Database Integrity Checking

Just as there are lies, damn lies, and statistics, there are, on a computer, productive work, a security overhead, and integrity checking. Database integrity checking is distinguished by its enormous cost in machine time, and by the paradox that it is only useful when it fails.

The dictionary definition of integrity - 'unimpaired or uncorrupted state; original perfect condition; soundness' - although it was coined long before the advent of data processing, provides a remarkably accurate description of its technical use in computing. In passing, it is very interesting how words such as 'corruption' and 'integrity', which fall naturally from the tongues of Puritan moralists, should also be the common currency of computer programmers in the 20th Century. The 17th Century Puritan's striving after moral perfection is quite accurately parallelled by our 20th Century striving after technical perfection, with the consequence of a single lapse being perdition in either case; 'fault-tolerance' in either religious beliefs or computer systems is subsequently developed because of the impossibility of reaching perfection.

Integrity in computer files is most easily described by a list of requirements, lapse from any one of which will cause integrity to be lost. The data must be readable by the hardware, it must conform to the format defined for the software-access method being used, and it must be reasonable in the field values presented to the application program. It can, however, retain its integrity while falling short of perfection in one important point; it may not be correct. If a human being has input to a computer system data that is entirely reasonable but happens to be incorrect, then it would not be useful to suggest that the computer system lacks integrity; it may be inadequately specified, if the error should have been detected, but, in terms of the specification to which it was written, it has retained its integrity.

A further point arises from this last one; ultimately, a computer system's integrity is defined in terms of the input it will accept from, and the output it will return to, human users. However, a prerequisite of that integrity is that the data on backing store shall retain its uncorrupted state, and the integrity discussed in this article is concerned with the data held on backing store by the computer system.

To understand why it is required, and how to do it at minimum cost, it is first necessary to understand the problem of database integrity. As a previous article in this column c, discussed ('Providing backup' Vol I No4 (March 1979), one of the consequences of the move from serially based batch systems to random-access-based real-time systems has been that automatic integrity checking is no longer provided. The serial update, in which yesterday's output file is today's input file and tomorrow's backup file, enforces a degree of automatic checking on the file that will later be used as a backup file. For it to be used successfully as an input file, all its records must be readable and, in the course of reading it, it is straightforward to arrange for control totals of important fields to be accumulated and checked against totals held in an end-of-file record. Therefore, by the time the file becomes the backup file, its read-ability and integrity can be assured. .It is always possible, of course, to discover at a very late stage that the file, and all its backup copies, are *incorrect*, but that is a slightly different matter; if integrity failures are like being handed a rotten apple instead of a good one, then incorrect systems are like being handed a lemon instead. No one wants rotten apples, but some people ask for (and get) lemons.

However, once we move on to random-access databases that are updated *in situ*, the updating process no longer assures us of any integrity in our backup copies. Problems can arise because of undetected failures both in the update and in the subsequent backup process.

What, in particular, are we talking about when we discuss database corruption or integrity? At the crudest level, we need to know that the computer is capable of taking the data off its disc storage and bringing it into buffers in main store. This implies that a very large number of procedures go correctly; the read heads must be able to position themselves over the surface of the disc and detect the presence or absence of bits on the track, the disc controller must be able to sort out the stream of bits into a sensible format, the central processor must be able to obtain the block from the disc controller, and parity checks, cyclic redundancy checks etc. must all be passed. However, viewed from the database system, the check is very simple; does it get the block that it asked for? If it does, for every block in the database, then the first level of checking, that of block readability, has been passed.

The next level of integrity is that of the block contents. The database management software must be able to find its way around the block, distinguishing block headers, pointers and records. If it cannot do so because, for example, a field that should contain a record length actually contains an impossibly large number, then, although the block is readable, its contents are corrupt.

There are, finally, two further parallel levels of integrity. On the one hand, pointers must point to somewhere sensible, e.g. the beginning, not the middle, of a: record, and, on the other hand, records must contain reasonable data, e.g. dates of births of living people must normally be in the last hundred years. If either of these two types of check fail, then, although the pointers or records are readable, their contents are corrupt.

It is plain when we reach this point that the individual user has a choice as to what he includes in a database's integrity and what he trusts to luck and correctness. The choice for record contents is straightforward; if he wishes, he can, in the course of integrity checking, repeat all the range checks etc. that were performed before the data was allowed into the database. As discussed below, because recovery from incorrect record contents does not normally give too much difficulty, their correctness is not usually made a crucial part of the definition of a database's integrity.

