Relational Theory

for Computer Professionals

What Relational Databases

Are Really All About

C. J. Date

Relational Theory for Computer Professionals
by C. J. Date

Printed in the United States of America.

Published by O'Reilly Media, Inc.
1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Printing History:
May 2013: First Edition.

Revision History:
2013-05-07 First release
See *http://oreilly.com/catalog/errata.csp?isbn=0636920029649* for release details.

ISBN: 978-1-449-36943-9
[LSI]

Mathematical science shows what is.
It is the language of unseen relations between things.

—Ada, Countess of Lovelace,
quoted in Dorothy Stein (ed.):
Ada: A Life and a Legacy (1985)

That is the essence of science:
Ask an impertinent question, and you are on the way to a pertinent answer.

—Jacob Bronowski:
The Ascent of Man (1973)

Hofstadter's Law: *It always takes longer than you expect,*
even when you take into account Hofstadter's Law.

—Douglas R. Hofstadter:
Gödel, Escher, Bach: An Eternal Golden Braid (1979)

——— ♦♦♦♦♦ ———

To my wife Lindy
and my daughters Sarah and Jennie
with all my love

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold nearly 900,000 copies at the time of writing and is used by several hundred colleges and universities worldwide. He is also the author of numerous other books on database management, including most recently:

- From Ventus: *Go Faster! The TransRelational™ Approach to DBMS Implementation* (2002, 2011)
- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, coauthored with Hugh Darwen, 2007)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007) and *Database Explorations: Essays on The Third Manifesto and Related Topics* (coauthored with Hugh Darwen, 2010)
- From Apress: *Date on Database: Writings 2000–2006* (2007) and *The Relational Database Dictionary, Extended Edition* (2008)
- From O'Reilly: *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition, 2012); *Database Design and Relational Theory: Normal Forms and All That Jazz* (2012); and *View Updating and Relational Theory: Solving the View Update Problem* (2013)

Mr. Date was inducted into the Computing Industry Hall of Fame in 2004. He enjoys a reputation that is second to none for his ability to explain complex technical subjects in a clear and understandable fashion.

Contents

Chapter 8

Transactions

For tyme ylost may nought recovered be
—Geoffrey Chaucer: *Troilus and Criseyde* (c. 1385)

The transaction concept is only tangentially relevant to most users. Well, this isn't *quite* true—if you're a programmer writing database applications, then you certainly need to know what the implications of transactions are for the way you write your code. But you don't need to know much more than that. And if you're an interactive user, you probably don't even need to know that much. On the other hand (as I've said several times previously, more or less), if you want to be able to lay honest claim to having at least a broad understanding of what database technology is all about, then you do need to have, among other things, a basic appreciation of what transaction management entails.

As I've also said previously, transaction management isn't really a relational topic as such. However, it's interesting to note that the original research—the research, that is, that put the concept on a sound scientific footing—was all done in a specifically relational context.[1]

WHAT'S A TRANSACTION?

A transaction is a program execution, or part of a program execution, that constitutes *a logical unit of work*. It begins when a BEGIN TRANSACTION statement is executed and ends when either a COMMIT statement or a ROLLBACK statement is executed, where:

- BEGIN TRANSACTION is normally (and desirably) explicit.
- COMMIT signifies successful termination, and is also normally explicit.
- ROLLBACK signifies unsuccessful termination, and is often implicit.

---

[1] The seminal paper was "The Notions of Consistency and Predicate Locks in a Data Base System," by K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger (*CACM 19*, No. 11, November 1976). The standard work on the subject is *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter (Morgan Kaufmann, 1993).

Thus, any given program execution involves a *sequence* of transactions running one after another[2] (where the number of transactions in the sequence might be just one, of course). See Fig. 8.1.

