University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science

North Haugh
St Andrews KY16 8SX

A Persistent Store Garbage Collector with Statistical Facilities

Persistent Programming
Research Report 29

Al Dearle.

A Persistent Store Garbage Collector with Statistical Facilities

Jack Campin
Malcolm Atkinson

University of Glasgow
Department of Computing Science
Glasgow G12 8QQ
Scotland

ABSTRACT

This report describes a suite of programs for compacting disk space used by the databases of the persistent programming language PS-algol, and providing statistical information about the objects contained in them.

Persistent Programming Research Report 29

1. Preface

This report describes a secondary store garbage collector that compacts databases used by the PS-algol persistent programming language, as implemented on the ICL Perq computers running PNX. It also documents a family of programs that create and manipulate statistical summaries of their contents.

It is intended to:

* give the reader a view of those aspects of the PS-algol persistent store technology that impinge on garbage collection,

* describe how to interpret the statistical information acquired during garbage collection or by the special-purpose statistical tools,

* make the error messages produced by these programs comprehensible.

It presumes that the reader will be familiar with the following:

* the PS-algol language as described in Persistent Programming Research Report 12, "PS-algol Reference Manual" (second edition).
* the implementation of the PS-algol type system as described in PPRR 11, "PS-algol Abstract Machine Manual", section 3 (some familiarity with this is needed to understand the statistical information these tools provide).
* the general concepts underlying the PS-algol persistent store, as described in PPRR 13, "The CPOMS Persistent Object Management System".

These utilities are distributed with Perq PS-algol; queries about them should be directed to the PISA Project at the University of Glasgow.

Terminology

This will be consistent with that of the other Persistent Programming Research Reports. In addition I use the phrase "persistent store directory" to mean the UNIX directory holding the files manipulated by a PS-algol process; the phrase "database directory" will always be used to mean the UNIX file used by PS-algol to hold information about its databases (see section 2 of this report, and PPRR 13, for a definition of *database* in the PS-algol context).

2. The CPOMS, the UNIX file system, and persistent programming methodology

PS-algol runs under the UNIX operating system and its persistent store (henceforth referred to as the CPOMS) is implemented on top of the UNIX file system. There may be a number of persistent stores, but a given PS-algol process may only use one at a time. These stores are a collection of UNIX files in a single directory (specified by the shell variable $PSDIR in the process's environment or else a default hardwired into the interpreter). A store is partitioned into a number of zones, called *databases*, to make it possible to garbage-collect the secondary store piecemeal; in general each database will be associated with a coherent set of PS-algol programs that present an "abstract data type" interface to the data and code it contains. See PPRR 12, "PS-algol Reference Manual", and PPRR 9, "Procedures as Persistent Data Objects", for a description at the source code level of how PS-algol uses databases.

Some possible examples:

[1] a bibliographic database: this might be manipulated by a single large PS-algol program providing a user interface (insertion, query, deletion, editing) and would consist of a large number of small PS-algol objects (chiefly strings) with long persistence (years) since items are unlikely to be deleted or edited once they have been stored. A database like this does not need to be garbage-collected at all.

[2] a text editor; one program, executed by a systems programmer, would add new versions of the editor to the database, while other programs written by application programmers would read the current version of the editor out of the database at their initialization and use the latest version to edit text in protected fields in the application's screens. The editors are likely to have a persistence of months, between successive releases; garbage collections need not be done any more frequently than this.

[3] a bitmap graphics editing system: this would contain successive versions of the editor and successive versions of the pictures. The images are likely be very large with short persistence, of the order of days or hours. Daily garbage collection would probably be essential.

A PS-algol program may access a number of databases in a single invocation. This may lead to systems like the following:

[4] editors for different classes of object (text, vector graphics, bitmap graphics, musical scores) may exist in different databases and application programs may store their own data objects in their own dedicated databases, using the editor databases as read-only program libraries. The application databases will probably accumulate garbage much faster than the editor databases.

[5] a font database, containing a large number of small objects (the bitmaps for the characters), which is used by application programs to create messages which will be stored as bitmap images in their own databases. The font database will only contain garbage in the very exceptional situation of a font being

updated.

The flexible binding time provided by PS-algol's higher-order procedures permits additional possibilities:

[6] a database like [4] where the text editors are structured as higher-order procedures which, given a font read in from a font database, return an editor that displays its text in that font. The application programmer will instantiate the procedure and store its result in her own database for future use to provide faster initialization of her programs than would be possible if the editor generator procedure were applied every time an editor was required. The font itself will not be copied into the application program database; what will be stored is a pointer in a procedure closure. This implies that cross-database references need to be taken into account by a persistent store garbage collector; the font cannot be deleted as long as the generated editor is accessible.

[7] a system like [4], but where the application program is structured as a higher-order procedure taking an editor as parameter and returning an end-user application. An end-user application generated in this way is stored in the database as a closure including a pointer to the version of the editor used at its generation; that version must be preserved as long as the application.

The minimal set of files used in a CPOMS persistent store are as follows:

(a) Files global to all databases:

* DIRIN, the database directory file, containing, for each database:
  - its name
  - its owner
  - its password
  - read/write locks
  - the root partition, a field used by the CPOMS address mapping scheme
* DIROUT, a temporary copy of DIRIN, used during write operations
* DIRLOCK, a temporary link whose existence is used as an atomic lock during write operations on DIRIN
* PARTSIN, a "partitions file", used to hold address mapping information
* PARTSOUT, a temporary copy of PARTSIN, used during write operations
* PARTSLOCK, a temporary link whose existence is used as an atomic lock during write operations on PARTSIN
* LINKFILE, an empty file the lock files are linked to

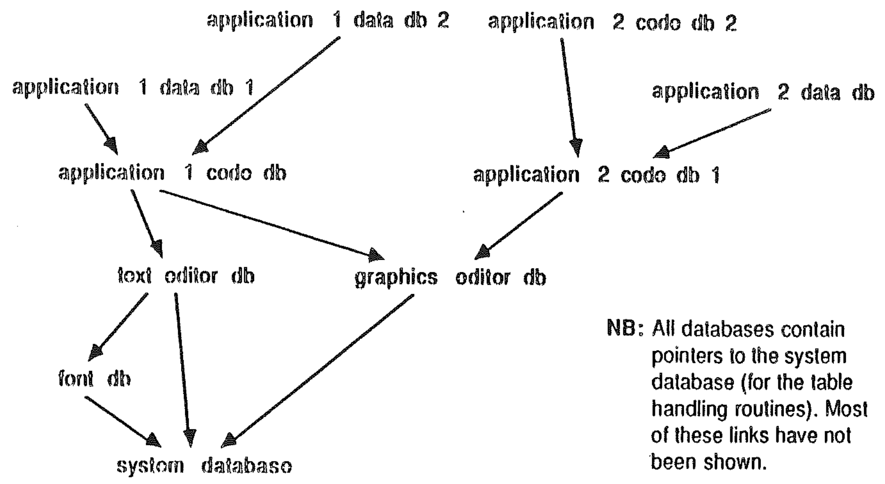
(b) Each database has two permanent files and a temporary file associated with it. These share a 5-digit suffix (which is actually the ordinal position of its entry in the database directory converted into octal and written backwards). These files are:

* DATAxxxxx, the data file, which contains the objects in the database, written consecutively
* INDXxxxxx, the index file, which contains offsets into the DATAxxxxx file; the position of an object's INDXxxxxx entry is computed from
  - its *persistent identifier* (henceforth "pid"), a 32-bit pointer uniquely identifying it in the persistent store, together with
  - the database housekeeping information
* BFRExxxxx, the before-look file, a temporary file used by the CPOMS to allow crash recovery when writing to the database.

A PS-algol program calls the standard function *commit()* to write to the persistent store. The effect of this is to append new objects to the DATAxxxxx file; once in the file an object is never overwritten (except for the very first object in the file, which is used to hold housekeeping information about the database and which is not directly accessible to the PS-algol programmer). The effect of overwriting is achieved by switching pointers in the INDXxxxxx file. The INDXxxxxx file is extended when it is necessary to acquire additional addresses for the database.

