Partly-Consistent Cuts of Databases

Marcin Skubiszewski and Nicolas Porteix
INRIA and O_2 Technology

Abstract. We introduce *partly-consistent cuts*, a mechanism analogous to causal cuts, defined in the context of object-oriented databases. A partly-consistent cut is a collection containing one or more copies of every page in a given database; the copies are made during the operation of the database, at different times. We consider four classes of partly-consistent cuts. Each class implies a different set of constraints imposed on the times when the copies are made. The consistency properties (*i.e.* the ability to correctly represent what happens in the actual execution of the database) of the cuts in each class are analyzed. One class, called *GC-consistent cuts*, can be used by a concurrent garbage collector to determine which objects to delete.

1 Introduction

In many situations, an observer needs to examine the current state of a complex system. For example, a bank employee may want to know how much money a given client has. This involves reading the balance of all the accounts held by the client; such accounts may be numerous, and may be stored in different computers.

Normally, the observer needs to examine all parts of the system at once. Otherwise, incorrect information may be obtained: for example, if money is transfered from an account x to another account y, and the bank employee examines x before the transfer and y after the transfer, then the money transfered will be incorrectly counted twice.

In many cases, however, it is complicated or costly to examine many different parts of a system at once. This difficulty generates a need for methods that make it possible to observe different parts of the system at different times, and still obtain correct information. One such method is known under the name of *causal cuts*.[1] Causal cuts are defined in *asynchronous distributed systems*, a class of distributed systems where information is transmitted between processes asynchronously, in the form of messages in communication channels. Asynchronous distributed systems are a theoretical concept, created to study the facts about computations that do not depend on the actual time and the actual speed at which operations are performed.

In this paper, we introduce *partly-consistent cuts*, a concept that results from transposing the philosophy behind causal cuts to object-oriented databases (OODB). Partly-consistent cuts can be used to observe various properties of a

[1] Chandy and Lamport [3] introduced a "global-state recording algorithm" that implements one kind of causal cut; then, Mattern [6] defined causal cuts in the general case. Babaoglu and Marzullo [1] describe causal cuts in detail.

database, without accessing the whole database at once, and without interfering with its normal operation.

We describe four different classes of partly-consistent cuts. Each class corresponds with a different *consistency property*, *i.e.* category of facts about which the cuts are guaranteed to deliver correct information. Correlatively, each class follows a different set of *timing constraints*, *i.e.* of constraints on the times when different parts of the system are observed.

Our results have an important practical implication: they lead to a new method for *concurrent garbage collection* in OODB, *i.e.* to a method that allows a garbage collector (GC) to perform its work without interrupting or disturbing the normal operation of the database. The method has been implemented in the commercial system O_2; it is described elsewhere [8].

The paper is organized as follows. Section 2 describes the properties of OODB and the assumptions about OODB that are used in this work. Section 3 defines partly-consistent cuts. Section 4 describes a consistency property shared by all partly-consistent cuts. In Sections 5–7, we describe the different classes of partly-consistent cuts, in a way that shows how the consistency properties relate to the corresponding timing constraints. Section 8 summarizes our results.

For the sake of brevity, we omit the proofs and we do not discuss the analogy which exists between partly-consistent cuts and causal cuts (this analogy is strong, although nonobvious). Publication [7] contains the proofs of all the facts stated in this paper, and discusses the analogy.

2 Definitions and assumptions about databases

Parts and transactions in databases. We view data in a database as being partitioned into parts numbered $0, ..., n-1$. Each object x belongs to exactly one part, noted $P(x)$. Usually, parts are database pages, but alternatively their rôle can be played by objects (in this case, $P(x) = x$) or by other entities.

An execution of a database management system is divided into chunks called *transactions*. Each transaction locks the parts to which it has access. A lock permits a transaction either only to read or both to read and to write objects in the specified part.

We assume that transactions are atomic and serializable; see [4] for the definitions of these terms. These assumptions allow us to consider transactions as null-duration events that take place in sequence.

The graphical notation and the transaction clock. We represent executions of database systems as shown in Fig. 1. Time flows from left to right. Each part is represented by a thin horizontal line. Each transaction is considered as an atomic null-duration event and represented by a thick black vertical line. If a transaction reads a part, the corresponding lines cross; if it also writes the part, an arrow is drawn at the crossing. For example, the leftmost transaction in the figure reads and writes part 0, reads part 1, and does not access part 2.

