

# Chapter 1

## Transactions on the Logical Database

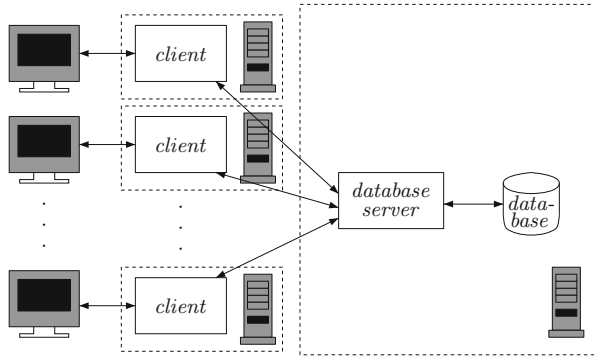
The database as seen by an application programmer is called the *logical database*. In most cases the logical database is a relational database, so that application programs operate on tuples in relations through an SQL interface. A *transaction* is a sequence of read and update actions on the logical database, performed on the database upon a sequence of requests from an application. The action sequence constituting a transaction is atomic in the sense that either the effects of all the actions are recorded permanently in the database (in which case the transaction is committed) or none are (in which case the transaction is aborted and rolled back).

Integrity constraints specified by the database designer on the logical database determine which database states (i.e., database contents) are legal. Each transaction should be programmed in such a way that it retains the integrity of an initially integral logical database. Upon requests from applications, the actions included in a transaction are performed one by one on the database by the *database server* whose responsibility is to ensure that the logical database as well as the underlying physical database retain integrity under many concurrently executing transactions.

In this chapter we present the basic concepts related to transactions on a relational database. We model a transaction as an action sequence consisting of a forward-rolling phase possibly followed by a backward-rolling phase of undo actions, where the forward-rolling phase can also contain partial rollbacks. For ease of treatment, we define a primitive transaction model called the *key-range transaction model*, which we shall use throughout this book when discussing different issues of transaction processing.

### 1.1 Transaction Server

The database server of a typical relational database management system functions as a *transaction server*, also called a *query server* (or a *function server*). Such a server offers an interface through which client application processes can send requests to



**Fig. 1.1** Typical two-tier transaction server

execute database actions (queries and updates) on the database stored at the server. The server processes the requests coming from the clients and returns the results to the clients. The requests are specified by SQL queries and update statements embedded in an application program or by a special database programming interface consisting of a collection of procedures and functions for communicating between an application program and the database server. Java Database Connectivity (JDBC) and Open Database Connectivity (ODBC) are examples of such interfaces.

The mode of operation of a transaction server is called *transaction shipping* or *query shipping* (or *function shipping*): a transaction (or a query or a function) is “shipped” from a client to the server, to be executed on the data stored at the server. Figure 1.1 shows a typical two-tier organization of many client machines connected to a database machine. The two “tiers” are the client tier and the database tier. For scalability, a large system with thousands of clients may be organized with additional tiers between the client and database tiers, such that a machine at a lower tier services only requests from a subset of machines at the next higher tier.

There are one or more *server processes* running at the server. Such a process is usually *multi-threaded*, that is, has several subprocesses or *threads* running concurrently in the same virtual address space. A typical server process services a number of clients, so that for each client application process there is a dedicated server-process thread.

A connection between a client application process and a server-process thread is created by the embedded SQL statement **connect to** *s*, where *s* is an identifier (network address) of the server. The statement creates a new server-process thread to service the application process or assigns an existing idle thread for the purpose.

## 1.2 Logical Database

A database application programmer sees the database as a *logical database*, a collection of data items defined and manipulated by the data definition and manipulation language offered by the database management system. In most cases

the logical database conforms to the relational data model extended with features from the object data model, with SQL as the data definition and manipulation language.

The logical databases that we consider in this book are assumed to be purely relational, if not explicitly stated otherwise. Thus, the logical database is a collection of *relations* or *tables*, which are multisets of *tuples* or *rows*. A *multiset* or *bag* is an unordered set where the same element can occur multiple times. All tuples of a relation conform to the same schema, that is, are of the same type. A *relation schema*  $r(Z)$  consists of the name  $r$  of the relation and the names and types  $Z$  of its attributes, along with possible integrity constraints defined on the relation (uniqueness of keys, referential integrity, etc.).

The operations of the relational model take relations as arguments and return a relation as a result. For example, the SQL statement

**delete from  $r$  where  $A = a$**

means the same as the assignment

$$r \leftarrow r \setminus \sigma_{A=a}(r),$$

where “ $\setminus$ ” denotes the relational difference operation and “ $\sigma$ ” denotes the relational selection operation, that is,  $\sigma_{A=a}(r) = \{t \in r \mid t[A] = a\}$ .

However, for the purpose of transaction management, this view of operations is too coarse grained. For one thing, as we will see shortly, we have to see a database transaction as a sequence of actions where each update action can be rolled back, that is, undone, if so desired. This requirement imposes a strong restriction on the complexity of actions: any single update action must be simple enough so that its undo action (inverse action) is easily defined and implemented. For another thing, modern database management requires that as much concurrency as possible be allowed among transactions active at the same time. This means that the granularity of data items used to synchronize transaction execution should be as small as possible. For example, we must allow several transactions to update a relation simultaneously, as long as the updates affect different tuples.

The usual solution (and the solution we adopt here) is to view the actions on the logical database as *tuple-wise actions*, so that each action always reads or updates (inserts or deletes) a single tuple only. Accordingly, in our view, the above **delete** statement is regarded as a sequence of tuple-wise actions:

delete  $t_1$  from  $r$ ; delete  $t_2$  from  $r$ ; ...; delete  $t_n$  from  $r$ ,

where  $\{t_1, t_2, \dots, t_n\}$  is the bag of tuples with  $A = a$ . For the deletion of a single tuple  $t$ , it is easy to define its undo action: it is the insertion of the same tuple.

Thus, for the purpose of transaction management, we assume that the collection of logical database actions consists of tuple-wise actions such as the following:

1. Reading a tuple  $t$  of relation  $r$ . The SQL query **select \* from  $r$  where  $C$**  generates a sequence of such read actions, one for each tuple satisfying the selection condition  $C$ .

2. Inserting a new tuple  $t$  into relation  $r$ . Such an insert action is generated by the SQL statement **insert into  $r$  values ( $t$ )**.
3. Deleting a tuple  $t$  from relation  $r$ . The SQL statement **delete from  $r$  where  $C$**  generates a sequence of such delete actions, one for each tuple satisfying the selection condition  $C$ .
4. Updating the value of an attribute  $A$  of a tuple  $t$  in relation  $r$ . The SQL statement **update  $r$  set  $A = e$  where  $C$**  generates a sequence of such update actions (if  $e$  is a constant) or a sequence of pairs of a read action followed by an update action (if  $e$  contains attributes whose values thus are read in computing the value of  $e$ ), one for each tuple satisfying condition  $C$ .
5. Creating a new relation  $r(Z)$  in the database. The input to the action is a relation schema, and the action is possible when there is no relation named  $r$  already in the database. The action creates a new, empty relation on which other actions can subsequently be performed. This action is generated by the SQL statement **create table  $r(Z)$** , where  $Z$  is the SQL definition of the attributes and integrity constraints of the schema.
6. Deleting an empty relation  $r$  from the database. After this action, the only action that can be applied to  $r$  is action (5). The SQL statement **drop table  $r$**  generates a sequence of tuple-deletion actions (3), one for each tuple in  $r$ , followed by the deletion of  $r$ .

