line character. This table shows all the special characters which can be introduced with a backslash. When any other character is preceded by a backslash, the backslash is ignored.

Sequence	Meaning To Db
\n	new line
\t	tab
\b	backspace
\f	form feed
\r	carriage return
\"	double quote
\\	\

Normally, a string constant must appear on a single line in a query. Db prints a diagnostic if it finds a string that is not terminated by a double quote on the same line. However, the sequence \newline or \newline tabs is ignored within a string constant, so a long string can be broken up over several lines if necessary.

Example.

"This is an example of a very long string constant
that is split into two lines"

5.4.6. The Conditional Operator

Db has a conditional operator similar to that of the C language. This facility allows one to use an "if ... then ... else ... " construct in an expression. The syntax of a conditional expression is

$$expr1 \; ? \; expr2 : expr3$$

If *expr1* turns out to be true, that is non-zero, the whole expression has the value returned by *expr2*. Otherwise the value of the entire conditional expression is *expr3*.

This operator can be used in a projection, say, to force a domain to take on positive values, or to assign a value to a domain only under certain conditions.

Example. The following query lists the absolute values of fields taken from domain **data1** of relation **data.**

data %% data1 abs = data1 > 0.0 ? data1 : -data1

The output is -

```
 data1    abs
   3.4    3.4
  -3.6    3.6
 0.001  0.001
   2.9    2.9
  10.1   10.1
```

Example. With the following query, one could list the **title** domain of **books** replacing all titles whose author is "Hardy" by a string of asterisks -

books %% title = author == "Hardy" ? "******" : title**

The output would be -

```
title
Persuasion
Emma
Pride And Prejudice
The Mill On The Floss
Silas Marner
********
********
********
********
Pygmalion
Nana
Women In Love
The Virgin And The Gypsy
Sons And Lovers
For Whom The Bell Tolls
```

5.4.7. Accessing Individual Bits From A Data Field

Db supports operators that allow one to retrieve single bits from *short* or *unsigned short* data fields. This feature is normally not required unless one is storing large quantities of data that could be compressed into a width of a few bits. This table shows the operators that can be used to access individual bits within a field -

Operator	Purpose
|	Bitwise Or
&	Bitwise And
^	Bitwise Exclusive Or
<<	Bitwise Left Shift
>>	Bitwise Right Shift
~	Unary Bitwise Complement

There are also assignment operators **\|=**, **&=**, **^=**, <<= and >>= (see section 5.3.7). All these operators have the same semantics as they do in the C language, except that they may only be applied to *short* or *unsigned short* operands. They all return *short* values. The shift operators are guaranteed to return a logical as opposed to an arithmetic shift.

Example. In order to retrieve tuples from a hypothetical relation **bits** where only the second or third bits of domain **bitwise** are set to 1, one would use this query -

bits :: (bitwise & (short)0x6) != 0

The cast **(short)** is required since the **&** operator does not accept a *long* integer as one of its operands. The parentheses are also required since the precedence of the **&** operator is less than that of **!=**.

5.5. Procedures Incorporated Into Db

A number of useful procedures are incorporated into Db. Some of these routines are useful for dealing with character strings and some for converting character strings to numeric values. There are also powerful routines for processing calendar dates. A summary of these procedures is in appendix III.

Built-in procedures can be used in any selection expression or in an assignment for a projection list.

An argument to one of these procedures can be a number, a string enclosed in double quotes or a name referring to a domain. An argument can even be the result of another procedure so nested calls to procedures are permitted.

Db complains if the number of arguments to a procedure is incorrect, or if the data type of an argument is wrong and cannot be converted.

5.5.1. Procedures For Character Strings

The built-in routines **strcat, substr, strlen** and **regex** all deal with character string fields.

Strcat returns a string which is the concatenation of its two arguments.

Substr returns a part of a string argument. The first argument is the string, the second the character position from which the sub-string is to be taken and the third is the length of the sub-string.

Strlen returns the number of characters in its argument.

Perhaps the most useful routine for dealing with strings is **regex.** This returns a value of one if a string matches a given regular expression pattern and zero otherwise. Regular expression syntax is the same as that provided inside the UNIX *ed* program. This procedure can be used to see if a field from a tuple matches a pattern and make a selection accordingly. Its first argument is the desired pattern, the second is the string to be matched.

5.5.2. Procedures For Converting Strings To Numeric Values

Db will not convert a character *string* to a numeric value unless explicitly instructed. The following procedures are available for such a conversion.

The procedures **stol**, **stod** and **stop31** take a single argument, which must be a string, and convert it to its equivalent numeric value. The difference is in the type of the returned value. **Stol** regards its argument as a long integer. **Stod** and **stop31** regard it as being a floating point number or as a 31 digit packed decimal number respectively.

None of these procedures complain if an argument is not a recognisable number. They will simply return a value of zero.

5.5.3. Miscellaneous Procedure

The routine **count** can be used to number consecutive tuples. The first time it is called, **count** saves its argument in an internal counter, and returns the same value. In subsequent calls the argument is ignored but the counter is incremented and returned.

Unfortunately there is no way to reset the counter. Nor is **count** very sensible if called from more than one place in a query.

Example. Using relation **books** again, the query

books %% author length = strlen(author)

produces -

author	length
Austen	6
Austen	6
Austen	6
Eliot	5
Eliot	5
Hardy	5
Hardy	5
Hardy	5
Hardy	5
Shaw	4
Zola	4
Lawrence	8
Lawrence	8
Lawrence	8
Hemingway	9

Again, the projection has created several duplicated tuples. A cleaner way of asking the same query is -

books %% author length = strlen(author) # author

This produces -

author	length
Austen	6
Eliot	5
Hardy	5
Hemingway	9
Lawrence	8
Shaw	4
Zola	4

Example. This query

books %% subtitle = substr(title, 5, 9) title

obtains a sub-string from each **title** field. Each sub-string is formed by taking nine characters from **title**, beginning at position five. For comparison, the **title** field is included here too. The output is -

subtitle	title
uasion	Persuasion
	Emma
e And Pre	Pride And Prejudice
Mill On T	The Mill On The Floss
s Marner	Silas Marner
Of The D	Tess Of The D'Urbervilles
The Obsc	Jude The Obscure
From The	Far From The Madding Crowd
Mayor Of	The Mayor Of Casterbridge
alion	Pygmalion
	Nana
n In Love	Women In Love
Virgin An	The Virgin And The Gypsy
And Love	Sons And Lovers
Whom The	For Whom The Bell Tolls

Example. To find those tuples where a domain matches a certain pattern, one would use the **regex** procedure. The query

books :: regex("The", title)

finds any tuples with "The" somewhere in the **title** domain. The output in this case is -

author	bookno	title
Eliot	9	The Mill On The Floss
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Lawrence	13	The Virgin And The Gypsy
Hemingway	15	For Whom The Bell Tolls

Example. Similarly, this query lists titles beginning with "The" -

books :: regex("^The", title) %% title

title
The Mill On The Floss
The Mayor Of Casterbridge
The Virgin And The Gypsy

Example. The procedure **count** is used to number tuples. For instance to obtain a list of numbered authors one could use this query -

books %% author # author %% number = count(1) author

with the output -

number	author
1	Austen
2	Eliot
3	Hardy
4	Hemingway
5	Lawrence
6	Shaw
7	Zola

Example. In order to obtain only the first four tuples from **books** one would use this query -

books :: count(1) <= 4

The output is -

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Eliot	9	The Mill On The Floss

5.5.4. Procedures For Calendar Dates

Db includes a powerful set of routines for processing calendar dates. These procedures accept as arguments dates represented by *long* integers in the form YYYYMMDD. For instance the date "31st December 1981" is represented by the integer 19811231.

These procedures will work with any valid Gregorian Calendar date. A year of less than 100 is assumed to refer to the 20th century, not the early Christian era.

If the date procedures are given a non-existent date, Db complains and the procedure continues, assuming another valid date. For instance, the date 19810229 is incorrect since Anno Domini 1981 was not a leap year, and hence there was no 29th February. Db would print a diagnostic and continue, assuming instead that the correct date is 19810301 (1st March 1981).

The procedures **dt_pretty** and **dt_ypretty** each return a string representing their date argument in a "pretty" or "very pretty" format. Similarly the procedures **dt_weekday** and **dt_month** return a string representing the weekday name and the month name of the given date.

Dt_days returns the number of days occurring between two dates. The procedure **dt_sundays** returns the number of inclusive Sundays lying between two dates.

The first argument to **dt_offset** is a date. The second is an offset, positive or negative, measured in days from that date. This procedure returns the new date.

The procedures **dt_ltos** and **dt_stol** convert a date to a character string and vice versa. In the case of **dt_ltos** the first argument is a date and the second argument is a delimiter string. This returns a string of the form "1981XXX12XXX31" where "XXX" is the delimiter. **Dt_stol** takes a string of the form "81X12X31" or "811231" and converts it to the equivalent long integer date format.

Dt_u3tol takes three arguments : a year number, month number and day number. It returns the equivalent date as a single long integer.

Finally, **dt_today** returns the date on which Db is invoked.

Example. These examples will use a new relation **loans** that was created by the command

dbcreate loans bookno/s personno/s datedue/l

The entire relation might be -

bookno	personno	datedue
5	1	811231
6	1	811231
7	1	820107
8	3	820110
13	2	820106
14	2	820106
15	7	810423

To print the **datedue** domain in a "pretty" format, one can use either the **dt_pretty** or **dt_vpretty** procedures. The query

loans %% datedue pretty = dt_pretty(datedue) vpretty = dt_vpretty(datedue)

produces this listing -

datedue	pretty	vpretty
811231	Thu 31 Dec 1981	Thursday 31 December 1981
811231	Thu 31 Dec 1981	Thursday 31 December 1981
820107	Thu 7 Jan 1982	Thursday 7 January 1982
820110	Sun 10 Jan 1982	Sunday 10 January 1982
820106	Wed 6 Jan 1982	Wednesday 6 January 1982
820106	Wed 6 Jan 1982	Wednesday 6 January 1982
810423	Thu 23 Apr 1981	Thursday 23 April 1981

Example. To find which of these dates is before today (assuming today's date to be 3rd December 1981), the query is -

loans :: dt_days(dt_today(), datedue) > 0

This produces -

bookno	personno	datedue
15	7	810423

Example. To add an offset of 100 days to fields from **datedue** and to list both the original and the new dates in a "pretty" format, the query is -

loans %% old = dt_pretty(datedue) new = dt_pretty(dt_offset(datedue, 100))

The output is -

old	new
Thu 31 Dec 1981	Sat 10 Apr 1982
Thu 31 Dec 1981	Sat 10 Apr 1982
Thu 7 Jan 1982	Sat 17 Apr 1982
Sun 10 Jan 1982	Tue 20 Apr 1982
Thu 31 Dec 1981	Sat 10 Apr 1982
Wed 6 Jan 1982	Fri 16 Apr 1982
Wed 6 Jan 1982	Fri 16 Apr 1982
Thu 23 Apr 1981	Sat 1 Aug 1981

5.6. Aggregation Of Tuples

Db possesses an *aggregation* operator that allows the user to group tuples together by their field values and to find, say, the minimum or maximum values within each group. This is a generalised operator by which the user can split up a relation into groups and perform some totaling or counting operation on the tuples of each group.

The syntax of the aggregation operator (@@) is roughly the same as that of the projection operator. On the left hand side is the source relation from which Db reads tuples. On the right hand side is a list of domain names or assignments.

Each domain name appearing by itself on the right hand side specifies how the source domain is to be grouped for the purposes of aggregation. The source relation is split up into groups of tuples based on the values in these domains. Therefore one can specify that Db should aggregate a relation grouping its tuples by their values in certain domains.

Assignments to new domains are paired together so that there are two assignments for each new domain of the output. The first initialisation assignment is performed only for the first tuple from each group of the source relation. The second assignment is performed for every tuple from the source relation.

In either assignment, one may refer to a field from a source tuple or to one from the output tuple. By referring to a field in the output tuple, one can, say, add a field from the input to this value and hence keep a running total in the output. In order to avoid ambiguity, domain names referring to the source relation are always preceded by a period (.). This operator can preserve the form of the input relation, that is, the output can be made to have exactly the same attributes as the source relation.

The aggregation operator does not automatically sort the source relation into order before grouping the tuples together. So in a typical query, it would be used in conjunction with one of the sort operators.

Example. These examples use the **data** and **loans** relations shown in sections 5.3.6 and 5.5.4.

