Chapter 8

Concurrency Control on Relational Databases

"Knowledge without wisdom is a load of books on the back of an ass" (Japanese proverb)

8.1 Goal and Overview of the Chapter

In this chapter, we take a closer look at semantic approaches to concurrency control. While there are many ways to do so, we restrict the attention to a particular data model, the relational model, as this is the most widely used and, as will be seen, allows for a variety of interesting observations and developments related to transactional concurrency control. In terms of what has been discussed in the previous chapters, we here consider a specific object model, in which the objects are relations that are manipulated as a whole by (query or update) operations. This is in line with layered object models, where, for example, operations on relations or sets of tuples are implemented through record-level operations, which in turn give rise to page operations. The important observation we have already made is that semantic knowledge available at a higher abstraction level can often be exploited in such a way that histories or schedules that are unacceptable from a read-write point of view become acceptable when considered from that higher level, as syntactic conflicts essentially disappear when the semantics of operations is taken into account. This is the kind of observation we will elaborate here for relational databases.

exploiting semantic knowledge

In the terminology of Chapter 6, we restrict ourselves to studying *flat* object schedules and histories, i.e., 2-level schedules with higher-level operations at level L_1 and read/write operations at level L_0 such that all L_1 operations are isolated a priori.

The generalization to non-flat or even non-layered schedules is straightforward, using the results of Chapters 6 and 7, and not specific to the subject of the current chapter. The higher-level operations we are particularly interested in are the SQL query and update operations. We will essentially look at three approaches which indicate the variety of studies that can be conducted in the context of a particular object model.

overview

The first is *predicate-oriented* concurrency control. Here the idea is to derive granules that are subject to locking from the relational query or update expressions that trigger the data accesses. For example, the `WHERE` condition in a `SELECT` expression determines a set of tuples (in a descriptive manner by means of a logical formula) which can serve as a unit of locking. Predicate-oriented concurrency control has already been mentioned in Section 7.3 in connection with layered locking.

The second approach is based on a model for *update transactions* in relational databases that takes SQL commands as the basic building blocks for database operations. A theory of concurrency control can be established for transactions formed from such updates that exploits semantic equivalence in a variety of ways. For example, while there are classes of serializable histories that resemble the classes FSR and CSR we have seen in the read-write model, there is also a rich hierarchy of history classes in between, all of which can be recognized in polynomial time. Moreover, functional dependencies, as a special calls of database consistency constraints, allow for a state-dependent notion of correctness, and goal-orientation marks a radical departure from serializability that emphasizes a user's view on transaction cooperation.

Finally, we will look at a way of exploiting *transaction-program semantic information* as it can be derived from SQL operations, by chopping transactions into semantically equivalent smaller pieces that can be scheduled independently.

Note that these three approaches may be understood as samples of what can be done to utilize semantic knowledge; a host of additional ways and options can be found in the literature. However, we also point out right away that, although such semantic approaches to concurrency control appear attractive and elegant from a theoretical point of view, up to now they are hardly of practical relevance. The reason is that an efficient implementation is difficult to achieve, which is not surprising as it is generally recognized in computer science that semantics is more complex than syntax.

Also, to avoid misunderstandings, we emphasize once again that the state of the art for concurrency control in relational database systems is record-level locking as discussed in Chapter 7 (in combination with index locking, to be discussed in Chapter 9). So the advanced approaches covered in the current chapter should be understood as additional, potential enhancements that provide specific options.

8.2 Predicate-Oriented Concurrency Control

In Section 7.3 we discussed layered locking and in particular layered 2PL as a way of applying locking-based concurrency control to layered schedules. An important point there was the fact that lock modes can be attached to operations (such that locks are in conflict if the corresponding operations do not commute), and scheduling can be done in a level-by-level fashion. The semantically rich, high-level operations of the previous chapter, such as "Select Name From Persons Where Age=30", already suggest that *predicates* that are employed to determine a certain set of objects can be associated with locks. We will look into this view in more detail next.

relational databases

We are using the relational data model as our platform of explanation. We assume the reader to be familiar with the relational data model and its standard query language SQL at an elementary level. For our purposes, it suffices to view a relational database as a set of relations. Each relation, also known as a "table", has a schema (aka. table header), which consists of a name R and a set of attributes (aka. columns) $A_1, A_2, \ldots, A_n$ and which is written in the form $R = (A_1, \ldots, A_n)$. Associated with each attribute A_i, $1 \leq i \leq n$ is a finite domain $\mathrm{dom}(A_i)$, so that a concrete relation (or an "instance" of a relation schema) can be considered as a set $r \subseteq \mathrm{dom}(A_1) \times \ldots \times \mathrm{dom}(A_n)$ whose elements are called *tuples*. Additionally, a relation schema may comprise a set of *integrity constraints* such as *functional* or *key dependencies*; in the presence of dependencies, a relation that is to be considered *valid* or *consistent* has to observe the constraints.

transactions on relations

When database relations are subject to transactional access, there are at least two extremes w.r.t. locking: entire relations and individual tuples. When a transaction locks an entire relation, it prevents other transactions from simultaneously accessing any subset of tuples from that relation; hence there will in general be unnecessary delays. Conversely, if a transactions locks each tuple that it touches individually, a large number of locks may result, which can create a lock administration problem. We postpone a discussion of this tradeoff to Chapter 10, where we will introduce a pragmatic solution known as *multiple granularity locking*. Unfortunately, locks on individual tuples only are problematic also from a correctness viewpoint, as the following example shows.

Example 8.1

Consider a relation schema *Emp* holding employee information, with attributes *Name*, *Department*, *Position*, and *Salary*. The following is a sample relation over this schema:

Emp	*Name*	*Department*	*Position*	*Salary*
	Jones	Service	Clerk	20000
	Meier	Service	Clerk	22000
	Paulus	Service	Manager	42000
	Smyth	Toys	Cashier	25000
	Brown	Sales	Clerk	28000
	Albert	Sales	Manager	38000

Now consider a transaction for reorganizing a department that consists of the following steps: (a) fire the manager of the service department, (b) hire a new manager for this department, (c) move all employees of the service department (other than the new manager) to the sales department, (d) hire someone new for the service department to do the work under the supervision of the new manager. These steps translate into the following SQL statements:

(a) Delete the former manager (strictly speaking, all former managers) of the service department:

```
DELETE FROM Emp
WHERE Department = 'Service'
AND Position = 'Manager'
```

This removes the employee named Paulus from the database.

(b) Insert a new manager of the service department named Smith with a salary of \$40.000:

```
INSERT INTO Emp
VALUES ('Smith', 'Service', 'Manager', 40000)
```

(c) Move all employees who are not managers from the service to the sales department:

```
UPDATE Emp
SET Department = 'Sales'
WHERE Department = 'Service'
AND Position <> 'Manager'
```

(d) Insert a new employee into the service department named Stone with a salary of \$13.000:

```
INSERT INTO Emp
VALUES ('Stone', 'Service', 'Clerk', 13000)
```

Now consider a second transaction that consists of a query asking for name, position, and salary of all employees of the Sales department. In SQL this is expressed as follows:

```
SELECT Name, Position, Salary
FROM Emp
WHERE Department = 'Service'
```

□

The issue that arises with this retrieval transaction is whether it can be interleaved with the four-step update transaction and if so, at which points the interleaving can be admitted. If the retrieval transaction ran right after step (a), the deletion of the old manager, it would see a department without a manager. If it ran in between step (b) and (c), it would see the department with a new manager and the old staff. Finally, if it ran in between (c) and (d), it would see a department with the new manager and no clerks for doing the actual work. So all conceivable interleavings are illegal in that they lead to an inconsistent view of the data.