Logical database actions also include certain *transaction-control actions* needed for managing transactions: beginning a new transaction, committing a transaction, aborting a transaction (i.e., starting rollback), and completing a rollback. The exact set of logical actions used in this book is fixed later, when the transaction model (the key-range model) is defined.

### 1.3 Integrity of the Logical Database

As is customary with treatments of relational database management, we may use the terms “database” and “relation” with two meanings: on the one hand, a logical database (resp. a relation) may mean a *database value* (resp. *relation value*), that is, some fixed contents of the database (resp. relation) as a bag of tuples, and on the other hand, it may mean a *database variable* (resp. *relation variable*), that is, a variable that can take a database (resp. relation) as its value. The context should make it clear which of the two meanings is assumed. Sometimes, when we wish to emphasize the distinction between the two meanings, we may talk about *database states* when referring to database values.

For each logical database and for each of its relations, considered with the latter meaning (as a variable), there is a set of associated *integrity constraints*, which have been specified at the time of creating the database or relation (with the SQL **create table** statement) or added afterwards (with the SQL **alter table** statement). An integrity constraint may be internal to a single relation such as a *key constraint*

(**primary key**, **unique**) or it may span over two relations such as a *referential integrity constraint* (**foreign key**).

Integrity constraints restrict the values that the database and relation variables are allowed to take. A logical database (i.e., its state) is *integral* or *consistent* if it fulfills the integrity constraints specified for the database.

An attempt to perform an update action that violates an integrity constraint specified by SQL on the database either returns with an error indication, or triggers a sequence of corrective actions so as to make the violated constraint to hold again if such corrective actions have been specified. In addition to constraints that can be specified by SQL **create table** and **alter table** statements, the logical database usually satisfies some application-specific constraints. Applications using the database must check for these and keep them satisfied.

As a running example of a relation, we use

$$r(\underline{X}, V),$$

where  $X$  and  $V$  are sequences of attributes. The underlining of  $X$  means that we have specified a key constraint on the relation: a relation that satisfies this constraint cannot contain two distinct tuples  $t$  and  $u$  with  $t[X] = u[X]$ . The attribute sequence  $X$  is called the *primary key*, or *key* for short, of the relation.

## 1.4 Transactions

When a database application process is running, it generates a sequence of requests to perform SQL queries and update operations on the database stored at the server. The server-process thread allocated to service the application process parses, optimizes, and executes the queries and update statements in the order in which they arrive from the application. From the view of the logical database, the execution of a query or an update statement is a sequence of tuple-wise actions on the relations in the database.

The sequence of SQL requests coming from the application process is divided into subsequences by issuing occasionally an SQL **commit** or **rollback** request. The sequence of tuple-wise actions on the logical database resulting from one such subsequence of requests, that is, one that extends from the action immediately following a **commit/rollback** request (or from the first action of the application) to the next **commit/rollback** request, is called a transaction.

A transaction is an action sequence that the database application programmer wants to see as forming an *atomic* (i.e., indivisible) unit of work: either (1) all changes to the logical database produced by the transaction are done or (2) none of the changes appear in the logical database. Requirement (1) must hold in the case of a *committed transaction*, obtained by terminating the transaction with a **commit** request, while requirement (2) must hold in the case of an *aborted transaction*, obtained by terminating the transaction with a **rollback** request. Requirement (2)

is also the only option in the case in which the database management system is unable to commit a transaction due to a process failure or a system crash occurring before or during servicing a **commit** request.

*Example 1.1* Assume that in the relation  $r(\underline{X}, V)$ ,  $V$  is a single numeric-valued attribute. The following fragment of an application program written in embedded SQL generates a transaction that doubles the  $V$  value of all tuples in  $r$ :

```
exec sql update r set  $V = 2 * V$ ;  
exec sql select sum( $V$ ) into :new_sum from  $r$ ;  
exec sql commit.
```

The application process sends to the server three requests, one by one, waiting for one request to be serviced before sending the next:

1. A request to execute the **update** statement
2. A request to execute the **select** query
3. A request to commit the transaction

Assuming that the contents of relation  $r$  are initially the set of tuples  $\{(x_1, v_1), \dots, (x_n, v_n)\}$ , request 1 results in performing the following action sequence at the server, where  $B$  denotes the action of beginning a new transaction,  $R[x, v]$  the action of reading tuple  $(x, v)$ , and  $W[x, u, v]$  the action of changing tuple  $(x, u)$  to  $(x, v)$ :

$$BR[x_1, v_1]W[x_1, v_1, 2v_1]R[x_2, v_2]W[x_2, v_2, 2v_2] \dots R[x_n, v_n]W[x_n, v_n, 2v_n].$$

Because the semantics of the **update** statement does not specify the order in which  $r$ 's tuples are processed, the system selects the most efficient order, probably the order in which the tuples are physically stored in  $r$ 's file. After performing the update, the server returns to the application an indication of successful completion. Then request 2 is sent, resulting in the following action sequence:

$$R[x_1, 2v_1]R[x_2, 2v_2] \dots R[x_n, 2v_n].$$

The computed sum is returned to the application, which assigns it to the program variable `new_sum`. Finally, request 3 is sent, resulting in the action

$C$ ,

denoting the commit of the transaction, assuming that no failures occur. □

In addition to atomicity, *durability* is required for all committed transactions. This means that the changes produced by a committed transaction need to actually happen and stay in effect in the logical database, even in the presence of process failures and system failures occurring after the transaction has successfully committed. The only way to undo updates produced by a committed transaction is to program a new transaction (or several new transactions) for effectively compensating for the updates.

In the above example, if the “commit the transaction” action is finished successfully, so that the transaction actually commits, all of the updates in the  $V$  attribute of

$r$ 's tuples need to stay in effect. If, however, the transaction does not commit (e.g., because of a system failure), then all of the  $V$  attributes have to retain their original values, so that none of them must be multiplied by two. To accomplish this, the database management system must be able to undo any updates that have already been done before the abort of the transaction.

## 1.5 Transaction States

Formally, we define a *transaction* as a pair  $(T, \alpha)$ , where  $T$  is the identifier and  $\alpha$  is the state of the transaction. The *transaction identifier* is a serial number or timestamp that uniquely identifies a transaction over a long period of time when the database system has been in use. More specifically, as will be evident later, transaction identifiers have to be unique over the set of transactions that either are currently active (i.e., have not yet committed or rolled back) or have committed or rolled back but still have traces (i.e., log records) retained in the available log files.

The *transaction state*,  $\alpha$ , is the sequence of actions performed for the transaction thus far. Each action in  $\alpha$  includes the values for the input arguments with which the action was performed on the logical database and the values of the output arguments returned. For example, the transaction state

$$BR[x_1, v_1]W[x_1, v_1, 2v_1]$$

represents the result of performing the first three actions of the transaction given in Example 1.1. The arguments  $x_1$  and  $v_1$  are constants;  $R[x_1, v_1]$  represents a read action that, when given input  $x_1$ , retrieved from the database the tuple  $(x_1, v_1)$ ; and  $W[x_1, v_1, 2v_1]$  represents an update action that, when given input  $x_1$ , returned the value  $v_1$  of the tuple with key  $x_1$  and replaced the value in the tuple by the value  $2v_1$ .