One might wish to group the **loans** relation by its **personno** domain, and then to count the number of tuples within each group. Firstly this relation should be sorted by **personno** and then aggregated into groups by this domain. The first assignment, performed for the first tuple in each group, initialises a new domain **count** by assigning it a value of zero. The second assignment, performed for every tuple from a group, increments **count** by one. Therefore, **count** will eventually contain the number of tuples for each group from **loans**. The query could be either -

loans ## personno @@ personno count = 0 count = count + 1

or -

loans ## personno @@ personno count = 0 count += 1

and the output is -

personno	count
1	3
2	2
3	1
4	1
7	1

Example. Similarly, to count the number of titles for each author from the **books** relation, the query is -

books ## author @@ author count = 0 count += 1

and the output is -

author	count
Austen	3
Eliot	2
Hardy	4
Hemingway	1
Lawrence	3
Shaw	1
Zola	1

Example. One might wish to find the minimum value of domain **data1** from relation **data**. In this case the relation is not to be split up into groups and the output relation will consist of one single tuple. Since the input relation is not to be grouped, there is no need for it to be sorted beforehand. The output domain might be called, say, **min1** and it is initialised from the first tuple of the source relation. Then, for every source tuple, both the input and output fields are examined to determine the lower value and this is assigned back to the output tuple. So if an output field has a lower value than the input, it is assigned back to itself. Therefore, the output field always contains the minimum valued field yet encountered from the source.

The conditional operator is used to decide which of the input or output fields has the lower value. Because there is a need to refer to **data1** from the source, this domain is preceded by a period in the query. The query to Db is -

data @@ min1 = .data1 min1 = .data1 < min1 ? .data1 : min1

The output is -

min1
-3.6

Example. In order to total domains **data1** and **data2** and to produce a listing with the same names for these domains, one uses this query -

data @@ data1 = 0.0 data1 += .data1 data2 = 0.0 data2 += .data2

Since both **data1** and **data2** are floating point, it is necessary to initialise the output domains to be floating point values, in this case 0.0. Again there is no need to sort the relation before the aggregation. The output is -

data1	data2
12.801	31

Example. Finally, to find the earliest value of **datedue** from the relation **loans**, again grouping this relation by **personno**, one would use this query -

```
loans ## personno @@ personno datedue = .datedue
datedue = .datedue < datedue ? .datedue : datedue
```

The output is -

personno	datedue
1	811231
2	820106
3	820110
4	811231
7	810423

5.6.1. More Concerning Aggregations

The first initialisation assignment can sometimes be omitted. If so, the output domain is of type *long* and is set to zero. Thus the first example of section 5.6 could have been written as -

```
loans ## personno @@ personno count += 1
```

Any arbitrary expression can appear in either assignment. So to concatenate each **title** field from relation **books** together, producing one output tuple for every **author**, one could use this query -

```
books ## author @@ author title = ""
title = strcat( title, strcat( .title, "/" ) )
```

In this example, **title** is initialised to a null string. Then for every tuple from the source relation, **title** becomes a concatenation of itself with a corresponding input field. There is a nested call to the built-in procedure *strcat* so that the titles are separated by slashes. The output is -

author	title
Austen	Emma/Persuasion/Pride And Prejudice/
Eliot	The Mill On The Floss/Silas Marner/
Hardy	Jude The Obscure/Tess Of The D'Urbervilles/Jude The Obscure/ ......
Hemingway	For Whom The Bell Tolls/
Lawrence	The Virgin And The Gypsy/Women In Love/Sons And Lovers/
Shaw	Pygmalion/
Zola	Nana/

5.7. Operators Processing Two Relations

All the relational operators that have been introduced so far process a single relation. The operators described below manipulate two relations. For instance, the *union* operator concatenates two relations together. Similarly, the *join* operator cross-references two relations by matching their respective tuples.

5.7.1. Union Of Two Relations

Two entire relations may be concatenated together by the *union* operator (++). This operator is similar to the familiar addition operator of standard arithmetic. One can regard it as "adding" two relations together.

Both relations must have the same degree. The domains need not have the same names in both relations, but the data types must be compatible with one another. So if the first domain of the left hand relation holds strings then so must the first domain of the right hand relation. Similarly, if a given domain in one relation holds floating point values, the corresponding domain of the second relation must hold floating point or integer fields, but not strings or packed decimal integers.

The result has domains with the same names and types as those from the left relation.

Example. Using a new relation **newbooks** which is -

author	bookno	title
Bronte	16	Wuthering Heights
Eliot	21	Daniel Deronda
Hemingway	19	The Old Man And The Sea
Hemingway	20	The Snows Of Kilimanjaro
Lawrence	17	Aaron's Rod
Lawrence	18	The Prussian Officer

the query

books ++ newbooks

produces -

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Shaw	2	Pygmalion
Zola	3	Nana
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls
Bronte	16	Wuthering Heights
Eliot	21	Daniel Deronda
Hemingway	19	The Old Man And The Sea
Hemingway	20	The Snows Of Kilimanjaro
Lawrence	17	Aaron's Rod
Lawrence	18	The Prussian Officer

Example. However, if one tried to concatenate **books** and **data** together, Db produces this error diagnostic -

```
db - Line 1 - Invalid domains author and event of relations books and data
```

meaning that these domains are incompatible with one another. (**Author** is a domain of character *strings* and **event** is a domain of *short* integers.)

5.7.2. Intersection Of Two Relations

The *intersection* operator (. .) shows which tuples from one relation correspond to those from another. Tuples from both relations are compared together to find any that match. This operator returns those tuples from the left operand matching any from the right operand. Thus the output consists of a subset of the left hand relation.

In both relations there must be at least one domain with the same name. It is these common domains that are examined to find matching tuples.

Should there be more than one domain with the same name in both relations then tuples are considered to match only if ALL the common domains have identical values.

Example. The query

books .. loans

is the way one would phrase the question "Which books are currently out on loan?" **Books** has domains **author, bookno** and **title. Loans** has domains **bookno, personno** and **datedue** so the only common domain from both relations is **bookno.** The output consists of tuples from **books** with a **bookno** value that appears somewhere in **loans.** The output is -

author	bookno	title
Austen	5	Pride And Prejudice
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls

Example. In the same way, the query

loans .. books

shows all the tuples from **loans** that match one or more tuples from **books** -

bookno	personno	datedue
5	1	811231
6	1	811231
7	1	820107
8	3	820110
13	2	820106
14	2	820106
15	7	810423

5.7.3. Difference Of Two Relations

The *difference* operator (- -) again compares tuples from two relations. It returns those tuples from the left hand operand that do not match any from the right hand operand. Thus the output consists of a subset of the left hand relation. This is the converse operation to intersection ; it returns tuples from the left hand relation not corresponding to any from the right hand relation.

Matching is performed in exactly the same way for the intersection operator. So again, there must be at least one commonly named domain from both relations.

Example. In order to answer the question "Which books are still in the library ?" the query to Db is -

books — loans

and the output is -

author	bookno	title
Austen	1	Persuasion
Shaw	2	Pygmalion
Zola	3	Nana
Austen	4	Emma
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	11	The Mayor Of Casterbridge
Lawrence	12	Women In Love

5.7.4. Join Of Two Relations

The *join* operator (* *) is perhaps the most useful of all the operators that process two relations. This operator can be used to cross-reference two relations.

The output from a join is the concatenation of every tuple from the left hand relation with every corresponding tuple from the right hand operand. Matching is done in the same way for the difference and intersection operators, so there must be at least one domain with the same name in both relations. The duplicated domains are removed from the output. Tuples from either relation not matching any from the other are discarded and do not form part of the result.

The join operator is commutative. That is, the same information is listed regardless of the way in which the operands are presented to Db, although the domains may appear in a different order. The difference and intersection operators are not commutative since the command **books .. loans** produces a very different output to **loans .. books**

Example. In order to obtain a full cross-reference listing of the relations **books** and **loans** one uses this query -

books ** loans

and the output is -

author	bookno	title	personno	datedue
Austen	5	Pride And Prejudice	1	811231
Hardy	6	Tess Of The D'Urbervilles	1	811231
Hardy	7	Jude The Obscure	1	820107
Hardy	8	Far From The Madding Crowd	3	820110
Lawrence	13	The Virgin And The Gypsy	2	820106
Lawrence	14	Sons And Lovers	2	820106
Hemingway	15	For Whom The Bell Tolls	7	810423

Example. The query -

loans ** books

produces -

bookno	personno	datedue	author	title
5	1	811231	Austen	Pride And Prejudice
6	1	811231	Hardy	Tess Of The D'Urbervilles
7	1	820107	Hardy	Jude The Obscure
8	3	820110	Hardy	Far From The Madding Crowd
13	2	820106	Lawrence	The Virgin And The Gypsy
14	2	820106	Lawrence	Sons And Lovers
15	7	810423	Hemingway	For Whom The Bell Tolls

These two examples have produced the same information although the order of output domains is different.

Example. This example uses a new relation called **people** which relates names and addresses to person numbers. This relation might be -

lname	fname	address	personno
Fischer	Denise	Biedermannsdorf	5
Francis	Michael	Vienna	6
Mednieks	Zig	Laxenburg	3
Medow	Serge	Baden	2
Novachkov	Asen	Vienna	7
Schweeger	Bernhard	Klosterneuburg	4
Ward	Robert	Vienna	1

Using this sample relation, the join of **loans** and **people** is formed by the query -

loans ** people

The output is -

bookno	personno	datedue	lname	fname	address
6	1	811231	Ward	Robert	Vienna
5	1	811231	Ward	Robert	Vienna
7	1	820107	Ward	Robert	Vienna
14	2	820106	Medow	Serge	Baden
13	2	820106	Medow	Serge	Baden
8	3	820110	Mednieks	Zig	Laxenburg
15	7	810423	Novachkov	Asen	Vienna

Example. One could join all three relations and eliminate uninteresting domains by the query -

books ** loans ** people %% fname lname address title

This produces -

fname	lname	address	title
Robert	Ward	Vienna	Tess Of The D'Urbervilles
Robert	Ward	Vienna	Pride And Prejudice
Robert	Ward	Vienna	Jude The Obscure
Serge	Medow	Baden	Sons And Lovers
Serge	Medow	Baden	The Virgin And The Gypsy
Zig	Mednieks	Laxenburg	Far From The Madding Crowd
Asen	Novachkov	Vienna	For Whom The Bell Tolls

Example. The join of **books** and **newbooks** would list only those tuples which are identical in both relations since they both have the same domains. In this case there are no such tuples.

Example. Db objects if there are no common domains from either relation. For instance this query -

```
books ** data
```

produces the message -

```
db - Line 1 - No common domains of relations books and data
```

5.7.5. Cartesian Product Of Two Relations

The *Cartesian product* (^^) forms new tuples from two relations by concatenating every tuple from the left hand relation with each tuple in turn from the right hand relation. That is, it forms all possible combinations of tuples from both relations.

So if the left hand relation contains, say, eleven tuples, and the right hand relation has twelve, the product of the two would have 132 tuples.

This product operator differs from the join, intersection and difference operators which require that there should be at least one commonly named domain from both operands. Instead, it insists that the domain names of both source relations all be different. This restriction exists so that the result tuples can be formed by concatenating the left and right hand tuples together while ensuring that the domain names of the result are unique.

The second example below shows how this operator can be used in conjunction with the aggregation operator to total up a relation over various ranges of values that are specified in another relation. The third example shows how this operator can be used to simulate a join of two relations.

Example. Using the relations **newbooks** and **people** again, one may form the product of these two relations with the query

```
newbooks ^^ people
```

The result would have domains **author bookno title lname fname address** and **personno** and would contain every possible combination of tuples from both relations.