The next question then is if standard locking protocols are good enough to prevent these illegal interleavings. Assume that both transactions lock exactly the tuples (or underlying stored records) of the *Emp* table that they are actually manipulating. So step (a), the deletion of the old manager Paulus locks the tuple with name Paulus, and nothing else. If the retrieval transaction ran at this point, there would be no tuple left for the query to "trip" on a lock conflict. The query could not lock the tuple for the new manager Smith either, for this tuple does not yet exist in the database. Such tuples that have been deleted or are not yet existing but will be inserted in the future appear as "phantoms" to the query: they have latent relevance for the query's outcome but cannot be seen by the query, hence not locked. For this analogy the problem situation discussed here is generally referred to as a *phantom problem*. A similar effect would occur if the retrieval transaction ran in between steps (c) and (d), as the query could no longer lock the now "invisible" former clerks of the service department.

phantom problem

This discussion seems to suggest that in the presence of `INSERT` or `DELETE` statements transactions need to lock entire tables (i.e., the *Emp* in the example), and such very coarse-grained locking would seem to be be necessary even for `UPDATE` statements as they implicitly insert or delete tuples from the set of tuples that satisfy a given search predicate. Table locking would indeed be a feasible solution to the above problem, but it would come at the very high price of restricting concurrency to at most one active transaction on the same table at every point of time. Fortunately, there is a much better alternative to table locking that eliminates the phantom problem, too. It is based on the observation that relation accesses are commonly expressed through *predicates* such as those occurring in the `WHERE` clause of an SQL `SELECT`, `UPDATE`, or `DELETE` statement. So rather than locking individual tuples in a relation, a lock request can refer to a predicate that specifies, in a descriptive way and independently of the current contents of tables, a subset of the database to be locked. In logic such predicates that capture the time-invariant

predicate locking

meaning of a specification are also called *"intensional"*, as opposed to "extensional" which refers to the current extension of a database.

To be more precise, we consider predicates like those used in the above example, which are either *simple* conditions of the form

attribute_name comparison_operator value,

e.g.,

Department = 'Sales',

or *conjunctions* of simple conditions, e.g.,

$$Department = \text{'}Service\text{'} \wedge Position = \text{'}Manager\text{'}$$

Clearly, SQL allows more complex conditions to be expressed, but conjunctions of simple conditions will be sufficient for our purposes.

conditions

In general, let $R = (A_1, \ldots, A_n)$ be a relation schema, and let C be a set of conditions over the attributes of R s.t. each condition in C is of the form "$A_i = a$" or "$A_i \neq a$," where $1 \leq i \leq n$ and a is a value from $\text{dom}(A_i)$. It is obvious what it means for a tuple over R to *satisfy* a condition or a set of conditions. For simplicity, we will only consider sets of conditions that are satisfiable, i.e., which do not contain conditions that are mutually exclusive.

A set C of conditions over a given attribute set specifies a set $H(C)$ of tuples in a straightforward manner:

$$H(C) = \{\mu \in \text{dom}(A_1) \times \ldots \times \text{dom}(A_n) \mid \mu \text{ satisfies } C\}$$

We use the notation $H(C)$ since such a set is sometimes called a "hyperplane" (in a multidimensional "vector space" generated by the n attributes) in the literature.

Note that, given a relation r over $R = (A_1, \ldots, A_n)$ and a condition C over the attributes of R, it may happen that

- $r \subseteq H(C)$, or
- $H(C) \subseteq r$, or
- $R \cap H(C) = \emptyset$, or just
- $R \cap H(C) \neq \emptyset$.

Thus, a hyperplane may specify a superset or a subset of the tuples in a given relation, it may have a non-empty intersection with the relation, or it may be disjoint from the relation. Notice that this resembles the situation we have encountered in the previous example, where we were tempted to lock tuples that moved out of the database (through a `DELETE` or an `UPDATE`) or that newly came into the database (through an `INSERT` or an `UPDATE`).

We can now imagine that an update or retrieval operation of transaction t comes with (a) a predicate or condition C_t specifying the tuples to be locked, and (b) a lock mode m_t stating whether the elements of $H(C_t)$ are to be locked in shared or exclusive mode, i.e., whether they are to be read or written. Next, two lock requests on behalf of transactions t and t' with associated predicates C_t and $C_{t'}$ as well as lock modes m_t and $m_{t'}$ are *compatible* if

compatibility of predicate locks

- $t = t'$, or
- both m_t and $m_{t'}$ are read (shared) mode, or
- $H(C_t) \cap H(C_{t'}) = \emptyset$ (i.e., no tuple satisfies both conditions).

Example 8.2

Returning to the previous example, consider the following predicates for the update transaction:

C_a: *Department = 'Service' ∧ Position = 'Manager'*
C_b: *Name = 'Smith' ∧ Department = 'Service' ∧ Position = 'Manager' ∧ Salary = 40000*
C_c: *Department = 'Service' ∧ Position ≠ 'Manager'*
C_c': *Department = 'Sales' ∧ Position ≠ 'Manager'*
C_d: *Name = 'Stone' ∧ Department = 'Service' ∧ Position = 'Clerk' ∧ Salary = 13000*

Note that the `UPDATE` statement for step (c) has two conditions associated with it, one referring to the tuples that are to be modified and one that characterizes the result of the modification. These two conditions need to be locked separately; so each of them sets a predicate lock. This is necessary because such an `UPDATE` statement affects queries for both the service department and the sales department. The retrieval transaction, on the other hand, needs to acquire a lock on the condition

C_q: *Department = 'Service'*

In addition, consider another retrieval transaction that reads all employees of the sales department and thus needs a lock on:

C_p: *Department = 'Sales'*

It is straightforward to verify that the following holds:

1. $H(C_a) \cap H(C_q) \neq \emptyset$, $H(C_b) \cap H(C_q) \neq \emptyset$, $H(C_c) \cap H(C_q) \neq \emptyset$, $H(C_d) \cap H(C_q) \neq \emptyset$
2. $H(C_c') \cap H(C_q) = \emptyset$
3. $H(C_a) \cap H(C_p) = H(C_b) \cap H(C_p) == H(C_c) \cap H(C_p) = H(C_d) \cap H(C_p) = \emptyset$
4. $H(C_c') \cap H(C_p) \neq \emptyset$

Thus, the `SELECT` operation for the sales department, which is based on C_p and which uses a lock in shared mode, would be compatible with the update statements of steps (a), (b), and (d), and also with the first one of the two predicate locks for step (c), but it would conflict with the second lock for step (c). The `SELECT` operation for the service department, based on C_q, on the other hand, is in conflict with all of the locks set by the four update steps except the lock for C_c' set by step (c). □

scheduling

A scheduler based on predicate locking has to keep track of the predicates for which locks have already been granted. When a new lock request arrives, the scheduler performs a compatibility test, and grants the lock only if the test shows no conflict (i.e., only disjoint predicates are locked in conflicting mode). Assuming SS2PL, when a transaction commits, its predicate locks are released, and the scheduler should now wake up blocked transactions that may be able to continue. Note, however, that with predicate locks, a transaction that has requested a lock for condition C may be blocked by two different predicates, say P and Q, which could themselves be disjoint. Thus, when one of these two predicate locks, say the one for P, is released, all the scheduler can do is to retry the lock request for C but this transaction would still have to wait until the lock Q is released, too. This brief consideration already indicates that the implementation of lock waiting queues and the management of transaction threads in a data server is more complex with predicate locking than for locks on "discrete", individually named, data items.