This implies that the primary target for a persistent store garbage collector is the DATAxxxxx files. The INDX files cannot be contracted except at the end because of the addressing architecture; and the entries at the end are those least likely to be garbage, since their identifiers are the most recently generated. The index files contain a free list indicating which persistent identifiers are no longer in use; by updating this, a garbage collector can prevent them growing inordinately - the CPOMS will recycle the positions on the free list. But they still grow monotonically as long as the database is in existence.

Databases can be deleted. This is done by a PS-algol standard function whose effect is to replace the database name field in the DIRIN file with the empty string. The database's files are unaffected, as is required by situations like [5] above; a database may have been deleted so that it is no longer directly accessible by name to any PS-algol program, but old bindings to it may force some of its data to be retained. However, PS-algol software systems are frequently structured so that referential dependencies between databases form a directed acyclic graph; in particular, some databases are not referred to by any others. If one of these is deleted, none of its objects remain reachable, and so both its DATAxxxxx and its INDXxxxxx files can be removed. The CPOMS can recognize that this has been done if the "root partition" field in its DIRIN file entry is set to a special value (-1); it will reuse the entry, and so the database number, when next asked to create a new database. So a secondary target for a persistent store garbage collector is to identify databases whose entire content is totally unreachable from anywhere in the persistent store, and to remove both their DATAxxxxx and INDXxxxxx files.



NB: All databases contain pointers to the system database (for the table handling routines). Most of these links have not been shown.

Diagram 2.1: A typical set of referential dependencies among databases

The examples above show that a PS-algol database may be strongly application-dependent, both in the kinds of object it holds and in the way its population varies with time. Statistical information about the objects in a persistent store is useful to the designer of future persistent stores; in future store designs such information will also be needed as a day-to-day diagnostic tool for dynamically optimizing data storage. Questions that might be asked include:

* how big are the objects?
* how are the objects distributed among the data types?
* how many pointers do objects have?
* how common are cross-database references?

A persistent store garbage collector is in a good position to provide answers to these questions, since it will probably be the only program in the system that can scan every object in a database or every object in the persistent store.

Questions that cannot be answered with our present persistent store technology include the following:

* How long do objects persist? (the CPOMS does not provide timestamps)
* How often are objects accessed? (the CPOMS does not keep use counts)

3. Overview of the garbage collection system

The PS-algol persistent store garbage collector presently consists of a suite of programs written in C to perform the functions described as desirable in the previous section. They are:

garbo - the garbage collector, which also gathers statistics
metro - a statistics collector using most of the same code as *garbo* but which leaves a database's DATAxxxxx and INDXxxxxx files unaffected
garbostat - prints the statistical information collected by *garbo* or *metro* in a humanly legible form
precis - summarizes the statistics files from a number of databases, producing a file displayable by *garbostat*
garbofixup - restores files in the event of a crash in *garbo* or *metro*
safecrack - unlocks databases after a crash in *garbo* or *metro*
lsdb - a new version of an existing utility to display the state of a persistent store, modified for greater reliability and to provide additional information that someone running *garbo* might need.

Using the garbage collector

The user may choose to garbage-collect randomly when space runs out, or systematically by the UNIX *at*(1) command (say, in the middle of the night). In the former case, she will probably have some idea which databases are in most need of garbage collection and will specify them by name or number. In the latter case, the **-a** flag is appropriate.

Garbo's running time is about 10-15 minutes per megabyte of data for the compaction phase on a moderately loaded Perq 2; if inter-database references are absent the cross-referencing phase is very short, otherwise it adds an overhead of perhaps 50% to this. A message is displayed at the end of each compaction or deletion showing what percentage of space has been reclaimed. Nothing is normally displayed during the cross-referencing phase, but using the **-f** flag, to display which files are currently being manipulated, will give the user some idea of what *garbo* is doing.

Metro is slightly faster because it does not need to write out a new data file.

Performing crash recovery

Only the PS-algol system administrator (who must have the username "ps" in the present PS-algol system) can do this, because of concurrency problems (since starting *garbofixup* while *garbo* is running could destroy database files). Recovery simply involves invoking *garbofixup*; there are no parameters to the command. If this fails, examining the persistent store directory by *lsdb* and "ls -l" should reveal what the problem is.

Using the statistics collecting facilities

The statistics files produced by *garbo* and *metro* are intended to be used in different ways. *Garbo* maintains a cumulative record of changes to each database in the STATxxxxx file. This file is updated at each garbage collection. Unless it is explicitly deleted by the PS-algol system administrator, it lasts as long as the database does. If the database it describes becomes empty and has its files deleted, STATxxxxx will be given a synthesized file name and retained in the persistent store's UNIX directory. *Metro*, on the other hand, is intended to give a "snapshot" of the state of the database, and its files cannot be updated; if the user wants to preserve ***metro***'s SNAPxxxxx files, she should move them to some other directory in the UNIX file system.

Garbostat is a program which displays the contents of a statistics file in a tabular report which should be understandable to anyone familiar with the description of the PS-algol type system in PPRR 11, "PS-algol Abstract Machine Manual", section 3. See the appendices for an example of a report produced by *garbostat*.

The *precis* utility is intended to make it possible to describe the overall state of a persistent store. It accumulates the information from a collection of STATxxxxx or SNAPxxxxx files into a single file; this file can be displayed by "garbostat". Usually *garbo* or *metro* will have been invoked with the **-a** flag, since the user is more likely to want to know about the entire persistent store than about a selection of databases from it. The statistics files need no longer be associated with a persistent store; thus snapshots of a persistent store can be kept outside it, and cumulative files from a defunct persistent store can still be *precis*'d Since it is not clear what the result of merging *garbo*'s cumulative files with *metro*'s snapshot files would mean, this operation is not permitted. Neither is it permitted to use *precis* to further precis its own output files.

All the statistical information generated by the programs described here is held in a single file format; see the Appendices for a specification of the statistics file format as a C ".h" file. The files contain flags that indicate which programs created them.

4. Design of the garbage collector

Garbo is a breadth-first copying garbage collector. The roots of garbage collection are of two kinds: the root objects of the named (non-deleted) databases, and inter-database references. A preliminary scan of the entire persistent store searches for these cross-references, building up a queue of references into each database to be garbage-collected. The second phase compacts the specified databases in sequence, copying the reachable objects in each to a new version of the data file and creating a new index file as it does so. There are two reasons for this design:

* garbage collection can be performed on individual databases rather than on the entire persistent store; so the garbage collector can avoid wasted effort trying to find garbage in databases that are known not to contain any, such as a typical font database.
* the databases can be garbage-collected in sequence, rather than by following pointers across databases; this is advantageous because it avoids frequent opening and closing of files, which is an expensive operation. (A depth-first algorithm would require this since UNIX imposes a small upper limit on the number of files that a process may have open at one time; the limit would easily be reached by a garbage collector that required access to all the persistent store's data and index files simultaneously.)

There is a cost to this: *garbo* cannot reclaim circular list structures that span more than one database, since the preliminary scan treats all cross-references as valid roots of garbage collection, even if they emanate from garbage objects. In practice such structures have not been used anywhere in the persistent stores we have examined. Another implication is that garbage does not necessarily disappear immediately. If the last reference to an object A is from a garbage object B in another database, then A won't be garbage-collected until the next run of *garbo* after the one that reclaims B.

The choice of a copying algorithm was motivated by considerations of error recovery. It would have been possible to garbage-collect the data file in place, using a before-look file to record the prior state of the data file as it was overwritten. This might have used filestore more economically, but would have greatly complicated recovery after a crash while the garbage collector was running. Since the garbage collector has to be one of the most trusted programs in the system, anything that might compromise its reliability has to be rejected. For similar reasons all the temporary files created by *garbo* are kept in the persistent store directory and there is no facility for using a remote machine's filestore for temporary storage. This would have been easy to implement using the Perq Newcastle Connection distributed file system, but there was a a real danger that a machine crash might produce an irreversible catastrophe, leaving essential recovery information scattered in untraceable pieces across the network. These design decisions conspire to force the PS-algol user to leave enough disk space free for a complete copy of the largest database in the persistent store.