Example. The product operator can be used in conjunction with the aggregation operator to form totals of domains whose fields lie within various ranges. Using the example of relation **data** again, one might wish to find totals of domains **data1** and **data2** for various ranges of **event** that are specified in a second relation. Suppose there is a relation **ranges** detailing the ranges of domain **event** to be summed up. This relation might be -

low	high	description
1	3	One To Three
1	5	Everything
2	4	Two To Four
3	5	Three To Five

Domains **low** and **high** contain the lowest and highest values of the ranges of **event** used in summing up **data1** and **data2** The entire query to sum up the relation over these ranges is -

```
ranges ^^ data
:: low <= event && event <= high
@@ low high description
   total1 = 0.0 total1 += .data1
   total2 = 0.0 total2 += .data2
```

The first line of this query forms the product of **ranges** and **data**. This product is -

low	high	description	event	data1	data2
1	3	One To Three	1	3.4	4.5
1	3	One To Three	2	-3.6	6.8
1	3	One To Three	3	0.001	3.2
1	3	One To Three	4	2.9	6.7
1	3	One To Three	5	10.1	9.8
1	5	Everything	1	3.4	4.5
1	5	Everything	2	-3.6	6.8
1	5	Everything	3	0.001	3.2
1	5	Everything	4	2.9	6.7
1	5	Everything	5	10.1	9.8
2	4	Two To Four	1	3.4	4.5
2	4	Two To Four	2	-3.6	6.8
2	4	Two To Four	3	0.001	3.2
2	4	Two To Four	4	2.9	6.7
2	4	Two To Four	5	10.1	9.8
3	5	Three To Five	1	3.4	4.5
3	5	Three To Five	2	-3.6	6.8
3	5	Three To Five	3	0.001	3.2
3	5	Three To Five	4	2.9	6.7
3	5	Three To Five	5	10.1	9.8

The second line of the query selects only those tuples from this intermediate relation where **event** is in the range specified by **low** and **high**. After the first two lines of the query, the data becomes -

low	high	description	event	data1	data2
1	3	One To Three	1	3.4	4.5
1	3	One To Three	2	-3.6	6.8
1	3	One To Three	3	0.001	3.2
1	5	Everything	1	3.4	4.5
1	5	Everything	2	-3.6	6.8
1	5	Everything	3	0.001	3.2
1	5	Everything	4	2.9	6.7
1	5	Everything	5	10.1	9.8
2	4	Two To Four	2	-3.6	6.8
2	4	Two To Four	3	0.001	3.2
2	4	Two To Four	4	2.9	6.7
3	5	Three To Five	3	0.001	3.2
3	5	Three To Five	4	2.9	6.7
3	5	Three To Five	5	10.1	9.8

Because the product operator has produced tuples in the correct order there is no need for them to be sorted before the aggregation. The final lines of the query aggregate the tuples to produce -

low	high	description	total1	total2
1	3	One To Three	-0.199	14.5
1	5	Everything	12.801	31
2	4	Two To Four	-0.699	16.7
3	5	Three To Five	13.001	19.7

Example. One can use the product operator to form a join of two relations if one explicitly states the domains that are to be matched. For instance this query -

ranges ^^ data :: event == low

joins **ranges** and **data** together by matching their domains **event** and **low** respectively. The output is -

low	high	description	event	data1	data2
1	3	One To Three	1	3.4	4.5
1	5	Everything	1	3.4	4.5
2	4	Two To Four	2	-3.6	6.8
3	5	Three To Five	3	0.001	3.2

5.7.6. Parentheses In Relational Expressions

Normally, relational operators group from left to right. For instance, Db evaluates this query by performing the union operation before the join -

books ++ newbooks ** loans

Parentheses can be used in relational expressions just as they can be used in arithmetic expressions. One may use parentheses to override the default order of evaluation.

Example. The following example takes tuples from **books** that do not match any from **loans**. It then concatenates them with tuples from **books** that do match. So this is a very indirect way of listing the relation **books.**

(books .. loans) ++ (books — loans)

The output is -

author	bookno	title
Austen	5	Pride And Prejudice
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls
Austen	1	Persuasion
Shaw	2	Pygmalion
Zola	3	Nana
Austen	4	Emma
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	11	The Mayor Of Casterbridge
Lawrence	12	Women In Love

Example. One might wish to produce a listing of **books** with some indication as to whether each book is currently out on loan. The following query is suitable -

```
( books .. loans %% author title out = "Yes"  )
++
( books — loans %% author title out = "No"  ) ## author title
```

The output is -

author	title	out
Austen	Emma	No
Austen	Persuasion	No
Austen	Pride And Prejudice	Yes
Eliot	Silas Marner	No
Eliot	The Mill On The Floss	No
Hardy	Far From The Madding Crowd	Yes
Hardy	Jude The Obscure	Yes
Hardy	Tess Of The D'Urbervilles	Yes
Hardy	The Mayor Of Casterbridge	No
Hemingway	For Whom The Bell Tolls	Yes
Lawrence	Sons And Lovers	Yes
Lawrence	The Virgin And The Gypsy	Yes
Lawrence	Women In Love	No
Shaw	Pygmalion	No
Zola	Nana	No

The first line of the query produces tuples from **books** which match any from **loans**. The second line produces tuples that do not match those from **loans**. These two sets of tuples are concatenated together by the union operator and then sorted in order of **author** and **title**.

Example. In order to find out how many books each person has out on loan, one would use this query -

```
( ( people ** loans ) @@ lname fname borrowed += 1 )
++
( ( people — loans ) %% lname fname borrowed = 0 ) ## lname fname
```

The first line of this query produces tuples from **people** who have borrowed one or more books from the library. An aggregation is used to count the number of books for each person. The final line produces tuples from **people** who have not borrowed any books. These two sets of tuples are concatenated together by the union operator, and then sorted in order of **lname** and **fname**. The output is -

lname	fname	borrowed
Fischer	Denise	0
Francis	Michael	0
Mednieks	Zig	1
Medow	Serge	2
Novachkov	Asen	1
Schweeger	Bernhard	0
Ward	Robert	3

5.8. Preserving The Output From Db

Db can be instructed to send its output to a file. The output can be either in relational format or in ascii. The former is normally more useful, since a new relation created by Db can then be used in further queries. Ascii output is useful when a listing is to be printed or viewed at a terminal.

When Db is instructed to send its output to a specific destination, it usually does not produce a listing on the standard output. That is, the operators described below normally suppress printed output.

5.8.1. Creating A New Relation

Db can be instructed to place its output in a new relation file instead of listing it to the standard output. The creation operator (<=) informs Db to place its output in a new relation. Once the new relation is formed, it can be used in further Db queries just as any other relation.

If the relation should already exist, its previous contents will be overwritten. The creation of the new relation is subject to the usual access restrictions imposed by the UNIX system. One must have the necessary permissions to create the new file.

On the right hand side of this operator is a relational expression that is evaluated and placed into a file whose name appears on the left hand side.

There is a second creation operator (=>) that has exactly the same purpose except that the flow of tuples goes from left to right. For this operator, the file name is on the right hand side and the expression on the left.

In most programming languages, assignments are usually made from right to left. However most of the Db operators introduced so far encourage one to think of a flow of tuples from left to right. The user should choose whichever form of this operator he prefers.

Example. This query is the same as the one shown in section 5.3.5 except that no listing is sent to the standard output. Instead the information is placed into a new relation **authorlist.**

authorlist <= books %% author # author

Example. The next query is the same as the one shown in section 5.6 but output is produced in a new relation **titlecount.**

books ## author @@ author count = 0 count += 1 => titlecount

Example. In the following example, the relations **books** and **newbooks** are concatenated together to create a new composite relation **allbooks.**

allbooks <= books ++ newbooks

5.8.2. Appending Output To A Relation

Db can append generated output to an existing relation. The append operators (<+ and +>) are very similar to the creation operators described above but output is added to an existing relation. Should the relation not exist, then it will be created. Again, one must have the necessary permissions to write to the relation file.

Any output appended to an existing relation must be compatible with that relation. For instance, the degree of the output must agree with that of the relation. The corresponding domains of the generated tuples and those of the destination relation must have compatible data types.

Db does not guarantee that new tuples will preserve the sorting order, if any, of the relation. There again, the sorting order will not be disturbed greatly. Tuples are inserted into the relation regardless of whether they are duplicates.

Example. In order to append the relation **newbooks** directly into **books** one uses this query -

books <+ newbooks

Alternatively, this query performs exactly the same purpose -

newbooks +> books

5.8.3. Relations Are Locked

Db does not permit one to copy part or all of one relation on to itself. No tuples can be placed into a relation if one is already reading data from that relation. For example, the query

books <+ newbooks

must be written so and not as

books <= books ++ newbooks

This would require tuples from **books** to be copied on to themselves. Db would produce the diagnostic -

db - Relation books - Open error : Relation already opened for writing

Db also incorporates a locking mechanism that protects a relation from being ruined by more than one person trying to update it at the same time. In order to protect the structure of a relation, Db forbids the user to update it when anyone else is updating or even reading information from it. Conversely, one is not allowed to read from a relation when it is being updated by someone else. Should a user be adding new data to **books** , say, then Db would refuse anyone else access to the same relation. Db would produce this diagnostic -

db - Relation books - Open error : Relation locked by another process

5.8.4. Listed Output To A File

Db can be instructed to send listed ascii output to a file, in the same way that one can tell it to send output to a relation. The operators are <== and ==>. They are used in a query in exactly the same way that the creation operators are used. The only difference is they produce not a relation but an ascii file that may be printed or viewed by standard UNIX programs such as *more*.

The listed output is influenced by the global **-h**, **-df** and **-dt** options presented to Db. These options were described in section 5.2. Again, one must have the necessary permissions to create the new file.

Example. This query produces an ascii listing of relation **people** in the file **printout**

printout <== people

5.8.5. Appending Listed Ascii Output

The operators <++ and ++> may be used to send ascii output to a file. They are similar to the listing operators described above, except that the file is not created afresh. Instead, Db opens the output file for appending more data.

Example. This query would append data from **loans** to the file **printout**

loans ++> printout

5.8.6. Printf Output To An Ascii File

One can call upon the standard C library procedure *printf* when listing tuples in ascii. This feature gives the user a great deal of power in formatting listed output. The user may choose an output format very different to the default method of printing used by Db. Printf is described fully in the UNIX Programmer's Manual.

This facility can be use with any of the four ascii listing operators described above. On the same side of the operator as the output file name, one may also incorporate a double quoted string. This string is then used as the format argument to repeated calls of *printf* to list successive tuples to the file.

The number of fields one requests *printf* to list must be no more than the degree of the output. The types of the output domains must match the conversion specifications in the format string. Thus, the conversion **%s** would be used to list a ***string*** domain, **%D** for a *long* integer, and so on.

Output produced by *printf* is not governed by the global printing flags described in section 5.2. Nor are any headers listed when using this facility.

Unfortunately, because *printf* is a standard C function, it cannot list packed decimal fields.

Example. One could list the entire contents of relation **people** into the output file **peopleout** according to some format string. Such a query and its output in **peopleout** might be -

peopleout "Name: %s, %s * Address: %s *** Personno: %d\n" <== people**

```
Name: Fischer, Denise *** Address: Biedermannsdorf *** Personno: 5
Name: Francis, Michael *** Address: Vienna *** Personno: 6
Name: Mednieks, Zig *** Address: Laxenburg *** Personno: 3
Name: Medow, Serge *** Address: Baden *** Personno: 2
Name: Novachkov, Asen *** Address: Vienna *** Personno: 7
Name: Schweeger, Bernhard *** Address: Klosterneuburg *** Personno: 4
Name: Ward, Robert *** Address: Vienna *** Personno: 1
```

Example. Similarly, one could list relation **data** in some special format. One might wish to list the floating point fields in a width of, say, six characters, with two after the decimal point. Such a query and its printed output in the file **datalist** might be -

data ==> "Event = %d, Data values = %6.2f, %6.2f\n" datalist

```
Event = 1, Data values =   3.40,   4.50
Event = 2, Data values =  -3.60,   6.80
Event = 3, Data values =   0.00,   3.20
Event = 4, Data values =   2.90,   6.70
Event = 5, Data values =  10.10,   9.80
```

5.8.7. Stdin And Stdout

Stdin and **stdout** are two special keywords used to force Db to read data from the standard input or to write relational data or an ascii listing to the standard output.

The keyword **stdin** may be used anywhere inside a query in place of a relation name. It tells Db to read a relation from the standard input. Thus relational data, created by a program such as Dbcreate, can be piped to the standard input of Db. (When Db is reading a relation from a pipe, the relation should normally be a heap without any overflow pages. Otherwise, Db may be unable to read the relation.)

When **stdout** is used with one of the <= or => operators, relational data is sent to the standard output instead of a named relation. Thus relational data formed by Db can be piped to a specialised user program to be processed further.

Stdout can also be used with any of the ascii listing operators <== ==> <++ or ++>. In this case, printed listings go to the standard output. One can also use the name **stdout** in place of a file name to make Db send output, formatted by ***printf*** , to the standard output.