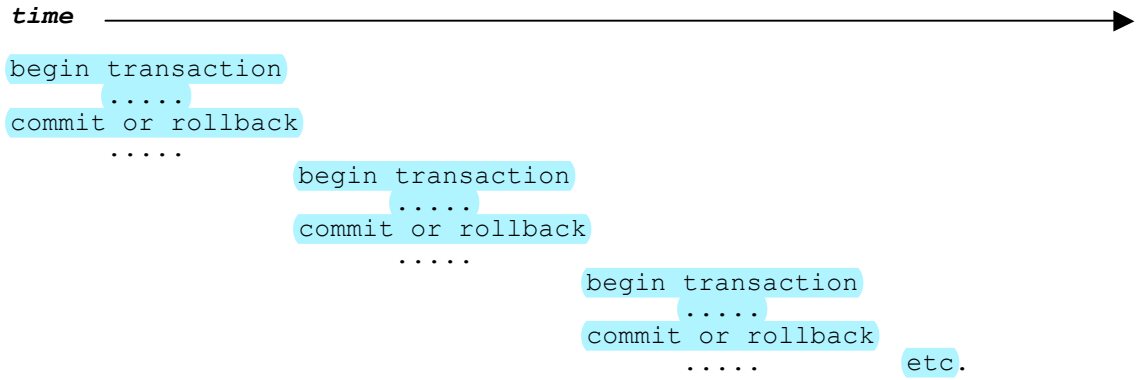

Fig. 8.1: Program execution is a sequence of transactions

RECOVERY

It's important to understand that all database updates must be done within the context of some transaction. The reason is that transactions aren't just a unit of work, they're also a unit of *recovery*. Let me explain. The standard example involves a banking application of some kind and a transaction that transfers a money amount, say $100, from one account to another. Here's the code—well, pseudocode—for such a transaction:

```
BEGIN TRANSACTION ;
   UPDATE account 123 : { BALANCE := BALANCE - 100 } ;
   IF error THEN ROLLBACK ;
   UPDATE account 321 : { BALANCE := BALANCE + 100 } ;
   IF error THEN ROLLBACK ;
COMMIT ;
```

As you can see, what's presumably meant to be a single atomic operation—"transfer $100 from account number 123 to account number 321"—in fact involves two separate updates to the database.[3] What's more, the database is actually in an incorrect state between those two updates, in the sense that it fails to reflect the true situation in the real world; clearly, a real world transfer

---

[2] Another way of saying this is: BEGIN TRANSACTION isn't allowed if a transaction is currently in progress.

[3] Actually it could be just one update in **Tutorial D**, thanks to the availability of the multiple assignment operator (see Chapter 6). For the purposes of the present discussion, it's necessary to assume that such an operator isn't available (or isn't being used, at any rate). Indeed, it's worth stating explicitly that multiple assignment wasn't even considered as a possibility when the original theoretical work was done on transactions.

of dollars from one account to another shouldn't affect the total number of dollars in the accounts concerned, but in the example the sum of $100 temporarily goes missing, as it were, between the two updates. Thus, the logical unit of work that's a transaction doesn't necessarily involve just a single database update. Rather, it involves a sequence of several such operations, in general, and the intent of that sequence is to transform a correct state of the database into another correct state, without necessarily preserving correctness at all intermediate points.

Note: At the risk of confusing you, let me now point out that, while the state of the database between the two updates in the example is certainly incorrect, it isn't *inconsistent*—it doesn't violate any known integrity constraints.[4] As I pointed out in Chapter 6, correctness implies consistency, but the converse isn't so: Incorrectness doesn't imply inconsistency. Thus, it would be more accurate to say the intent of a transaction is to transform a *consistent* state of the database into another *consistent* state, without necessarily preserving *consistency* at all intermediate points. I'll have more to say on this particular issue in the subsection "The ACID Properties" below.

Anyway, it's clear that what mustn't be allowed to happen in the example is for the first update to be done and the second not, because that would leave the database overall in an incorrect state. Ideally, we would like an ironclad guarantee that both updates will succeed. Unfortunately, it's impossible to provide any such guarantee—there's always a chance things will go wrong, and go wrong moreover at the worst possible moment. For example, a system crash might occur between the two updates, or an arithmetic overflow might occur on the second of them. But proper transaction management does provide the next best thing to such a guarantee; specifically, it guarantees that if the transaction executes some updates and then a failure occurs before the transaction reaches its planned termination, then those updates will be "rolled back" (i.e., undone). Thus the transaction *either* executes in its entirety *or* is totally canceled (meaning it's made to look as if it never executed at all). In this way, a sequence of operations that's fundamentally not atomic can be made to look as if it were atomic from an external point of view. And the COMMIT and ROLLBACK operations are the key to achieving this effect:

- *Successful termination:* COMMIT "commits" the transaction's updates (i.e., causes them to be installed in the database) and terminates the transaction.

