Technical Report

UCAM-CL-TR-1
ISSN 1476-2986

Number 1

# The JACKDAW database package

M.F. Challis

October 1974

ISSN 1476-2986

TECHNICAL REPORT No. 1.

# THE JACKDAW DATABASE PACKAGE

by

M. F. Challis

University of Cambridge Computing Service

Series Editor:

M.F.Challis
University of Cambridge
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG
England

October 1974

## Summary

This report describes a general database package which has been implemented in BCPL on an IBM 370/165 at the University of Cambridge. One current application is the provision of an administrative database for the Computing Service.

Entries within a database may include (in addition to primitive fields such as 'salary' or 'address') links to other entries: each link represents a relationship between two entries and is always two-way.

Generality is achieved by including within each database class definitions which define the structure of the entries within it; these definitions may be interrogated by program.

The major part of the package presents a procedural interface between an applications program and an existing database, enabling entries and their fields to be created, interrogated, updated and deleted. The creation of a new database (or modification of an existing one) by specifying the class definitions is handled by a separate program.

The first part of the report describes the database structure and this is followed by an illustration of the procedural interface. Finally, some of the implementation techniques used to ensure integrity of the database are described.

M. F. Challis
October 1974

## CONTENTS

## 1 Introduction

This report describes a general database package which has been implemented on an IBM 370/165 at the University of Cambridge.

Entries held within a database maintained by the package may include (in addition to primitive fields such as 'salary' or 'address'), links to other entries: these 'link elements' are the way by which relationships between entries are expressed in the database. A characteristic feature of the JACKDAW package is that all such links are two-way: if entry X is linked to entry Y then the package automatically ensures that Y is linked to X, and we can truly speak of a relationship 'between' the two entries.

Generality is achieved by including within each database a set of 'class definitions' which define the structure of entries within it. The package enables the definitions in an existing database to be altered, so that new kinds of entry and new relationships between entries can be added as a particular database evolves.

The interface between the package and an applications program is a procedural one, and access to entries and their fields and relationships is by name. As a consequence, all applications programs are completely independent of data representations - indeed, additions to entry structure (as described in the paragraph above) may be made without affecting existing programs in any way.

The proper functioning of an applications program, then, depends only on the existence of the entries it accesses and the existence of the class and field names to which it refers; but even these dependences may be avoided. It is, of course, possible to determine whether a particular entry exists, and to access one by one all those entries belonging to a particular class - indeed, few useful applications could be implemented without these facilities. But the package also provides analogous facilities for class definition interrogation: a program may discover whether a particular class or field exists, and may even access and examine each class definition in turn, thus determining by enquiry the structure of all entries in the database. In this way it is possible to write programs which are entirely independent of database structure: for example, general updating, enquiry and listing programs are feasible.

The remainder of this report is divided into four main sections. The first describes the database structure in some detail, and this is followed by an illustration of the procedural interface. The third section discusses some of the implementation techniques used to ensure integrity of the database, and the final section gives some details of a major application.

## 2 Structural Description

### 2.1 Entries and classes

Information is stored in a database within named entries, each of which must be of some 'class'; for example, there might be entries named FRED and BILL (these are 'entry identifiers') which belong to the class STAFF. Information within an entry is held in 'primitive fields' and 'link fields'; each field of an entry is referred to by its 'field identifier'. Primitive fields contain simple values, and link fields specify the relationships which exist between different entries in a database. The number, nature and identifiers of the fields of an entry are determined by the class of that entry: in other words, all entries of one class have the same structure.

### 2.2 Primitive fields

The definition of a primitive field includes the 'type' of value that the field may contain. Four types are allowed: 'word' (any 32-bit quantity), 'string' (variable length of up to 255 characters), 'bool' (a truth value) and 'vector' (a variable length indexable sequence of words).

Values of the appropriate type may be stored in and retrieved from primitive fields of entries and, in addition, existing elements of a vector field may be individually accessed. Whenever a value is stored, the field in that entry is said to be 'set'; it is possible to determine whether a field is set or not, and also to unset (or 'delete') it.