We use the symbol  $T$  (or subscripted,  $T_i$ ) to denote both the transaction identifier and the transaction  $(T, \alpha)$  as a whole; accordingly, we may say that transaction  $T$  is at state  $\alpha$ . Also, when the state of the transaction is the interesting part, we may even talk about “transaction  $\alpha$ .”

We distinguish between four different types of transaction states: (1) forward-rolling, (2) committed, (3) backward-rolling, and (4) rolled back. Every transaction starts as a *forward-rolling transaction*; in general, the state of such a transaction is an action sequence of the form

$$B\alpha, \tag{1.1}$$

where  $B$  is the *begin-transaction* action and  $\alpha$  is a sequence of read actions and normal (forward-rolling) update actions. The action sequence  $\alpha$  forms the *forward-rolling phase* of the transaction.

The begin-transaction action can be thought of as an action that is needed to introduce a new transaction into the system, including the generation of a new transaction identifier. We assume that with database interfaces such as (embedded)

SQL that have no explicit request for starting a transaction, the first read or update action triggers the begin-transaction action as the first action of the transaction.

A forward-rolling transaction can be continued with read and update actions until a commit or an abort action is performed. The state of a *committed transaction* is an action sequence of the form

$$B\alpha C, \quad (1.2)$$

where  $B$  and  $\alpha$  are as in a forward-rolling transaction and  $C$  is the *commit-transaction* action, the result of a successfully processed **commit** request. A committed transaction cannot be continued with any more actions.

The state of a *backward-rolling transaction* is an action sequence of the form

$$B\alpha\beta A\beta^{-1}, \quad (1.3)$$

where  $B\alpha\beta$  is the state of a forward-rolling transaction,  $A$  is the *abort-transaction* action, and  $\beta^{-1}$  is a sequence of undo actions for the suffix  $\beta$  of the forward-rolling action sequence  $\alpha\beta$ . The action sequence  $\beta^{-1}$  forms the *backward-rolling phase* of the transaction. Such a transaction has rolled back the forward-rolling update actions in  $\beta$ ; the update actions in  $\alpha$  are still to be undone.

The abort action can be seen as marking the start of the service of an SQL **rollback** request or the start of the rollback of a transaction aborted due to an outside event such as a system failure. The service of the **rollback** request also includes performing the backward-rolling phase of the transaction back to the undo action for the first forward-rolling update action, after which it marks the transactions as rolled back.

The *undo sequence* for a forward-rolling action sequence  $\beta$ , denoted  $\beta^{-1}$  or  $\text{undo}(\beta)$ , consists of undo actions for the sequence of update actions  $o_1 o_2 \dots o_n$  contained in  $\beta$ , in reverse order:

$$\text{undo}(\beta) = \beta^{-1} = o_n^{-1} o_{n-1}^{-1} \dots o_1^{-1} = \text{undo}(o_n) \text{undo}(o_{n-1}) \dots \text{undo}(o_1).$$

The *undo action* or *inverse action* for an update action  $o$  is defined separately for each action:

1.  $\text{undo}(\text{insert tuple } t \text{ into } r) = \text{delete } t \text{ from } r.$
2.  $\text{undo}(\text{delete tuple } t \text{ from } r) = \text{insert } t \text{ into } r.$
3.  $\text{undo}(\text{change the value of attribute } A \text{ in tuple } t \text{ of } r \text{ from } u \text{ to } v) = \text{restore the value of attribute } A \text{ in tuple } t \text{ of } r \text{ to } u.$
4.  $\text{undo}(\text{create relation } r(Z) \text{ in the database}) = \text{drop empty relation } r(Z) \text{ from the database.}$
5.  $\text{undo}(\text{drop empty relation } r(Z) \text{ from the database}) = \text{create relation } r(Z) \text{ in the database.}$

Unlike a forward-rolling transaction, a backward-rolling transaction cannot be continued arbitrarily. The next action to be performed is always uniquely defined:



the undo action for the last forward-rolling update action still undone is the one to be performed next. Thus, any backward-rolling transaction will eventually enter in a state in which all its forward-rolling updates have been undone; then the transaction is recorded to have rolled back.

The state of a *rolled back transaction* is an action sequence of the form

$$B\alpha A\alpha^{-1}C, \quad (1.4)$$

where  $B\alpha A\alpha^{-1}$  is the state of a backward-rolling transaction and the  $C$  action marks the transaction as being rolled back. Thus, a rolled back transaction has, as a result of its backward-rolling phase  $\alpha^{-1}$ , rolled back all of its forward-rolling updates.

The definition of a rolled back transaction implies that such a transaction has no permanent effect on the state of the logical database: whatever updates it does in its forward-rolling phase are all undone in the backward-rolling phase. As will be seen later, the physical database that stores the logical database, however, is not necessarily restored into its original state by the undo actions. Note, for example, that undoing a tuple deletion need not bring the tuple back to its original data page but may insert it to some other page allocated to the same relation.

We use the same action name,  $C$ , to mark the end of both committed and rolled back transactions. This is because from the point of view of transaction processing, the same procedure is performed in both cases, only the result indication to be returned to the application being different.

A committed or rolled back transaction is said to be *terminated*. A terminated transaction cannot be further advanced with any action. A forward-rolling or backward-rolling transaction is *active*. A backward-rolling or rolled back transaction is *aborted*.

*Example 1.2* Let us change the embedded SQL fragment of Example 1.1 so that instead of committing the transaction, it is aborted and rolled back:

```
exec sql update r set V = 2 * V;
exec sql select sum(V) into :new_sum from r;
exec sql rollback.
```

The following action sequence is generated:

$$BR[x_1, v_1]W[x_1, v_1, 2v_1]R[x_2, v_2]W[x_2, v_2, 2v_2] \dots R[x_n, v_n]W[x_n, v_n, 2v_n] \\ R[x_1, 2v_1]R[x_2, 2v_2] \dots R[x_n, 2v_n] \\ AW^{-1}[x_n, v_n, 2v_n] \dots W^{-1}[x_2, v_2, 2v_2]W^{-1}[x_1, v_1, 2v_1]C.$$

Here an undo action  $W^{-1}[x_i, v_i, 2v_i]$  restores the previous  $V$  value  $v_i$  of the tuple with key  $x_i$ .

Should instead a system failure occur during performing the **update** statement, the following transaction is generated, assuming that the update on tuple  $(x_i, v_i)$  is the last that is found recorded (in the log saved on disk) at the time the system recovers from the failure:

$$BR[x_1, v_1]W[x_1, v_1, 2v_1]R[x_2, v_2]W[x_2, v_2, 2v_2] \dots R[x_i, v_i]W[x_i, v_i, 2v_i] \\ AW^{-1}[x_i, v_i, 2v_i] \dots W^{-1}[x_2, v_2, 2v_2]W^{-1}[x_1, v_1, 2v_1]C.$$

We conclude that in both cases, the logical database is restored to the state in which it was before the transaction started.