One feature of the CPOMS that has not been reflected in the garbage collector is ownership of databases; this is recorded in a field of the database directory entry (as a UNIX username). The garbage collector ignores this; any user can garbage-collect any other user's databases. Further, the PS-algol system administrator is no different in this respect; system databases can be garbage-collected in the same way as ordinary users'. The system administrator, however, is the only person permitted to run the crash recovery program; this is because concurrent attempts at garbage collection and crash recovery could destroy data.

Initialization, persistent addressing, and the partition map

Pointers to objects in the persistent store are *persistent identifiers*, henceforth referred to as *pids*. The format of a pid is:

bit 31	set, to distinguish pids from in-core pointers
bits 16 - 30	partition number 1 - 32767
bits 0 - 15	object number 0 - 65535 within a partition

Partitions are units of persistent address space. A given partition may only be allocated to one database; thus it is possible to deduce which database an object is in from its partition number. The allocation of partitions to databases is stored in the partitions file (PARTSIN).

The partition number of an object also leads to its entry in the index file. A list of the partitions allocated to a database, in the order in which they have been allocated, is held in a database's *partitions vector*. When a new partition is allocated to a database, a new 64Kword block of the index file also starts. So the block of the index file in which the object's entry is located is the index into the partitions vector that indexes its partition number; the partitions vector is a $^1/_{65536}$ scale replica of the index file. The actual offset into the index file for an object is then found by adding its object number to the offset of its partition block. (This scheme obviously cannot find the partitions vector itself; the root object of the database contains the offset of its partition to make it possible to bootstrap the addressing mechanism.)

This address mapping information has to be represented efficiently in the garbage collector. The solution is the simplest possible; a static array, indexed by partition number, of pairs <database number, partition's block offset in index file>. This array is called the *partition map* in what follows; it performs the same function as the PTODI (partition to database and index map) in the CPOMS (see PPRR 13, section 4.3). The first entry is filled in from the partitions file; the second is derived from the partitions vectors of all the databases in the persistent store during the cross-reference phase. A consistency check is performed at the same time; if the partitions file and a partitions vector disagree, *garbo* aborts.

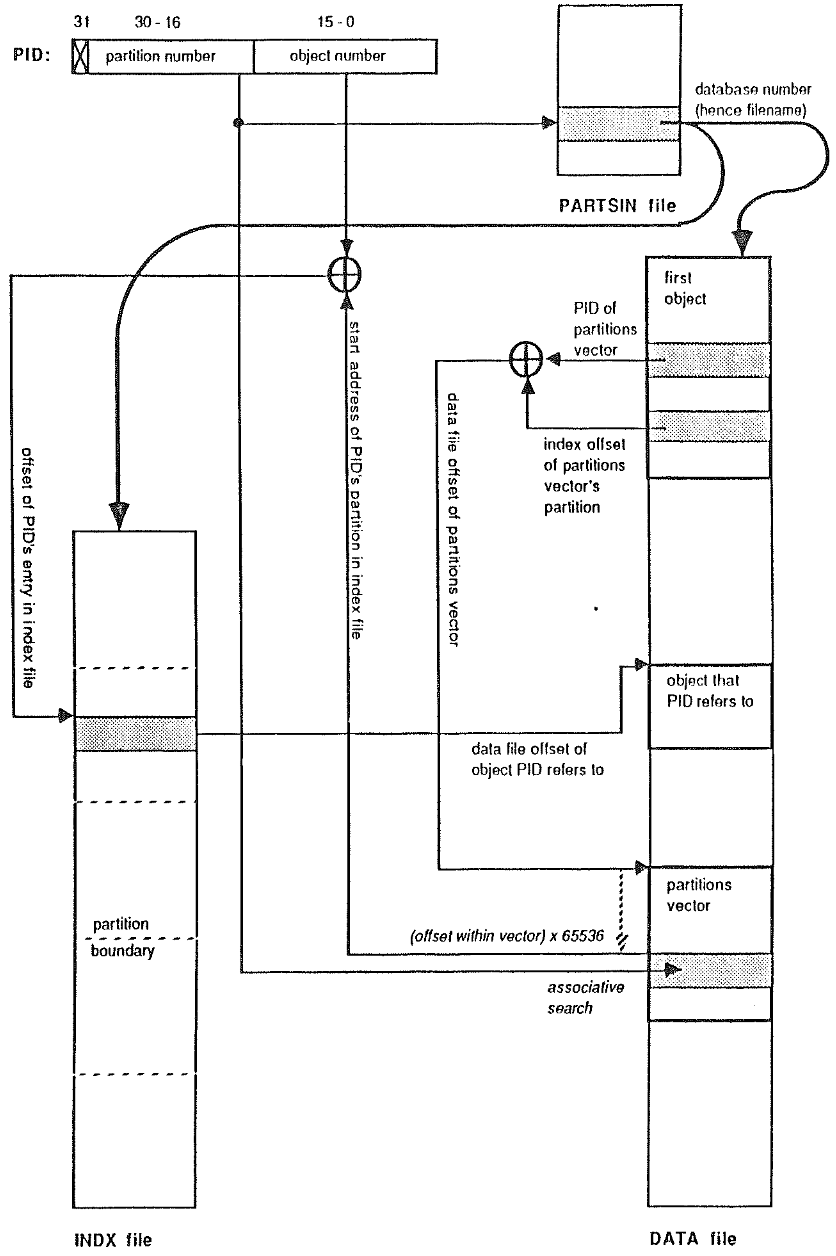

31
30 - 16
15 - 0
PID:
partition number
object number
database number (hence filename)
PARTSIN file
first object
PID of partitions vector
index offset of partitions vector's partition
data file offset of partitions vector
start address of PID's partition in index file
offset of PID's entry in index file
object that PID refers to
data file offset of object PID refers to
partitions vector
(offset within vector) x 65536
associative search
partition boundary
INDX file
DATA file

Diagram 4.1: The CPOMS addressing architecture

The cross reference phase

Since *garbo* can be used to garbage-collect a single database, it is desirable to avoid scanning the whole persistent store in this phase. This is possible since each database maintains a list of databases it has references to in the *linked databases vector*, a PS-algol vector of integers (database numbers) which is pointed to directly from the database root object. This may be out of date, since the CPOMS commit may remove cross-references between databases by updating objects but has no way of deleting an entry in the vector - but a database will certainly have no references to databases that do *not* appear in it. So *garbo* will only scan a database if its linked databases vector has an entry for one of the databases to be compacted.

The scan traverses the whole index file; entries in this are 4-byte integers which, if positive, are offsets into the data file to the start of an object. The header of the object is read in; this indicates whether the object contains pointers or not. If it does, all the pointers in the object are examined and those that are external references to one of the command-line databases are put into a cross-reference table for it. (Internal pointers are disregarded.) These tables are sorted arrays with a fixed size of 10K entries; this is considerably more than the number of cross-references so far met with. Lookup is done by bisection search. The tables are allocated on demand since most databases will not be referred to by others (like those at the top of the graph in diagram 2.1).

The compaction phase

Two things may happen to a database in the second phase of the garbage collection. If (a) it has been deleted, (b) there are no pointers to it from other databases, and (c) no other database has it as an entry in its linked databases vector, then its files will be deleted, its directory file entry marked for reallocation, and its partitions deallocated (this is done by creating a new version of the partitions file in which the entries previously holding the number of the recycled database are set to -1). Otherwise "garbo" will attempt to compact it.

The qualification (c) about linked database vector references is not a requirement of the CPOMS addressing architecture but a quirk of its implementation; on initialization it "eagerly" opens all the databases it may need, consulting the linked databases vector, and so will fail if a database pointed to by the vector has been recycled.

This leads to the disconcerting situation that a database may be contracted down to just its root object and partitions vector, the minimum necessary for CPOMS initialization, but will not be recycled until some other database gets its linked databases vector contracted by a garbage collection.

Before doing the compaction, *garbo* checks to see if there is enough space for its temporary files (mainly the copies of the data and index files) and its statistics file. Since *garbo* has no way of knowing how much of a database is garbage before attempting to compact it, it makes the pessimistic assumption that it will not recover any space and leaves room for a complete copy of the database. If there is insufficient

space, *garbo* will abandon the compaction and continue with the next database. It may happen that a second run will succeed because space has been freed by the compaction of smaller databases; however, *garbo* does not retry automatically. (Perq PNX does not maintain a tally of free space in the superblock, so the Perq implementation does the space check by forking to *df(1)*. This is an expensive operation and prone to failure, but essential since PNX behaves unpredictably when disk space runs out.)