Although conceptually elegant, predicate locking is not exactly the method of choice in practical systems. This is mostly for the reason that testing whether two predicates are compatible, i.e., whether $H(C_t) \cap H(C_{t'}) = \emptyset$ holds for given C_t and $C_{t'}$, amounts to testing a Boolean formula for satisfiability, which is known to be an NP-complete problem. In other words, predicate locking is prohibitively expensive in general. However, very limited special cases are of practical interest, and we will reconsider a very simple form of predicate-oriented locks in Chapter 9 on concurrency control for index structures.

precision locking

A variant of predicate locking called *precision locking* avoids the satisfiability test, but still is not a light-weight protocol. The scheduler grants all lock requests right away, i.e., without testing predicates for compatibility. However, when a transaction reads or writes a tuple, this tuple is tested for compatibility with all other outstanding predicates of other transactions. A tuple read or write is rejected if the tuple in questions intersects the hyperplane of another transaction's predicate and the corresponding lock modes conflict.

So precision locking tests "discrete" (or, in logic terminology, "extensional") writes against predicate-oriented ("intensional") reads; there are no predicates associated with write operations. Therefore, to detect conflicts between predicate-oriented update statements, such a statement needs to be handled as a predicate-oriented read (for the statement's retrieval part) follow by a "discrete" writes of the qualifying tuples. For example, the predicate conflict between the two operations:

```
DELETE FROM Emp
WHERE Department = 'Toys' And Position = 'Clerk'
```

and

```
INSERT INTO Emp VALUES
('Johnson', 'Toys', 'Clerk', 19000)
```

would be detected by conceptually splitting the DELETE statement into a query with the search condition *Department = 'Toys'* $\wedge$ *Position = 'Clerk'* and a sequence of simple, record-oriented, deletions for the tuples in the query result. The inserted tuple for the new employee "Johnson" would satisfy the query's predicate, and a conflict would be signaled.

8.3 Relational Update Transactions

In this section we take a different look at relational concurrency control where insert, update, and delete operations are taken as the constituents of flat object-model transactions. The theory presented here has been investigated for update operations only; we leave it as an exercise to extend it in such a way that queries are covered as well (see Exercise 6). So the transactions to be considered next will consist of update operations of the form shown earlier in example 8.1, i.e., they will be sequences of insert, delete, and modify (formerly called update) operations, hence the name *IDM transaction model*.

8.3.1 Syntax and Semantics

We continue to use the conditions that were introduced in the previous section for describing update operations and introduce a simple language for expressing updates next. Let R be a relation schema:

relational update operations

(i) An *insertion* is an expression of the form $i_R(C)$, where the conditions in C specify a complete tuple over R. This corresponds to the SQL statement `INSERT INTO R VALUES C`.

(ii) A *deletion* is an expression of the form $d_R(C)$, where C is a set of conditions over R. In SQL we would write `DELETE FROM R WHERE C`.

(iii) A *modification* is an expression of the form $m_R(C_1; C_2)$, where C_1 and C_2 are sets of conditions over R, and for each attribute A of R either $C_1|_A = C_2|_A$ or "$A = a$" $\in C_2$ holds; the equalities occurring in C_2 but not in C_1 state how the tuples in H(C_1) are modified. In SQL, we would write `UPDATE R SET C2 WHERE C1`.

In the following, an *update operation* is an insertion, a deletion, or a modification.

The semantics of the update operations, called their *effect* should intuitively be the following. Let r be a relation (set of tuples) over schema R:

(i) The *effect* of $i_R(C)$ is
$$\text{eff}[i_R(C)](r) := r \cup \{C\}$$

(ii) The *effect* of $d_R(C)$ is
$$\text{eff}[d_R(C)](r) := r - H(C)$$

(iii) The *effect* of $m_R(C_1; C_2)$ is
$$\text{eff}[m_R(C_1; C_2)](r) := (r - H(C_1)) \cup \{m_R(C_1; C_2)(\mu) \mid \mu \in H(C_1) \cap r\}$$
Here, the modified version $m_R(C_1; C_2)(\mu_1)$ of a tuple $\mu_1 \in H(C_1)$ under $m_R(C_1; C_2)$ is the tuple $\mu_2 \in H(C_2)$ s.t.
$$\mu_2(A) = \begin{cases} \mu_1(A) & \text{if } C_1|_A = C_2|_A \\ a & \text{if "}A = a\text{"} \in C_2 \end{cases}$$
for each $A \in X$, where $\mu_i(A)$, $i = 1, 2$, stands for the A component of tuple μ_i.

Thus, as mentioned above, an insert operation inserts a completely specified new tuple. A delete operation deletes all tuples satisfying a given condition, and a modify operation takes out tuples to be modified and reinserts their modified versions.

With these preparations, we can now formally define the IDM relational transaction model:

IDM transactions

Definition 8.3 (IDM transaction)

An *IDM transaction* over a database schema D is a finite sequence of update operations over D.

If $t = u_1 \ldots u_m$ is an IDM transaction over a given database, the *effect* of t, written eff(t), is defined as
$$\text{eff}(t) := \text{eff}[u_1] \circ \text{eff}[u_2] \circ \ldots \circ \text{eff}[u_m]$$
where $\circ$ denotes function composition.

This definition of (the semantics of) a transaction immediately leads to a notion of transaction equivalence:

equivalence of IDM transactions

Definition 8.4 (Transaction equivalence)

Two IDM transactions t and t' over the same database schema are *equivalent*, written $t \approx t'$, if $\text{eff}(t) = \text{eff}(t')$, i.e., if t and t' have the same effect.

It has been shown in the literature (see the bibliographical notes for this chapter) that equivalence of IDM transactions is "easy" to test, i.e., for given IDM transactions t and t' it can be decided in polynomial time whether $t \approx t'$. There are two ways to exhibit a constructive proof of this statement: One is through the specification of a sound and complete axiomatization of equivalence, the other is through a graphical illustration of transaction effects. We will not look into details of any of these, but concentrate on a subset of the set of axioms that cover update *commutativity*.