We leave as an exercise to figure out what should be done if a system failure occurs during performing the **rollback** statement.  $\square$

## 1.6 ACID Properties of Transactions

The implementor of a database application must take care that each committed transaction  $T$  produced by the application process is *logically consistent*: when run alone without outside disturbances or failures on a logically consistent database,  $T$  keeps the database consistent.

Naturally, it has to be possible to program the transactions without taking care of the physical structure of the database, other concurrent transactions, or system failures. The function of the database management system is to ensure that both logical and physical consistencies are preserved when there are multiple logically consistent transactions running at the same time and system failures can occur.

A rolled back transaction is always trivially logically consistent, regardless of what it does in its forward-rolling phase and what the integrity constraints of the database are: in any case, the backward-rolling phase undoes any updates done in the forward-rolling phase.

A forward-rolling (and thus uncommitted) transaction does not need to be logically consistent. In some cases it may be difficult or even impossible to retain a referential integrity constraint in effect between two update actions; in such cases it is meaningful to require integrity checking to be enforced only at transaction commit.

In general, transactions are required to satisfy four named properties, called the *ACID properties*, of which we have already defined the following three:

1. *Atomicity*: all the updates performed by a committed transaction appear in the database, and all the updates performed by an aborted transaction are rolled back.
2. *Consistency*: committed transactions retain the consistency of an initially consistent logical database.
3. *Durability*: all the updates performed by a committed transaction remain permanently in the database.

The fourth ACID property is:

4. *Isolation*: each transaction has—more or less—the impression that it is running alone, without any other transactions interfering. More specifically, isolation means restricting how early updates by a transaction become visible to other transactions and how early values read by a transaction can be overwritten by other transactions. When transactions are run in full isolation, an update by a transaction becomes visible to, or a value read by a transaction is overwritten by, another transaction only when the first transaction has committed; then it seems

that all the transactions are executed serially, one transaction at a time, although in reality some transactions are running concurrently.

Two of the above properties, namely, atomicity and durability, are maintained solely by the database management system, according to the transaction boundaries, that is, the placement of the **commit** or **rollback** statements, as designated by the application programmer. The other two properties are maintained partly by the application programmer and partly by the database management system.

For consistency, the application programmer must ensure that each of her committed transactions, when run alone in a failure-free environment on a consistent logical database, retains the consistency of the database, while the system must ensure that the consistency of the logical database is also preserved in the presence of multiple concurrently running transactions, to the extent possible under the isolation levels set for the transactions. If all the updating transactions are set to run at the highest isolation level (i.e., serializable), the system is expected to provide full preservation of the consistency of the logical database.

For isolation, the application programmer must ensure that each of her transactions is designated to run at an isolation level high enough so that database consistency is preserved to a sufficient extent, while the system must ensure that the designated isolation level indeed is achieved in the presence of concurrent transactions.

*Example 1.3* Assuming the primary keys in  $r(\underline{X}, V)$  are integers, the following embedded SQL program fragment reads the maximum key  $x$  from  $r$  and inserts into  $r$  a new tuple with key  $x + 1$ :

```
exec sql select max( $X$ ) into : $x$  from  $r$ ;
exec sql insert into  $r$  values(: $x$  + 1,  $v'$ );
exec sql commit.
```

When run alone without interference from other transactions and in the absence of failures, the program fragment generates a committed transaction of the form

$$B \dots R[x, v] \dots I[x + 1, v']C,$$

where the action  $R[x, v]$  retrieves the tuple  $(x, v)$  with the maximum key  $x$  from  $r$  and the action  $I[x + 1, v']$  inserts the tuple  $(x + 1, v')$ . (To determine the maximum key  $x$ , also some other read actions may be needed.)

The above transaction is obviously logically consistent: when run alone, it retains the consistency of an initially consistent logical database. The designer of the transaction (the application programmer) is allowed to assume that no other transaction can change the maximum key between the executions of the read action  $R[x, v]$  and the insert action  $I[x + 1, v']$ .

A violation of the primary-key constraint would occur if some other transaction inserted a tuple with key  $x + 1$ . This other transaction could, for example, be generated from another instance of the same application, and both transactions may read the same maximum key  $x$  and hence try to insert a tuple with the key  $x + 1$ .

It is the responsibility of the application programmer to designate the transaction to be run at a level of isolation that disallows the occurrence of such a violation of database integrity, while the concurrency-control mechanism of the database management system has to ensure that such an isolation level is indeed achieved.

The system has also to ensure that if the transaction indeed commits, the insertion of the new tuple is recorded permanently in the database and that otherwise if the transaction after all does not commit, the transaction is aborted and rolled back, undoing the insertion (if needed).  $\square$

Allowing transactions to be run at an isolation level lower than full isolation usually means more efficient transaction processing, because of the increased concurrency between transactions. However, with lower than full isolation, the execution of transactions may not correspond to any of their serial executions, being thus incorrect with respect to the usual notion of correctness. In many cases this is acceptable (see Sects. 5.5 and 9.5). Additionally, however, as shown in Chap. 5, if a transaction is run at a low isolation level, it may not be possible for the system to maintain logical consistency. For example, it would be highly risky to run the transaction of Example 1.3 at any isolation level lower than full isolation (serializable).

The program steps between two transaction boundaries in a database application program can be viewed as a mapping from a vector of input values to a vector of output values, where the input values are relations, constants, and values of program variables given as input to database actions, and the output values are results returned by those actions. The output values include both relations updated in update actions and the values returned by read actions.

It is in line with the above definition of the ACID property “C” and the responsibility laid on the programmer in this respect to regard such a mapping as *correctly programmed* so that an input vector is correctly mapped to an output vector when the program is run alone on a consistent logical database in the absence of failures. Again, serial executions are guaranteed to retain correctness, while correctness may be lost in nonserial executions.

*Example 1.4* Assuming the  $V$  values in  $r(\underline{X}, V)$  are numeric, the following embedded SQL program fragment generates a transaction,  $T_1$ , that scans the relation  $r$  and computes the sum of the  $V$  values of all tuples in  $r$ :

```
exec sql select sum( $V$ ) into : $s$  from  $r$ ;  
exec sql commit.
```

The mapping defined by this program fragment maps the set of  $V$  values retrieved from  $r$  to the sum stored into the program variable  $s$ .

Another transaction,  $T_2$ , generated from the following program fragment is run concurrently with  $T_1$ :

```
exec sql insert into  $r$  values ( $x, u$ );  
exec sql insert into  $r$  values ( $y, v$ );  
exec sql commit.
```

The mapping defined by this program fragment maps the constants  $x$ ,  $u$ ,  $y$ ,  $v$  and the current  $r$  to the relation obtained from  $r$  by inserting the tuples  $(x, u)$  and  $(y, v)$ .

In serial execution, the sum computed by  $T_1$  includes both of the values  $u$  and  $v$  if  $T_2$  is executed first and includes neither of them otherwise. Both executions must be regarded as correct.