Example. The C Shell command

dbcreate -s stdout -like payroll | db stdin

invokes the Dbcreate and Db programs simultaneously. Data in relational format is piped from Dbcreate to Db. The keyword **stdin** makes Db read a relation from its standard input, in this case the piped data. Db would then simply print all the data given to it by Dbcreate.

Example. This query makes Db send its output, in relational format, to the standard output for further processing.

```
stdout <= books ** loans
```

Example. One may use both **stdin** and **stdout** at the same time. For instance, three programs - Dbcreate, Db and a special user program - could be linked together by pipes in this C Shell command -

```
dbcreate -s stdout -like loans | db 'stdout <= stdin %% datedue' | a.out
```

Db uses a projection to remove all domains except **datedue** from the relation read from the standard input. In turn, Db sends this relational output to be read by the specialised program **a.out**. This user program would make use of routines in the Access Methods library to read the data produced by Db.

Example. This query would make Db print the **data** relation according to the given *printf* format. The listing goes to the standard output.

```
data ==> "Event = %d, Data values = %6.2f, %6.2f\n" stdout
```

5.8.8. Relational Assignments Are Expressions

Any of the relational or ascii output operators may appear in part of a more complicated relational expression. These operators may be thought of as filters that divert output into a file or to the standard output. They all yield a result that is the same as their operand. Thus, the result from any one of these operators may be further processed in the same command.

When any of the assignment operators described above appear in a query, Db normally suppresses any listing to the standard output. However, one may explicitly instruct Db to print to the standard output by using the keyword **stdout**. Thus output can be diverted into a file as well as being listed on the standard output. One must then tell Db to send output to both destinations.

Example. This query produces both a listing of **loans** on the standard output and a copy of the relation in **newloans**.

```
newloans <= stdout <== loans
```

Example. This query copies a part of **books** to **subset1** and a part of **loans** to **subset2**. These smaller relations are then joined together and the final output is placed into **final**.

```
final <= ( subset1 <= books :: bookno >= 6 )
         **
         ( subset2 <= loans :: bookno <= 13 )
```

The output placed into **subset1** is -

author	bookno	title
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls

The output in **subset2** becomes -

bookno	personno	datedue
5	1	811231
6	1	811231
7	1	820107
8	3	820110
13	2	820106

The output in **final** is then -

author	bookno	title	personno	datedue
Hardy	6	Tess Of The D'Urbervilles	1	811231
Hardy	7	Jude The Obscure	1	820107
Hardy	8	Far From The Madding Crowd	3	820110
Lawrence	13	The Virgin And The Gypsy	2	820106

5.8.9. Interleaved Output

Db can send multiple streams of ascii output to the same file. This is useful when one is aggregating a relation. For instance, one can list tuples from a relation with a total of their domains produced at the bottom of the output.

Tuples are interleaved in the output only when there are no intermediate sorts between the listing operations.

Example. One could list all tuples from relation **data**, with the total of domains **data1** and **data2** produced at the end of the output. The second line of the following query tells Db to list the source tuples. They are then aggregated and printed according to the printf format on the last line -

```
data
==> stdout
@@ data1 = 0.0 data1 += .data1 data2 = 0.0 data2 += .data2
==> stdout "Total of data1 is %6.2f, Total of data2 is %6.2f\n" ;
```

The output is -

```
event              data1              data2
   1                 3.4                4.5
   2                -3.6                6.8
   3               0.001                3.2
   4                 2.9                6.7
   5                10.1                9.8
Total of data1 is  12.80, Total of data2 is  31.00
```

Example. One could expand the example of section 5.7.5 so that Db produces a list of tuples with their aggregation listed after each group. The query is -

```
ranges ^^ data
:: low <= event && event <= high
==> stdout
@@ low high
  total1 = 0.0 total1 += .data1
  total2 = 0.0 total2 += .data2
==> stdout "Low = %d,High = %d,Total1=%6.2f,Total2=%6.2f\n"
```

The third line of the query forces Db to list each tuple before it is aggregated. The final line prints the aggregated tuples according to the given printf format. The output is interleaved, with the result of each aggregation appearing after the corresponding group of tuples. The output is -

```
low  high  description      event  data1    data2
  1    3   One To Three       1      3.4      4.5
  1    3   One To Three       2     -3.6      6.8
  1    3   One To Three       3   0.001       3.2
Low = 1,High = 3,Total1= -0.20,Total2= 14.50
  1    5   Everything         1      3.4      4.5
  1    5   Everything         2     -3.6      6.8
  1    5   Everything         3   0.001       3.2
  1    5   Everything         4      2.9      6.7
  1    5   Everything         5     10.1      9.8
Low = 1,High = 5,Total1= 12.80,Total2= 31.00
  2    4   Two To Four        2     -3.6      6.8
  2    4   Two To Four        3   0.001       3.2
  2    4   Two To Four        4      2.9      6.7
Low = 2,High = 4,Total1= -0.70,Total2= 16.70
  3    5   Three To Five      3   0.001       3.2
  3    5   Three To Five      4      2.9      6.7
  3    5   Three To Five      5     10.1      9.8
Low = 3,High = 5,Total1= 13.00,Total2= 19.70
```

5.9. Identifiers Containing Special Characters

Identifiers, or names, are used in Db to signify relations or domains. By default, a name should start with a letter or the underscore character (_) and continue with a sequence of letters, digits or underscores.

Sometimes is is necessary to have file names containing other characters, for instance a slash (/). An alternative way of specifying a name is to surround any sequence of characters with single quotes (′).

Characters within a single quoted identifier that are preceded by a backslash have the same special meaning that they would have in a double quoted string.

If one is presenting a query to Db in an argument list, one must ensure that Db "sees" the single quotes. The C Shell strips away the quotes unless they are preceded by a backslash.

Example. In order to create a new relation in the file **/tmp/relations/newpeople** one could use a query such as -

′/tmp/relations/newpeople′ <= people

If this query were presented to Db on the command line, then one would have to precede the quotes by a backslash. Otherwise the C Shell would remove the quotes and they would not be passed on to Db. The following C Shell command would be suitable -

db \′/tmp/relations/newpeople\′ \<= people

5.10. Keywords

The following identifiers have a special meaning to Db. Preferably, they should not be used as the names of relations or domains.

char short unsigned long int float double pack15 pack31 stdin stdout

If it is necessary to use one of these keywords in the context of a relation or domain name, the special meaning may be suppressed by enclosing the identifier in single quotes (′).

Example. One might wish to list a domain **stdout** from a relation **float**. This query is suitable -

```
'float' %% 'stdout'
```

5.11. More Than One Command In A Query

One may present Db with a sequence of commands in one query. Each command is separated by a semi-colon (;). The commands are executed one after the other. An error detected at any stage prevents any further processing.

Example. The example shown in section 5.8.8 could have been written as -

```
subset1 <= books :: bookno >= 6 ;
subset2 <= loans :: bookno <= 13 ;
final <= subset1 ** subset2 ;
```

Example. One might wish to view relations **books** and **loans** and then the join of these two. This query is suitable -

```
books ; loans ; books ** loans
```

5.11.1. Macros And Temporary Relations

There is one last assignment operator (=) that is used to assign a relational expression to a name. Wherever that name appears in subsequent commands, it is expanded as a macro to become the full relational expression.

One may think of this feature as being an assignment to a temporary relation that is thrown away once the entire query is executed.

Macros can improve the readability of a query. They are generally more efficient than using a real relation to hold temporary results. An intermediate real relation must usually be written out to a disc file and then read back in again. Macros avoid this inefficiency.

However, the fact that this operator represents a macro implies that Db must evaluate the expression as many times as it is subsequently used. One should choose carefully whether to use this macro facility or to create a real relation that can be removed once the query is finished.

Example. The query shown in section 5.7.5 could have been written as -

```
a = ranges ^^ data ;
b = a :: low <= event && event <= high ;
b @@ low high description
    total1 = 0.0 total1 += .data1
    total2 = 0.0 total2 += .data2
```

Example. Similarly, the query of section 5.7.6 could have been written as -

```
a = books .. loans %% author title out = "Yes" ;
b = books -- loans %% author title out = "No" ;
c = a ++ b ;
c ## author title ;
```

5.11.2. Shell Commands

Db can call upon a Shell to execute some UNIX command from a query. The syntax to execute a shell is an exclamation mark (!) followed by a command enclosed in double quotes. This feature can be used, say, to delete unwanted temporary relations, or to echo a comment from a query.

Db tries to invoke one's usual shell program to execute the command. This works as follows : if the **SHELL** variable is set in the user's environment, then Db calls upon that shell to execute the given command. If it is not set, the standard UNIX Shell (*Sh*) is used. One may use the C Shell command *printenv* to determine whether the **SHELL** variable is set.

A non-zero return code from the shell halts any further processing by Db.

In an effort to conserve the number of files it opens, Db may close the standard input. If one wishes to start up an interactive shell from a query, one should direct the input to be from the terminal.

Example. In order to introduce a comment before a listing of a relation one could use a query such as -

! "echo This Is A Complete Listing Of Relation Data" ; Data

Example. In order to start up an interactive C Shell from a query, one would use the command -

! "csh </dev/tty" ;

5.12. Pre-Processor Input

When Db reads input from a terminal or from a file, the query is sent firstly through the C pre-processor. This pre-processor allows one to define macros more advanced than with the simple facility outlined above. The C pre-processor is explained fully in Kernighan/Ritchie (1978).

Macros defined in pre-processor statements can be used to improve the syntax of a Db query. For instance, one may find the pre-processor useful to define a suitable form of an aggregation syntax, that can then be used in subsequent statements.

The pre-processor can include other files into a query. For instance, one can make several global definitions in one file that is then shared amongst several queries. In this way, common definitions can be kept in one central place.

Db accepts options **-D**, **-U** and **-I** as do the UNIX C compiler and *lint*. There may be any number of these options in an argument list to Db.

A **-D** option defines a name to the pre-processor, just as if one had used a '# define' statement in a query. Therefore one may make external definitions to a query. An option of the form

-DNAME=definition

tells the pre-processor to define **NAME** as **definition**. An option of the form

-DNAME

defines **NAME** as 1.

The **-U** option removes any initial definition of a name. It is similar to an '# undef' pre-processor statement.

The **-I** option informs Db of a directory in which to search for any '# include' files. For instance, the option

-I/tmp/dbdefs

tells the pre-processor to look for '# include' files in the directory **/tmp/dbdefs.**

Db also accepts a **-E** option. This runs only the pre-processor. Thus one can view how the pre-processor transforms a query, instead of it being executed by Db.

Example. One might wish to improve upon the standard Db syntax for aggregations. One could use a '# define' statement in order to have a definition for totaling a floating point domain. In order to total up domains **data1** and **data2** of relation **data**, one could use this query -

```
# define    SUMUP( dom )    dom = 0.0    dom += .dom
data @@ SUMUP( data1 ) SUMUP( data2 )
```

One can see how the pre-processor expands this query by invoking Db with its **-E** option. The output would be -

```
data @@ data1 = 0.0 data1 += .data1 data2 = 0.0 data2 += .data2
```

Example. One could extend this definition of **SUMUP** so that it could be used with any data type of field. In this example, an extra parameter is given to **SUMUP** so that the initialisation gives the correct data type to the output domain. For instance -

```
# define    SUMUP( dom, type )    dom = (type)0  dom += .dom
data @@ SUMUP( event, short ) SUMUP( data1, float ) SUMUP( data2, float )
```

Example. In a similar fashion, one can form a definition to extract the minimum value of any domain. Such a definition and its usage is -

```
# define    MIN( dom )    dom = .dom dom = dom < .dom ? dom : .dom
data @@ MIN( data1 ) MIN( data2 )
```

Example. One can use the pre-processor to read from a file before continuing with the rest of the query. Definitions such as **SUMUP** and **MIN** could be kept in a file **defines.db** and then accessed in a query by the pre-processor statement -

```
# include    "defines.db"
```

Example. One might have a query, say, that creates a temporary real relation during its evaluation. Normally, this temporary relation would be deleted once the query has finished running. However, when debugging the query, it might be preferable to keep the relation afterwards. By using the following query, one could choose whether to keep or delete the relation -

```
# ifndef    DEBUG
! "rm temprel" ;
# endif
```

When debugging the query one invokes Db by the command -

```
db -DDEBUG
```

This defines the name **DEBUG** to the pre-processor. After the query has been debugged, one invokes Db without this option. Db would then call upon a shell to execute the **rm temprel** command.