The compaction is a straightforward breadth-first scan. A queue is initialized with the pids found in the cross-reference phase. As an object is removed from the queue, it is copied to the new data file, its new data offset is written to the new index file, statistics are taken on it, and any pointer it may contain is put on the queue if it has not already been processed (which will be true if and only if it has an entry in the new index file). This continues until the queue is empty. If *garbo* finds a pointer to another database, the statistics functions note the fact and a bitmap of all databases used for updating the linked databases vector is marked; the pointer is not put on the queue.

There is one deviation from the breadth-first order which is required for statistics collection. In the PS-algol abstract machine, bitmap images are implemented as structures (of fixed size) containing size information and a pointer to a vector of bitplanes. The bitplanes vector is an ordinary vector of pointers (to bitmaps for each plane) and is not given a special type. Neither are the bitmaps themselves; they are just integer vectors. There is no way to identify a bitmap as such in the persistent store without knowing what points to it. Since bitmaps may take up substantial space in a Perq PS-algol database, users will want to know just how much. So when an image object is encountered, the algorithm switches to depth-first for the two levels that comprise its components; the bitplanes vector and the bitmaps never get on the queue. This has the additional advantage of putting the image descriptor, bitplane vector, and bitmaps together on the disk; they will usually be accessed together.

Finalization

Some essential housekeeping tasks remain to be performed after the compaction.

(1) The linked databases vector needs to be updated. The information needed for this is collected during the compaction phase; a new vector is constructed and appended to the new data file.

(2) The CPOMS can reuse pids that have become garbage if their index file positions are put on a free list; entries on this are negated. So the garbage collector scans the new index file and threads together a free list of the entries that have not been written with a positive offset into the new data file. For each such entry, it also consults the old files to take statistics on the corresponding object; since such objects are garbage, they will not have been encountered during the compaction phase.

(3) The first object in the data file has a field for the size of the data file and for the start of the free list in the index file; these are updated.

Virtual memory

PS-algol objects can be extremely large - bitmaps are typically the biggest, and can be an entire Perq screen. So they have to be paged in some way. Since the implementation began on versions of PNX that did not have virtual memory, a simple virtual memory system was constructed. Each data and index file (old or new) is given a set of pages threaded into a doubly linked list; this is looked up by linear search with the most recently accessed page being moved to the front. This subsystem accounts for most of the cpu time, but still performs fairly well; in contrast, an early version of *garbo* loaded entire index files into memory and used a large buffer for objects - this was considerably slower, probably because of the amount of swapping it forced the Perq to do.

The virtual memory provided by PNX 5 has not been used; its performance for very large processes is unpredictable. The simple scheme chosen keeps the garbage collector core image fairly small; it is easily possible to run the garbage collector as a background process and still have enough processing power left for tasks involving user interaction.

The statistics collector

Metro, the statistics collector, differs from *garbo* in two ways: it does not change any files used by the CPOMS, and the statistics files it produces are not cumulative; they give a "snapshot" of the state of a database. The algorithm used is identical to *garbo*'s, but a new data file is not generated. A new index file is: entries for accessible objects are put into it as a way of identifying garbage objects by comparison with the old file. The new index file is discarded when statistics collection on a database finishes. *Metro* does not need to create beforelook files and requires simpler recovery mechanisms.

5. Error recovery and locking

The error recovery mechanisms used by *garbo* are designed for intelligibility rather than minimality; a profusion of links is used. These are of two types. Diagnostic links are used to indicate the state *garbo* is currently in; if the machine crashes, their presence or absence will indicate the correct recovery action to the recovery programs or the PS-algol system administrator. These are all links to the empty file LINKFILE. Nonempty links are used to hold temporary files such as beforelooks or new versions under construction. All links manipulated by *garbo* are in the persistent store directory ($PSDIR or the default).

Similar error recovery mechanisms are used by the *garbofixup* recovery program and by *garbo* itself (both for actual error conditions and to abandon garbage collection in response to a user interrupt).

Garbo's first action is to create a link called GARBORUNNING, which it will only remove just before it exits. This is of no significance to the CPOMS or the error recovery routines; its purpose is (1) to tell a user why the entire persistent store is locked, (2) to tell the PS-algol system administrator why the store is in the state it's in if a crash occurs, (3) to stop *garbo* from being retried after an unrepaired crash.

Garbo must not run concurrently with any PS-algol program (hence its name - "I want to be alone"). To prevent this, it uses the same locking protocol as the CPOMS (see PPRR 13, section 3.6 and PPRR 13, appendix 1). It first locks the database directory (by the creation of a link called "DIRLOCK"); it will stay locked as long as the persistent store may be in an anomalous state. *Garbo* then write locks all databases before attempting to open any of their files. If any database already has a read or write lock, *garbo* will abort. This write locking provides no extra security beyond that given by locking the directory - simply checking whether any database was already locked would do - but it gives users a way of figuring out what might be happening (the persistent store directory is not publicly readable, but the *lsdb (1)* utility would show that all the databases in the system were write locked). The locking is done the same way as in the CPOMS; the directory file is not written to but a new version, with the locks set, is created and the old one unlinked.

It would be disastrous to garbage-collect a database that was in an inconsistent state after a crash during the CPOMS *commit()* routine. While a commit is in progress, a number of files (actually links to the same file) with the prefix "BFRE" are in existence. *Garbo* searches the persistent store directory for any such file, and aborts if one exists. (It does the same for its own BEF backup files, but is unlikely to get to this point without deliberate meddling by the PS-algol administrator since the "GARBORUNNING" link will cause it to fail first.)

No additional error recovery is required during the cross-reference phase, since no files are modified during it; recovery at this point consists of (1) unlocking the databases, (2) removing the DIRLOCK file, (3) removing GARBORUNNING.

During the compaction/deletion phase, only one database at a time can be in an inconsistent state. A link is created to indicate which database this is; this uses the same naming scheme as the CPOMS. If the database is to be compacted, the link is called DOINGxxxxx, where xxxxx is the database number in reverse octal. If the database is to be deleted, the link is called DELETINGxxxxx. Recovery from a crash during compaction is by rollback; if a crash occurs during deletion, recovery is by roll-forward.

Three things have to be done during recovery from a crash during deletion: any remaining database files have to be removed (and statistics files renamed - see later) , the DIRIN file entry for the database has to be marked for recycling, and the partitions file has to be updated so that the database's partitions can be reallocated. Then the DELETING link can be removed and (1), (2), and (3) above can be done.

Rollback from a failed compaction is done by relinking files rather than by reconstructing them. The DOING link is not created until backup links to the database files have been made; for as long as the DOING link is in existence, these links are guaranteed to represent the original state of the database. They are given the same names as the database files, prefixed by "BEF". So the way to recover from this point is: find the database with a DOINGxxxxx link, remove all files with an xxxxx suffix except for the BEF files, then rename the BEF files, remove the DOINGxxxxx link, and do (1), (2), and (3) above.

The compaction phase does not write files under the old filenames; instead it writes to a new set, whose names are the old ones prefixed by "NEW". This creates unnecessarily redundant links (the old DATAxxxxx, INDXxxxxx, and STATxxxxx are not unlinked until the compaction has finished) but provides additional user information in the event of a crash. NEW files can always be removed without losing any information; so can BEF files with no corresponding DOING file (these will have been left by a crash either before garbo started its compaction phase or after it got the database files back into a consistent state). *Garbofixup* removes such files.

Garbo creates one further file, called *df_output*, which is used to hold the output of a fork to *df*(1) (this inelegant way of finding out how much disk space is left is required by a number of colluding bugs in Perq PNX). This file can always be deleted.

6. Statistical information

Garbo and *metro* both create auxiliary files containing statistical information. This was an early design decision and probably a mistake - if the statistics had been recorded in a statistics database as ordinary PS-algol data, it would have been easier to make changes in the statistics collected because of PS-algol's ability to handle dynamically typed structures, and it would also have been easier to manipulate the data with existing and future PS-algol software tools. As it is, the data is held in a fixed format consisting of a single large nested C structure with vacant slots for a small amount of future expansion. The version and release number of the program that created the file is recorded at the start, so some upward compatibility is possible.