However, if  $T_1$  is run at an isolation level lower than full isolation (serializable), the sum computed by  $T_1$  may include only one of the values  $u$  and  $v$ . For example, if  $x < y$  and  $T_1$  scans  $r$  in ascending key order, it may happen that  $T_2$  inserts  $(x, u)$  after  $T_1$  has already scanned a key greater than  $x$ , but inserts  $(y, v)$  and commits before  $T_1$  has scanned the greatest key less than  $y$ . In this case  $v$  is included in the sum, but  $u$  is not.

This phenomenon, called the *phantom phenomenon*, is possible if  $T_1$  is run at an isolation level that permits an isolation anomaly called “unrepeatable read.” It is up to the application programmer to decide whether or not that is acceptable.  $\square$

The ACID properties stated above only pertain to the logical database. The database management system is solely responsible for maintaining *physical consistency*, that is, integrity of the underlying physical database, regardless of whether or not logical consistency is maintained. This means, for instance, that a B-tree index structure for a relation is maintained in a consistent and balanced state even in the presence of logically inconsistent transactions and process failures and system crashes.

## 1.7 The Read-Write Model

In order to study transaction management in more detail, we need to define a database and transaction model that specifies the actions that are used in transactions. For the sake of simplicity, we assume that our logical database consists of only one relation with the scheme  $r(\underline{X}, V)$ . The tuples of the relation are pairs  $(x, v)$ , where  $x$  is the unique *key* of the record and  $v$  is the *value* of the tuple (i.e., the values of the other attributes in the tuple). Transactions always operate on  $r$ 's tuples using the key  $x$ .

In the simplest transaction model, called the *read-write model*, a transaction on the database  $r$  can contain, besides the begin-transaction action  $B$ , the abort-transaction action  $A$ , and the commit-transaction (or complete-rollback) action  $C$ , the following two types of forward-rolling database actions:

1. *Read actions* of the form

$$R[x, v] \tag{1.5}$$

for reading the tuple  $(x, v)$  with key  $x$ . The key  $x$  is an input parameter for the action. The action fetches the unique tuple  $(x, v)$  with key  $x$  from  $r$ . If the tuple is not found, the action fails. A shorthand notation for the action is  $R[x]$ .

2. *Write actions* of the form

$$W[x, u, v] \quad (1.6)$$

for updating the value of the tuple with key  $x$ . The key  $x$  and the new value  $v$  are input parameters, and the old value  $u$  is an output parameter. The action replaces the value of the tuple with key  $x$  in the relation; the former value  $u$  will be replaced by  $v$ . If the tuple does not exist, the action fails. Shorthand notations are  $W[x, v]$  and  $W[x]$ .

*Example 1.5* The action sequence

$$BR[x, u]R[y, v]W[z, w, u + v]$$

is a forward-rolling transaction that reads the values  $u$  and  $v$  of the tuples with keys  $x$  and  $y$ . Then, in the tuple with key  $z$ , the transaction replaces the previous value  $w$  by the sum  $u + v$ . When augmented with the commit action  $C$ , this sequence becomes a committed transaction:

$$BR[x, u]R[y, v]W[z, w, u + v]C.$$

□

In the backward-rolling phase of an aborted transaction or in a partial rollback to a savepoint, the *undo actions* of the forward-rolling write actions are performed in reverse order.

3. For the write action  $W[x, u, v]$ , the *undo-write action*

$$W^{-1}[x, u, v] = \text{undo-}W[x, u, v] \quad (1.7)$$

restores the value  $u$ , that is, its effect on the logical database is that of  $W[x, u, v]$ .

*Example 1.6* The action sequence

$$BR[x, u]R[y, v]W[z, w, u + v]AW^{-1}[z, w, u + v]$$

is a backward-rolling aborted transaction, which has undone its only write action. When augmented with the action  $C$ , this sequence becomes a rolled-back aborted transaction:

$$BR[x, u]R[y, v]W[z, w, u + v]AW^{-1}[z, w, u + v]C.$$

□

Insertions and deletions of tuples cannot be represented in the read-write model. When transaction management based on the read-write model is examined in the literature, the targets of the actions are not tuples but abstract uninterpreted “data items,” which are not precisely defined. If the sets of tuples of the logical database, for instance, the relations of a relational database, are selected as the data items, it is possible to model insertions and deletions of records by write actions on the

set (relation). Similarly, if the pages of the physical database where the tuples are stored are selected as data items, insertions and deletions of tuples can be modeled as write actions on a page. In both cases, transaction management is rather coarse grained: the log must record changes to a whole relation or page, and the unit of synchronization for managing concurrency (the lockable unit) is a whole relation or page.

In modern database management systems, individual tuples are used in log records and in locking (as the most fine-grained units). In this case, when a transaction  $T_1$  has updated a page and is still active, another transaction  $T_2$  can update the same page. It is also allowed that, due to a structure modification (such as a page split in a B-tree) caused by  $T_2$ , the tuple that  $T_1$  updated on page  $p$  will be moved to another page  $p'$  while  $T_1$  is still active.

## 1.8 The Key-Range Model

In the transaction model used throughout this book and termed the *key-range model*, read actions return the least key in a given *key range*, and the update actions are insertions and deletions of tuples with a given key.

The set of key values for the tuples is assumed to be totally ordered. The ordering is denoted  $\leq$  and its inverse relation  $\geq$ ; the respective irreflexive ordering relations are referred to in the familiar way as  $<$  and  $>$ . The least possible key value is  $-\infty$ , and the largest is  $\infty$ ; we assume that these do not appear in any tuple in the actual database.

The key-range model is sufficient to model the most important principles that are used in physiological log-based recovery and tuple-level concurrency control (key-range locking of a unique key). The model can also be used to describe, in a natural manner, isolation anomalies (see Sect. 5.3) coming from concurrent key-range reads and insertions and deletions of individual tuples (for instance, the phantom phenomenon; see Sects. 1.6 and 5.3 and Example 1.9 below).

In the key-range model, a transaction on the database  $r$  can contain, besides the begin-transaction action  $B$ , the abort-transaction action  $A$  and the commit-transaction (or complete-rollback) action  $C$ , the following four types of forward-rolling database actions:

1. *Read-first actions* of the form

$$R[x, \geq z, v] \tag{1.8}$$

for reading the tuple  $(x, v)$  with the first key value  $x$  that satisfies the condition  $x \geq z$ . The key value  $z$  is an input parameter, and the key  $x$  and the value  $v$  are output parameters. The action fetches the tuple  $(x, v)$  whose key is the least key satisfying  $x \geq z$  and  $(x, v) \in r$ . If no such tuple exists,  $(\infty, 0)$  is returned. Shorthand notations for this action are  $R[x, \geq z]$ ,  $R[x, v]$ , or  $R[x]$ .

2. *Read-next actions* of the form

$$R[x, >z, v] \quad (1.9)$$

for reading the tuple  $(x, v)$  with the key value  $x$  next to  $z$ . The key  $z$ ,  $-\infty \leq z < \infty$ , is an input parameter, and the key  $x$  and the value  $v$  are output parameters. The action fetches the tuple  $(x, v)$  whose key is the least key satisfying  $x > z$  and  $(x, v) \in r$ . If no such tuple exists,  $(\infty, 0)$  is returned. Shorthand notations for this action are  $R[x, >z]$ ,  $R[x, v]$ , or  $R[x]$ .