Example. A query might normally use a default input relation. Sometimes, though, one might wish to use a different source relation. The pre-processor could set up a standard default. For example, one can set up the default relation to be **people** by including the following lines -

```
# ifndef    INPUT
# define    INPUT    people
# endif
```

When the query is run, any occurences of **INPUT** are automatically replaced by **people.** However, when Db is invoked by the command -

```
db -DINPUT=newpeople
```

the pre-processer replaces any references to **INPUT** by **newpeople.**

5.13. Comments ○

Any text surrounded by /* and */ is ignored within a query.

On the implementation of Db at IIASA, any text after // and up to the end of a line is also ignored.

Example.

```
/*
** This is an example of a comment in a query
*/

// This is another comment
```

5.14. Historical Syntax ○

For historical reasons, there is an alternative syntax for the union, intersection, difference and join operators. In some circumstances, one may use the single character operators + . - or * respectively. As a word of warning, these operators have an undefined precedence. They are archaic and so should not be used.

6. Dbedx - Interactive Screen Editor For Relations

The program Dbedx allows one to use the screen editor *edx* to edit a relation. Alternatively, one may use Dbedx to append new tuples to a relation or to delete tuples from a relation. The screen editor *edx* is described in Pearson (1980).

A relation is locked while being edited. Only one person may edit a relation at any time.

There are three modes of using Dbedx. The default mode is 'edit and replace'.

6.1. Edit And Replace Mode

In this mode, one uses *edx* to edit specified tuples in a relation. The modified tuples are put back into the relation and replace their original versions.

Tuples are edited one at a time. A 'template' of domain names is listed down the left-hand side of the screen and each field appears on a separate line.

The user invokes Dbedx with a command similar to a selection expression of Db. Only those tuples that match the expression are edited so that one can specify exactly which tuples are to be edited from a large relation. For example the C Shell command -

dbedx books :: bookno == 4

tells Dbedx to edit only tuples from **books** where the domain **bookno** has a value of four. The selection expression to Dbedx must usually be carefully quoted so that the C Shell does not impose its own interpretation on special characters. For instance, in order to edit those tuples from **books** where the **author** field is 'Austen', say, one would use a C Shell command such as

dbedx 'books :: author == "Austen"'

Another possibility would be the command -

dbedx books :: author == \"Austen\"

Once a tuple is displayed on the screen, one may make any required changes to the fields. The template, however, must be left undisturbed. In order to replace the current tuple into the relation and move on to the next, one types **escape w**. The edited tuple is placed back into the relation and the next tuple is displayed.

Dbedx refuses to move onto the next tuple if there are any detectable mistakes in the edited version or if the template has somehow been changed. The user must reedit the tuple correctly before Dbedx will accept it.

Once all the matching tuples have been edited, Dbedx prompts with a message

Overwrite (y/n) ?

A reply of **y** tells Dbedx to overwrite the original relation with all the edited changes. A reply of **n** tells Dbedx to leave the original relation unchanged as though no editing had been done. Any changes are then discarded. Therefore if one has made some mistake in editing a relation, one still has a chance to retrieve back the original data.

6.1.1. Leaving Dbedx

One has the option of leaving the Dbedx program before all the matching tuples have been edited. Instead of typing **escape w** after a tuple had been changed, one may type **escape q**. Dbedx prompts with the message

Continue (y/n/d) ?

A reply of **n** puts back the edited tuple and then finishes the Dbedx session. This gives the user a chance to finish editing before all the matching tuples have been shown. One must then reply to the "Overwrite (y/n)?" question.

A reply of **d** deletes the displayed tuple from the relation. Dbedx then continues with the remainder of the matching tuples. Thus one can step through a relation, deleting any unwanted tuples.

A reply of **y** places the tuple back into the relation just as if the user had initially typed **escape w**. Dbedx then continues with the next tuple.

Example. Using the example of relation **books** again, one might wish to edit those tuples where the **author** field is 'Hardy'. The C Shell command is -

dbedx 'books :: author == "Hardy"'

Dbedx displays the first tuple on the screen. This appears as -

```
|author ............... Hardy
|bookno ............... 6
|title ................ Tess Of The D'Urbervilles
```

Example. One must always provide Dbedx with a selection expression even if one wishes to edit all the tuples in a relation. Thus to edit all tuples in relation **data**, say, the command is -

dbedx data :: 1

The first image on the screen appears as -

```
|event ................ 1
|data1 ................ 3.4
|data2 ................ 4.5
```

6.2. Append Mode

Dbedx can be used to append new tuples to a relation. The user invokes Dbedx with a **-a** , for "append", option. Dbedx displays a blank template down the left hand side of the screen. One can fill in the template with some field values and then type **escape w** to insert the tuple. The tuple will be placed into the relation and a new blank template will be drawn.

To leave Dbedx, one types **escape q** after a tuple has been completed. Dbedx responds with the

Continue (y/n/d) ?

question and the user replies **y**, **n** or **d**. A reply of **y** causes Dbedx to display a new blank template. A reply of **n** terminates the session and a reply of **d** deletes the new tuple.

Again, Dbedx will complain if a tuple cannot be read properly. The user must reedit the tuple and then retype **escape w** before Dbedx will move onto the next.

Once the user decides to leave Dbedx by typing **escape q** , he must answer the

Overwrite (y/n) ?

question. A reply of **y** causes all the new tuples to be placed into the new relation. A reply of **n** discards the new tuples and leaves the original relation untouched.

One may use the *control- c control- z* feature of *edx* to reproduce lines on the terminal. This can save one from having to reenter many similar tuples over again. The first tuple can be entered and then be repeated down the screen. One can then make small changes to each tuple and insert them all together by typing a single **escape w**. This also applies when one is using 'edit and replace' mode to change existing tuples.

Example. The C Shell command

dbedx -a people

allows one to add new tuples to **people.** Dbedx draws a blank template on the screen. This appears as -

```
|lname .................
|fname .................
|address ...............
|personno ..............
```

One then fills in the template. For instance, it could be completed so that the screen appears as -

```
|lname ................. Godwin-Toby
|fname ................. William
|address ............... Vienna
|personno .............. 8
```

One can now make *edx* move to the top of the screen and duplicate the first four domains by typing

control-g control-c 4 control-z control-x

The screen then becomes -

```
|lname ................ Godwin-Toby
|fname ................ William
|address .............. Vienna
|personno ............. 8
|lname ................ Godwin-Toby
|fname ................ William
|address .............. Vienna
|personno ............. 8
```

One could make some small change to the second tuple and insert both together by typing **escape w**.

6.3. Delete Mode

Dbedx can also be used to delete tuples from a relation. One invokes Dbedx with a **-D** , for "delete", flag. One must give Dbedx a selection expression, just as when using it in 'edit and replace' mode. Only those tuples that match the expression are removed. This option does not use the *edx* editor. Instead Dbedx just reports how many tuples would be deleted from the relation. The user must then respond to the

Overwrite (y/n) ?

question before Dbedx destroys the tuples.

Example. In order to remove all those tuples from **books** whose **author** field is 'Lawrence', the command is -

dbedx -D 'books :: author == "Lawrence"'

Dbedx then informs the user how many such tuples there are -

```
dbedx - 3 tuple(s) deleted

dbedx - Overwrite (y/n) ?
```

One would then reply **y** or **n** according to whether the tuples are to vanish forever.

Example. In order to delete entries from **people** where the **address** field is 'Vienna', one could use the *regex* procedure to locate these tuples. Such a command is -

dbedx -D 'people :: regex("Vienna", address)'

6.4. Possible Problems With Dbedx

Unfortunately one must go through the tuples in sequence. There is no possibility to go back and reedit a tuple. Another difficulty is that Dbedx copies the entire relation into the user's current directory before any editing occurs. It is this copied relation that is edited, not the original. This operation can be very time consuming for editing a large relation. Copying the file does at least give one a chance to throw away any mistaken changes and so preserve the original relation.

Internally within Dbedx, binary 03 and 04 bytes have a special meaning and so Dbedx cannot edit a relation which contains these characters. In practice, this technicality presents no problem.

One must use the backslash convention of Dbcreate and Db to enter a string field containing new-line characters (see section 5.4.5). Fields containing such special characters are displayed using the same convention.

Space left by replacing or deleting tuples is not reclaimed. Moreover, Dbls may report incorrect values for the minimum and maximum valued fields of each domain. Both these problems can be corrected by cleaning up the relation with Dbmodify.

7. Dbmodify - Changing The Internal Storage Structure Of A Relation

The program Dbmodify is used to change the internal storage structure of a relation. It reclaims unused disc storage left by deleting or replacing tuples with Dbedx. Dbmodify can also restore the sorting order of a relation after it has been disrupted by appending tuples with Db, Dbappend or Dbedx.

By resorting a relation, Dbmodify ensures that later access by Db is optimised. Db can sometimes locate selected tuples from a large relation without having to read the whole file.

7.1. Arguments To Dbmodify

By default, Dbmodify sorts an entire relation into order, freeing any unused disc space as it does so. The first argument to Dbmodify is the name of a relation. Subsequent arguments inform it on which domains the relation is to be sorted.

7.2. Preserving Duplicate Tuples

Dbmodify normally does not include any duplicated tuples in the modified relation. A **-k** or **-heap** option instructs Dbmodify to preserve duplicate tuples.

Example. The **books** relation was created in section 2.5. The sorting order of this relation was disrupted by adding new tuples (see section 3). Presently, the entire relation is -

author	bookno	title
Austen	1	Persuasion
Austen	4	Emma
Austen	5	Pride And Prejudice
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Hardy	11	The Mayor Of Casterbridge
Shaw	2	Pygmalion
Zola	3	Nana
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls

One can permanently modify the order of this relation by the C Shell command -

```
dbmodify books bookno
```

This instructs Dbmodify to sort **books** on the domain **bookno**. Afterwards, one can view the relation by the command -

```
db books
```

The resulting relation is -

author	bookno	title
Austen	1	Persuasion
Shaw	2	Pygmalion
Zola	3	Nana
Austen	4	Emma
Austen	5	Pride And Prejudice
Hardy	6	Tess Of The D'Urbervilles
Hardy	7	Jude The Obscure
Hardy	8	Far From The Madding Crowd
Eliot	9	The Mill On The Floss
Eliot	10	Silas Marner
Hardy	11	The Mayor Of Casterbridge
Lawrence	12	Women In Love
Lawrence	13	The Virgin And The Gypsy
Lawrence	14	Sons And Lovers
Hemingway	15	For Whom The Bell Tolls

Example. Similarly, the order of tuples in relation **people** can be permanently changed by the command -

dbmodify people personno

lname	fname	address	personno
Ward	Robert	Vienna	1
Medow	Serge	Baden	2
Mednieks	Zig	Laxenburg	3
Schweeger	Bernhard	Klosterneuburg	4
Fischer	Denise	Biedermannsdorf	5
Francis	Michael	Vienna	6
Novachkov	Asen	Vienna	7

7.3. Storage Modes

A storage mode reflects the way in which the tuples are ordered inside a relation. Any relation can have one of three different storage modes. A storage mode should be chosen carefully when a relation is being formed by Dbcreate or cleaned up by Dbmodify. Db can locate selected tuples very quickly if the correct mode has been chosen.

A *heap* storage mode is perhaps the most useful for small relations or for relations which are accessed infrequently. The tuples in a heap relation are contained in a random order. No optimisation is possible to locate tuples within a heap since no ordering information is kept. A heap relation is formed by using the **-heap** flag of Dbcreate or Dbmodify. Although it may be more inefficient for Db or other programs to access such a relation, it is faster to create or modify one since no sorting is necessary.

In general, a *sort* storage mode is the most appropriate and hence this is the default for Dbcreate and Dbmodify. Db can locate selected tuples very efficiently if one has used a selection expression requesting tuples that match some given qualification. Db looks at the selection expression and decides whether it can avoid reading the entire relation from disc. If possible Db uses a binary search strategy to locate the requested tuples. For this to happen, the relation must have been sorted on the key domain whose name appears in the qualification. If the selection has been specified for some other domain, it is still necessary to read the whole relation.

Hash relations are formed by a **-hash** flag to Dbcreate or Dbmodify. Although the tuples appear to be in a random order, Db can locate specific ones very swiftly provided that all the key values have been specified and that they all are exact equalities. When forming a hash relation, one can also specify a fill-factor. A **-f** option followed by a floating-point number indicates the factor by which one eventually expects the relation to grow in size. Sufficient space is reserved for the file : new tuples can be appended without disturbing the storage structure.