When talking about a database execution, we use a special real-valued global clock called *transaction clock*. This clock takes value 0 at some time before the

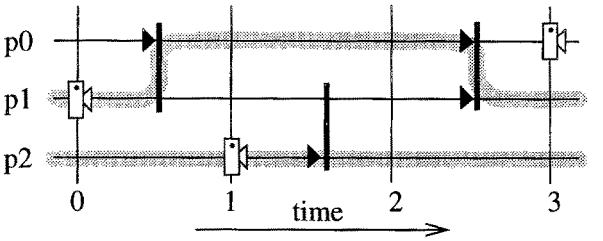

Fig. 1. Example execution of a database.

first transaction, then, in an execution including n transactions, it takes each integer value $t \in [1\,..\,n-1]$ at some time between the t-th and the $t+1$-th transaction. Value n is taken at some time after the n-th transaction. For every t, during the t-th transaction the value of the clock is strictly included between $t-1$ and t. (We do not assume that the database system has access to the transaction clock; we only use the clock to talk about the system.)

Two elements in Fig. 1, namely cameras and very thick gray lines, will be explained later, respectively in Sections 3 and 5.

Reachability, garbage, and garbage collection. We use a classical model of reachability, based on the fact that before accessing an object, a transaction must first access a pointer to it.

The database is assumed to contain a fixed set of indestructible objects called *roots*. Pointers to roots are system constants, to which all transactions have access. Every transaction has access to the objects that it has created. Outside of these two cases, an object can only be accessed by a transaction once this transaction has read a pointer to the object, from a pointer field in another object present in the database. There are no other possibilities for a transaction to obtain a pointer value (*e.g.* it is illegal to perform pointer arithmetic or to store pointers in places other than pointer fields of objects).

This model is usually enforced in OODB. This is necessary for the garbage collection to operate correctly: without such a precise model, the GC could never ascertain that a given object is unreachable, and could therefore not delete objects.

An object is said to be *reachable* at a given time t iff it exists at time t and the first transaction that will take place after t can access it. According to the rules above, the following is a correct definition of reachability.

Definition 1 (reachability in databases). *The objects* reachable *in a database execution E at time t form the smallest set such that*

1. *roots are reachable*
2. *and if at time t object x is reachable and object y exists and x contains a pointer to y, then y is reachable at time t.*

An object that exists but is not reachable is called *garbage*. It is the purpose of the garbage collector to delete garbage objects.

3 Cuts in databases

The definition. The copy of part number i, taken at time t, is noted (i,t) and is called a *snapshot*. Since we consider transactions as atomic events, we only accept the possibility of taking snapshots between transactions, *i.e.* at integer times. On figures, snapshots are represented by cameras. For example, Fig. 1 shows the snapshots $(0,3)$, $(1,0)$ and $(2,1)$.

A set of snapshots containing at least one snapshot of each part is called a *cut*. A cut is called *simple* iff it contains one and only one snapshot of each part; otherwise, it is called *multiple*.

We define the *time interval* of a cut to be the interval from the time when the first snapshot in the cut is taken, to the time when the last snapshot is taken, inclusively. If an event happens during the time interval of a cut C, we say that it happens *during* C.

Reachability and garbage in cuts. Among others, cuts may be used to determine the reachability of objects. For this purpose, the following definitions are used.

Definition 2 (presence; inconsistent presence). *Let C be a cut. An object x is* present *in C iff C contains a snapshot that contains a copy of x,* i.e. *a snapshot of $P(x)$ taken at a time when x exists; otherwise, x is* absent *from C. x is* inconsistently present *in C iff C contains both a snapshot of $P(x)$ taken when x exists, and a snapshot of $P(x)$ taken when x does not exist.*

Definition 3 (reachability in cuts). *Let C be a cut. Objects reachable in C form the smallest set such that*

1. *roots are reachable in C;*
2. *objects inconsistently present in C are reachable in C;*
3. *if object x is reachable in C and a copy of x present in C contains a pointer to object y and y is present in C, then y is reachable in C.*

To understand this definition, let us first assume that the cut C is simple. Under this assumption, no objects are inconsistently present in C, and rule 2 in the definition is not applied. Then, the definition implements the idea that a cut should be treated as if it represented a state of an execution, copied at some time (in reality, a cut is usually not such a state, because different snapshots are taken at different times). In the context of simple cuts, Definition 3 is indeed equivalent to what we would obtain by substituting the words "at time t" with words "in cut C" in Definition 1, which defines reachability in databases.

Now, consider the case of multiple cuts. In this case, rule 2 in the definition applies. To explain the rationale of the rule, let us observe that an object x may only be inconsistently present in C if it has been created or deleted during C. This, in turn, implies that some transaction had access to x during C. x was therefore reachable at some time during C, and, for C to be a faithful representation of what happened in the actual system, it should be reachable in C.

An object x that is present in a cut C and not reachable in C is said to be *garbage* in C.