- *Unsuccessful termination:* ROLLBACK "rolls the database back" to the state it was in when the transaction began (thereby undoing the transaction's updates) and terminates the transaction.

- *Implicit ROLLBACK:* The code in the dollar transfer example includes explicit tests for errors and forces an explicit ROLLBACK if it detects one. But we obviously can't assume,

---

[4] What's more, there's no reasonable integrity constraint we could write that would be violated by the dollar transfer in the example (right?).

> nor would we want to assume, that transactions always include explicit tests for all possible errors that might occur. Therefore, the system will force an *implicit* ROLLBACK for any transaction that fails to reach its planned termination for any reason, where "planned termination" means either an explicit COMMIT or an explicit ROLLBACK. Moreover, if there's a system crash (in which case the transaction will fail through no fault of its own), then, when the system restarts, it will typically try to run the transaction again.

In the example, therefore, if the transaction gets through the two updates successfully, it executes a COMMIT to force those updates to be installed in the database. If it detects an error on either one of those updates, however, it executes a ROLLBACK instead, to undo all of the updates (if any) it has made so far. And if a totally unlooked-for error occurs, such as a system crash, then the system will force a ROLLBACK anyway on the transaction's behalf.

The Recovery Log

You might be wondering how it's possible to undo an update. Basically, the answer is that the system maintains a *log* in persistent storage (usually on disk), in which details of all updates—in particular, values of updated objects before and after each update, sometimes called *before-* and *after-images*—are recorded.[5] Thus, if it becomes necessary to undo some particular update, the system can use the corresponding log record to recover the original value of the updated object (that is, to restore the updated object to its previous value). Of course, in order for this process to work, the log record for a given update must be physically written to the log before that update is physically written to the database.[6] This protocol is known as *the write ahead log rule*.

> *Aside:* Of course, there's rather more to the write ahead log rule than that, as I'm sure you'd expect. To be more specific, the rule requires, among other things, (a) all other log records for a given transaction to be physically written to the log before the COMMIT record for that transaction is physically written to the log, and (b) COMMIT processing for a given transaction not to complete until the COMMIT record for that transaction has been physically written to the log. *End of aside.*

It follows from all of the above that (as noted at the beginning of this section) transactions aren't just a unit of work, they're also a unit of recovery. Which brings us to our next topic, viz., the so called *ACID properties* of transactions.

---

[5] You can think of "objects" here as tuples if you like, though in practice they're likely to be something more physical, such as disk pages.

[6] By "physically written" here, I mean the record in question isn't just waiting in main memory somewhere to be written out at some later time—rather, it has actually been forced out to persistent storage.

The ACID Properties

ACID is an acronym, standing for *atomicity – consistency – isolation – durability*, which are four properties that transactions in general are supposed to possess. Briefly:

- *Atomicity:* Transactions are all or nothing.
- *Consistency:* Transactions transform a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points.
- *Isolation:* Any given transaction's updates are concealed from all other transactions until the given transaction commits.
- *Durability:* Once a transaction commits, its updates survive in the database, even if there's a subsequent system crash.

As we've seen, the atomicity and durability properties mean the transaction is a unit of work and a unit of recovery, respectively; what's more, the consistency property means it's also a unit of *integrity*—but see the paragraph immediately following—and the isolation property means it's a unit of *concurrency* as well. Now, we've seen how the atomicity and durability properties are achieved; however, the other two require a little more explanation.