Example:
STAFF entries might include a primitive string field called NAME and a primitive word field STAFFNUMBER.

### 2.3 Link fields

A link field consists of a number (possibly none) of 'link elements', each of which refers to an entry of a particular class; this class is called the 'class' of the link field.

A fundamental property of database structure is that links between entries must always be reciprocal. In detail, this means that if an entry X1 of class C1 contains a link element referring to an entry X2 of class C2, then entry X2 will contain a link element of class C1 referring to X1. A corollary is that if a class C1 contains a link field of class C2, then class C2 must contain a link field of class C1.

Example:
Suppose that members of staff are working on various projects, and that we wish to represent these relationships in a database. We define a new class PROJECT with entries OS, TSO etc. for each project, and then define a link field PROJECTS for the class STAFF and a corresponding link field MEMBERS for the class PROJECT.

If FRED belongs to the OS project, then his PROJECTS field will contain a link element referring to entry OS; if BILL works on both OS and TSO, then his link field will contain two link elements. As a consequence of this state of affairs, the MEMBERS link field in OS will have two link elements referring to the STAFF entries FRED and BILL, whereas TSO's link field will contain only one link element (for BILL). The structure is shown pictorially as:

PROJECT:
OS
TSO
STAFF:
FRED
BILL

The two-way nature of links is automatically maintained by the package, so that the addition of a new MEMBERS link element to a PROJECT entry automatically results in the addition of a new PROJECTS link element to the appropriate STAFF entry. As a consequence, it is as easy to discover the members of staff working on a particular project as it is to determine the projects to which a particular staff member is attached, regardless of the way in which the information was originally recorded.

## 2.4 Marks

It is possible to select one of the link elements within a link field and to 'mark' it; marks are given names ('mark identifiers') so that more than one mark can be associated with the same link field. In the example above, we might define a mark LEADER for the MEMBERS link field of the class PROJECT; the link element so marked would refer to the member of staff who was the project leader.

Marks are strictly unnecessary, as the same effect can be achieved by defining additional link fields, but it is particularly cheap to implement and turns out to be practically of great value.

## 2.5 Link element parameters

It is sometimes necessary to hold information which is associated with a link between two entries rather than with either entry itself. As an example, suppose we have classes CIRCULATIONLIST and SUBSCRIBER with a link between to indicate membership, and we wish to record the number of copies required of each circulation list by each subscriber. This number is not a property of the circulation list or of the subscriber, but belongs to the link itself.

The JACKDAW package provides this facility by allowing primitive fields to be held within link elements: these are called link element parameters. For example, if the link from SUBSCRIBER to CIRCULATIONLIST is called LISTS then a primitive word field COPIES might be defined which would occur in each link element of every LISTS field.

## 3 Creating databases

The program for constructing a new database needs to read in class definitions and translate them into a format suitable for use by the rest of the package. The translation process involves assigning offsets to the various fields and is very similar to the compilation of data structure declarations for a programming language.

A suitable language for expressing class definitions has been designed, and an example is given below:

```
ADD CLASS STAFF (WORD STAFFNUMBER, STRING NAME)
ADD CLASS PROJECT
ADD LINK (PROJECTS, MEMBERS(MARK LEADER)) FROM STAFF TO PROJECT
```

Other commands enable the modification of existing definitions in a database; fields and classes may be renamed, added, or removed:

e.g.
```
AMEND CLASS STAFF
BEGIN
    RENAME NAME AS SURNAME
    DELETE STAFFNUMBER
    ADD STRING ADDRESS
END

AMEND LINK FIELD PROJECTS IN STAFF
BEGIN
    ADD MARK MAJORPROJECT
END
```

Note that certain of these commands (e.g. DELETE) may result in the destruction of fields of entries - or even the entries themselves: to minimise the possibility of accidental information loss, the system demands explicit authorisation before any entry or field in a database is actually destroyed.