The file records which program created it - *garbo*, *metro*, or *precis* (and if the latter, whether it is a precis of files created by *garbo* or by *metro*). The naming conventions used for statistics files are as follows:

* the cumulative statistics maintained on a database by *garbo* are held in files STATxxxxx, where xxxxx is the reverse-octal suffix used by the CPOMS for the other files of the database. These files, like the CPOMS database files, are given protection mode 0600 (owner read/write) so that only the PS-algol system administrator, or programs impersonating her by the UNIX "setuid" mechanism, can touch them.

* the snapshot statistics acquired by *metro* are held in a SNAPxxxxx file, where xxxxx is the reverse-octal suffix. The user is expected to move this file to her own directory after creating it, or else delete it; if *metro* is invoked on a database that already has a SNAP file, it will issue a warning and refuse to overwrite it. SNAP files are created with protection mode 0666 (public read/write) to enable users other than the PS-algol system administrator to remove them from the persistent store directory. Users can find the suffix for a database from *lsdb*.

* when *garbo* recycles a database, any STAT or SNAP files it has will be renamed with synthesized names "statyyyyyyyy" and "snapyyyyyyyy" respectively, where yyyyyyyy is the date (in seconds since 1/1/1970) in hexadecimal. This is necessary to avoid an old statistics file being associated with a new database when the recycled entry is reused by the CPOMS.

* *precis* uses the names PRECSTAT or PRECSNAP for its output files, depending on whether it has been asked to summarize cumulative or snapshot files. The output files are created with public read/write protection mode.

All of these files have the same format and can be displayed by *garbostat*.

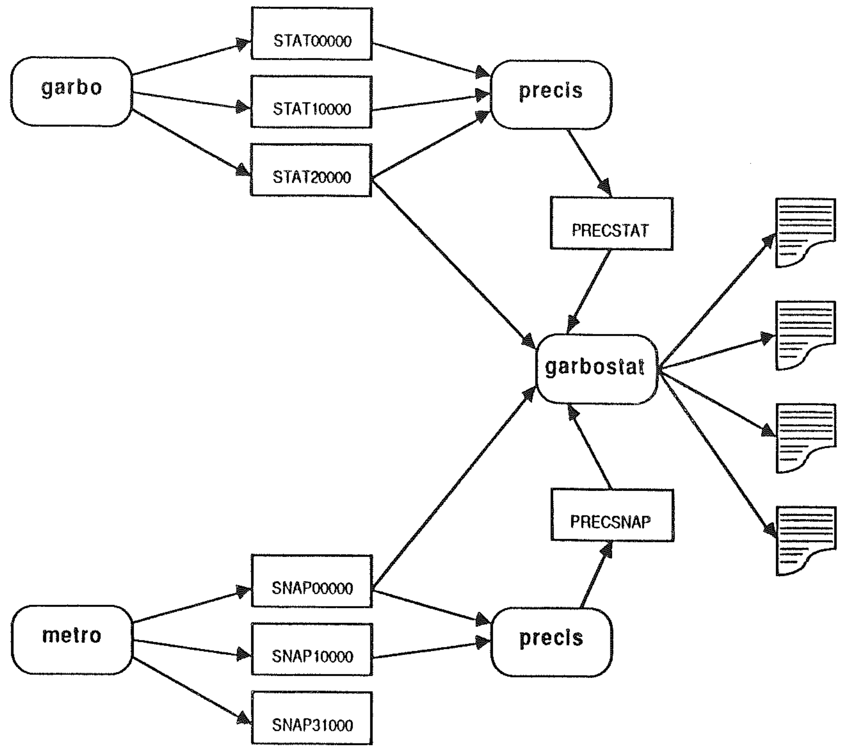


Diagram 6.1: Flow of statistical information

The information contained in these files is:

* the number of garbage collections performed on the database to date
* the maximum size reached by the queue during the compaction phase
* the number of references to the database from other databases
* the size of the data file at the end of the last garbage collection
* the current size of the data file (this may be different if the statistics file was produced by "metro")
* the number of files summarized, if the file was produced by *precis*
* for both accessible and garbage objects:
  * the number of nil pids
  * the number of reserved pids
  * the number of pids pointing to other databases
  * the number of pids pointing into the database
  * the amount of space added to the data file between the last two garbage collections
  * the total space found so far in all garbage collections to date
  * for each abstract machine type, a record of the objects of that type:
    * how many of them there are

    * their maximum size
    * the total of their sizes
    * the sum of the squares of their sizes (for computing the variance)
    * their maximum number of pointers
    * the total number of pointers in them
    * the sum of squares of the number of pointers in them.

Data on bitplane vectors and bitmaps is held in the same format as that for the true abstract machine types; bitplane objects are also recorded as pointer vectors, and bitmaps are also recorded as integer/boolean vectors. *Garbostat* includes figures for these subtypes in its reports, with the type names in parentheses: (BITMAP) and (BITPLANE). These figures are also included in those for pointer vectors and integer/boolean vectors.

Appendices

UNIX manual pages:

- garbo
- garbofixup
- garbostat
- lsdb
- metro
- precis
- safecrack

Sample output from *lsdb* and *garbostat*

Statistics file format

NAME

garbo — persistent store garbage collector (version 2.1)

SYNOPSIS

garbo [—f] —a | dbname1 dbname2 ... | —o dbnum1 dbnum2 ...

DESCRIPTION

Garbo garbage collects PS—algol databases by recovering space in their DATA files. The databases to be compacted may be specified in three ways:
(1) by the **—a** flag (do all of them)
(2) by database names: *dbname1*, *dbname2*, ...
(3) by the **—o** flag followed by a sequence of 5-digit reverse octal numbers: *dbnum1*, *dbnum2*, ... which are the suffixes of the database's files (these are displayed by *lsdb (1)).* This is useful for garbage-collecting large deleted databases.

It searches for database files in the directory specified by the shell variable **$PSDIR**, or in */usr/lib/ps/dbs* if this is not set.

Garbo builds up statistics on the contents of the databases in a file called STATxxxxx, where xxxxx is the reverse octal suffix of the database's files. The information in this file is in a form displayable by *garbostat (1)* or summarizable by *precis (1).* These statistics are cumulative.

The **—f** option makes the program announce file operations: create, open, close, link and unlink. This is useful if you don't know how long the garbage collection is likely to take and want some feedback on what's happening.

FILES

DATAxxxxx	data file to be compacted
INDXxxxx	database index file
STATxxxxx	cumulative statistics on the database
BEFDATAxxxxx	backup link to original data file
BEFINDXxxxxx	backup link to original index file
BEFSTATxxxxx	backup link to original statistics file
NEWDATAxxxxx	temporary version of data file
NEWINDXxxxxx	temporary version of index file
NEWSTATxxxxx	temporary version of statistics file
GARBORUNNING	link set while garbo is in action; does nothing, purely to tell inquisitive minds why their programs aren't working
DOINGxxxxx	linked to LINKFILE for crash diagnostics to indicate which db is being compacted
DELETINGxxxxx	linked to LINKFILE for crash diagnostics while files with suffix xxxxx are being deleted
statyyyyyyyyy	copy of statistics file made to retain statistics if the database is deleted
snapyyyyyyyyy	copy of metro (1) statistics file made for the same reason
df_output	scratch file of no significance for recovery
DIRIN	db directory file (read only)
DIROUT	db directory file (write only)
DIRLOCK	link to LINKFILE for locking db directory
PARTSIN	partitions file (read only)
PARTSOUT	partitions file (write only)
PARTSLOCK	link to LINKFILE for locking partitions file
LINKFILE	empty file used for linking to

SEE ALSO

lsdb (1), garbostat (1), precis (1), garbofixup (1), safecrack (1).

DIAGNOSTICS AND RECOVERY PROCEDURES

Garbo works in two phases; in the first it locks the database directory and all the databases. (If a before-look (BFRExxxxx) or *garbo* backup (BEF*xxxxx) file is present, it will refuse to proceed further.) It then constructs tables of cross-references between databases. If it crashes in this phase, unlocking the directory and the databases will restore the status quo.