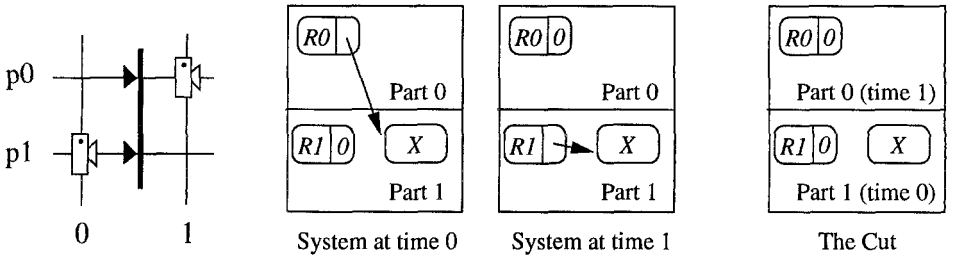

Fig. 2. Object X is reachable in the execution, and garbage in the cut ($R0$ and $R1$ are roots; X is not a root).

4 Arbitrary cuts

Theorem 4 (preservation of garbage). *Every object that is constantly garbage in a database execution E during a cut C of this execution, is garbage in C.*

(Let us recall that the proofs of all theorems can be found in [7].)

The converse of this theorem is not true: from the counterexample shown in Fig. 2, it results that an object that is constantly reachable in an execution while a cut is being taken may be garbage in the cut.

The theorem goes against the common-sense idea that no reliable information can be obtained by observing different parts in a database at arbitrary times, with no effort to guarantee consistency. It implies, for example, that by examining different objects in the system at arbitrary times, we can estimate the number and the total size of garbage objects, with no risk of underestimation.

5 GC-consistent cuts

Definition 5 (path). *Let E be a database execution, comprising n transactions; we assume that the database contains m parts. A* path *in E is a function H that goes from the set of integer times of the transaction clock to the set of parts (in symbols: $H : \{0, ..., n\} \to \{0, ..., m-1\}$) and that satisfies, for every $t > 0$ belonging to its domain, one of the following conditions:*

1. $H(t) = H(t-1)$
2. *or the transaction that takes place between times $t-1$ and t locks part $H(t-1)$ for reading or writing, and part $H(t)$ for writing.*

A path represents the way in which a pointer present at time n in some part i may have been successively copied during E in order to reach this part. According to the definition, $H(t-1)$ and $H(t)$ either are equal (this corresponds with the situation where a pointer value is not copied) or are chosen so that the transaction that happens between times $t-1$ and t has the possibility to copy a pointer from part $H(t-1)$ to part $H(t)$. The latter means that the transaction locks part $H(t-1)$ for reading or writing, and part $H(t)$ for writing.

In Fig. 1, two example paths are represented by very thick gray lines (other paths exist in this execution). The lower one is straight. This corresponds with a constant path—a path that stays in the same part during the whole execution. The upper one shows that a pointer value located in part 1 at time 3 might be there because between times 2 and 3 it was copied there from part 0, after being copied from part 1 to part 0 between times 0 and 1.

Definition 6 (GC-consistent cut). *Let E be a database execution. A cut C of E is* GC-consistent *iff it crosses every path,* i.e. *iff for each path H in E there exists some time t satisfying $(H(t), t) \in C$.*

Theorem 7 (absence of false garbage). *Let C be a GC-consistent cut of a database execution E. Then, an object can be garbage in C only if it is garbage in E, at some time during C.*

The theorem implies that an anomaly similar to the one in Fig. 2 cannot occur with a GC-consistent cut.

A GC-consistent cut can be used for garbage detection in a concurrent GC: instead of looking at the real system (whose content cannot be examined in a simple way, because it can change at any time and in arbitrary ways), the GC examines a GC-consistent cut of the system, and determines which objects are garbage in the cut. These objects are then deleted from the system. This method does not jeopardize the safety of the GC, because according to Theorem 7, only objects that are actually garbage in the system can be garbage in the cut. Similarly, it results from Theorem 4 that the use of a GC-consistent cut does not jeopardise the *liveness* of the GC: the objects which are garbage in the system during the operation of the GC are also garbage in the cut, and are therefore deleted by the GC.

An industrial GC has been built according to this principle, and is part of the commercial OODB management system O_2.[2]

6 A consistency property of simple GC-consistent cuts

Simple GC-consistent cuts (that is, cuts that are both simple and GC-consistent) can be used to solve the problem described in the introduction to this paper, *i.e.* to compute the total balance of several bank accounts stored in a database. Let us describe this problem formally. If v is a variable stored in the database, then the value of v at time t is noted v^t. The value of v present in the simple cut C is noted v^C. (This makes sense because C is a simple cut, and therefore contains one and only one copy of v. We do not define v^C in the case where C is a multiple cut.)