For the exposition that follows, we mention that the effects of update operations over distinct relations can be shown to be independent of each other, so that it suffices to consider transactions over a single relation in what follows. Most of the time, we even look at the simplest schema possible, which has a single attribute only. Thus, when we write $i(3), d(2)$, or $m(1;4)$ in the sequel, we mean that value 3 is inserted into, 2 is deleted from, and 1 is modified to 4 in the single-attribute relation at hand.

8.3.2 Commutativity and Simplification Rules

When performing update operations on a relational database, it may or may not matter in which order a certain sequence of operations is applied. For example, two insert operations commute if they refer to distinct tuples. In general, the following *commutativity rules* for IDM transactions hold, with C_1, C_2, C_3, and C_4 being sets of conditions that describe pairwise disjoint hyperplanes:

commutativity rules

1. $i(C_1)i(C_2) \approx i(C_2)i(C_1)$
2. $d(C_1)d(C_2) \approx d(C_2)d(C_1)$
3. $d(C_1)i(C_2) \approx i(C_2)d(C_1)$ if $C_1 \neq C_2$
4. $m(C_1;C_2)m(C_3;C_4) \approx m(C_3;C_4)m(C_1;C_2)$ if $C_3 \neq C_1, C_2$ and $C_1 \neq C_4$

5. $m(C_1;C_2)i(C_3) \approx i(C_3)m(C_1;C_2)$ if $C_1 \neq C_3$
6. $m(C_1;C_2)d(C_3) \approx d(C_3)m(C_1;C_2)$ if $C_3 \neq C_1, C_2$

Formally, these rules generate a relation "$\leftrightarrow$" on the set of all transactions (for a fixed database), whose reflexive and transitive hull we denote by $\overset{*}{\leftrightarrow}$; thus, $t_1 \overset{*}{\leftrightarrow} t_2$ means that the equivalence of t_1 and t_2 can be proved using only the commutativity rules. Notice that $t_1 \overset{*}{\leftrightarrow} t_2$ is sufficient for $t_1 \approx t_2$, but not necessary.

Another group of axioms needed for a complete characterization of equivalence consists of *simplification rules*, which are to be applied from left to right; as before, let C_1, C_2, and C_3 be sets of conditions describing pairwise disjoint hyperplanes:

simplification rules

1. $i(C_1)i(C_1) \Longrightarrow i(C_1)$
2. $d(C_1)d(C_1) \Longrightarrow d(C_1)$
3. $i(C_1)d(C_1) \Longrightarrow d(C_1)$
4. $d(C_1)i(C_1) \Longrightarrow i(C_1)$
5. $m(C_1;C_1) \Longrightarrow \epsilon$
6. $m(C_1;C_2)i(C_2) \Longrightarrow d(C_1)i(C_2)$
7. $i(C_1)m(C_1;C_2) \Longrightarrow m(C_1;C_2)i(C_2)$
8. $m(C_1;C_2)d(C_1) \Longrightarrow m(C_1;C_2)$
9. $m(C_1;C_2)d(C_2) \Longrightarrow d(C_1)d(C_2)$
10. $d(C_1)m(C_1;C_2) \Longrightarrow d(C_1)$
11. $m(C_1;C_2)m(C_1;C_3) \Longrightarrow m(C_1;C_2) \quad (C_1 \neq C_2)$
12. $m(C_1;C_2)m(C_2;C_3) \Longrightarrow m(C_1;C_3)m(C_2;C_3)$

We emphasize again that the commutativity rules as well as the simplification rules alone are not enough for a complete axiomatization of transaction equivalence; however, for our purposes in this chapter these will suffice.

8.3.3 Histories and Final-State Serializability

ies

We are now ready to define a straightforward notion of histories for IDM transactions. Let $T = \{t_1, \ldots, t_n\}$ be a given set of IDM transactions. Then a history s for T contains all operations from each $t \in T$ in a way that preserves their original order. A history is *serial* if all transactions appear strictly one after the other. We could also consider termination operations here, but for simplicity just assume that all transaction eventually commit.

Example 8.5

Let $T = \{t_1, t_2\}$ with $t_1 = d(3)m(1;2)m(3;4)$ and $t_2 = d(3)m(2;3)$. Then

$$s = d_2(3)d_1(3)m_1(1;2)m_2(2;3)m_1(3;4)$$

is a history for T.

Notice that, in a static situation, we could even imagine that a history is optimized before it gets processed. For example, we could obviously apply some simplification rules to s in order to obtain the following history:

$$s' = d(3)m(1;4)m(2;4)m(3;4)$$

We have $\text{eff}(s) = \text{eff}(s')$; however, the latter history does no longer exhibit from which transaction an update stems. □

We can define a notion of serializability for IDM transactions that is analogous to final-state serializability that we have earlier defined for read-write page-model transactions; for the close similarity between the two notions, we can even keep the same name:

IDM final-state serializability

Definition 8.6 (Final-state serializability)

A history s for a set $T = \{t_1, \ldots, t_n\}$ of IDM transactions is *final-state serializable* if $s \approx s'$ for some serial history s' for T.

Let FSR_{IDM} (or $\text{FSR}_{IDM}(T)$) denote the class of all final-state serializable histories (for T).

Notice that this notion is now defined via the semantics of a history considered as an IDM transaction, where the semantics is captured in the effects of the updates that are present.

Example 8.7

Consider again $t_1 = d(3)m(1;2)m(3;4)$, $t_2 = d(3)m(2;3)$, and

$$s = d_2(3)d_1(3)m_1(1;2)m_2(2;3)m_1(3;4)$$

as in the previous example. The final effect of s is that value 3 is deleted, and 1 and 2 get modified to 4 (provided these values are present). We now find that $s \not\approx t_1 t_2$, since the latter serial history would first delete 3, then modify 1 and 2 to 3, and leave everything else unchanged. Next, we also find $s \not\approx t_2 t_1$, since now the serial history would delete 2 and 3, then modify 1 to 2. Thus, $s \notin \text{FSR}_{IDM}$.

According to common understanding, s would hence not be considered acceptable. However, the IDM model can allow for a more relaxed approach if transaction simplification is applied. Indeed, we can optimize the first transaction to omit the last update, which is redundant since value 3 is already deleted by the first update (provided it occurs in the current state). In other words, $t_1 \approx t_1' = d(3)m(1;2)$. If the same update is removed from s, we find for the resulting history s':

$$s' = d_2(3)d_1(3)m_1(1;2)m_2(2;3) \approx t_1 t_2$$

Thus, s' is acceptable as an execution sequence for T. Though, s' is no longer a history in the strict sense, since not all updates from the given transactions are still there; s' is only *equivalent* to some serial history for T and can therefore be considered a member of FSR_{IDM}. □

In general, we can eventually improve histories by removing redundant updates or by simplifying individual transactions and hence the history itself; as the above example indicates, this can make the difference between serializability and non-serializability.

A second generalization of the notion of history derives from another form of optimization, now applied to the updates of several *distinct* transactions occurring in a history. If we consider history s' from the previous example as an individual transaction, we find that

$$s' \approx s'' = d(3)m(1;3)m(2;3)$$

s'' is another simplification of s' in terms of the tuple operations to be executed, although the updates remaining in s'' can no longer be directly associated with t_1 and t_2. We will not consider this form of generalization further here, but we mention that it can easily be exploited in the context of a concurrency control device that does some preprocessing on the (prefix of a) history that it is about to schedule.