The second phase is the garbage collection proper; *garbo* creates the BEF backup files before starting this potentially destructive operation. Immediately after this it creates the DOINGxxxxx link so that *garbofixup* or *garbo's* own recovery routines can find out whether a database's files may have been corrupted if *garbo* crashes. The DOING file is unlinked immediately after *garbo* has gotten the DATA, INDX and STAT files into a consistent state again.

If there is a DOING file, or any BEF or NEW files, present in the dbs directory after *garbo* has failed, *garbofixup* can be used to restore the database files to their state before *garbo* was applied. *Garbofixup* does not unlock the database affected; use *restoredb* to do that, or, if a deleted database is locked, get the PS—algol system owner 'ps' to run *safecrack (1)* which will forcibly unlock all databases.

Because *garbo* creates new DATA and INDX files rather than compacting in place, it needs space for the new versions. It will skip over any databases whose DATA, INDX and STAT files are collectively too big to be copied. So - to garbage collect a big database, remove as many other files as you can spare, and garbage collect smaller databases first.

If a database has no accessible data at all in it, its files will be deleted and its partitions reclaimed. A DELETINGxxxxx file will be in existence while this is happening. After this, *lsdb (1)* will not display the deleted database unless invoked with the —a flag. To retain any statistical information that may have been collected on the database, if it has a non-empty STATxxxxx or SNAPxxxxx file this will be moved to 'statyyyyyyyyy' or 'snapyyyyyyyyy' respectively, where 'yyyyyyyyy' is the time of deletion since 1/1/1970 in hexadecimal.

As from version 1.3, the error/interrupt handling routines should restore the persistent store to a usable state for any failure short of a kill or machine crash.

BUGS AND LIMITATIONS

Garbo does not recover empty partitions in the index file unless a database is deleted. (these are in fact very unlikely to occur anyway).

This version is for the Perq only; databases cannot be locked by the Berkeley 4.2 file locking mechanism.

Garbo can't handle databases that refer to more than 256 other ones.

WARNING

Databases that have been written to by early versions of the PS—algol interpreter may give meaningless figures for the amount of garbage present. To circumvent this, garbage-collect all databases and remove all statistics files. Then start afresh, making sure that no version of the interpreter earlier than 4.0 is still present on the system.

NAME

garbofixup (version 2.1) — restore database files after garbo (1) has crashed

SYNOPSIS

garbofixup

DESCRIPTION

Garbofixup searches for database files in the directory specified by the shell variable **$PSDIR**, or in */usr/lib/ps/dbs* if this is not set.

If *garbo (1)* has crashed while compacting a database it will leave a file named DOINGxxxxx in the dbs directory, where xxxxx is the database number in backwards octal. *Garbofixup* will look for this and restore the database files from the backups (BEFDATAxxxxx, BEFINDXxxxxx, and BEFSTATxxxxx) made by *garbo* before starting the compaction.

Garbofixup will remove any of *garbo's* temporary files that it finds, also any STATxxxxx file found to have length 0.

If *garbo* has started deleting a database's files, there will be a DELETINGxxxxx file present; *garbofixup* will continue with the file deletion if such a file exists and will move any non-empty STATxxxxx file found to statyyyyyyyyy and any non-empty SNAPxxxxx file to snapyyyyyyyyy, where yyyyyyyyy is the time since 1/1/1970 in hexadecimal (this is the same convention used by *garbo).*

Garbo or *metro* will leave all databases locked after a crash and *garbofixup* does not unlock them; use *restoredb (1)* or *safecrack (1)* to do that.

WARNING

Don't run *garbofixup* concurrently with *garbo* or you could destroy any database *garbo* happens to be working on.

FILES

DOINGxxxxx	identifies corrupt databases; removed
DATAxxxxx	may be corrupt if DOINGxxxxx exists; removed
INDXxxxxx	may be corrupt if DOINGxxxxx exists; removed
STATxxxxx	may be corrupt if DOINGxxxxx exists; removed
BEFDATAxxxxx	moved to DATAxxxxx if DOINGxxxxx exists
BEFINDXxxxxx	moved to INDXxxxxx if DOINGxxxxx exists
BEFSTATxxxxx	moved to STATxxxxx if DOINGxxxxx exists
NEWDATAxxxxx	removed unconditionally
NEWINDXxxxxx	removed unconditionally
NEWSTATxxxxx	removed unconditionally
DELETINGxxxxx	linked to LINKFILE for crash diagnostics
statyyyyyyyyy	copy of statistics file made to retain statistics if the database is deleted
snapyyyyyyyyy	copy of metro (1) statistics file made for the same reason

SEE ALSO

garbo (1), garbostat (1)

BUGS

See WARNING.

NAME

garbostat — PS—algol database statistics file displayer (version 2.1)

SYNOPSIS

garbostat —f filename | [—m] —o

DESCRIPTION

Garbostat prints a report on the cumulative statistics gathered by *garbo (1)* or the snapshot statistics gathered by *metro (1)*.

The statistics file to be displayed can be specified in one of three ways:

(1) by the name *(dbname)* of the database it relates to;

(2) with the —o flag, by its database's 5-digit reverse octal number *dbnum* (these numbers are used as file suffixes by the CPOMS, and are the only way to refer to deleted databases);

(3) by filename, using the —f flag; this is used for displaying summary files produced by *precis (1)*. They need not retain the names *precis* gave them. The -m flag is required to display a summary of SNAP files.

In (1) and (2) the program will search the directory **$PSDIR** if this shell variable is set, or /usr/lib/ps/dbs otherwise. For these cases the —m flag can also be added; this will make *garbostat* look for SNAPxxxxx files produced by *metro (1)* instead of STATxxxxx files. The files are of identical format except for a field that identifies the program that made them.

The —f option is intended for examining statistics files that have been removed from the database directory in which they were created, or for summary files produced by *precis (1)*.

The report is 100 characters wide by 63 lines long (needs most of a landscape Perq screen to display it).

INTERPRETING THE STATISTICS

The types listed in the breakdown of objects by type are the types of the PS—algol abstract machine, not those directly visible to a PS—algol programmer. See Persistent Programming Research Report 11, "PS—algol Abstract Machine Manual", for more details.

The type breakdown includes two "types", (BITPLANE) and (BITMAP), that are actually subtypes of others: PNTR VEC and INT/B VEC respectively. They represent the objects reachable from image descriptors; bitmap vectors may be a significant part of the total space taken up by a database. The data for these is recorded twice, in the figures for the subtype and for the parent type.

The numbers produced by *garbo* and *metro* have different semantics; *garbo* produces a cumulative record of the space it has garbage-collected, whereas *metro* takes a "snapshot" of the database at a single instant, though it will use some of the cumulative information provided by *garbo* if there is any.

Garbo only counts accessible objects once; it does this by seeing whether an object's DATA file address is past the end of the file left by the last garbage collection - 'commit()' creates new objects by appending them. An object can also only become garbage once. A consequence of this is that the figure for "total accessible space scanned to date" may be misleading; some of it may have been found to be accessible at some past garbage collection, but might have become garbage since. There is no way to identify the class of very long-lived objects.

The statistics files record details of new objects (accessible or garbage) created since the last use of *garbo;* this information is retrieved from the old STATxxxxx file before *garbo* or *metro* create their new statistics files.