Consider an execution E of a database containing integer variables $v_0, ..., v_l$. We assume that each variable represents a bank account. Money can be transfered between the accounts, but the total balance remains constant: the value

[2] See [8] for a description of the garbage collector, and [2] for a description of O_2. For general information about garbage collection and for a description of other methods for concurrent garbage collection, see Wilson [9] or Jones and Lins [5]. For references about garbage collection in databases, see also [7].

	Preservation of garbage	Absence of false garbage	Sum of bank accounts	Dangling pointers
Defined in theorem number	*4*	*7*	*8*	*10*
Causal cuts	×	×	×	×
Simple GC-consistent cuts	×	×	×	
GC-consistent cuts	×	×		
Arbitrary cuts	×			

Table 1. Consistency criteria and classes of cuts.

$s = \sum_{j=0}^{l} v_j^t$ does not depend on t. The following theorem implies that s can be computed using a simple GC-consistent cut of E.

Theorem 8 (sum of bank accounts). *Let E be an execution of a database that contains several integer variables $v_0, ..., v_l$. We assume that the sum of the variables remains constant over time, and is equal to s. Then, for any simple GC-consistent cut C of E, we have $s = \sum_{j=0}^{l} v_j^C$.*

There is no equivalent to this theorem for GC-consistent cuts in general, *i.e.* there is no general method to compute the value s using a GC-consistent cut of E: a counterexample quoted in [7] shows that the same multiple GC-consistent cut may correspond with two different executions, where the values s are different.

7 Causal cuts of databases

In this section, we introduce *causal cuts of databases* (not to be confused with the unqualified "causal cuts").

Definition 9 (causal cut in a database). *A cut C of a database execution E is causal iff it crosses every path H of E in exactly one point,* i.e. *iff for every path H, there is one and only one time t such that $(H(t), t) \in C$.*

It results from this definition that every causal cut of a database execution is also a simple GC-consistent cut of this execution.

Let us exhibit a consistency property of causal cuts of databases, that simple GC-consistent cuts do not have. We say that a pointer is *dangling* in a database execution at time t if, at time t, it points to an object that currently does not exist. In a cut C, a pointer is dangling iff it is present in C, but the pointed-to object is not present in C.

A simple GC-consistent cut may contain a dangling pointer even if such pointers never appear in the underlying execution, and even if no objects are deleted during the execution (to understand why we quote the latter condition, compare this sentence with Theorem 10 below). An example illustrating this possibility is quoted in [7]. The situation is different for causal cuts:

Theorem 10. *Let E be a database execution in which no pointer is ever dangling and during which no objects are deleted. Then, no pointer is dangling in a causal cut of E.*

The theorem implies that in a system that does not delete objects, we can use causal cuts to detect dangling pointers in an execution, and no false alarms will be issued (although, conversely, some dangling pointers may remain undetected). Simple GC-consistent cuts cannot be used for this purpose.

8 Summary

We have introduced partly-consistent cuts, a principle that can be used to obtain meaningful information about the state of a database by observing different parts of the database at different times. We have studied four classes of partly-consistent cuts. The classes are ordered by strict inclusion, as follows: arbitrary cuts of databases (this is the biggest class), GC-consistent cuts, simple GC-consistent cuts, and causal cuts of databases.

Table 1 summarizes the consistency criteria that are met by the different classes of cuts. Each criterion is described with a reference to the theorem where it is first used, and with a name. A cross in the table means that all the cuts in a class meet a criterion. Conversely, for every cross missing, we know that some cuts in the class fail to satisfy the criterion.

A garbage collector based on cuts has been developed in O_2, a commercial OODB management system.

References

1. Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55–96, chapter 4. Addison-Wesley, 1993, second edition.
2. Francois Bancilhon, Claude Delobel, and Paris Kannellakis. *Building an Object-Oriented Database: the O_2 Story.* Morgan Kaufmann, 1991.
3. K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
4. Jim Gray and Andreas Reuter. *Transaction processing: Concepts and Techniques.* Morgan-Kaufmann, 1993.
5. Richard Jones and Rafael Lins. *Garbage Collection.* Wiley, 1996.
6. Friedmann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.
7. Marcin Skubiszewski and Nicolas Porteix. GC-consistent cuts of databases. Research Report 2681, INRIA, April 1996. Available from `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2681.ps.gz`.
8. Marcin Skubiszewski and Patrick Valduriez. Concurrent garbage collection in O_2. In *International Conference on Very Large Data Bases (to appear)*, 1997.
9. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.