3. *Insert actions* of the form

$$I[x, v] \quad (1.10)$$

for inserting the tuple  $(x, v)$  into  $r$ . Input parameters are the key  $x$  and the value  $v$ . The action inserts the tuple  $(x, v)$  into relation  $r$ . If  $r$  already contains a tuple with key  $x$ , the action fails. A shorthand notation is  $I[x]$ .

4. *Delete actions* of the form

$$D[x, v] \quad (1.11)$$

for deleting the tuple  $(x, v)$  with key  $x$  from  $r$ . The key  $x$  is an input parameter, and the value  $v$  is an output parameter. The action deletes the tuple  $(x, v)$  with key  $x$  from relation  $r$ . If the tuple is not found, the action fails. A shorthand notation is  $D[x]$ .

For an insert or delete action  $o[x]$ , we define the *undo action*  $o^{-1}[x]$  or *undo- $o$*  $[x]$  as follows:

5. The *undo-insert action*

$$I^{-1}[x, v] = \text{undo-}I[x, v] \quad (1.12)$$

undoes the action  $I[x, v]$  by deleting the tuple  $(x, v)$  from  $r$ .

6. The *undo-delete action*

$$D^{-1}[x, v] = \text{undo-}D[x, v] \quad (1.13)$$

undoes the action  $D[x, v]$  by inserting the tuple  $(x, v)$  into  $r$ .

*Example 1.7* The forward-rolling transaction

$$BR[x_1, \geq x', v_1]R[x_2, >x_1, v_2]R[x_3, >x_2, v_3]D[x, v]I[x, v_1 + v_2 + v_3]$$

reads three tuples with consecutive keys and replaces the value in the tuple with key  $x$  by the sum of the values in the three tuples. This transaction is rolled back by first performing the abort action  $A$  and then undoing the two updates and finally completing the rollback:



$$BR[x_1, \geq x', v_1]R[x_2, > x_1, v_2]R[x_3, > x_2, v_3]D[x, v]I[x, v_1 + v_2 + v_3] \\ AI^{-1}[x, v_1 + v_2 + v_3]D^{-1}[x, v]C.$$

The original value  $v$  of the tuple with key  $x$  is thus restored.  $\square$

Let  $r$  be the logical database (a relation) and  $o[\bar{x}]$  an action,  $o \in \{B, R, W, I, D, C, A\}$  and  $\bar{x}$  a sequence of constant arguments. We define when the action  $o[\bar{x}]$  *can be run* on  $r$  and what is the database (relation)  $r'$  *produced* by the action—this is denoted  $(r, r') \models o[\bar{x}]$ .

1.  $(r, r) \models o$ , when  $o \in \{B, C, A\}$ .
2.  $(r, r) \models R[x, \theta z, v]$ , if  $(x, v) \in r$  and  $x$  is the least key in  $r$  that satisfies  $x \theta z$ . Here  $\theta$  is the operator  $\geq$  or  $>$ .
3.  $(r, r) \models R[\infty, \theta z, 0]$ , if  $r$  contains no tuple with key  $x$  satisfying  $x \theta z$ .
4.  $(r, r') \models W[x, u, v]$ , if  $r' = (r \setminus \{(x, u)\}) \cup \{(x, v)\}$ .
5.  $(r, r') \models I[x, v]$ , if the key  $x$  does not appear in  $r$  and  $r' = r \cup \{(x, v)\}$ .
6.  $(r, r') \models D[x, v]$ , if  $(x, v) \in r$  and  $r' = r \setminus \{(x, v)\}$ .
7.  $(r, r') \models W^{-1}[x, u, v]$ , if  $(r', r) \models W[x, u, v]$ .
8.  $(r, r') \models I^{-1}[x, v]$ , if  $(r', r) \models I[x, v]$ .
9.  $(r, r') \models D^{-1}[x, v]$ , if  $(r', r) \models D[x, v]$ .

To simulate an *exact-match read action*, we may write  $R[x, \geq x, v]$ . By (2),  $(r, r) \models R[x, \geq x, v]$ , if  $(x, v) \in r$ .

The action sequence  $\alpha$  *can be run* on database  $r$  and *produces* the database  $r'$ , denoted  $(r, r') \models \alpha$ , if either (1)  $\alpha = \epsilon$  and  $r' = r$  or (2)  $\alpha$  is of the form  $\beta o$ , where the action sequence  $\beta$  can be run on  $r$  and produces a database  $r''$  and action  $o$  can be run on  $r''$  and produces  $r'$ .

*Example 1.8* The transaction of Example 1.7 can be run on every database that contains the tuples  $(x_1, v_1)$ ,  $(x_2, v_2)$ ,  $(x_3, v_3)$ , and  $(x, v)$  with  $x' \leq x_1 < x_2 < x_3$  but no other tuples with keys in the range  $[x', x_3]$ . The forward-rolling portion of the transaction produces a database where the tuple  $(x, v)$  has been changed to  $(x, v_1 + v_2 + v_3)$  and the entire rolled back transaction restores the original database.  $\square$

*Example 1.9* The transactions of Example 1.4 can be modeled in the key-range model as follows:

$$T_1 = BR[x_1, > -\infty, v_1]R[x_2, > x_1, v_2] \dots R[x_n, > x_{n-1}, v_n]R[\infty, > x_n, 0]C, \\ T_2 = BI[x, u]I[y, v]C,$$

where  $\{(x_1, v_1), \dots, (x_n, v_n)\}$  is the set of tuples in the relation  $r(\underline{X}, V)$  and  $x$  and  $y$  are keys not found in  $r$ . The phantom phenomenon occurs if

$$x_i < x < x_{i+1} < y = x_j$$

for some  $i$  and  $j$ , and the actions are executed in the following order (from left to right):

$$T_1 : BR[x_1, v_1] \dots R[x_{j-1}, v_{j-1}] \qquad R[x_j, v_j] \dots R[\infty, 0]C \\ T_2 : BI[x, u]I[y, v]C$$

Thus, the first tuple,  $(x, u)$ , inserted by  $T_2$  is not among the tuples read by  $T_1$ , while the second tuple,  $(y, v) = (x_j, v_j)$ , is.

We also note that the phantom phenomenon is not prevented by the simple locking protocol in which transactions obtain commit-duration shared locks (read locks) on the keys of tuples read and commit-duration exclusive locks (write locks) on the keys of tuples inserted, deleted, or updated.  $\square$

## 1.9 Savepoints and Partial Rollbacks

The transaction model of SQL allows for *partial rollbacks* of transactions: a subsequence of the update actions performed by a forward-rolling transaction is rolled back without aborting and rolling back the entire transaction. After performing a partial rollback, the transaction remains in the forward-rolling phase and can thus perform any new forward-rolling actions.