FILES

DIRIN	database directory file
STATxxxxx	cumulative statistics created by garbo (1)

SNAPxxxxx	snapshot statistics created by metro (1)
statyyyyyyyyy	copy of a STAT file from a db deleted by garbo (1) - yyyyyyyyy is the date of deletion in hexadecimal statistics if the database is deleted
snapyyyyyyyyy	copy of a SNAP file from a db deleted by garbo (1) - yyyyyyyyy is the date of deletion in hexadecimal
PRECSTAT	summary of a collection of STAT files made by precis (1)
PRECSNAP	summary of a collection of SNAP files made by precis (1)

SEE ALSO

garbo (1)
metro (1)
precis (1)
lsdb (1)

WARNING

If the figures for the amount of garbage present are nonsensical (typically *garbo* will report 0% compaction but *garbostat* says there was some garbage in the database) you have allowed an early version of the PS—algol interpreter to write to a database. The fix for this is to garbage-collect all databases and start afresh, ensuring that only versions of the interpreter from 4.0 onwards can still write to databases.

```
$ garbostat -m setPAIL
SNAPSHOT FILE FROM 'metro' v2.1    DATABASE NAME: "setPAIL"
**************************************************************************************************

DATA FILE SIZE IN BYTES AFTER LAST 'garbo': 0    DATA FILE SIZE IN BYTES AT LAST 'metro': 48388

MAXIMUM QUEUE SIZE: 253    # OF INWARD REFERENCES: 0

**** accessible objects **************************************************************************

            -- object count -- * ---- space occupied in bytes ---- * ------ number of pointers ------
TYPE           NUMBER * %TOTAL  *  MEAN  STD.DEV  MAXIMUM   %TOTAL  *  MEAN  STD.DEV  MAXIMUM   %TOTAL

STRING            318   57.30%    43.23   129.69     1604   28.41%        -        -        -        -
FILE                -        -        -        -        -        -        -        -        -        -
STRUCTURE          44    7.93%    26.64    52.80      364    2.42%     4.57    13.14       90   13.87%
PNTR VEC           27    4.86%    61.33    77.93      256    3.42%    12.33    19.48       61   22.98%
PROC VEC           14    2.52%    90.86   109.23      420    2.63%    19.71    27.31      102   19.05%
INT/B VEC           3    0.54%    16.00     0.00       16    0.10%        -        -        -        -
REAL VEC            -        -        -        -        -        -        -        -        -        -
FRAME               6    1.08%   386.67   293.11      996    4.79%    58.83    46.48      145   24.36%
CODE VEC          143   25.77%   197.01   719.29     6484   58.22%     2.00     0.00        2   19.74%
IMAGE DESC          -        -        -        -        -        -        -        -        -        -

(OVERALL)         555  100.00%    87.19   387.59     6484  100.00%     2.61    11.17      145  100.00%

(BITPLANE)          -        -        -        -        -        -        -        -        -        -
(BITMAP)            -        -        -        -        -        -        -        -        -        -

-- pointers -------------------------------------------------------------------------------------
NILS: 411   RESERVED: 45   TO THIS DB: 980   TO OTHER DBS: 13   TOTAL: 1449
% OF ACCESSIBLE DATA FILE SPACE TAKEN UP BY POINTERS: 11.98

-- pointers per object (taken over just those objects that have them) ----------------------------
MEAN:   6.19      STD.DEV:   16.55


**** garbage objects *****************************************************************************

            -- object count -- * ---- space occupied in bytes ---- * ------ number of pointers ------
TYPE           NUMBER * %TOTAL  *  MEAN  STD.DEV  MAXIMUM   %TOTAL  *  MEAN  STD.DEV  MAXIMUM   %TOTAL

STRING             40   47.06%    35.00    49.41      304   21.82%        -        -        -        -
FILE                -        -        -        -        -        -        -        -        -        -
STRUCTURE           2    2.35%    30.00    14.00       44    0.94%     2.50     0.50        3    1.99%
PNTR VEC           15   17.65%    25.07    17.80       88    5.86%     3.27     4.45       19   19.52%
PROC VEC            4    4.71%    54.00    29.05      100    3.37%    10.50     7.26       22   16.73%
INT/B VEC           -        -        -        -        -        -        -        -        -        -
REAL VEC            -        -        -        -        -        -        -        -        -        -
FRAME               2    2.35%   314.00   106.00      420    9.79%    55.50    26.50       82   44.22%
CODE VEC           22   25.88%   169.82   179.21      812   58.23%     2.00     0.00        2   17.53%
IMAGE DESC          -        -        -        -        -        -        -        -        -        -

(OVERALL)          85  100.00%    75.48   121.48      812  100.00%     2.95     9.72       82  100.00%

(BITPLANE)          -        -        -        -        -        -        -        -        -        -
(BITMAP)            -        -        -        -        -        -        -        -        -        -

-- pointers -------------------------------------------------------------------------------------
NILS: 60   RESERVED: 32   TO THIS DB: 159   TO OTHER DBS: 0   TOTAL: 251
% OF GARBAGE DATA FILE SPACE TAKEN UP BY POINTERS: 15.65

-- pointers per object (taken over just those objects that have them) ----------------------------
MEAN:   5.58      STD.DEV:   12.80
$
```

Sample output from "garbostat"

NAME

lsdb — list PS—algol databases

SYNOPSIS

lsdb [—a]

DESCRIPTION

Lsdb lists PS—algol databases, using information in the database directory file *(DIRIN* or *DIROUT)* in the directory $PSDIR if this is set, or in the default directory */usr/lib/ps/dbs* otherwise.

Lsdb displays the following:
- the pathname of the directory file referred to;
- if the directory is locked, the lock file's pathname;
- the pathname of any GARBORUNNING or METRORUNNING file left by a crash in *garbo* or *metro;*
- for each database, its:
  * name
  * octal suffix
  * root partition number
  * lock state: 0 - unlocked
    -1 - write locked
    +n - read locked by n readers
  * size of data file (bytes), or "DELETED" if absent
  * size of index file (bytes), or "DELETED" if absent
  * whether any STAT or SNAP files exist for it; this is displayed in the S column:
    - if there is an "S", a STAT file exists
    - if there is an "s", a SNAP file exists
  * owner

  and, if 'root' or 'ps' is running the program,
  * password.

If a database has a root partition of -1, meaning that its database directory entry can be reused and its files may be deleted, *lsdb* will not display it unless invoked with the —a flag.

USING THE INFORMATION

The usual reasons for using this utility are:

(a) Some of the databases are locked so that programs don't run; this is due to a crash while one of *psr (1), garbo (1),* or *metro (1)* was running. If the crash occurred while *psr* was doing a commit, there may be a before-look (BFRE) file present. Use *restoredb (1), garbofixup (1),* and/or *safecrack (1)* to get things back to normal.

(b) Filestore space is running low; in this case, compact the data files of the databases with *garbo (1).*

(c) The user wants to know what statistics files exist for a particular database.

SEE ALSO

psr (1), garbo (1), metro (1), garbofixup (1), restoredb (1), safecrack (1), garbostat (1).

```
$ lsdb -a
DB DIRECTORY FILE = /ps/src/garbo/testdbs/DIRIN
db directory locked: file /ps/src/garbo/testdbs/DIRLOCK exists
"metro" running or crashed: file /ps/src/garbo/testdbs/METRORUNNING exists
NAME            PASSWORD      SUFFIX ROOT   LOCKS DATAsize    INDXsize    S  OWNER
sys.database    *fred*pete*   00000  1      -1    51200       3072        S- ps
FONTS           friend        10000  2      -1    65536       3584        S- ps
test1           gc            20000  3      -1    512         512         S- ps
test2           gc            30000  4      -1    5120        2560        Ss ps
test3           gc            40000  5      -1    512         512         S- ps
setPAIL         aleph         50000  6      -1    48640       2560        -- ps
                aleph         60000  7      -1    240128      20480       S- ps
                friend        70000  -1     -1    DELETED     DELETED     -- ps
                friend        01000  -1     -1    DELETED     DELETED     -- ps
                friend        11000  -1     -1    DELETED     DELETED     -- ps
                friend        21000  -1     -1    DELETED     DELETED     -- ps
                aleph         31000  -1     -1    DELETED     DELETED     -- ps
$
```

Sample output from "lsdb"

(executed by the PS-algol system owner, hence the displayed passwords)

NAME

metro — persistent store statistics collector (version 2.1)

SYNOPSIS

metro [—f] —a | dbname1 dbname2 ... | —o dbnum1 dbnum2 ...

DESCRIPTION

Metro takes measurements on PS—algol databases. The databases to be examined may be specified in three ways:
(1) by the **—a** flag (do all of them)
(2) by database names: *dbname1, dbname2, ...*
(3) by the **—o** flag followed by a sequence of 5-digit reverse octal numbers: *dbnum1, dbnum2,* ... which are the suffixes of the database's files (these are displayed by *lsdb (1)*). *This is useful for* be referred to by name.

It searches for database files in the directory specified by the shell variable $PSDIR, or in */usr/lib/ps/dbs* if this is not set.

Metro produces statistics on the contents of the databases in files called SNAPxxxxx, where xxxxx is the reverse octal suffix of a database's files. The information in these files is displayable by *garbostat (1)* or summarizable by *precis (1)*.