Actually I don't want to say too much more about the consistency property, other than to point out that it's more than a little suspect, because it assumes that integrity constraints aren't checked until commit time ("deferred checking"). And while it's true that some integrity checking, both in the SQL standard and in certain commercial DBMSs, is indeed deferred in this sense, the fact remains that to defer checking in this way is logically incorrect (as was mentioned briefly in Chapter 6, and as *SQL and Relational Theory* explains in detail).[7] That's why, again as we saw in Chapter 6, the relational model requires all constraint checking to be immediate—implying that, at least so far as the relational model is concerned, the "unit of integrity" is not the transaction but the *statement*.

So that leaves the isolation property. Here there's quite a bit more to say, and it deserves a section or two of its own.

CONCURRENCY

Most of the time in this chapter so far, I've been tacitly assuming there's just one transaction running in the system at any given time. But now suppose there are two or more, running concurrently. Then, as indicated in Chapter 1, controls are needed in order to ensure that those

---

[7] It's ironic, therefore, that in the literature "the C property" is often said to stand not for consistency but for *correctness*.

transactions—which we assume, reasonably enough, to be totally independent of one another—don't interfere with each other in any way. For example, consider the scenario illustrated in Fig. 8.2. That figure is meant to be read as follows: First, transaction *TX1* updates some object *p*; subsequently, transaction *TX2* retrieves that same (but now updated) object *p*; finally, transaction *TX1* is rolled back. At that point, transaction *TX2* has seen, and therefore become dependent on, an update that in effect never occurred.

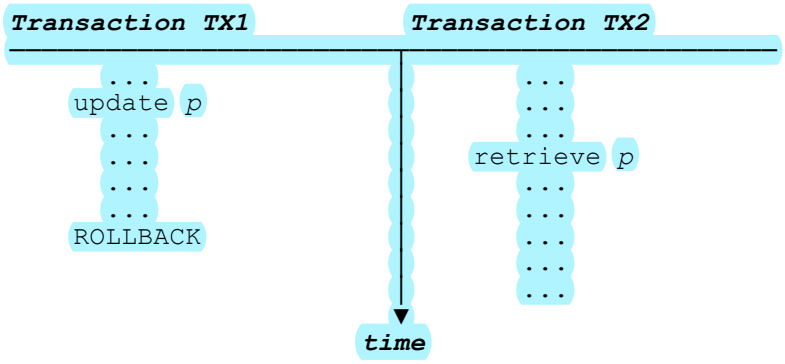

Fig. 8.2: Transaction *TX2* is dependent on an uncommitted update

The scenario illustrated in Fig. 8.3 is arguably even worse: Here, transactions *TX1* and *TX2* both retrieve the same object *p*; subsequently, transaction *TX1* updates that object *p*; and then transaction *TX2* also updates that same object *p*, thereby overwriting *TX1*'s update (at which point *TX1*'s update is said to be "lost").[8]

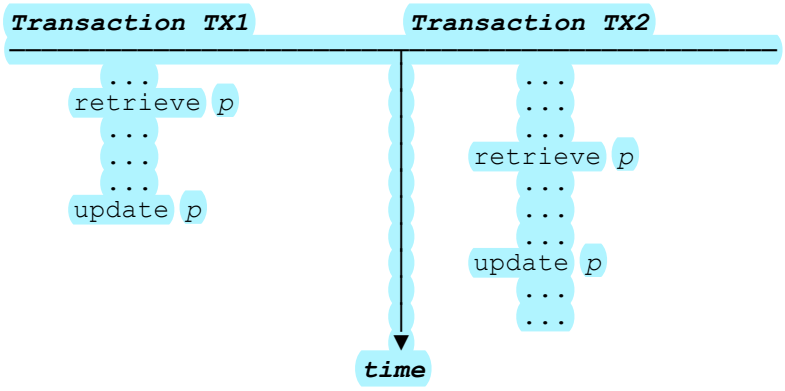

Fig. 8.3: Transaction *TX1*'s update is lost

Now, what has happened in both Fig. 8.2 and Fig. 8.3 is that one transaction has effectively interfered with another, concurrent, transaction, to produce an overall incorrect result. By the

[8] Such a situation isn't exactly unknown in real life, incidentally. Have you ever found someone else already sitting in your airline seat?

way, the problems illustrated in those figures—dependence on an uncommitted update in Fig. 8.2 and loss of an update in Fig. 8.3—aren't the only problems that can occur, absent suitable controls, if transactions are allowed to run concurrently; however, they're probably the easiest ones to understand. But the point is: We do need those controls. And the kind of control most commonly found in practice—though not the only kind possible—is called *locking*.