The actions to be rolled back in a partial rollback constitute a sequence of actions from the latest update action by the transaction back to preset *savepoint*. Savepoints are set in the application program using the SQL statement

**set savepoint  $P$**

where  $P$  is a unique name for the savepoint. The SQL statement

**rollback to savepoint  $P$**

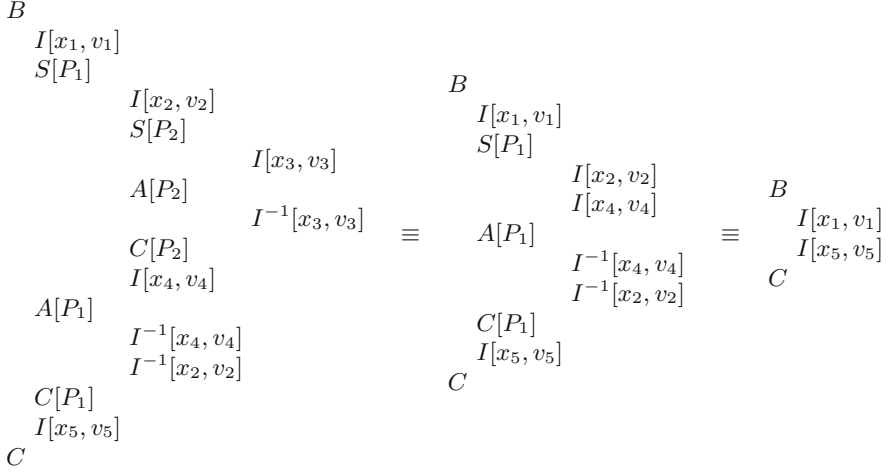
executes a partial rollback to savepoint  $P$ : all forward-rolling update actions performed by the transaction after setting  $P$  that are not yet undone are undone.

*Example 1.10* Partial rollbacks can be nested (Fig. 1.2).

```
insert into  $r$  values  $(x_1, v_1)$ ;
set savepoint  $P_1$ ;
insert into  $r$  values  $(x_2, v_2)$ ;
set savepoint  $P_2$ ;
insert into  $r$  values  $(x_3, v_3)$ ;
rollback to savepoint  $P_2$ ;
insert into  $r$  values  $(x_4, v_4)$ ;
rollback to savepoint  $P_1$ ;
insert into  $r$  values  $(x_5, v_5)$ ;
commit.
```

The statement **rollback to savepoint  $P_2$**  deletes from relation  $r$  the inserted tuple  $(x_3, v_3)$ . The statement **rollback to savepoint  $P_1$**  deletes from  $r$  the inserted tuples  $(x_4, v_4)$  and  $(x_2, v_2)$ . At the end of the transaction, an initially empty relation  $r$  contains only the tuples  $(x_1, v_1)$  and  $(x_5, v_5)$ .  $\square$

We now add partial rollbacks to our transaction model. For that purpose, we define the following actions:



**Fig. 1.2** Nested partial rollbacks. Action  $S[P]$  sets savepoint  $P$ , and partial rollback to  $P$  is begun by action  $A[P]$  and completed by action  $C[P]$ . Rolled-back segments of the transaction are shown *indented*. Because of the rollbacks, the overall effect on the logical database is the same as that of the transaction  $BI[x_1, v_1]I[x_5, v_5]C$

7.  $S[P]$ : set savepoint  $P$ .
8.  $A[P]$ : begin partial rollback to savepoint  $P$ .
9.  $C[P]$ : complete the partial rollback to savepoint  $P$ .

Now in the forward-rolling phase  $B\alpha$  of a transaction, string  $\alpha$  may contain set-savepoint actions  $S[P]$ , completed rollbacks  $S[P] \dots A[P] \dots C[P]$ , and maybe one initialized but not yet completed partial rollback  $S[P] \dots A[P] \dots$ . Formally, the *forward-rolling phase* of a transaction can now be of any of the following three forms:

- (a) A sequence  $\alpha$  of actions  $R$ ,  $I$ ,  $D$ ,  $W$ , and  $S$
- (b) An action sequence of form  $\alpha S[P] \beta A[P] \beta^{-1} C[P] \gamma$ , where  $\alpha$ ,  $\beta$ , and  $\gamma$  are of form (a) or (b)
- (c) An action sequence of form  $\alpha S[P] \beta \delta A[P] \delta^{-1}$ , where  $\alpha$ ,  $\beta$ , and  $\delta$  are of form (a) or (b)

In case (b) the subsequence  $S[P] \beta A[P] \beta^{-1} C[P]$  represents a completed partial rollback to savepoint  $P$ . In case (c) the subsequence  $S[P] \beta \delta A[P] \delta^{-1}$  indicates that the transaction is rolling back to savepoint  $P$ .

The *undo sequence* for  $\alpha$ , denoted  $\alpha^{-1}$  or *undo*( $\alpha$ ), is now defined depending on its form: For a sequence  $\alpha$  of form (a), the undo sequence  $\alpha^{-1}$  is defined as before. For a sequence of form (b), the undo sequence is  $\gamma^{-1} \alpha^{-1}$ . For a sequence of form (c), the undo sequence is  $\beta^{-1} C[P] \alpha^{-1}$ .

Savepoints and partial rollbacks constitute an important database programming paradigm: transactions can be programmed freely to update the database immediately even if some subsequent event forces the update to be rolled back and

another avenue to be followed so as to complete the transaction. In fact, with partial rollbacks, every transaction can be programmed to terminate with a **commit** request and never with a **rollback** request (i.e., total rollback). The effect of a total rollback can be achieved by setting a savepoint before the first update and then, at the end, by performing a partial rollback to that savepoint and committing the transaction.

## 1.10 Multiple Granularity

We may extend our key-range transaction model by adding *multiple granularity*, so that tuples can be grouped into relations. The relations of the database constitute a set that is totally ordered according to their identifiers. We denote this set by

$$\{(r_1, R_1), \dots, (r_{n_i}, R_{n_i})\},$$

where  $r_i$  is an identifier that uniquely identifies a relation in the database and  $R_i$  is the relation schema ( $= \underline{X}_i V_i$ ). Tuple  $(x, v)$  in relation  $r$  is then uniquely identified by the pair  $(r, x)$ .

The tuple-wise forward-rolling actions in the transaction model are now:

1.  $R[r, x, \geq z, v]$ : reading of the first tuple  $(x, v)$  with  $x \geq z$  from relation  $r$ .
2.  $R[r, x, > z, v]$ : reading of tuple  $(x, v)$  next to  $z$  from relation  $r$ .
3.  $W[r, x, u, v]$ : update of tuple  $(x, u)$  in relation  $r$ .
4.  $I[r, x, v]$ : insertion of tuple  $(x, v)$  into relation  $r$ .
5.  $D[r, x, v]$ : deletion of tuple  $(x, v)$  from relation  $r$ .

New actions include:

- (a)  $R[r', \theta r, R']$ : browsing the schema  $R'$  of relation  $r'$ .
- (b)  $I[r, R]$ : creation of a new relation  $r(R)$  into the database, corresponding to the SQL statement **create table**  $r(R)$ .
- (c)  $D[r, R]$ : deletion of an empty relation  $r$  from the database, corresponding to the SQL statement **drop table**  $r$  for an empty relation  $r$ .

Additional levels could be added to the granule hierarchy by grouping relations into databases (for different owners); new actions would then include ones corresponding to the SQL statements **create database** and **destroy database**.

## Problems

**1.1** The personnel database of an enterprise contains, among others, the relations created by the following SQL statements:

```
create table employee(empnr integer not null,
    name varchar(40) not null, address varchar(80) not null,
```

```

    job varchar(20), salary integer, deptnr integer not null,
    constraint pk primary key (empnr),
    constraint dfk foreign key (deptnr) references department);
create table department(deptnr integer not null,
    name varchar(20) not null, managernr integer,
    constraint pk primary key (deptnr),
    constraint efk foreign key (managernr) references employee).

