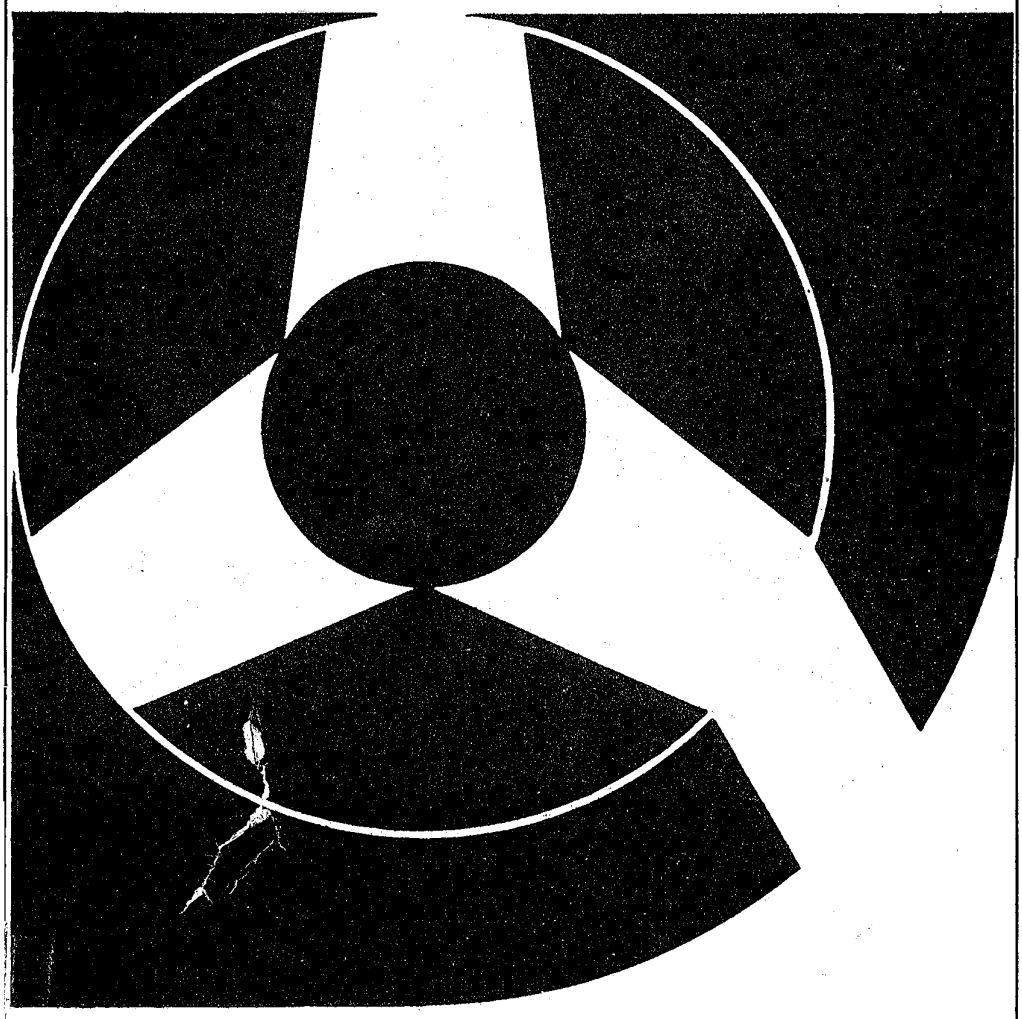
UNIVERSIDADE FEDERAL DE MINAS GERAIS
COMISSÃO DE COORDENAÇÃO DAS ATIVIDADES
DE PROCESSAMENTO ELETRÔNICO-CAPRE

IV SEMINÁRIO SOBRE DESENVOLVIMENTO INTEGRADO DE SOFTWARE E HARDWARE



1977 VOLUME II

UNIVERSIDADE FEDERAL DE MINAS GERAIS
COMISSÃO DE COORDENAÇÃO DAS ATIVIDADES DE PROCESSAMENTO ELETRÔNICO - CAPRE/PNTC

IV SEMINÁRIO SOBRE DESENVOLVIMENTO INTEGRADO DE SOFTWARE E HARDWARE

VOLUME II

ORGANIZAÇÃO:

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO - ICEX-UFMG

CENTRO DE COMPUTAÇÃO DA UFMG

FUNDAÇÃO DO DESENVOLVIMENTO DA PESQUISA (FUNDEP)

COMISSÃO ORGANIZADORA:

PROF. JOSÉ MARCOS SILVA NOGUEIRA

PROF. NEWTON ALBERTO DE CASTILHO LAGES

COMISSÃO TÉCNICA

PROF. DALTRO JOSÉ NUNES

PROF. IVAN DA COSTA MARQUES

PROF. WILSON DE PÁDUA PAULA FILHO

LOCAL:

AUDITÓRIO DA REITORIA DA UFMG

DATA:

25 A 29 DE JULHO DE 1977

"INTEGRITY TECHNIQUES IN THE JACKDAW DATABASE PACKAGE"

Presenter: Michael F. Challis

Author: Michael F. Challis

University: Departamento de Informatica,
Pontificia Universidade Catolica,
Rio de Janeiro

Abstract

The Jackdaw Database Package was designed and developed by the author whilst working at Cambridge University, England, and has now been implemented on the IBM 370/165 at PUC-RJ.

The units of information held in a Jackdaw database are called "entries", and are classified according to their type: for example, a database might contain ROOM and PERSON entries. Basic information (such as the NAME of a PERSON) may be stored in the "primitive fields" of an entry, and relationships between entries (such as the ROOM in which a PERSON works) are represented by pointers held in "link fields" of the related entries.

One of the major applications of Jackdaw at Cambridge was the provision of an administrative database in which information about users of the Computing Service was held. This information included their names, addresses, departments, and allocations of computing resources. The database was updated interactively by the Receptionist and Tape Librarian on a day-to-day basis, and reports were generated in the usual way in batch mode.

One design requirement of paramount importance was that the database itself should not suffer damage in the event of system or application program failures, and various implementation techniques were used to try to achieve this end.

The paper commences with an introduction to the Jackdaw package facilities, which is followed by a detailed description of the techniques used to ensure integrity.

SECTION I

The Jackdaw Database Package

Status

The Jackdaw package is written in a portable software programming language called BCPL [1], and is currently available in Brasil on the IBM 370/165 at PUC-RJ.

The "nucleus" of the system provides a library of "interface procedures" which may be called from BCPL programs to interrogate and update entries in a database. Several general-purpose programs are also available for "non-programmers" to use including an enquiry/update program designed for interactive use, and a report generator for producing formatted listings.

An essentially "network" approach is adopted for the representation of data relationships, although the concepts involved are somewhat more straightforward than those of CODASYL [2].

Introduction

The main body of this paper, Section II, is devoted to the description of a general technique for ensuring the integrity of a database. This technique has been successfully employed by the author in the Jackdaw database package, and the first section provides a brief introduction to this system. Finally, in Section III, some further applications of the technique in a multi-user environment are suggested.

Structural Description

The way in which information may be represented in a Jackdaw database is best illustrated by example - for a more complete account, see [3].

The following "definition-statements" define a database in which information about shops and the items they have for sale is held:

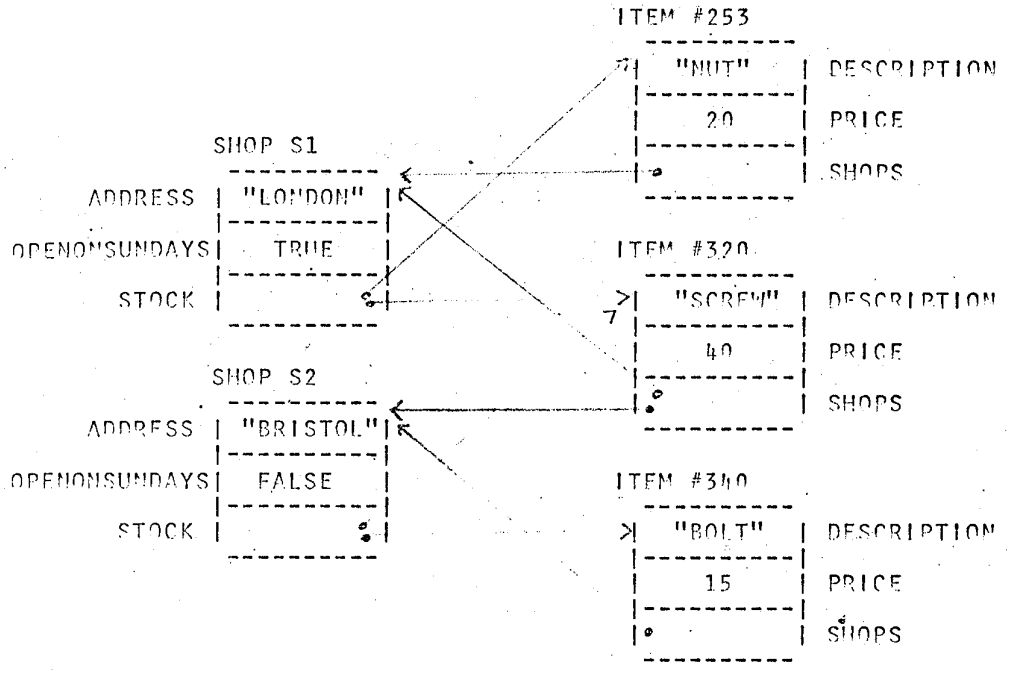
```
ADD CLASS SHOP
BEGIN
  STRING ADDRESS
  BOOL OPENONSUNDAYS
END

ADD CLASS ITEM
BEGIN
  STRING DESCRIPTION
  INT PRICE
END

ADD LINK (SHOPS, STOCK) FROM ITEM TO SHOP
```

The "ADD CLASS" statements define the basic structure of the different kinds of entry that may appear in the database: for example, each SHOP entry has a field called ADDRESS of type STRING and a boolean field called OPENONSUNDAYS. In addition, every entry must have an identifer (unique within its class) by which it is referenced; in the case of ITEM entries this might be the item's code-number (although it could equally well be the name of the item provided that this was unique).

The "ADD LINK" statement defines a two-way relationship between SHOP and ITEM entries. Unlike the CODASYL "set" concept the relationship is many-to-many (see [4]): one item may be stocked by several shops, and one shop may stock several items. The information about which shops stock a particular item X is held in a "link field" called SHOPS in the entry for ITEM X; similarly, the items stocked by shop Y are referred to by the link field called STOCK in SHOP entry Y.

Whenever a new link is created from one entry X to another entry Y, the corresponding link from Y to X is automatically inserted by the package, and so an applications programmer does not need to be concerned about the "direction" of the relationship: he may add ITEMs to the STOCK field of a SHOP entry or add SHOPs to the SHOPS field of an ITEM entry according to whichever is most convenient.

The following diagram illustrates two SHOP entries and three ITEM entries together with relationships between them:



S1 stocks #253 and #320

S2 stocks #320 and #340

We conclude this section with some examples of statements in the enquiry/update language which illustrate how entries and their fields may be manipulated.

Statement	Effect
TYPE SHOP S1 ADDRESS	/prints "LONDON"
TYPE SHOP S1 STOCK	/prints "#253, #320"
ITEM #320 PRICE=60	/to update the price
NEW SHOP S3 (ADDRESS=LEEDS OPENONSUNDAYS=FALSE ITEMS=(#340, #253))	/to create and / initialise / a new SHOP entry
TYPE ITEM #340 SHOPS	/prints "S2, S3"

SECTION II

Database Integrity

Its Importance

A database, especially in an on-line environment, is a very valuable object and it is essential to provide as much protection for it as possible from the effects of faulty applications programs. For by its very nature a database is central to many separate applications, and a database corrupted by one program will probably not be useable by any other. This is in contrast to the "classical" data processing environment where a corrupt master file only delays applications dependent on that file.