The key domains of sorted or hashed relations should be chosen on the anticipated type of access to those relations. Although this may appear to be an unnecessary difficulty, doing so can help Db to locate specific tuples. This is especially important when Db examines large relations.

Example.

dbmodify -heap books

This command modifies **books** so that it becomes a heap relation. Db cannot optimise access to such a relation and the entire file must be read in whenever it is processed. However this modification would reclaim any unused disc space.

Example. The following command sorts **people** on domain **personno**-

dbmodify people personno

A query such as -

people :: persono == 7

or -

people :: personno <= 9 && lname == "Ward"

enables Db to find the requested tuples very quickly. However a query such as -

people :: fname == "Robert"

still forces Db to read the whole relation since the qualification is not in terms of **personno.**

Example. A command to sort **books** firstly by **author** and then by **title** is -

dbmodify books author title

This relation becomes -

author	bookno	title
Austen	4	Emma
Austen	1	Persuasion
Austen	5	Pride And Prejudice
Eliot	10	Silas Marner
Eliot	9	The Mill On The Floss
Hardy	8	Far From The Madding Crowd
Hardy	7	Jude The Obscure
Hardy	6	Tess Of The D'Urbervilles
Hardy	11	The Mayor Of Casterbridge
Hemingway	15	For Whom The Bell Tolls
Lawrence	14	Sons And Lovers
Lawrence	13	The Virgin And The Gypsy
Lawrence	12	Women In Love
Shaw	2	Pygmalion
Zola	3	Nana

Dbls informs one of how a relation is structured. A mode of *Sort* shows the relation has been marked as sorted. The order of the sort is shown underneath the column *Key*. A value of 1 here indicates that the sort is firstly on **author.** The value 2 in the *Key* column shows that the secondary sort key is **title.** The output would be -

```
       Mode Tuples Deg Pages Fix Max  Fl Modified              Size
books  Sort    15    3 1+ 0   12  50 VU Jan  7 18:25 1982     2048
               Domain Ty Key  Fl Print Offs Fix Max Smallest Largest
                author S   1 VA     9    2        9
                bookno s             6    6   2              1     15
                 title S   2 VA    26    8       26
```

This storage structure would be useful for Db queries that qualify tuples by the domain **author**. For instance -

books :: author == "Austen"

or -

books :: regex("Madding", title) && author <= "Hardy"

Example. A hash storage structure is suitable when one wishes to retrieve tuples given all their key values. For example, the command -

dbmodify -hash -f3 people lname fname

modifies **people** so that it can be accessed quickly by Db when given values of both **lname** and **fname**. The **-f3** flag tells Dbmodify that the relation is eventually expected to grow in size by a factor of three. Again, Dbls would report the keyed domains of this relation -

```
       Mode Tuples Deg Pages Fix Max Fl Modified              Size
books  Hash     7    4 3+ 0   16  54 V  Jan  7 18:25 1982    4096
                 Domain Ty Key Fl Print Offs Fix Max Smallest Largest
                  lname S   1  V     9    2        9
                  fname S   2  V     8    6        8
                address S           15   10       15
               personno s            8   14   2              1       7
```

This storage mode would be useful for the query -

people :: lname == "Ward" && fname == "Robert"

However it would not be suitable for a query such as -

people :: lname == "Ward"

or -

people :: lname <= "Ward" && fname == "Robert"

In the first case only one key domain has been specified. All the key domains need to be qualified so that Db can quickly locate specific tuples from a hashed relation. In the last example, both domains have been specified but they need to be exact equalities for a hashed relation.

7.4. When Optimised Access Is Possible

As a word of warning, optimised access to a relation is only possible when a key domain is being compared to a specific value. It is not possible to optimise access to tuples selected by procedures such as *regex*.

Example. This query makes a selection according to a specific value. In this example, the required tuples could be found efficiently.

books :: author == "Zola"

Example. The following query requires the entire relation to be read in -

books :: regex("Zola", author)

7.5. Appending Tuples To A Relation

Because there is no ordering information for heap relations, new tuples are placed at the end of a relation file.

For sorted relations, tuples are placed as closely as possible to where they should go according to the sorting order. However no existing tuples are ever moved to make room for the new ones. Adding many new tuples to a relation will eventually destroy the original order of the relation. A similar slow destruction of the internal format also happens to some extent with hashed relations.

This implies that the strategy employed by Db to locate specific tuples eventually becomes degraded. When new tuples are added to a relation, the sorting order is slowly disrupted. Db is then forced to read in more of a relation file than would otherwise be necessary.

It also means that appending yet more tuples becomes slower since programs such as Db or Dbappend must read more of a relation in order to determine where each new tuple should be placed.

7.6. Primary And Overflow Pages

When a relation is created, or cleaned up by Dbmodify, all the tuples are placed into blocks of disc storage termed *primary pages*. Afterwards, when further tuples are added, the primary pages become full and the additional tuples are put on to *overflow pages*. These are the two figures reported under the *Pages* column of Dbls. When the number of overflow pages becomes large compared with the number of primary pages, the strategy used by Db or other programs to locate specific tuples becomes less efficient. If this happens, it is time for the relation to be reformatted by Dbmodify.

7.7. When To Use Dbmodify

When extensive modifications have been made to a relation by Dbedx, it is advisable to use Dbmodify to reclaim disc space left by deleted or changed tuples.

Because the Db programs always attempt to find the most suitable place to insert a new tuple in a sorted or hashed relation, it can sometimes be useful to modify it to a heap format beforehand. This speeds up insertion of new tuples since neither Dbappend nor Db need search the file to see where each tuple should be placed. New tuples simply go at the end of the file. Once the tuples have been successfully appended to the relation, Dbmodify could then be used again to return it back to its original format.

7.8. Secondary Indices

As described above, Dbmodify can form a sorted or hashed relation so that accessing tuples by a particular key value is very efficient. Sometimes it may be desirable to access a large relation by different key domains.

When one wishes to examine a relation on other key values one can construct a *secondary index*. This is small table placed into the relation file that enables Db to access tuples on other key domains very quickly. A relation may have any number of secondary indices. Db looks at a relation to see if there are any suitable indices by which to improve access.

A secondary index is formed by a -i flag to Dbmodify.

Secondary indices become useful for large relations, for instance relations larger than about 100 kbytes. Each secondary index is updated when new tuples are added or deleted. Unfortunately, when Dbmodify reformats the primary relation itself, it destroys these indices and they must then be regenerated.

Example. One may usually wish to access tuples from **books** given values of **bookno** as keys. This command structures the relation so that Db quickly finds these tuples -

dbmodify books bookno

Sometimes, though, it might also be desirable to find tuples when given values of **author.** It would then be useful to construct a secondary index for this domain. Such a command is -

dbmodify -i books author

8. Practical Usage

This chapter discusses the more practical aspects of using Db. When maintaining a large relational database, it may be important to understand how to make a Db query more efficient. Possible problems that may occur are also mentioned.

8.1. Restrictions

Few restrictions are imposed by the Db programs. This section comments on the present limitations.

Domain names are limited to a maximum of twenty characters. Each domain within a single relation must have a unique name. There is no limit on the number of domains in any relation.

There is no inherent limit on the size of a relation file. A relation may grow to any length, given that there is sufficient disc space and that no other external constraints apply.

Although there is no maximum length imposed on the length of any field, there is a restriction on the total size of each tuple. No tuple may have a length exceeding 1000 bytes. In practice this limit is generous and should not be reached.

This imposition also applies to any tuples generated internally by Db. For instance a projection could possibly create very wide tuples.

The complexity of any command in a query is limited by the maximum number of open files permitted by the system. Db sometimes uses real temporary relations for sorting purposes and so the maximum number of real relations that can be accessed is usually less than this number.

8.2. Optimising Dbcreate

By default, Dbcreate sorts the input data into order. Dbcreate may be faster if the **-heap** option is specified so that it does not have to sort the input data.

8.3. Optimising Dbappend

Dbappend is faster when appending tuples to a heap relation since the order of new tuples is then unimportant. It does not have to search the relation to decide where each new tuple should be placed.

If one is appending tuples to a sorted or hashed relation, Dbappend can be made faster by specifying the **-k** option so that it does not have to search the relation for duplicated tuples.

8.4. Optimising Access To A Relation

The purpose of Dbmodify to format a relation so that Db can quickly access specified tuples has already been mentioned. This issue was discussed in section 7.3.

8.5. Optimisation Of A Db Query

A Db query may generally be written in one of several different ways, each one producing the same output. Some methods will be more efficient than others. Db does not rearrange a query to make it run faster but instead executes a query exactly as it is written. In terms of efficiency, it can be useful to know how best to formulate a query.

8.5.1. Optimisation Of Selections

Whenever possible, one should use a *selection* as the first step in a query to limit the quantity of processed tuples. The alternative, to process all the input tuples somehow and then select the required ones, is less efficient.

It is usually more efficient to select tuples by specifying exact values for their keyed domains, rather than by using procedures such as *regex*.

Example. This query selects specified tuples and then sorts them into order -

books :: author == "Hardy" ## title

This is a more efficient query than the following, which sorts the entire relation before selecting the required tuples -

books ## title :: author == "Hardy"

8.5.2. Optimisation Of Projections

Projections should usually occur as late as possible in a query. A projection is typically used to form new domains in the output. The flow of data may be minimised by avoiding the creation of extra domains until they are needed. Processing time may be further reduced by combining successive projections together into one step.

Example. This query forms new domains **sum** and **diff** from the relation **data.** The output is sorted by **data1.**

data ## data1 %% event data1 data2 sum = data1 + data2 diff = data1 - data2

This is a more efficient query than the following, that creates the extra domains before sorting them into order. The volume of data to be sorted is therefore increased.

data %% event data1 data2 sum = data1 + data2 diff = data1 - data2 ## data1

Example. An efficient query for forming the domains **sum** and **diff** from relation **data** is -

data %% event data1 data2 sum = data1 + data2 diff = data1 - data2

This is a more efficient method than the following which uses two projections to form the same output -

data %% event data1 data2 sum = data1 + data2
%% event data1 data2 sum diff = data1 - data2

8.5.3. Optimisation Of Sorting

The *sort* operator is frequently used to order listed output, to remove duplicate tuples from a relation, or to ensure that a subsequent aggregation receives tuples in order. However, sorting a large relation can be a slow operation and should be avoided unless necessary. The output from a *join*, *intersection* or *difference* operation is always sorted in ascending order of the domains used for matching the two relations. There is then no point in explicitly resorting the output on the same domains.

Example. One might wish to view the join of **loans** and **people** and sort the output in order of domain **personno.** In this case, the common domain used for matching tuples from both relations is **personno** and so there is no need for explicitly sorting the output. A suitable query is -

loans ** people

and the output is -

bookno	personno	datedue	lname	fname	address
6	1	811231	Ward	Robert	Vienna
5	1	811231	Ward	Robert	Vienna
7	1	820107	Ward	Robert	Vienna
14	2	820106	Medow	Serge	Baden
13	2	820106	Medow	Serge	Baden
8	3	820110	Mednieks	Zig	Laxenburg
15	7	810423	Novachkov	Asen	Vienna

8.5.4. Optimisation Of Temporary Relations

Using the macro feature to form a "temporary" relation is generally more efficient than using a real disc relation to hold an intermediate result. One must be careful, however, that the macro is not expanded more than once. Moreover, the maximum number of open files permitted by the system cannot be exceeded.

Example. In order to repeat the third example of section 5.7.6, an equivalent query is -

```
a = people ** loans @@ lname fname borrowed += 1 ;
b = people — loans %% lname fname borrowed = 0 ;
c = a ++ b ;
c ## lname fname ;
```

The following example is less efficient : the query uses real relations to hold the temporary results. In this case, real files **tmp1**, **tmp2** and **tmp3** are written out and then read back in again. Using the macro facility avoids this inefficiency.

```
tmp1 <= people ** loans @@ lname fname borrowed += 1 ;
tmp2 <= people — loans %% lname fname borrowed = 0 ;
tmp3 <= tmp1 ++ tmp2 ;
tmp3 ## lname fname ;
! "rm tmp1 tmp2 tmp3" ;
```

One may sometimes be able to avoid rereading a relation file. This is possible by using the output from a list or creation operator, instead of discarding it.

Example. One might wish to form a listed output of **people** on the standard output, and a copy of this relation in **pcopy**. An efficient way of producing both printed output and the copied relation is -

```
pcopy <= stdout <== people
```

A less efficient method of achieving the same result is to read the **people** relation twice. Such a way is -

```
pcopy <= people ;
stdout <== people ;
```

8.5.5. Optimisation Of The Product Operator

Because of the way the *product* operator is implemented, it may be more efficient if the right hand operand is the smaller of the two relations.