LOCKING

The basic idea behind locking is simple: When transaction *TX1* needs an assurance that some object it's interested in won't be updated while its back is turned, as it were, it *acquires a lock* on that object (in a manner to be explained). The effect of acquiring that lock is to "lock other transactions out of" the object in question (again, in a manner to be explained), and thereby in particular to prevent them from updating it. *TX1* is thus able to continue its processing in the certain knowledge that the object in question won't be updated by any other transaction, at least until such time as *TX1 releases* the lock in question (which of course it won't do until it has finished with it).

Typically, two kinds of locks are supported, "shared" or S locks (also called read locks) and "exclusive" or X locks (also called write locks). Loosely speaking, S locks are compatible with one another, but X locks aren't compatible with anything. To elaborate: Let *TX1* and *TX2* be distinct but concurrent transactions. Then:

- So long as *TX1* holds an X lock on object *p*, no request from *TX2* for a lock of either type on *p* can be granted.
- So long as *TX1* holds an S lock on object *p*, no request from *TX2* for an X lock on *p* can be granted, but a request from *TX2* for an S lock on *p* *can* be granted.

Thus, any number of transactions can hold an S lock on *p* at the same time, but at most one transaction can hold an X lock on *p* at any given time—and in this latter case, no other transaction can hold any lock on *p* at all at the time in question.

The system now uses the mechanism just described in order to enforce a protocol that will guarantee that problems such as those illustrated in Figs. 8.2 and 8.3 can't occur. Here in outline is how that protocol works:[9]

- A transaction's request to retrieve some object *p* causes an implicit request for an S lock on that object *p*.

---

[9] The formal name for the protocol is *strict two-phase locking*. In practice, real systems typically adopt various improvements to the protocol as described here. Details of what's possible in this connection are beyond the scope of this book, however.

- A transaction's request to update some object *p* causes an implicit request for an X lock on that object *p*. *Note:* If the transaction in question already holds an S lock on *p* (as indeed it very likely will in practice), then that implicit lock request is treated as a request to *upgrade* that S lock to an X lock.

- In both cases, if the implicit lock request can't be granted (because some other transaction currently holds a conflicting lock on the object in question), then the requesting transaction enters a wait state, waiting for that conflicting lock to be released.

- Finally, COMMIT and ROLLBACK both cause all locks held by the terminating transaction to be released.

Now let's see how the foregoing protocol solves the problems illustrated in Figs. 8.2 and 8.3. Fig. 8.4 is a modified version of Fig. 8.2, showing what happens to the interleaved execution of transactions *TX1* and *TX2* from that figure under the protocol. First, *TX1* updates the object *p*, acquiring as it does so an X lock on *p*. Subsequently, *TX2* attempts to retrieve that same (but now updated) object *p*; however, *TX2*'s implicit request for an S lock on *p* can't be granted at that point, and so *TX2* goes into a wait state. Then *TX1* is rolled back and its X lock on *p* is released. Now *TX2* can come out of the wait state and acquire its requested S lock on *p*; but *p* is now as it was before *TX1* ran, and so *TX2* is no longer dependent on an uncommitted update.