Another important difference is in the manner of updating. In the classical situation, a new master file is usually a modified copy of the original, which may be reinstated if anything goes wrong during the (batch) update run. A database, however, will be updated "in-situ", possibly by many applications simultaneously, and it is usually a difficult if not impossible task to "back-up" both the database and all other applications in the event that one application should corrupt the database. (This situation is similar to that of a multi-programmed operating system where failure of one user program must not affect other, independent users).

For the purposes of the Jackdaw system, a corrupt database is defined as one which cannot be processed correctly by the package: examples of corruption would be inconsistent indexes, or a relationship between two entries represented by a link in one direction only.

The techniques described below to ensure integrity efficiently protect a Jackdaw database against faulty applications programs and against operating system failures, but do not attempt to solve the problem of deliberate attacks by malicious users.

Applications Programs Errors

A Jackdaw database is held as a sequence of fixed size blocks on disc, which are read into core, updated and written back to disc as necessary. In the current implementation (as a standard user program on the IBM 360/370 series) package code, buffers and application program (AP) code all share the same region of core store, and so errors in the AP may result in either package code or buffers being overwritten. In the former case, we can expect that the package will abnormally terminate if the corrupted code is ever exercised, and so the net effect is similar to that of an operating system crash. The latter case is more difficult to deal with, but in practice it appears that simple internal checks carried out on buffers before they are written back to disc are adequate. (For example, each buffer includes its own block number which is checked for accuracy before the buffer is written back).

The possibilty of overwriting buffers is also minimised by never providing the AP with pointers to areas within the buffers. Indeed, there is never any need for an AP to write or read directly from a buffer area, since all information passed between the AP and the database is always copied to or from an AP-supplied data area by the package itself. Addresses for such data transfers are of course checked by the package to ensure that they lie within the AP's part of the region.

Other kinds of parameter passed from the AP to the package can be directly checked, since they are values which were previously created by the package. For example, the following three Jackdaw interface procedures might be called to read the value of the PRICE field of the ITEM entry "#340":

```
X := FINDENTRY(ITEM, "#340")
N := READWORD(X, PRICE)
RELEASEENTRY(X)
```

The value X, used to represent the located entry "#340", is the address of a small data-structure created by the package to describe the location and structure of the entry and to enable subsequent references to fields of the entry to be processed rapidly. Thus the package knows that the first parameter of "READWORD" must be the address of such a data-structure and so can easily check its validity.

It is worth noting that if the package code and buffers were held in a separate region of store, inaccessible to the AP except by procedure call, then much of the checking described above would not be necessary. Unfortunately, OS/MVT does not provide facilities of this kind for normal user programs.

System Crashes

The case of a system-wide failure differs from that of an AP failure in that it is not possible for the package to "trap" the failure and try to tidy up afterwards, and so protection of a database against such failures cannot depend on knowledge of what took place immediately prior to the crash. Indeed, the only version of the database in existence after a system crash will be that on disc, and so the solution to the integrity problem is to ensure that the database on disc is always consistent.

Physical and Logical Databases

The disc file containing a Jackdaw database is organised as a sequence of fixed-size blocks (which may be accessed at random) numbered from 0 to some maximum, say MP. This is called the "physical database".

The "logical database" is organised in a similar way with blocks numbered from 0 to ML (<MP), and a mapping from logical to physical block number describes where the blocks of the logical database appear within the physical database.

Pointers within the database (representing link fields, for example) are held as logical block number/offset pairs, and the major part of the package code operates in terms of such "logical addresses": only the lowest level (where paging and buffer allocation is performed) needs to be aware of physical addresses, and a clean interface separates this from the rest of the package.

The logical-to-physical (LP) mapping is itself held in the physical database in such a way that it can be read into core without knowledge of itself: the first block of the mapping is held in a standard location in the physical file, and itself contains the physical block number of the next mapping block, and so on.