We now briefly look into the question of how to test a given history for membership in class FSR_{IDM}. As for read-write transactions, we can obviously do this for given s and $T = \{t_1, \ldots, t_n\}$ by checking for each permutation ρ of $\{1, \ldots, n\}$ whether $s \approx t_{\rho(1)} \ldots t_{\rho(n)}$. Again, it can be verified that most likely there is no significantly simpler test:

Theorem 8.8

The problem of testing whether a given history is in FSR_{IDM} is NP-complete.

We are not giving the proof of this theorem in full detail here, but just mention two things about it. First, testing whether a given history s is in NP is as straightforward as it was for read-write transactions; just guess a permutation of the transactions and test whether s is equivalent to it. The latter can be done in polynomial time by looking at the corresponding transition specifications. Second, to show that the problem is indeed NP-complete, the problem of testing whether a directed graph contains a Hamiltonian cycle can be reduced to it. To this end, with a given directed graph G with n nodes a set of $n+1$ IDM transactions and a history s can be associated in such a way that G has a cycle containing each node exactly once iff $s \in \mathrm{FSR}_{IDM}$. Details of this reduction can be found in the literature.

This result is not surprising, since it is unreasonable to expect that testing membership in FSR_{IDM} becomes any easier when semantic information on transactions is available. On the other hand, it does not become any harder than in the read-write setting either, and as before it motivates looking for restrictions of the correctness notion that are decidable in polynomial time.

8.3.4 Conflict Serializability

A near-at-hand restriction of final-state serializability results from establishing a suitable notion of conflict for IDM transactions; this is now easy (and fully in line with what we have done in the page model) using the commutativity rules shown earlier:

IDM conflict serializability

Definition 8.9 (Conflict serializability)

A history s for a set T of n transactions is *conflict-serializable* if

$$s \overset{*}{\leftrightarrow} t_{\rho(1)} \dots t_{\rho(n)}$$

for some permutation ρ of T, i.e., if the equivalence of s to a serial history can be proven using the commutativity rules alone.

Let CSR_{IDM} denote the class of all conflict-serializable histories.

Thus, for conflict serializability we just take commutativity rules into account (in a similar way as we did for page-model transactions in Chapter 3), and nothing else.

In particular, the simplification rules we have listed above (or other rules that would contribute to an axiomatization of equivalence) are not considered for the moment.

As we could do for read-write transactions, we can test membership of a given history in class CSR_{IDM} by testing a suitably defined *conflict graph* that shows violations of commutativity rules through cycles:

IDM conflict graph

Definition 8.10 (Conflict graph)

Let T be a set of IDM transactions and s a history for T. The *conflict graph* $G(s) = (T, E)$ of s is defined by
$(t_i, t_j) \in E :\Longleftrightarrow (\exists\, t_i, t_j \in V, j \neq i)\ (\exists\, u \in t_i)\ (\exists\, u' \in t_j)\ (u <_s u'$ and $uu' \not\approx u'u)$.

In words, the conflict graph of a given history has the transactions as nodes and contains an edge of the form (t, t') between two transactions t and t' if an update from t does not commute with a subsequent update from t'.

Example 8.11

Consider $s = d_1(0)m_1(0;1)m_2(1;2)m_1(2;3)$. Then $G(s) = (V, E)$, where $V = \{t_1, t_2\}$, has exactly one cycle which results from the fact that $m_2(1;2)$ commutes neither with its left nor with its right neighbor. □

Now we have:

Theorem 8.12

Let s be a history for a set T of transactions. Then $s \in \mathrm{CSR}_{IDM}$ iff $G(s)$ is acyclic.

Proof. Let $G(s)$ be acyclic. We show by induction on $|T|$ that $s \overset{*}{\leftrightarrow} s'$ holds for a serial history s' for T. Clearly, the claim is trivial for $|T| = 1$. Let it be valid for all sets of less than n transactions, and consider a set T s.t. $|T| = n$; let s be a history for T s.t. $G(s)$ is acyclic. Then there exists a node $t \in V$ with out-degree 0, i.e., that has no outgoing edge. For each pair (u, v) of updates s.t. $u \in t$, $v \notin t$, and $u <_s v$ it follows that u and v commute. Therefore, $s \overset{*}{\leftrightarrow} s_0 t$, where $s_0 = \pi_{T-\{t\}}(s)$. Since $G(s_0)$ is a subgraph of $G(s)$, $G(s_0)$ is again acyclic. Using the induction hypothesis we may thus conclude that $s_0 \overset{*}{\leftrightarrow} s_1$, where s_1 is a serial history for $T - \{t\}$. Hence $s \overset{*}{\leftrightarrow} s_1 t$, i.e., $s \in \mathrm{CSR}_{IDM}$.

Conversely, let s be a history for T s.t. $G(s) = (V, E)$ has a cycle, and suppose that $s \in \mathrm{CSR}_{IDM}$. The latter implies that there is a serial history s' for T s.t.

$s \overset{*}{\leftrightarrow} s'$. Since $G(s)$ is cyclic, s' cannot contain the transactions from T in topological order. Thus, there exist transactions $t_i, t_j \in V$ for which $s = \ldots t_i \ldots t_j \ldots$, but $(t_j, t_i) \in E$. In addition, there exist update operations $u_j \in t_j$ and $u_i \in t_i$ s.t. $s = \ldots u_j \ldots u_i \ldots$ and $u_j u_i \not\approx u_i u_j$. This implies that the equivalence of s' and s cannot be shown by using only the commutativity rules, a contradiction. □

We conclude this subsection by mentioning that, not surprisingly,

$$\mathrm{CSR}_{IDM} \subset \mathrm{FSR}_{IDM}.$$

By definition, each conflict-serializable history is final-state serializable. Moreover, the inclusion is strict due to the fact that sometimes conflicts between pairs of updates from distinct transactions get compensated by their context.

Example 8.13

Let $s = m_2(1;2)m_1(2;3)m_2(3;2)$. Here $G(s)$ is cyclic, so $s \notin \mathrm{CSR}_{IDM}$. On the other hand, $s \approx m_1(2;3)m_2(1;2)m_2(3;2)$ and hence $s \in \mathrm{FSR}_{IDM}$. □

We also mention that a straightforward scheduling protocol for dynamically generating the class CSR_{IDM} of conflict-serializable IDM histories is obtained by adapting the protocol of serialization-graph testing (SGT) appropriately.

8.3.5 Extended Conflict Serializability

So far we have established a direct analogy to the page model of transactions, namely "global" (final-state) serializability with an NP-complete decision problem, and "local" (conflict-based) serializability, where the latter is (again) a strong restriction of the former. A natural question to ask then is whether there is something in between. As will be seen next, it is here where the full power of IDM concurrency control arises.

Recall that conflict serializability only takes local conflicts into account, whereas final-state serializability looks at the entire context at once. An intermediate approach would therefore be to look at some portion of the context only, and we will show next that in the IDM model of transactions this can be done in various ways.