This program is intended to provide a "snapshot" of the database, and the SNAPxxxxx files do not form a cumulative record as the STATxxxxx files created by *garbo* do. If a SNAPxxxxx file already exists for a database, *metro* will refuse to analyze it, so the SNAPxxxxx file cannot be accidentally overwritten.

The **—f** option makes the program announce file operations: create, open, close, link and unlink.

FILES

DATAxxxxx	data file to be compacted
INDXxxxxx	database index file
NEWINDXxxxxx	temporary version of index file
SNAPxxxxx	snapshot statistics file
METRORUNNING	link set while metro is in action; does nothing, purely to tell inquisitive minds why their programs aren't working
DOINGxxxxx	linked to LINKFILE for crash diagnostics
DIRIN	db directory file (read only)
DIROUT	db directory file (write only)
DIRLOCK	link to LINKFILE for locking db directory
LINKFILE	empty file used for linking to

DIAGNOSTICS AND RECOVERY PROCEDURES

Metro works in two phases; in the first it locks the database directory and all the databases. (If a before-look (BFRExxxxx) file or a snapshot (SNAPxxxxx) file is present, it will refuse to proceed further.) It then constructs tables of cross-references between databases. If it crashes in this phase, unlocking the directory and the databases will restore the status quo.

The second phase is the statistics collection proper; *metro* creates the DOINGxxxxx link so that *garbofixup* or *metro's* own recovery routines can see whether there may be some temporary files to remove if *metro* crashes. The DOING file is unlinked immediately after *metro* has deleted them.

If there is a DOING file, or any NEWINDX files, present in the dbs directory after *metro* has failed, *Garbofixup* can be used to restore the database files to their state before *metro* was applied. *Garbofixup* does not unlock the database affected; use *restoredb* to do that, or, if a deleted database is locked, get the PS—algol system owner 'ps' to run *safecrack (1)* which will forcibly unlock all databases.

As from version 1.3, the error/interrupt handling routines should restore the persistent store to a usable state for any failure short of a kill or machine crash.

SEE ALSO

lsdb (1)
garbostat (1)
precis (1)
garbofixup (1)
safecrack (1)

NAME

precis — compress statistics files from *garbo (1)* or *metro (1)* (version 2.1)

SYNOPSIS

precis [-m] [-d directory] [-f filename1

DESCRIPTION

Precis generates a file summarizing the information contained in a set of statistics files in the format produced by *garbo (1)* or *metro (1)*. These files are specified as follows:

If the -m flag is used, *precis* will look for STATxxxxx and statyyyyyyyyy cumulative statistics files produced by *garbo (1)* and will produce a summary in a file called PRECSTAT. if not, it will look for SNAPxxxxx and snapyyyyyyyyy statistics files produced by *metro (1)* and will produce a summary in a file called PRECSNAP. The summary files retain a memory of which program produced the original data they were made from; this affects *garbostat (1)'s* display. It makes no sense to mix these two types, so *precis* won't let you.

If the -f or -d flags are not given, *precis* will look in the directory **$PSDIR**, or in **/usr/lib/ps/dbs** if this variable is not set, and create the summary file there.

If the -f flag is used, *precis* will take the remaining arguments as a list of filenames of statistics files and create the summary file in the current directory.

If the -d flag is used, *precis* will treat all files in 'directory' as potential statistics files and will create the summary file in that directory.

The summary files can be displayed by *garbostat (1)*.

WARNINGS

Precis will not permit its own output files to be further summarized.

Databases produced with early versions of the interpreter may give meaningless figures for the amount of garbage present. See the manual entry for *garbo (1)*.

SEE ALSO

garbo (1)
metro (1)
garbostat (1)

NAME

safecrack — forcibly release locks on all databases (version 2.1)

SYNOPSIS

safecrack

DESCRIPTION

Safecrack releases all locks on any databases in the PS—algol system, whether or not they have been deleted. Before doing this it checks to see if there are any beforelook (BFRE) or garbo backup (BEF) files present; it will refuse to continue if there are.

DIAGNOSTICS

A file called SAFECRACKING is created while the program is working on the database directory file. *Safecrack* will refuse to unlock databases while there are any before-look (BFRE) or *garbo* backup (BEF) files present; this is to avoid catastrophic corruption of the corresponding databases. In this situation, use *restoredb (1)* or *garbofixup (1)* first, whichever is appropriate.

SEE ALSO

restoredb (1)
garbofixup (1)

```
/* ************************************************************** */
/* STATISTICS DEFINITIONS MODULE - stat.h                          */
/* ************************************************************** */

/* last edited: Tue Jun 24 16:13:28 GMT 1986                       */

/* This module defines the record types for a STAT file.           */

#define GARBO  1  /* bit 0 value for filetype field   */
#define METRO  2  /* bit 1   "     "      "        "    */
#define PRECIS 4  /* bit 2   "     "      "        "    */

typedef struct {
      int tally;     /* number of objects of this type        */
      int maxsize;   /* size of type's biggest object (bytes) */
      int size;      /* total size of this type (bytes)       */
      int size_2;    /* sum of squares of sizes               */
      int maxpntrs;  /* largest number of pointers found      */
      int pntrs;     /* number of pointers in objects         */
      int pntrs_2;   /* sum of squares of number of pointers  */
} typestats;

typedef struct {
      int nils;       /* nil pids                             */
      int reserved;   /* reserved pids                        */
      int nonpids;    /* with bit 31 unset                    */
      int outptrs;    /* pids from other dbs                  */
      int ownrefs;    /* pids from this db                    */
      int newspace;   /* new objects since last garbo (bytes) */
      int sofar;      /* total space found in all gcs (bytes) */
      int slot1;      /* for afterthoughts                    */
      int slot2;      /* ditto                                */
      typestats s;    /* strings                              */
      typestats f;    /* files                                */
      typestats p;    /* structures                           */
      typestats vp;   /* vectors of pointers                  */
      typestats vpr;  /* vectors of procs                     */
      typestats vib;  /* vectors of int/bools                 */
      typestats vr;   /* vectors of reals                     */
      typestats fr;   /* frames                               */
      typestats pr;   /* code vectors                         */
      typestats im;   /* images                               */
      typestats bm;   /* bitmap vectors                       */
      typestats bp;   /* bit plane info                       */
      typestats pro;  /* Prolog objects from Aberdeen         */
      typestats t1;   /* for an additional type               */
      typestats t2;   /* for an additional type               */
} breakdown;
```

```
typedef struct {
        int  version;     /* version number                       */
        int  release;     /* release number                       */
        int  gc_tally;    /* number of gc's done on db so far     */
        int  maxqsize;    /* maximum size reached by queue        */
        int  bodycount;   /* dead pids found so far               */
        int  inrefs;      /* references inward from other dbs     */
        char filetype;    /* one of GARBO or METRO bits must be   */
                          /* set; the PRECIS bit is optional      */
        int  dummy1;      /* for afterthoughts                    */
        int  dummy2;      /* for afterthoughts                    */
        int  sizeIn;      /* size of DATA file in ints at the end */
                          /* of a garbo - not including padding   */
                          /* to 512-byte blocks                   */
        int  sizeOut;     /* size of DATA file in ints at the end */
                          /* of a garbo, not including padding,   */
                          /* or else the amount of accessible     */
                          /* data found by metro                  */
        int  filecount;   /* for precis - number of files used    */
        breakdown saved;  /* info on accessible objects           */
        breakdown trash;  /* info on garbage objects              */
} statistics;
```

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.
Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics"; The Association for Computing Machines, 11 West 42nd St., New York, NY 10036; University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

In Preparation

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : A DBMS based on the functional data model", to be submitted.

Atkinson, M.P. & Buneman, O.P.
"Database programming languages design", submitted to ACM Computing Surveys - see PPRR-17-85.

Theses

The following Ph.D. theses have been produced by member of the group and are available from:

The Secretary,
Persistent Programming Group,
University of Glasgow,
Department of Computing Science,
Glasgow G12 8QQ,
Scotland.

W.P. Cockshott
Orthogonal Persistent, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 28th October 1986.

Copies of documents in this list may be obtained by writing to:

The Secretary,
The Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - third edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00