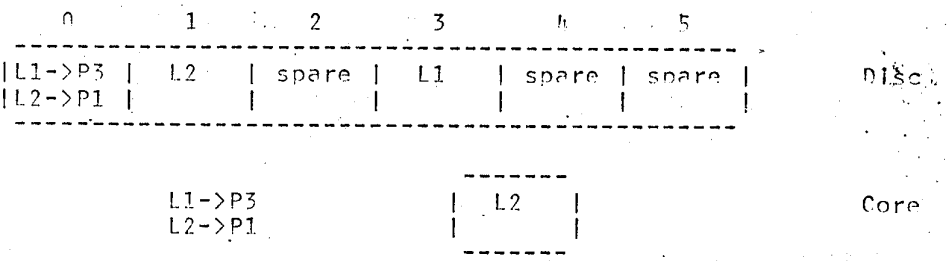
The Updating Process

When the database is first opened, the LP mapping is read into core and used to translate logical addresses to physical addresses as requests are made to read logical blocks into core buffers. When one of these buffers is first updated a spare physical block is allocated to the corresponding logical block and the LP mapping in core is updated to reflect this change. Thus when it becomes necessary to write the buffer back to disc, it will go to a new location in the physical database instead of overwriting the original version of the corresponding logical block. The effect is that the physical database now contains two logical databases: the original one described by the LP mapping on disc, and the updated one described by the LP mapping in core. At some suitable moment

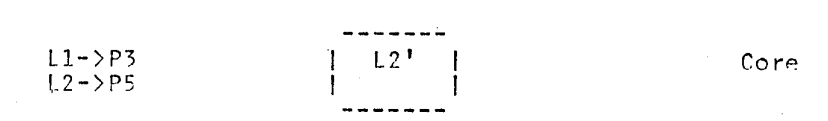
(see next section) the LP mapping in core is written back to disc so that the updated logical database is now described by both the disc and core mappings. If we ensure that only LP mappings describing consistent logical databases are written to disc, then only consistent databases will be defined by the physical database which is thereby protected from the effects of arbitrary termination of application programs.

The following example illustrates this technique for a logical database of three blocks held in a physical database of six blocks:

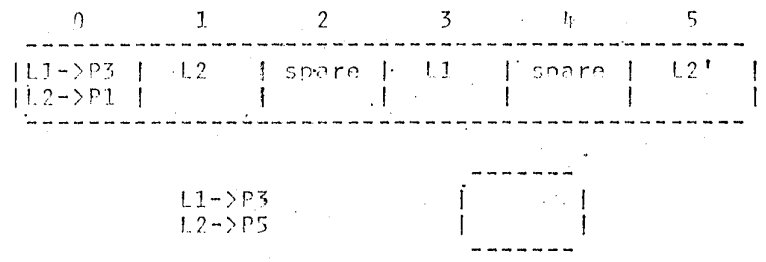
1) Open the database and copy the LP mapping (here held entirely in physical block 0) into core; next read logical block 2 into a buffer:



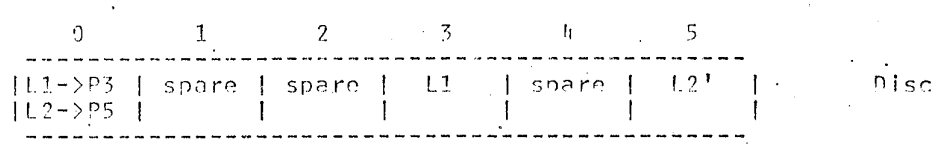
2) Update L2 to make L2', and assign a spare physical block to L2 in the mapping in core:.



3) When we need the core buffer for some other purpose, we must write back the altered block to disc. Note that at this stage the disc file contains two versions of the same logical block (L2 and L2'):



4) Finally the LP mapping in core is copied back to the disc and physical blocks referenced by the old mapping but not by the new are freed:



A small but important point concerns the writing of a new LP mapping from core to disc. The new mapping should always be written to spare blocks in the physical file, so that if the system crashes in the middle of the writing process the old mapping is still available. This technique also increases the likelihood of successful recovery in the event of parts of the disc file itself becoming unreadable.

Logical Consistency

The action of writing a new LP mapping from core to disc to define a new logical database is called "remaking" the disc file, and must always be done before the database is closed after a series of updates. During the execution of an AP it is only necessary to remake the file if the number of spare blocks becomes dangerously low, or if the AP explicitly requests a remake in order to provide a "checkpoint" from which it may be restarted.