We will first look at an approach that seems particularly amenable to dynamic scheduling, as the idea is to look at conflicts between single updates u from one transaction and prefixes of other transactions occurring in a given history prior to u. As for ordinary conflicts, a notion of serializability can be defined by means of an appropriately chosen graph:

extended IDM conflict graph

Definition 8.14 (Extended conflict graph and serializability)

Let s be a history for a set $T = \{t_1, \ldots, t_n\}$ of transactions.

(i) The *extended conflict graph* $EG(s) = (T, E)$ of s is defined as follows:

$$(t_i, t_j) \in E :\Longleftrightarrow (\exists\, u \in t_j)\ s = s'us'' \text{ and } \pi_{\{t_i\}}(s')u \not\approx u\pi_{\{t_i\}}(s')$$

(Here $\pi_{\{t_i\}}(s')$ denotes the projection of s' onto $\{t_i\}$.)

(ii) s is *extended conflict-serializable* if $EG(s)$ is acyclic.

Let ECSR_{IDM} denote the class of all extended conflict-serializable histories.

Thus, while ordinary conflict serializability looks at pairs of updates from distinct transactions only and whether they commute, extended conflict serializability looks at a transaction prefix as well as an update from another transaction and tests whether they commute. Now the effect of the entire prefix is taken into account, no longer just that of an arbitrary update within that prefix. As can be seen, this is a relaxation of conflict serializability, since inside a prefix there may exist updates that compensate or even completely reset the effect of others.

Example 8.15

Let s be as in Example 8.11. Then $s \in \mathrm{ECSR}_{IDM}$, since

$$s \approx m_2(1;2)d_1(0)m_1(0;1)m_1(2;3) = t_2t_1. \quad \square$$

Example 8.16

Consider $s = m_1(0;1)m_2(1;2)m_1(2;3)d_1(1)d_1(3)$. Then $s \notin \mathrm{ECSR}_{IDM}$, but $s \in \mathrm{FSR}_{IDM}$. □

Notice that an application of just the commutativity rules to pairs of updates is not sufficient for testing whether a prefix projection of a history s of the form $\pi_{\{t_i\}}(s')$ commutes with a subsequent update $u \in t_j$, since this projection and update u might commute even if u does *not* commute with each individual update in the projection. Thus, we here exploit the transaction's semantics, and we can easily test whether given $\pi_{\{t_i\}}(s')$ and u commute by looking at the corresponding transition specifications.

Since the extended conflict graph of a given history can also be constructed and tested for acyclicity in polynomial time, membership in class ECSR_{IDM} is again efficiently decidable.

Theorem 8.17

For every database schema, $\text{CSR}_{IDM} \subset \text{ECSR}_{IDM} \subset \text{FSR}_{IDM}$.

Proof. Clearly, $\text{CSR}_{IDM} \subseteq \text{ECSR}_{IDM}$, since the extended conflict graph of a given history s is acyclic if the same holds for the ordinary conflict graph of s. We next show by induction on $|T|$ that $\text{ECSR}_{IDM} \subseteq \text{FSR}_{IDM}$ for every T: The claim trivially holds for $|T| = 1$. Now suppose it holds for every T s.t. $|T| < n$. Let T be a set of transactions s.t. $|T| = n$, and let s be a history for T s.t. $s \in \text{ECSR}_{IDM}$. Since, by assumption, $EG(s)$ is acyclic, there exists a node t in EG with out-degree 0. Then s can be modified in such a way that all updates from t occur at the end: Let u be the first update from t occurring in s. Since t has no outgoing edge, u can be moved to the right in s by using the commutativity rules, until it hits the second update, say, u' from t. If u' is followed by an update v from another transaction, the pair uu' and v can be exchanged (since t has out-degree 0 and is hence not in conflict with a subsequent transaction). In this way, we can "collect" all updates from t and move them to the end of s. As a result, we obtain a history

$$s' = \pi_{T-\{t\}}(s)t,$$

for which $s' \approx s$ holds. Since $\pi_{T-\{t\}}(s)$ has less than n transactions and $EG(\pi_{T-\{t\}}(s))$ is acyclic, our induction hypothesis applies; hence $s \in \text{FSR}_{IDM}$.

We have seen in examples already that the two inclusions are strict. □

As for class CSR_{IDM}, a protocol that can generate histories in class ECSR_{IDM} can be based on SGT scheduling, since a decision on conflicts is made based on the knowledge a scheduler already has about prefixes of transactions as well as on individual operations.

8.3.6 Serializability in the Presence of Functional Dependencies

We next present a further possibility to exploit semantic information for enriching existing notions of serializability. The major difference to what we have done so far in this chapter will be that we are now going to take *state information* into account. In particular, we will assume that for each relational database we run transactions upon, there will not only be relations and attributes, but also *functional dependencies* (FDs) as semantic integrity constraints. For a relation R with attributes $A_1, \ldots, A_n$

a functional dependency is a logical invariant of the form

$$A_{i_1} \ldots A_{i_k} \rightarrow A_j (k \leq n)$$

with the meaning that for all tuples t, t' in relation R, either present or conceivable, the logical condition

$$t.A_{i_1} = t'.A_{i_1} \wedge \ldots \wedge t.A_{i_k} = t'.A_{i_k} \Rightarrow t.A_j = t'.A_j$$

must be true. In words, if two tuples have the same value for the left-hand-side attributes of a FD, they must have the same value for the right-hand-side attribute as well. Primary key constraints are obviously a special case, but FDs can capture a wider class of integrity constraints.

Intuitively, when testing a given history for serializability, we can then restrict our attention to the effect of the history on states that satisfy the FDs. In other words, a history s is no longer required to have the same effect as a serial one for the same transactions on *every* state, but only on those respecting the FDs.

serializability with regard to FDs

This intuition could suggest the following definition: Let R be a relation schema with attributes X and a set F of FDs over X. A history s for a set T of transactions over R is *serializable with regard to* F if there exists a serial history s' for T s.t. $\text{eff}(s)(r) = \text{eff}(s')(r)$ holds for every relation r over schema R that satisfies F. Here, $\text{eff}(s)(r)$ denotes the effect of s on r, analogously for s'.