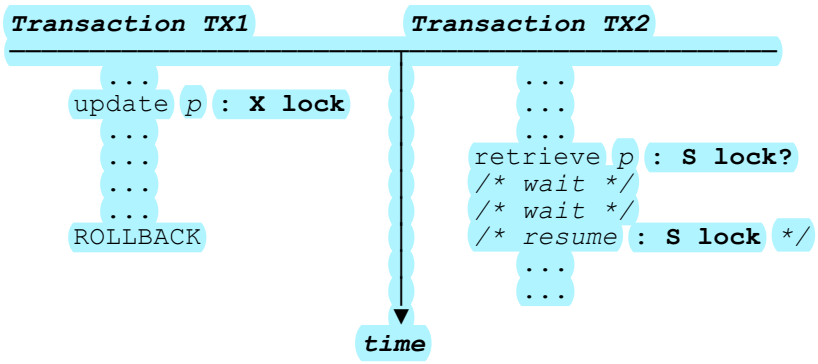

Fig. 8.4: Transaction *TX2* is no longer dependent on an uncommitted update

To refer back to the so called ACID properties for a moment, observe how the locking protocol has served to "isolate" transactions *TX1* and *TX2* from each other in Fig. 8.4.

I turn now to Fig. 8.5, which is a similarly modified version of Fig. 8.3. First, *TX1* retrieves the object *p*, acquiring as it does so an S lock on *p*. Subsequently, *TX2* retrieves that same object *p* and also acquires an S lock on it. *TX1* then attempts to update *p*; that attempt causes a request for an X lock, which can't be granted, and so *TX1* goes into a wait state. Then *TX2* also attempts to update *p*, and for analogous reasons also goes into a wait state. Neither

transaction can now proceed, so there's certainly no question of any update being lost. On the other hand, no useful work is being done, either!—in other words, we've solved the lost update problem by reducing it to another problem (but at least we've solved the original problem). The new problem is called *deadlock*.

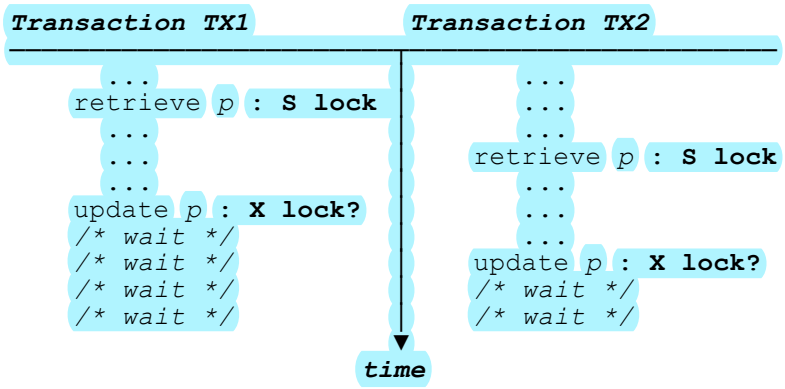

Fig. 8.5: No update is lost, but deadlock occurs

The typical solution to the deadlock problem involves (a) choosing one of the deadlocked transactions—typically the youngest one, meaning the one that began most recently—as the "victim" and rolling it back, thereby releasing its locks, and then (b) starting the victim again as a new transaction.

A REMARK ON SQL

Since I don't plan to say any more about transactions in subsequent chapters (in Part III of this book in particular), let me close this chapter by noting that everything I've been saying about transactions in general applies to SQL systems in particular (more or less!). Thus, SQL does support explicit BEGIN TRANSACTION, COMMIT, and ROLLBACK statements (so does **Tutorial D**, incidentally).[10] It also has no explicit reliance on locking as such—that is, it has no explicit syntax for requesting or releasing locks—which means the system is free to use some mechanism other than locking, if it chooses, as a basis for concurrency control. (Again, the same is true of **Tutorial D**.) But what SQL does have is all kinds of additional features, concepts, and statements having to do with transactions in general that (a) **Tutorial D** most certainly doesn't have and (b) are far beyond the scope of an introductory book of this nature. One "simple" example of such a feature is the ability to have a transaction begin implicitly, without requiring execution of an explicit BEGIN TRANSACTION statement—but there's much, much more to the matter than that. *Caveat lector*.

---

[10] Actually SQL has START TRANSACTION in place of BEGIN TRANSACTION.