On the other hand, there are times when we must not remake the file in order to avoid the appearance of inconsistent logical databases on disc. For example, if a new link is to be created between two entries the package must update both entries (probably in separate logical blocks) before the database is consistent.

Most such situations can be accommodated by ensuring the availability of a small number of spare physical blocks before the updating process is started: if sufficient are not immediately available, the disc file is remade in order to free blocks referenced by the previous mapping but no longer required by the new one.

Certain situations, however, may require a large number of spare blocks: the worst possible case would be the deletion of an entry which had links to at least one entry in every block of the database, thereby requiring the updating of every logical block before the database is consistent. In such cases, the package

records extra information on the disc when the file is remade to indicate that some (as yet incomplete) operation is in progress; if the system crashes before a further remake there is then enough information on disc to enable the completion of the operation when the database is next opened.

By careful use of these techniques, the current implementation of the Jackdaw package not only ensures that only consistent databases appear on disc, but also that each interface procedure is "indivisible"; in other words, any interface procedure call will either complete its specified action or do nothing at all.

SECTION III

Future Developments

The current implementation of Jackdaw allows only one AP to update a database at a time, although many users may read simultaneously. Future versions will allow concurrent update by several users, and new procedures need to be defined to allow cooperation and controlled interaction between individual users.

One problem in an on-line database environment with many users continually updating the database is that of obtaining consistent reports (see [5]). For example, consider the case of a program which generates a summary of items in stock followed by a detail report showing the location of these items by warehouse. If stock figures are updated whilst this program is being executed, the totals in the two reports will not tally.

This problem may be solved by "freezing" the LP mapping used by the report generator for the duration of its run. During this period, any updates performed by other concurrent processes continue to be reflected in new LP mappings in the usual way, except that physical blocks referred to by the "frozen" mapping are not reused. The effect is that the physical database continues to hold the "frozen" as well as the current logical database until the reports are complete.

A similar technique may be used to safely test new applications programs on a "live" database. When the database is opened in "test" mode, a copy of the current LP mapping is made for the program being tested. Any updates made are reflected in this copy, but not in the "real" LP mapping. At the end of the run, the copy mapping would normally be thrown away, but could alternatively be preserved to enable "post-mortem" programs to see how the test had progressed. Once again, the effect is that the physical database reflects more than one logical database: in this case the "real" database together with that modified by the program under test.

Summary

This paper has described a general technique for protecting a database on disc from the effect of system and application program errors, essentially dependent on the ability of a single physical database to reflect more than one logical database.

Further applications of the technique in a multi-user environment were suggested, inlcuding a facility for obtaining consistent reports in an ever-changing environment, and the ability to debug new applications against a "live" database.

This technique is employed by the Jackdaw database package which has been used to support an Administrative database at the University of Cambridge Computing Service for almost four years. As an indication of the success of these techniques it may be noted that it has never been necessary to recover the administrative database from a backup copy on tape, although updates are made interactively on a daily basis.

Acknowledgements

Part of the research described in this paper was realised whilst the author was employed by the University of Cambridge Computing Service, England, and further development of the Jackdaw package is now being partly supported by CNPq at Pontificia Universidade Catolica, Rio de Janeiro.

References.

1) Richards, M., 1974. "The BCPL Programming Manual", Computer Laboratory, University of Cambridge, Cambridge, England.

2) CODASYL, 1971. "CODASYL Data Base Task Group Report", April 1971, ACM, New York.

3) Challis, M.F., 1974. "The Jackdaw Database Package", TR1, Computer Laboratory, University of Cambridge, Cambridge, England.

4) Date, C.J., 1975. "An Introduction to Database Systems", pp231-241, Addison-Wesley Publishing Company, London.

5) Palmer, I., 1975. "Database Systems: A Practical Reference", pp2.20-30, C.A.C.I Inc., London.

CURRICULUM VITAE

Michael F. Challis, Ph. D. - Universidade de Cambridge na área de Linguagem de Programação. Professor no Departamento de Informática da PUC/RJ.