Example. Given two relations, **large** and **small**, the following query will usually be more efficient than if the operands were reversed -

```
large ^^ small
```

8.6. Measuring Disc Read And Write Operations

When one of the Dbcreate, Dbappend, Dbedx or Dbmodify programs is invoked with a **-p** option, the program reports the number of disc read and write operations before completion.

If Db is invoked with this option, it reports the number of disc reads and writes after each command in a query. It also reports the quantity of processing steps applied to each tuple.

These figures, especially those derived from Db, may be useful to locate inefficiencies in processing.

Example. The following message might be produced by Dbmodify when invoked with its **-p** option. It indicates that Dbmodify read in four and wrote out two disc blocks during processing.

```
dbmodify - 4 Page Reads, 2 Page Writes
```

Example. The following message might be produced by Db after listing the relation **books** -

```
db - 1 Page Reads, 0 Page Writes, 31 Coroutine Calls
```

In this example, Db has read only one disc block. Thirty-one tuples have been processed. This figure is derived as follows : 15 tuples have been read in, 15 tuples have been printed and a final end-of-file tuple has been processed.

8.7. Common Problems When Using Db

This section details the most common errors that occur when using Db.

For each command in a query, Db performs whatever checking is required and then attempts to execute that command. Unless noted otherwise, an error at any stage will generally halt processing.

In terms of efficiency, it is usually preferable to use the macro facility to form temporary results. Doing so, however, may make Db try to open more relation files than the system will allow. This is especially true if Db has to perform an intermediate sort on the data, in which case it uses one or more real temporary files. The solution is to split up the processing by using a real relation file to hold temporary results. Unfortunately, the number of real files needed for sorting purposes may be dependent on the data and cannot always be predicted in advance.

Db complains if one tries to concatenate together two relations of different degrees. It also objects if any domains of the two relations cannot be concatenated because their data types are incompatible.

There must be at least one common domain between the two relations for a join, intersection or difference operation. Should there be more than one commonly named domain, tuples are considered to match only if ALL their common domains have identical values. The data types of the these matching domains must allow them to be comparable.

Conversely, the operands of a Cartesian product must have no commonly named domains.

The sort operator will generally sort a relation only on the specified domains. The order of any other domains is unpredictable. Even if tuples have been previously sorted in a specific way, any original ordering is likely to be disrupted unless one instructs Db otherwise. (This also applies when using Dbmodify to sort a relation.)

Db usually permits scalar expressions involving mixed data types and converts the type of an expression if necessary. String values, however, are not converted to a numeric type unless one explicitly uses one of the procedures

stol, **stod** or **stop31**. Packed decimal values can only be converted to one another or to *long* integers. A conversion to a floating point type can happen only through an explicit intermediate conversion to a *long* integer. The bit operators all require *short* or *unsigned short* operands. An integer constant must be cast to one of these types if it is to be used as an operand.

An invalid argument to a scalar procedure will usually generate a diagnostic but processing will continue.

Arithmetic interrupts, for instance a division by zero, are trapped and generate a diagnostic. Db continues if possible, but some exceptions cannot be recovered and so halt the program.

Should an I/O error occur - for instance a disc file system might become full while Db is producing a new relation - Db will halt processing.

8.8. The PDP-11/70 Implementation

The versions of the Db programs running on the VAX-11/780 and the PDP-11/70 are almost identical in terms of their functionality. One difference is that the PDP does not support packed decimal integers.

The VAX computer has a much larger main memory space than the PDP. The Db programs make use of this potential memory area in order to optimise sorting of relations. Therefore a query run on the VAX version of Db tends to execute faster than on the PDP version.

When Db is processing a query on the PDP, it may occasionally run out of necessary main memory. This problem can usually be solved if one uses real files to hold temporary results and so break up a complicated command into smaller units. It is also helpful if one is processing relations whose degree is small, since Db no longer needs to allocate internal information to process extra domains.

9. Conclusion

This chapter lists some subjective thoughts concerning the Db relational database management system.

9.1. Current Applications For Db

At the time of writing, Db has been fully operational for almost twelve months. It is being used to manage several databases at IIASA.

One application has been implemented for the Personnel Department at IIASA. This database is used to maintain information concerning employees at the Institute. Since most people who maintain this database are unfamiliar with the UNIX system, a C Shell command processor is used to interface between the DBMS and the users. This command processor hides many details of the database implementation from those who maintain the data. The database is updated frequently by using Dbedx to change or add new information.

Another application has been formed for the Resources And Environment Task at IIASA. A database has been constructed to store large quantities of experimental data. This database currently has a size of over 1 megabyte and contains relations with a cardinality of over 60000 tuples. The typical queries presented to Db are not complex but involve processing joins over such large relations. The database is eventually expected to grow to a size of perhaps 10 megabytes.

A third example of its current use at IIASA is an application for the Budget And Finance Department. Db maintains a database of monetary transactions on a set of accounts. It is used to calculate the tax returns on these accounts and to perform various totaling operations on the data. For this application the cardinality of the input data is small (approximately 2000 new tuples per month). However, the queries that have been developed for the tax calculations and for evaluating the movements on each account are complicated. Despite the small quantity of input data, Db may generate over one million internal tuples during processing. Because of the financial nature of the task, it is essential that all arithmetic processing be accurate. Therefore packed decimal arithmetic is used throughout.

9.2. The Design Objectives Of Db

A primary goal for the design of Db was to integrate it as well as possible into the existing framework of the UNIX operating system. The Db system does not attempt to create a new environment for the user. Unlike some relational database systems, notably Ingres (Stonebraker et al. 1976), Db does not impose a new world of commands on to the user. Nor does it duplicate features already present within UNIX. For instance, Db has no special commands to print a relation or to delete a relation from a database.

There is no notion of a database catalogue that must be updated whenever a relation is changed. Instead, all information concerning a relation is contained within one file. This simplifies the implementation, since there is no need to duplicate existing facilities. For instance, relations can be protected by the standard UNIX commands *chmod* and *chown*.

One of the major goals of the Db implementation has been efficiency of processing. The response time to a query and the resources used by a DBMS depend partly on how well it interfaces with the data on disc. Using an efficient set of procedures to access disc data improves performance. The routines developed for the Db programs allow fast access to disc data and ensure that it is kept in a compact format.

How a DBMS processes the data once it has been retrieved from disc also affects performance. The Db query processor takes advantage of a pipe-line algorithm to transform or to combine data. This pipe-line ensures that temporary results need not normally be written out to an intermediate disc file.

9.3. Further Improvements To The Db Programs

For the proposed applications at IIASA, it was necessary that Db should run on both a VAX-11/780 and a PDP-11/70. The implementation has therefore been written almost entirely in portable C. The only exception is a few lines of assembler code in the VAX version that provide an interface to the machine's packed decimal instruction set. One or two frequently called procedures could be made faster by rewriting them in assembler language. This might well benefit the overall execution time of typical queries.

It is doubtful whether the access methods library could otherwise be improved. The algorithms used to locate or store tuples in a relation appear to be reasonably fast. The sorting algorithm that is invoked when Db must join, intersect, or find the difference of two relations, has been heavily optimised. It does, however, suffer on the PDP version where there is a lack of main memory space.

To improve performance, Db should recognise whether a new relation being formed is sorted in any way. If so, then the new relation should be marked as being sorted. Presently, all relations formed by Db are heaps. Db should also take advantage of any sorting order in relations when joining them together. At the moment, any relations to be joined are resorted, despite any initial ordering they may have. These two points may be corrected in the future.

Some improvements could be made to the Db query processor. For instance, it would perhaps be beneficial if one could explicitly state the matching domains for a join, intersection or difference operation. The ability to define one's own relational procedures in a query might also be useful.

It would be of benefit if there were some means by which a user's code could be directly executed on relational data. Presently, one must use the access methods library to retrieve data from a relation if one wishes to apply some specialised routines. One should be able to compose a routine in C or in Fortran that could then be compiled and the resulting object module called directly from Db.

9.4. Overview

Db is currently being used to maintain several databases each of which has its own special requirements. The fact that the Db family of programs may be integrated into a UNIX environment has facilitated construction of these databases.

Any data processing system can be ultimately judged by how well it can be applied to solving real problems. Despite its limitations, the Db system seems to have been largely successful in this respect.

10. Bibliography

1. C. J. Date,
An Introduction To Database Systems, Second Edition,
Addison-Wesley Publishing Company, Reading, Massachusetts (1977).

2. W. Joy,
An Introduction To The C Shell,
Computer Science Division, Department Of Electrical Engineering And Computer Science, University Of California, Berkeley (1980).

3. B. W. Kernighan and D. M. Ritchie,
The C Programming Language,
Prentice-Hall, Englewood Cliffs, New Jersey (1978).

4. M. M. L. Pearson,
Using The Computer To Communicate : An Introduction To To Text Processing At IIASA - The Edx And Nroff Programs, (Working Paper WP- 80- 111)
IIASA, A-2361 Laxenburg, Austria (July 1980).

5. D. M. Ritchie and K. Thompson,
The UNIX Time- Sharing System,
Communications Of The ACM, Volume 17, No. 7 (July 1974).

6. G. Sandberg,
A Primer On Relational Database Concepts,
IBM Systems Journal, Volume 20, No. 1 (1981).

7. M. Stonebraker, E. Wong, P. Kreps and G. Held,
The Design And Implementation Of Ingres,
ACM Transactions On Database Systems, Volume 1, No. 3 (September 1976).

11. Acknowledgements

I would like to thank James Kulp who suggested many useful design features for Db and who assisted with debugging the system.

I am also very grateful to Carolyn Lathrop and Martina Joestl for their careful and patient assistance in producing this paper.

Appendix I - Summary Of The Available Data Types

The following table summarises the scalar data types supported by the Db programs. For each data type, it shows the type character for Dbcreate and Dbls, the corresponding type for Db, the space it occupies in a relation, and the permitted range of values.

Data Type	**Type Char**	**Db Name**	**Byte Width**	**Range Of Values**
Char integer	c	char	1	-128 to 127
Short integer	s or i	short or int	2	-32768 to 32767
Unsigned integer	u	unsigned or unsigned short	2	0 to 65535
Long integer	l	long	4	-2147483648 to 2147483647
Floating point	f	float	4	$\pm10^{\pm38}$
Double precision	d	double	8	$\pm10^{\pm38}$
Packed decimal	p	pack15	8	$\pm10^{15}-1$
Packed decimal	P	pack31	16	$\pm10^{31}-1$
Character string	S		Variable	

Appendix II - Summary Of The Scalar Operators

The following table summarises the scalar operators available in Db. The table is listed in order of increasing precedence of the operators.

Operator	Purpose	Precedence	Binds	Returns	Remarks
? :	Conditional	1	right		
||	Logical Or	2	left	short	
&&	Logical And	3			
|	Bit Or	4	left	short	Operands must be *short* or *unsigned short*
^	Bit Exclusive Or	5			
&	Bit And	6			
==	Comparison Operators	7	left	short	
!=					
<					
<=					
>					
>=					
<<	Bit Left Shift	8	left	short	Operands must be *short* or *unsigned short*
>>	Bit Right Shift				
+	Addition	9	left	Depends On Operands †	
−	Subtraction				
*	Multiplication	10			
/	Division				
%	Modulus				Operands cannot be *float* or *double*
−	Unary Minus	11	right		
!	Unary Logical Not			short	
~	Unary Bit Complement				Operand must be *short* or *unsigned short*

† The type returned by an arithmetic operator depends on the types of its operands. The following table summarises the types returned by these operators. Any combinations of data types not found in this table are impermissible.

Left Operand	Right Operand	Type Returned
char	char	char
	short	short
	unsigned short	unsigned short
	long	long
	float	float
	double	double
short	char	short
	short	short
	unsigned short	unsigned short
	long	long
	float	float
	double	double
unsigned short	char	unsigned short
	short	unsigned short
	unsigned short	unsigned short
	long	long
	float	float
	double	double
long	char	long
	short	long
	unsigned short	long
	long	long
	float	float
	double	double
	pack15	pack15
	pack31	pack31
float	char	float
	short	float
	unsigned short	float
	long	float
	float	float
	double	double
double	char	double
	short	double
	unsigned short	double
	long	double
	float	double
	double	double
pack15	long	pack15
	pack15	pack15
	pack31	pack31
pack31	long	pack31
	pack15	pack31
	pack31	pack31

Appendix III - Summary Of The Scalar Procedures

The following table summarises the scalar procedures currently incorporated into Db. It describes the arguments for each procedure and their expected types.