Unfortunately, it can be shown that the class of histories which is serializable w.r.t. a given set F of FDs coincides with FSR_{IDM} (for each F), so that an alternative definition of serializability is required for appropriately exploiting the additional semantic information provided by FDs. The following example motivates how such a definition can be obtained:

motivating example

Example 8.18

Consider a relation schema R with two attributes A and B as well as an FD $A \rightarrow B$, and let the following history s be given (for two transactions t_1 and t_2 over R):

$$\begin{aligned} s \;=\; & m_1(A=0, B=0; A=0, B=2) m_2(A=0, B=0; A=0, B=3) \\ & m_2(A=0, B=1; A=0, B=3) m_1(A=0, B=1; A=0, B=2) \end{aligned}$$

It is easily verified that $s \notin \text{CSR}_{IDM}$, $s \notin \text{ECSR}_{IDM}$, and even $s \notin \text{FSR}_{IDM}$. However, a detailed analysis of the conflicts occurring in s reveals the following: The first two modifications which imply a conflict between t_1 and t_2 apparently affect tuple (0, 0); similarly, the last two modifications affect tuple tuple (0, 1). Therefore, the first conflict is relevant only to a relation containing (0, 0), while the second is relevant only to a relation containing (0, 1). The point now is that these two tuples cannot occur *simultaneously* in a relation satisfying $A \rightarrow B$! Thus, in any valid relation at most one of the two conflicts will be relevant, so that depending on the current state, s is always equivalent either to t_1t_2 or to t_2t_1. □

The above observation motivates a novel notion of serializability for which equivalence to seriality is state-dependent:

IDM state serializability

Definition 8.19 (State serializability)

Let R be a relation schema with attributes X and FDs F. A history s for a set $T = \{t_1, \dots, t_n\}$ of transactions is *state-serializable w.r.t.* F if for each relation r satisfying F there exists a serial history s_r for T which depends on r s.t. $\text{eff}(s)(r) = \text{eff}(s_r)(r)$.
Let SSR^F (or more precisely $\text{SSR}^F(T)$) denote the class of all histories which are state-serializable w.r.t. F.

Notice that the particular permutation whose effect on r equals that of s is thus dependent on r itself. Now Example 8.18 demonstrates that there exist histories $s \notin$ FSR which are state-serializable w.r.t. a given set F of FDs. More generally, it can be shown that FSR_{IDM} is a strict subset of SSR^F for each F containing at least one non-trivial FD. In addition, the tests for membership in CSR_{IDM} and in FSR_{IDM} we have described earlier in this chapter can be generalized in such a way that state information is taken into account.

We are not presenting the latter in full detail here, but just sketch how it can be accomplished for the case of class CSR_{IDM}. The basic idea follows directly from Example 8.18. For a given history s and a set F of FDs, a conflict graph $G(s)$ is constructed in which the auxiliary information of which hyperplane is relevant to what conflict edge is represented. Next, if a given database state d has an empty intersection with *every* hyperplane that is relevant to an edge (t_i, t_j), then the conflict between t_i and t_j (represented by this edge) can be ignored when history s is executed on state d. If $G(s)$ contains a cycle, it can be tested using a particular form of the well-known chase procedure for enforcing FDs whether there exists a state d satisfying the FDs which has a non-empty intersection with at least one relevant hyperplane of each edge on the cycle; if no such state can be found for any cycle of $G(s)$, history s is apparently state-serializable w.r.t. F, i.e., $s \in \text{SSR}^F$.

We mention that the test just sketched is a *sufficient* condition for membership in class SSR^F only, i.e., not every state-serializable history can be recognized in this way. Correspondingly, the class of histories recognized is denoted CSSR^F (for "conflict-state-serializable histories"). Moreover, the algorithm for testing membership in CSSR^F has exponential time complexity (due to the fact that there are

exponentially many choices for sets of hyperplanes to which the chase procedure is applied). Instead of going further into details, we present another example showing the essence of what has been discussed in this subsection.

Example 8.20

Consider a relation schema R with attributes ABC and FDs $A \to C, C \to B$. Let the following history be given:

$$\begin{aligned} s = {} & d_1(A=3)m_1(A=0,B=0,C\neq 0;A=3,B=0,C\neq 0)d_2(A=2)\\ & m_2(A=0,B=0,C\neq 0;A=2,B=0,C\neq 0)d_2(A=0,B\neq 0,C=1)\\ & d_2(A=1,C=0)d_1(A=5)m_1(A=0,B\neq 0,C=1;A=5,B\neq 0,C=1)\\ & d_1(A=1,C=0)m_1(A=0,C=0;A=1,C=0) \end{aligned}$$

Note that both t_1 and t_2 are consistency-preserving when considered in isolation. Obviously, $s \notin \text{CSR}_{IDM}$, since its conflict graph has edges (t_1, t_2) and (t_2, t_1) and is hence cyclic. However, we can associate labels with these edges as follows:

$$l((t_1,t_2)) = \{C_1\}$$
$$l((t_2,t_1)) = \{C_2, C_3\}$$

where

$$C_1 = \{A=0, B=0, C\neq 0\}$$
$$C_2 = \{A=0, B\neq 0, C=1\}$$
$$C_3 = \{A=0, C=0\}$$

Thus, there are two choices for sets of hyperplanes, $\{C_1, C_2\}$ as well as $\{C_1, C_3\}$. For both, the chase procedure will not be successful, meaning that no state will intersect a hyperplane from each edge simultaneously. It follows that s is conflict-state-serializable w.r.t. F, i.e., $s \in \text{CSSR}^F - \text{CSR}_{IDM}$. □

The following theorem summarizes what can be shown for the scenario incorporating integrity constraints and hence state information we have outlined above:

Theorem 8.21

For each set F of FDs with at least one non-trivial element, the following holds:

$$\text{CSR}_{IDM} \subset \text{CSSR}^F \subset \text{SSR}^F$$

8.3.7 Summary

The IDM transaction model as described in this section is a model of transactions that is essentially "located" at the conceptual level of a database, where a user is not concerned with pages or other system-internal objects, but only looks at relations, their attributes, their values, and maybe their integrity constraints. As has been indicated, this view, as a particular case of flat object schedules as treated in Chapter 6, renders a number a investigations possible, of which we have here presented a few only: commutativity-based serializability, conflict serializability that takes an entire history prefix into account, or state-based serializability. There are various other issues to investigate. For example, it can be shown that there is an infinite hierarchy of classes of serializable histories between CSR_{IDM} and FSR_{IDM} which, in a sense, approximates the latter; the more time a scheduler is willing to invest, the closer it can get to unrestricted serializability. Yet there is still room for further study, in particular when it comes to scheduler design. Indeed, efficient, low-overhead management of the semantic information available in a model like this is a non-trivial problem, as we have also seen in previous chapters. For this particular transaction model, efficient implementations are yet to be seen.

8.4 Exploiting Transaction-Program Knowledge

The third approach to exploiting semantic information we describe in this chapter is not particularly tied to the relational model, but again fits well with it. The ideas described in this section can be combined with predicate-oriented concurrency control as discussed earlier in this chapter. They are also closely related to the approach of interleaving specifications as introduced in Section 3.10, with the key difference, however, that we will now preserve serializability despite a more sophisticated interpretation of isolation whereas the approach in Section 3.10 deliberately allowed deviating from serializability. The approach developed in the current section can be viewed as a way of automatically deriving, by means of complete knowledge of transaction programs, those interleaving specifications that are compatible with serializability.

8.4.1 Motivating Example

The approach described in this section aims to reflect the basic tuning principle of "making transactions short" in order to reduce the probability of many conflicts and long delays. It is motivated by application scenarios along the following lines.

Example 8.22

Suppose that a bank allows customers to withdraw a certain amount in cash per day from their accounts. At night, an update transaction inspects each account that has been accessed during the day, updates its balance, and then updates the corresponding branch total. The update transaction will typically be of long duration, and due to account accessibility through the Internet, the bank wants to move away from running the update transaction only at night. In order to improve service, solutions along the following lines are conceivable:

bank example

1. The single large update transaction is replaced by many small ones, each of which updates one customer's account and the corresponding branch total. Clearly, the small transactions can now execute in parallel, and the effects on the individual accounts will be the same. Also, if there are additional transactions that read a single customer's account balance, these will not see any difference compared to the original setting either.
2. As the small update transactions may now interfere with each other when updating the branch total, each of the the small transactions could be decomposed further into two transactions, one that updates the customer's account and one that updates the corresponding branch total.