## 4 The Procedural Interface

The package proper consists of a set of procedures which may be called upon to create, update, interrogate and delete entries and their fields within a database of fixed structure: procedures are also provided to interrogate the class structure, but no alterations may be made.

The package is written in BCPL (ref [1]), and at present the interface is also BCPL: this is not strictly necessary, but is certainly convenient!

### 4.1 Illustration

The example below (in which the interface procedure calls are underlined) serves to illustrate the way in which database facilities are provided by the interface procedures; its effect is to list the names of those members of staff who are working on project OS, noting which one (if any) is the project leader.

Lines (1) to (5) serve to translate class, field and mark identifiers into values which are to be used as parameters to further interface procedures. These values are more convenient internally than strings and avoid the need for repeated look-up by the package.

In line (6), we locate the PROJECT entry OS and assign a value representing its location to the variable OS. This value remains valid until it is explicitly released (in line (16)), and, while it is valid, the entry OS and all its fields and link elements will be available to the program.

```
$(  LET STAFF = LOOKUPCLASSID("STAFF")                    - (1)
    LET NAME = LOOKUPPRIMID(STAFF, "NAME")                - (2)
    LET PROJECT = LOOKUPCLASSID("PROJECT")                - (3)
    LET MEMBERS = LOOKUPLINKID(PROJECT, "MEMBERS")        - (4)
    LET LEADER = LOOKUPMARKID(MEMBERS, "LEADER")          - (5)

    LET OS = FINDENTRY(PROJECT, "OS")                     - (6)
    LET L = SETUPLINKS(OS, MEMBERS)                       - (7)

    WHILE NEXTLINK(L) DO                                  - (8)
    $(  LET E = REFERENCEDENTRY(L)                        - (9)
        LET WORKSPACE = VEC 64                            -(10)

        WRITES(READSTRING(E, NAME, WORKSPACE))            -(11)
        IF ISMARKEDLINK(L, LEADER) DO                     -(12)
            WRITES("(Project Leader)")                    -(13)
        NEWLINE()                                         -(14)
        RELEASEENTRY(E)   $)                              -(15)

    RELEASEENTRY(OS)   $)                                 -(16)
```

Program 1

In line (7) we locate the MEMBERS link field of this entry; NEXTLINK(L) will now cause L to represent each link element in this field in turn, returning the value TRUE so long as there are any link elements left. Link elements are held in alphabetical order, and so the effect of the loop (8) to (15) is to execute lines (9) to (15) twice: once with L describing the link element referring to BILL, and once for the link element referencing FRED.

In line (9) we locate the STAFF entry referred to by L - say BILL.

In line (11) the call of READSTRING causes the contents of the primitive string field NAME of entry E to be copied to the vector WORKSPACE (declared in line (10)): the address of WORKSPACE is also returned as result so that the library routine WRITES (for Write String) writes BILL's name.

In line (12) we determine whether link element L is marked or not: if BILL is a project leader, line (13) will add "(Project Leader)" after his name.

Line (15) releases entry E before returning to (8), and line (16), executed after all link elements in the field have been processed, finally releases entry OS.

## 4.2 Other procedures

Over sixty interface procedures are defined, most of which are analogous to those in the example: they allow creation and deletion of entries, addition, location and deletion of link elements, setting and unsetting of primitive fields etc.

An important set of procedures not illustrated by program 1 are those which enable the class definitions to be interrogated. Program 2 uses some of these procedures to list all the classes defined in a database.

```
$(  LET WORKSPACE = VEC 64                          - (1)
    LET CLASS = FIRSTCLASS()                        - (2)

    UNTIL CLASS = 0 DO                              - (3)
    $(  WRITES(IDOFCLASS(CLASS, WORKSPACE))         - (4)
        NEWLINE()                                   - (5)
        CLASS := NEXTCLASS(CLASS)     $)     $)     - (6)
```

Program 2

Line (2) sets the variable CLASS to the (translated form of the) first class definition; in line (6), the call of NEXTCLASS returns a value representing the next class definition if one is present, or 0 if not. The loop (3) to (6) is therefore executed with CLASS representing each class definition in turn, and line (4) causes the identifier of each class to be printed.