Pointers, on the other hand, present more of a problem. If a pointer points to a reasonable, but incorrect, record, then the work of unscrambling the problem may be very great indeed. There is therefore a good case for including a high degree of pointer checking in deciding on a database's integrity. Two devices, both of which must be included during database design, are useful. Backward pointers can prove that a record is not the correct one to be pointed to; if record A points to record B, and record B should point back to record A, then, if it does not, there is clearly a problem. Unfortunately, this method does not provide positive proof of correctness. On the other hand, inclusion of higher-level keys in records can provide positive assurance of correctness. In a geographical database, if a 'Road' record points to a series of 'House' records, then, if each House record contains the Road name, we can be assured that the pointer structure is, or is not, correct. We can provide ourselves with the means of checking for integrity at the cost of additional storage space.

A crucial factor in the discussion of database integrity is the time taken to recover from failures. It is, in principle, possible to recover from any failure provided enough backup copies are kept and enough machine time is available for reprocessing after the error has been corrected. If a database integrity failure is not discovered until a month after it occurred, because it was on a seldom-accessed portion of the file, then it is clearly possible, in principle, to identify and correct the cause of the error, reload the database to its state of a month previously, and reprocess a complete month's input. However, in most circumstances, this will be impractical. An important factor in deciding on an approach to integrity checking is the amount of reprocessing time that could be tolerated in the worst. possible circumstances. In most installations, this will be more than 12 h, which could probably be done in a weekend, and less than 3 days, which could not.

One approach to database integrity bypasses the entire reprocessing problem by assuming that any database corruption can be 'fixed' on detection by a database-mending program, probably using hexadecimal patching, which takes very little time to run. Apart from the fact that, although the database mend may be short in duration, the subsequent database backup copy-taking may not, there is a limit to the extent of a database corruption which can be mended reliably by this means. The amount varies enormously, depending on the power of the mending tools and the ability of the technicians, but it is probably fair to suggest that database integrity failures extending over more than a few physical blocks will normally be impossible to mend. Certainly it is not sensible to plan for database maintenance on the assumption that all cases of corruption can be dealt with by mending. Even if the 'mending' approach to recovery is taken, it is still necessary to be able to check out the database afterwards, to give confidence in the correctness and completeness of the fix.

There is, however, the possibility of reducing the time to reprocess while recovering from a database error; if the entire database is partitioned, and the database update processes are structured so that, on reprocessing, updating can be constrained to specified partitions, the reprocessing time is reduced correspondingly. This assumes, of course, that the error has been identified and corrected before reprocessing takes place. Also, partitioning of update processes can only be achieved if the structure is designed into the programs right from the start.

The discussion so far has not answered the question of why we should want to do integrity checking at all. It is completely unproductive, and does not even provide direct help in recovering from any errors that it may uncover. The answer is the same as that which we would give to someone who refuses to go to the doctor because he is afraid that he has a disease and does not want it confirmed; this would only be a rational attitude if there were grounds for belief that he would suffer worse from the cure than from the disease. Fortunately, that is no longer usually true of database management systems, and the sooner database corruption is diagnosed the more easily it can be cured.

The ideal situation is a regular 100% check of the whole database. With some application systems, it is possible to incorporate at least part of this into production runs, particularly if there is a requirement for 100% processing of the records of a part of the database. Normally, however, the checking has to be done by specially written programs, unconnected with other production work, which progress through the database ensuring that all possible access paths remain correct. It is possible to incorporate record-contents checks at the same time, but this is not essential, because records with incorrect contents can normally be recovered by deleting them and reinserting them, without any fixing or reprocessing being necessary.

Full database checking on a regular basis may or may not be possible, depending on the operational schedule, although, if it is not possible in normal circumstances, a question mark must be raised about the installation's capacity to function if some of the equipment fails. In any case, the checking programs must be written and available, because there will inevitably be times after recovering from errors when a confidence check is required.

In addition to this, there are very positive advantages to incorporating as much checking as possible into the real-time programs that access the database. Many database errors are immediately apparent to the system, because it finds itself unable to carry out the requested task, but some are not so obvious. If, to take the example used above, a House record becomes attached to the wrong Road, it is much better for the system to discover this at once rather than for a VDU user to find himself in a confusing and incomprehensible situation.

Integrity checking, like a medical checkup, is about confidence. It will cost both design effort and machine time to give a database a regular clean bill of health, but it can pay for itself in user confidence and ease of recovery. *J Moffett*