```

Consider the transaction on the database produced by the SQL program fragment:

```

exec sql select max(empnr) into :e from employee;
exec sql select max(deptnr) into :d from department;
exec sql insert into department values (:d + 1, 'Research', :e);
exec sql insert into employee values (:e + 1, 'Jones, Mary',
    'Sisselenukuja 2, Helsinki', 'research director', 3500, :d + 1);
exec sql update department set managernr = :e + 1
where deptnr = :d + 1;
exec sql insert into employee values (:e + 2, 'Smith, John',
    'Rouvienpolku 11, Helsinki', 'researcher', 2500, :d + 1);
exec sql commit.

```

- Give the string of tuple-wise actions (readings and insertions of single tuples) that constitutes the transaction. We assume that the tuples of the relations *employee* and *department* reside in the data pages in an arbitrary order and that there exist no index to the relations.
- Repeat (a) in the case in which there exists an ordered (B-tree or ISAM) index to the relation *employee* on attribute *empnr* and an ordered index to the relation *department* on attribute *deptnr*.
- Are the transactions created in (a) and (b) logically consistent? That is, do they preserve the integrity constraints of the database?

**1.2** The following SQL program fragments operate on relation  $r(\underline{X}, V)$ . Describe the transaction produced by the program fragments (1) in the read-write model of transactions and (2) in the key-range model of transactions.

- update**  $r$  **set**  $V = V + 1$  **where**  $X = x$ ;  
**update**  $r$  **set**  $V = V + 1$  **where**  $X = y$ ; **commit**.
- update**  $r$  **set**  $V = V + 1$  **where**  $X = x$ ;  
**update**  $r$  **set**  $V = V + 1$  **where**  $X = y$ ; **rollback**.

**1.3** Explain the meaning of the transaction

$$\begin{aligned}
 &BI[r, x_1, v_1]S[P_1]I[r, x_2, v_2]S[P_2]I[r, x_3, v_3]A[P_2] \\
 &I^{-1}[r, x_3, v_3]C[P_2]I[r, x_4, v_4]A[P_1] \\
 &I^{-1}[r, x_4, v_4]I^{-1}[r, x_2, v_2]C[P_1]I[r, x_5, v_5]A \\
 &I^{-1}[r, x_5, v_5]I^{-1}[r, x_1, v_1]C.
 \end{aligned}$$

Give SQL statements to generate this transaction.

#### 1.4 A banking database contains the relations

*account*(number, balance),  
*holder*(card number, account number),  
*card*(number, holder name, holder address, crypted password),  
*transaction*(site, date time, type, amount, account number, card number),

where the relation *transaction* stores information about every completed or attempted withdrawal and deposit and about every balance lookup.

Give an embedded SQL program for a transaction for a withdrawal of  $s$  euros using card  $c$  with password  $p$ , where the withdrawal is allowed only if no overdraft will occur. The transaction also shows the balance that remains in the account after completing the withdrawal. We assume that the program includes the following statement:

**exec sql whenever sqlerror goto  $L$ ,**

where  $L$  is a program address to which the program control is transferred whenever an error status is returned by the execution of some SQL statement.

**1.5** Consider extending the read-write transaction model with an action  $C[x]$ , which *declares* the updates on  $x$  as *committed*. This action can appear in the forward-rolling phase of the transaction, and it has the effect that even if the transaction eventually aborts and rolls back, the updates on  $x$  done before  $C[x]$  will not be undone. Accordingly, complete the transaction

$BR[x, u]W[x, u, u']R[y, v]W[y, v, v']$   
 $C[x]R[z, w]W[z, w, w']W[y, v', v'']$   
 $C[y]W[x, u', u'']A$

to a rolled back transaction. Consider situations in which this feature might be useful.

**1.6** Our key-range transaction model assumes that the tuples  $(x, v)$  are only referenced via the unique primary keys  $x$ . Extend the model to include relations  $r(\underline{X}, Y, V)$ , where tuples  $(x, y, v)$  can be referenced by either the primary key  $x$  or by the (non-unique) secondary key  $y$ .

**1.7** We extend our key-range transaction model by a *cursor mechanism* that works as follows. When a transaction starts, it allocates a main-memory array (the *cursor*), private to the transaction, to store the tuples returned by all the read actions performed by the transaction, in the order of the actions. Each read action  $R[x, v]$  appends the tuple  $(x, v)$  to the next available entry in the cursor. In a more general setting, we would have several cursors, with special actions to open and close a cursor.

Now if the transaction performs a partial rollback to a savepoint, the contents of the cursor should be restored to the state that existed at the time the savepoint was set. Obviously, to that effect, we should define an undo action,  $R^{-1}[x]$ , for a read action  $R[x]$ . Elaborate this extension of our transaction model.

## Bibliographical Notes

The foundations of transaction-oriented database processing were laid by Eswaran et al. [1974, 1976] and Gray et al. [1976]. Transactions with partial rollbacks were already implemented in the System R relational database management system described by Astrahan et al. [1976], Gray et al. [1981], Blasgen et al. [1981, 1999], and Chamberlin et al. [1981]. The authors of System R state that the execution of a transaction should be atomic, durable, and consistent, where the last property is described as: “the transaction occurs as though it had executed on a system which sequentially executes only one transaction at a time,” that is, in isolation [Gray et al. 1981].

Gray [1980] presents a general formal model for transactions and their processing. Gray [1981] reviews the transaction concept and implementation approaches, analyzes the limitations of ordinary “flat transactions,” and argues for the need of “nested transactions” as a means of modeling long-lived workflows. In their textbook on the implementation of transaction processing systems, Gray and Reuter [1993] also analyze thoroughly the transaction concept and review the history of the development of the transaction concept and transaction processing systems.

In many textbooks on database management and transaction processing, most notably in the classic works by Papadimitriou [1986] and Bernstein et al. [1987], the read-write transaction model (Sect. 1.7) was adopted as the basis for discussing transaction-oriented concepts. In this setting the database was considered to be a set of uninterpreted abstract data items  $x$  that could be read (by action  $R[x]$ ) and written (by action  $W[x]$ ). Schek et al. [1993] consider the enhanced model in which the backward-rolling phase of an aborted transaction is represented explicitly as a string of undo actions; this feature is also included in the read-write model (also called the page model) used in the textbook by Weikum and Vossen [2002].

The key-range transaction model defined in this chapter and used previously by Sippu and Soisalon-Soininen [2001], Jaluta [2002], and Jaluta et al. [2003, 2005, 2006] was inspired by the model used by Mohan [1990a, 1996a] and Mohan and Levine [1992] to describe the ARIES/KVL and ARIES/IM algorithms. The transaction model considered by Gray et al. [1976] included the feature that a transaction can declare an update as committed before the transaction terminates, although this feature was only allowed for transactions run at the lowest isolation level (Problem 1.5).

Transaction Processing  
Management of the Logical Database and its  
Underlying Physical Structure  
Sippu, S.; Soisalon-Soininen, E.  
2014, XV, 392 p. 64 illus., Hardcover  
ISBN: 978-3-319-12291-5