Given a particular class, analogous procedures enable the identifier and type of each primitive field and the identifier and class of each link field to be determined.

## 5 Implementation topics

### 5.1 Integrity

One of the major design goals of any database package should be that of integrity of the database. By this I mean that the effects of non-malicious program malfunctions and system crashes should be minimised: dealing with deliberate attacks is quite a different matter and outside the scope of this paper.

### 5.2 Applications program errors

Errors in applications programs may result in package code or buffer areas being overwritten or in incorrect parameter values being passed to the package. If code is overwritten, we can expect chaos to ensue rapidly and for the program to terminate abnormally; many internal consistency checks ensure abnormal termination in the case of buffer overwriting as well, and so both these cases can be treated as system crashes. Most parameters are values which were originally provided by the package itself (e.g. PROJECT in line (6) and L in line (8) of program 1) and so can be directly checked; others can be checked to be of the right type (e.g. the second parameter of FINDENTRY must be a string).

If it were possible to install the package (code and data) in an area of core inaccessible to the applications program except by procedure call then most of this checking would be unnecessary: in no case does the package require a program to read directly from or write directly to an area under package control.

### 5.3 System crashes

A database is held as a file on disc, and, as it is processed, portions are paged into core. To guard against system crashes we must ensure that the version on disc is always consistent - that is, if a sequence of mutually dependent updates has to be made then either all or none of them will appear in the disc version. This is achieved as follows.

### 5.4 Disc file maintenance

The database is held on disc as a sequence of 'physical blocks' all of the same size, whereas pointers (or addresses) within the database refer to 'logical blocks'; the first physical block of the datafile contains the mapping from logical to physical blocks. Not all physical blocks are allocated to logical blocks, so that there are always some spare. This is illustrated below, where a logical database of 3 blocks is held in a physical file of 7 blocks:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Physical datafile: | L1->P6<br>L2->P3<br>L3->P1 | L3 | spare | L2 | spare | spare | L1 |

When the datafile is opened, the logical/physical mapping is read into core and is used to translate logical addresses to physical addresses; blocks from disc are then paged into core as necessary.

Whenever a logical block is updated (i.e. when an entry is changed in some way), a new spare physical block is allocated to it, and the changed mapping is noted in core but not on the disc. Since the original physical block is not yet made available for re-use, the disc file still retains the information that describes the original (unaltered) logical database - i.e. the original logical/physical mapping together with the physical blocks described by it - but the core mapping together with its allocated physical blocks describes the new updated logical database. This is illustrated below, where we suppose that logical block 3 has been updated:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Physical datafile: | L1->P6<br>L2->P3<br>L3->P1 | L3 | L3' | L2 | spare | spare | L1 |

| | |
|---|---|
| Core mapping: | L1->P6<br>L2->P3<br>L3->P2 |

The disc mapping and blocks P1, P3 and P6 define the original database, whereas the core mapping and blocks P2, P3 and P6 define the updated database.

At the end of a run or at some other suitable moment (see next section), any altered blocks in core are written back to disc finishing with the new logical/physical mapping itself, thus bringing the disc file up-to-date; we say that the disc file has been 'remade'. We see that the disc file defines the original logical database right up to the moment when the new mapping is written back: provided this block is written successfully, the disc file will thereafter define the new logical database and all physical blocks allocated in the old database mapping but not in the new become available for re-use again:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Physical datafile: | L1->P6<br>L2->P3<br>L3->P2 | spare | L3 | L2 | spare | spare | L1 |

To guard against the remote possibility of the new mapping being only partially written back, a slight modification is made to the scheme described above by arranging that mappings are written alternately to the first two physical blocks of the datafile so that a new mapping never physically obliterates the preceding one. Each mapping is allocated a sequence number one greater than its predecessor, so that we can distinguish between the mappings available in the first two blocks of the file: the current mapping is that with the greater sequence number.