Procedure	Returns	Type Returned	Types Expected
regex	1 if pattern #1 matches string #2. 0 if pattern does not match string	short	string, string
strcat	Concatenation of #1 with #2	string	string, string
substr	Substring of #1 beginning at position #2 (origin of 1) and of length #3	string	string, unsigned, unsigned
strlen	Length of string #1	unsigned	string
count	For 1st call sets and returns internal counter with #1. Afterwards increments and returns counter	long	long
stol	Equivalent *long* integer of string #1	long	string
stod	Equivalent *double* of string #1	double	string
stop31	Equivalent 31 digit packed decimal value of string #1	pack31	string
dt_days	Number of days between two dates #1 and #2	long	long, long
dt_ltos	String of form "1981X12X31" where X is #2	string	long, string
dt_month	Month name of date #1	string	long
dt_offset	New date from date #1 and offset in days #2	long	long, long
dt_pretty	Pretty format of date #1	string	long

Procedure	Returns	Type Returned	Types Expected
dt_stol	Long integer date from string #1 in format "811231" or "81X12X31" where X is any single character	long	string
dt_sundays	# inclusive Sundays between two dates #1 and #2	long	long, long
dt_today	Date on which Db is invoked	long	
dt_u3tol	Equivalent long integer date from year #1, month #2 and day #3	long	unsigned, unsigned, unsigned
dt_vpretty	Very pretty format of date #1	string	long
dt_weekday	Name of weekday of date #1	string	long

Appendix IV - Summary Of The Relational Operators

The following table summarises the relational operators implemented in Db. The table is listed in order of increasing precedence of the operators. It also lists the section in which each operator is described.

Operator	Purpose	Precedence	Binds	Section
<==	Ascii Create	1	right	5.8.4
<++	Ascii Append			5.8.5
<=	Relation Create			5.8.1
<+	Relation Append			5.8.2
==>	Ascii Create	2	left	5.8.4
++>	Ascii Append			5.8.5
=>	Relation Create			5.8.1
+>	Relation Append			5.8.2
::	Selection	3	left	5.3.1
%%	Projection			5.3.2
@@	Aggregation			5.6
#	Sort (removing duplicate tuples)			5.3.5
##	Sort (keeping duplicate tuples)			5.3.4
++	Union			5.7.1
..	Intersection			5.7.2
--	Difference			5.7.3
**	Join			5.7.4
^^	Cartesian Product			5.7.5

Appendix V - Summary Of The Syntax Of The Db Query Language

This appendix formally describes the syntax of the query language accepted by Db. This language is described downwards from the highest level syntactic construction recognised by Db.

Symbols appearing down the left hand margin represent the names of non-terminal symbols being defined. Their corresponding definitions appear on the right hand side. Some definitions are recursive. Multiple definitions appear on successive lines. For instance, a *commandlist* is defined as either a *commanditem* or as a *commandlist* followed by a *commanditem*.

Syntactic items in *italic* type represent non-terminal symbols. Items in **bold** type represent terminal symbols.

The syntax definitions of constants are not given here.

program :	*command*
	commandlist command
commandlist :	*commanditem*
	commandlist commanditem
commanditem :	*command* **;**
command :	*rel_exp*
	rel_macro
	shell
	commanditem
shell :	**!** *const_string*
rel_macro :	*identifier* **=** *rel_exp*
rel_exp :	*rel_primary*
	selection
	projection
	aggregation
	join
	intersection
	union
	difference
	product
	sort
	rel_print
rel_print :	*identifier* **<+** *rel_exp*
	rel_lvalue **<=** *rel_exp*
	rel_plvalue **<==** *rel_exp*
	rel_plvalue **<++** *rel_exp*
	rel_exp **+>** *identifier*
	rel_exp **=>** *rel_lvalue*
	rel_exp **==>** *rel_plvalue*
	rel_exp **++>** *rel_plvalue*

rel_primary :	*relation*
	(*rel_exp*)
relation :	*identifier*
	stdin
rel_lvalue :	*identifier*
	stdout
rel_plvalue :	*rel_lvalue*
	const_string rel_lvalue
	rel_lvalue const_string
selection :	*rel_exp* :: *scalar_exp*
	rel_exp : *scalar_exp*
projection :	*rel_exp* **%%** *projlist*
	rel_exp **%** *projlist*
projlist :	*projitem*
	projlist projitem
projitem :	*projdomain*
	identifier = *scalar_exp*
	identifier projassign scalar_exp
projassign :	+=
	-=
	*=
	/=
	%=
	<<=
	>>=
	&=
	|=
	^=
aggregation :	*rel_exp* @@ *projlist*
	rel_exp @ *projlist*
join :	*rel_exp* ** *rel_exp*
	rel_exp * *rel_exp*
product :	*rel_exp* ^^ *rel_exp*
	rel_exp ^ *rel_exp*
intersection :	*rel_exp* .. *rel_exp*
	rel_exp . *rel_exp*
union :	*rel_exp* ++ *rel_exp*
	rel_exp + *rel_exp*
difference :	*rel_exp* — *rel_exp*
	rel_exp - *rel_exp*

```
sort :                  rel_exp ## sortlist
                        rel_exp # sortlist

sortlist :              sortitem
                        sortlist sortitem

sortitem :              identifier
                        - identifier

projdomain :            identifier
                        . identifier

scalar_exp :            scalar
                        scalar_exp < scalar_exp
                        scalar_exp <= scalar_exp
                        scalar_exp > scalar_exp
                        scalar_exp >= scalar_exp
                        scalar_exp == scalar_exp
                        scalar_exp != scalar_exp
                        scalar_exp || scalar_exp
                        scalar_exp && scalar_exp
                        scalar_exp ^ scalar_exp
                        scalar_exp | scalar_exp
                        scalar_exp & scalar_exp
                        scalar_exp << scalar_exp
                        scalar_exp >> scalar_exp
                        scalar_exp + scalar_exp
                        scalar_exp - scalar_exp
                        scalar_exp * scalar_exp
                        scalar_exp / scalar_exp
                        scalar_exp % scalar_exp
                        scalar_exp ? scalar_exp : scalar_exp

scalar :                scalar_primary
                        ( cast ) scalar
                        - scalar
                        ! scalar
                        ~ scalar
                        scalar_function
                        ( scalar_exp )

scalar_function :       identifier ( )
                        identifier ( arglist )

arglist :               argitem
                        arglist , argitem

argitem :               scalar_exp

cast :                  char
                        short
                        unsigned
                        unsigned short
                        long
```

```
                          pack15
                          pack31
                          int
                          float
                          double

scalar_primary :          projdomain
                          constant

constant :                const_long
                          const_double
                          const_string
```

Appendix VI - A Demonstration Session Of The Db Programs

The following text demonstrates an example session using the Db programs. Text printed by the computer is shown in roman type : text typed by the user is shown in **bold** type.

Firstly, two new relations **german** and **french** are created. The former relates English and German nouns, the latter relates English and French nouns. In each case the input data is read from the user's terminal and is terminated by a control-D on the final line.

The program Dbcreate is used to form a new relation from the input data. In the first example, relation **german** is formed. Input fields are delimited by the @ character. The **-heap** option tells Dbcreate not to sort the input data and to keep any duplicated tuples. The **-p** option instructs it to report the number of tuples placed into the new relation.

```
% dbcreate german english/S article_g/S german/S -df@ -heap -p
>garden@das@Garten@
>shirt@der@Hemd@
>umbrella@der@Regenschirm@
>cupboard@der@Schrank@
>girl@das@Maedchen@
>horse@das@Pferd@
>bridge@die@Bruecke@
>book@das@Buch@
>cow@die@Kuh@
>^D
dbcreate - 9 Tuples Inserted, 1 Primary Pages Written
dbcreate - 0 Page Reads, 1 Page Writes
```

Dbcreate is again invoked to form a relation of English and French nouns. Because there is no **-heap** option, it sorts the input data into order. Any duplicated tuples would be removed.

```
% dbcreate french english/S article_f/S french/S -df/
>garden/la/jardin/
>umbrella/la/parapluie/
>shirt/le/chemise/
>cupboard/le/bord/
>woman/la/femme/
>bridge/le/pont/
>book/le/livre/
>horse/le/cheval/
>house/la/maison/
>girl/la/fille/
>^D
```

One can now use Db to view both relations. Db is invoked and waits for the user to type a query on the terminal. The input is terminated by a control-D on the final line.

```
% db
german ; french
^D
```

```
english    article_g  german
garden     das        Garten
shirt      der        Hemd
umbrella   der        Regenschirm
cupboard   der        Schrank
girl       das        Maedchen
horse      das        Pferd
bridge     die        Bruecke
book       das        Buch
cow        die        Kuh

english    article_f  french
book       le         livre
bridge     le         pont
cupboard   le         bord
garden     la         jardin
girl       la         fille
horse      le         cheval
house      la         maison
shirt      le         chemise
umbrella   la         parapluie
woman      la         femme
```

Dbls supplies information about the data in these relations -

```
% dbls german french
          Mode   Tuples  Deg  Pages  Domains
french    Sort        9    3   1+ 0  english/S article_f/S french/S
German    Heap       10    3   1+ 0  english/S article_g/S german/S
```

In order to find all feminine German nouns, one uses the following query. When a query is given to Db in an argument list, it is generally advisable to enclose the query within single quotes.

```
% db 'german :: article_g == "die"'
```

```
english   article_g   german
bridge    die         Bruecke
cow       die         Kuh
```

To find those French nouns for which there is no stored German equivalent, one uses this query -

```
% db 'french — german'
```

```
english   article_f   french
house     la          maison
woman     la          femme
```

One might wish to form the join of **french** and **german**. The output is formed from tuples in both relations which have the same values in their commonly named domains. In this example, the common domain is **english**. Tuples from either relation that do not match any from the other are not listed.

```
% db 'german ** french'
```

```
english    article_g   german        article_f   french
book       das         Buch          le          livre
bridge     die         Bruecke       le          pont
cupboard   der         Schrank       le          bord
garden     das         Garten        la          jardin
girl       das         Maedchen      la          fille
horse      das         Pferd         le          cheval
shirt      der         Hemd          le          chemise
umbrella   der         Regenschirm   la          parapluie
```

In order to form a full cross-reference listing of both relations, with a '?' substituted for any unknown entries, one uses the following query. The first line of the query forms a relation of tuples from both input relations that match one another. The second line forms tuples from **german** that have no corresponding entries in **french**. The third line forms tuples from **french** that do not match any in **german**. The final line concatenates the three sets of tuples together and sorts them in order of domain **english**.

```
% db
a = german ** french ;
b = german — french %% english article_g german article_f = "?" french = "?" ;
c = french — german %% english article_g = "?" german = "?" article_f french ;
a ++ b ++ c ## english ;
^D
```

english	article_g	german	article_f	french
book	das	Buch	le	livre
bridge	die	Bruecke	le	pont
cow	die	Kuh	?	?
cupboard	der	Schrank	le	bord
garden	das	Garten	la	jardin
girl	das	Maedchen	la	fille
horse	das	Pferd	le	cheval
house	?	?	la	maison
shirt	der	Hemd	le	chemise
umbrella	der	Regenschirm	la	parapluie
woman	?	?	la	femme

One could add more tuples to **german** using the program Dbappend. Each input field is delimited by the @ character.

```
% dbappend german -df@
>house@das@Haus@
>woman@die@Frau@
>mouse@das@Maus@
^D
```

The updated relation now becomes -
% db german

english	article_g	german
garden	das	Garten
shirt	der	Hemd
umbrella	der	Regenschirm
cupboard	der	Schrank
girl	das	Maedchen
horse	das	Pferd
bridge	die	Bruecke
book	das	Buch
cow	die	Kuh
house	das	Haus
woman	die	Frau
mouse	das	Maus

One could make a permanent modification to the order of tuples in **german** using the program Dbmodify. The following command instructs Dbmodify to sort **german** in ascending order of the domain **english**.
% dbmodify german english
% db german

english	article_g	german
book	das	Buch
bridge	die	Bruecke
cow	die	Kuh
cupboard	der	Schrank
garden	das	Garten
girl	das	Maedchen
horse	das	Pferd
house	das	Haus
mouse	das	Maus
shirt	der	Hemd
umbrella	der	Regenschirm
woman	die	Frau