EXERCISES

8.1 (From *An Introduction to Database Systems*, 8th edition, Addison-Wesley, 2004.) The term *schedule* is used to refer to the overall sequence of database retrieval and update operations that occurs when some set of transactions run interleaved (i.e., concurrently). The schedule below represents the sequence of operations in a schedule involving transactions *T1*, *T2*, ..., *T12* (*a*, *b*, ..., *h* are objects in the database):

```
time t00    ..........
time t01  (T1)   : RETRIEVE a ;
time t02  (T2)   : RETRIEVE b ;
  ...     (T1)   : RETRIEVE c ;
  ...     (T4)   : RETRIEVE d ;
  ...     (T5)   : RETRIEVE a ;
  ...     (T2)   : RETRIEVE e ;
  ...     (T2)   : UPDATE e ;
  ...     (T3)   : RETRIEVE f ;
  ...     (T2)   : RETRIEVE f ;
  ...     (T5)   : UPDATE a ;
  ...     (T1)   : COMMIT ;
  ...     (T6)   : RETRIEVE a ;
  ...     (T5)   : ROLLBACK ;
  ...     (T6)   : RETRIEVE c ;
  ...     (T6)   : UPDATE c ;
  ...     (T7)   : RETRIEVE g ;
  ...     (T8)   : RETRIEVE h ;
  ...     (T9)   : RETRIEVE g ;
  ...     (T9)   : UPDATE g ;
  ...     (T8)   : RETRIEVE e ;
  ...     (T7)   : COMMIT ;
  ...     (T9)   : RETRIEVE h ;
  ...     (T3)   : RETRIEVE g ;
  ...     (T10)  : RETRIEVE a ;
  ...     (T9)   : UPDATE h ;
  ...     (T6)   : COMMIT ;
  ...     (T11)  : RETRIEVE c ;
  ...     (T12)  : RETRIEVE d ;
  ...     (T12)  : RETRIEVE c ;
  ...     (T2)   : UPDATE f ;
  ...     (T11)  : UPDATE c ;
  ...     (T12)  : RETRIEVE a ;
  ...     (T10)  : UPDATE a ;
  ...     (T12)  : UPDATE d ;
  ...     (T4)   : RETRIEVE g ;
time t36    ..........
```

Assume that RETRIEVE *p* (if successful) acquires an S lock on *p*, and UPDATE *p* (if successful) promotes that lock to X level. Assume also that all locks are held until end of transaction. At time *t36*, which transactions are waiting for which other transactions? Are there any deadlocks at that time?

ANSWERS

8.1 At time *t36* no transactions are doing any useful work at all! Transactions *T1*, *T5*, *T6*, and *T7* have all terminated (*T1*, *T6*, and *T7* successfully, *T5* unsuccessfully). There's one deadlock, involving transactions *T2*, *T3*, *T9*, and *T8*; in addition, *T4* is waiting for *T9*, *T12* is waiting for *T4*, and *T10* and *T11* are both waiting for *T12*. We can represent the situation by means of a graph (the *wait-for graph*), in which (a) the nodes represent transactions; (b) a directed edge from node *Ti* to node *Tj* indicates that *Ti* is waiting for *Tj*; and (c) the edge from note *Ti* to node *Tj* is labeled with the name of the database object and the kind of lock that *Ti* is waiting for (see Fig. 8.6 below). Observe that a cycle in the graph corresponds to a deadlock.

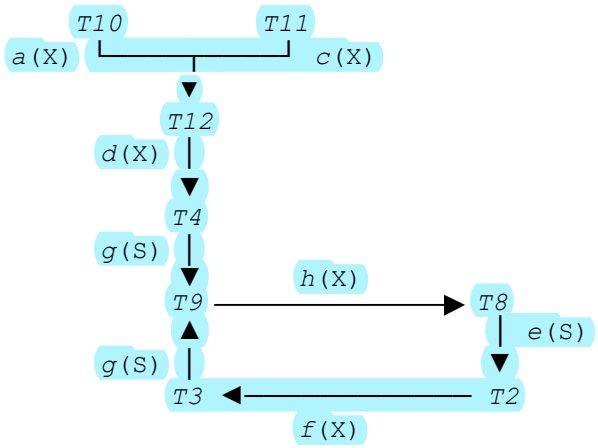

Fig. 8.6: Wait-for graph for Exercise 8.1