## 5.5 Consistency and indivisibility

By avoiding remaking the disc file during a sequence of mutually dependent updates, we can avoid the possibility of an inconsistent logical database appearing on the disc.

Often we know that only a certain (small) number of blocks will be affected by the series of updates, and so we can reasonably require that sufficient spare physical blocks are available before embarking on the series: if enough are not immediately available we remake the disc file first, hoping that sufficient will be freed by this process. This technique is used when the package creates a new link between two entries: a new link element must be added to each entry, and, since the entries probably lie in different blocks, two blocks will need to be updated.

Sometimes, however, the number of blocks to be updated may be very large, and it becomes unreasonable to insist that a sufficient number of spare blocks be available: for example, deleting an entry will involve changes to all entries to which it is linked. In these cases, the package arranges to record extra information on the disc whenever the file is remade whilst the updating operation is in progress. This extra information enables the operation to be completed when the file is next opened if some failure prevents its completion during the current run.

By judicious application of these two techniques, the database package ensures both the consistency of the database on disc (in the sense that no entries will appear 'half linked' or 'partially' updated) and the 'indivisibility' of each interface procedure (in the sense that any procedure call will either complete its specified action or do nothing at all).

## 6 Application

The major application to date has been the provision of an administrative database for the University Computing Service. Computing resources are allocated to 'projects' and 'fileowners': each project is given a number of shares which controls the rate of working on that project, and each fileowner is given space allocations which control the amount of disc space which he may use. Each person who wishes to use the computing facilities is allocated a 'user identifier' by which he is known to the system and is given access to one or more projects and fileowners to satisfy his computing needs. Thus in the database we have classes USER, PROJECT and FILEOWNER with SHARES and SPACE primitive fields for PROJECTs and FILEOWNERs, and links between USERs and the other two classes.

For accounting purposes, projects are grouped together into departments which are themselves part of a 'tree structure' of superior departments and faculties etc., and further classes and link fields represent this structure.

Other information maintained in the database includes magnetic tapes (each linked to its owner), names and addresses of users, and mailing lists. At present, the database includes approximately 2500 USER entries, 1800 PROJECT entries, 1500 FILEOWNER entries and 2500 TAPE entries, and is about one megabyte in size.

The database is usually interrogated and updated using an interactive program at a terminal, but the same program may be used in batch mode. Other applications programs in use include a general purpose listing program (which allows listings of entries selected according to conditions on both primitive and link field values) and a program which produces user names and addresses on 'sticky labels' for mailing purposes.

The system has been in full operation for about one year with updates taking place almost daily. During this period, many operating system crashes whilst the database is being interactively updated have provided a practical test of the integrity techniques: in fact, although the database is dutifully dumped to magnetic tape once a week, we have not yet needed to back up from such a tape copy.

## 7 Conclusions

The characteristic features of the JACKDAW database package are its automatic reciprocal linking mechanism and its generality.

The linking mechanism allows a relationship between classes to be represented in its most natural way, without favouring either class; new examples of this relationship may then be set up between particular entries in either direction. For example, the interactive program for maintaining the administrative database (see previous section) allows an extra user to be added to a project whilst updating a project entry, or an extra project may be added to a user whilst updating a user entry.

The generality of the package results partly from the ability to interrogate the class definitions and partly from the procedural interface: in the Introduction we showed how these two features affected the dependence of a particular applications program on a particular database structure. Experience has shown that completely general purpose programs are extremely difficult to design: for example, it is very difficult to design an output format for displaying an entry of arbitrary structure. However, useful compromises are attainable in which a program can continue to function over quite large changes in database structure; and completely general programs are always of use, of course, when major structure changes have to be made before 'tailor made' software is completed.

## 8 Acknowledgement

The structural aspects of the JACKDAW package arose from ideas originally put forward by Dr. J. Larmouth as the basis of a project for the provision of an administrative database; I am indebted to him for many helpful discussions during the design of this package.

## Reference

[1] - Richards, M. 'The BCPL Programming Manual', The Computer Laboratory, University of Cambridge, April 1973.