Note that the short transactions that result from this kind of "chopping" are not only perfectly tolerable from an application viewpoint, but they do in fact behave as if the original, long, transactions were executed in a serializable manner. So it is conceivable that the chopping is automatically inferred from the transaction programs themselves, without any need for an application expert to provide additional semantic hints. However, the correctness of the above decomposition crucially builds on the fact that there are no other transaction programs that could, for example, access the accounts of all customers and compare their total to the sum of the bank's branch totals. Such a transaction would have an inconsistent view of the data, but if we knew that this type of access pattern does not occur, the chopping would be acceptable. □

The sample scenario just described makes a point for *short* transactions, since long ones acquire more locks and therefore cause potentially more lock contention. However, as indicated by the example, whether or not a transaction can be broken into smaller pieces depends on which other transactions may possibly be running concurrently.

To cast the above considerations into a precise model, let us make the assumption that we know exactly which transactions may be running concurrently during some time period. To this end, let us continue the last example and describe the transactions in terms of a straight-line sequence of parameter-less SQL calls. For simpler notation, we will abbreviate the SQL calls, which are very simple SELECT and UPDATE statements anyway, as if they were read and write steps on the account and branch objects. Later we will reconsider the approach and its practicality in the light of real application programs that have rich control flow and contain parameterized SQL calls.

Example 8.23

Let us return to the bank example we considered in the previous example and let us from now on assume there are the three types of transactions we discussed above as an alternative decomposition:

1. a transaction updating a single customer's account as well as the corresponding branch balance;
2. a transaction reading a customer's account balance;
3. a transaction comparing the grand total of all account balances with the sum of the branch balances.

Assume further that accounts A_1, A_2, and A_3 belong to branch B_1, and that A_4 and A_5 belong to B_2. Let the transactions be as follows:

$$\begin{aligned}
t_1 &= r_1(A_1)w_1(A_1)r_1(B_1)w_1(B_1)\\
t_2 &= r_2(A_3)w_2(A_3)r_2(B_1)w_2(B_1)\\
t_3 &= r_3(A_4)w_3(A_4)r_3(B_2)w_3(B_2)\\
t_4 &= r_4(A_2)\\
t_5 &= r_5(A_4)\\
t_6 &= r_6(A_1)r_6(A_2)r_6(A_3)r_6(B_1)r_6(A_4)r_6(A_5)r_6(B_2)
\end{aligned}$$

Transactions t_1–t_3 update a customer's account balance as well as the corresponding branch balance, transactions t_4 and t_5 just read a customer's balance, and transaction t_6 compares the grand totals of customer accounts to the sum of the branch totals.

Note once again that the transaction programs actually consist of SQL statements rather than read and write steps. So t_1, for example, would in reality be as follows:

```
SELECT Balance INTO :oldbalance FROM Accounts WHERE AccountNo = A_1;
UPDATE Accounts SET Balance = :newbalance WHERE AccountNo = A_1;
SELECT Total INTO :oldtotal FROM Branches WHERE BranchNo = B_1;
UPDATE Branches SET Total = :newtotal WHERE BranchNo = B_1;
```

It is only for more concise notation that we have denoted these statements as r and w steps. □

8.4.2 Transaction Chopping

Assume that there are n transaction programs that can execute within some given interval, leading to n possibly concurrent transaction executions. If a program can be invoked multiple times within the considered interval, we treat it as if these were two different programs. We further assume that each program consists of a straight-line sequence of SQL statements with parameter-less WHERE clauses. Decomposing, or *chopping*, a transaction program amounts to changing the program as follows:

transaction chopping

Definition 8.24 (Transaction chopping)

Let t_i be a transaction program. A *chopping* of t_i is a decomposition of t_i into ordered pieces $t_{i_1}, \ldots, t_{i_k}$ ($k \geq 1$, most often $k \geq 2$) s.t. every database operation invoked by t_i is contained in exactly one piece and the order of operation invocations is preserved.

In practice this transformation could be done simply by inserting additional `COMMIT WORK` statements into the original programs at all points where one piece ends and the next one begins. This is straightforward as long as the programs are indeed straight-line sequences of SQL statements, without any branching or loops. Also, the assumption that the pieces that result from the chopping of a transaction program obey the execution order of the original program is virtually self-guaranteed in our simplified setting.

If a piece that results from the chopping of a transaction program is aborted due to a deadlock it needs to be restarted repeatedly until it commits. If a piece is aborted due to a program-initiated rollback (e.g., as a result of an explicitly coded consistency check), no other pieces of the same transaction should execute, and only the first piece is allowed to contain such program-initiated rollback statements. These assumptions need to be guaranteed by the programs themselves in combination with the underlying runtime system. In particular, the property that once the first piece is successfully committed, all subsequent pieces must be executed to completion exactly once, so that the atomicity of the original transaction is preserved, requires elaborate considerations on recovery. We will come back to this issue in Chapter 17 on application recovery, where a straightforward solution based on transactional message queues will be presented.

Our next goal now is to test if a given chopping for a set of transaction programs is correct in that it preserves serializability of the transactions that would be spawned by the original programs. This can be done by a simple form of *static program analysis*. For simplicity, we will again use the convenient *read/write* notation rather than SQL, and we will henceforth no longer explicitly distinguish between a straight-line transaction program and the totally ordered transaction or sequence of pieces that originate from the program. For the analysis, we introduce a particular kind of *chopping graph* defined as follows:

chopping graph

Definition 8.25 (Chopping graph)

Let T be a given set of transactions, and let a chopping of the transactions in T be given. A *chopping graph* $C(T)$ is an undirected graph s.t.

(i) The nodes of $C(T)$ are the transaction pieces occurring in the chopping.

(ii) Let p, q be pieces from two different transactions. If p and q contain operations that are in conflict, $C(T)$ contains an undirected edge between p and q labelled with a c (for "conflict").

(iii) If p and p' are pieces from the same transaction, $C(T)$ contains an undirected edge between p and p' labelled with an s (for "sibling").

Note that no edge in a chopping graph can have both an s and a c label. We say that a chopping graph contains an *sc cycle* if it contains a cycle with at least one s and at least one c edge.

Example 8.26

Consider the following three transactions:

$$t_1 = r_1(x)w_1(x)r_1(y)w_1(y)$$
$$t_2 = r_2(x)w_2(x)$$
$$t_3 = r_3(y)w_3(y)$$

Let t_1 be chopped into two pieces as follows:

$$t_{11} = r_{11}(x)w_{11}(x)$$
$$t_{12} = r_{12}(y)w_{12}(y)$$

The corresponding chopping graph is shown in Figure 8.1. Obviously, there is no *sc* cycle. □

Our goal is to deal only with choppings that are "correct" in the following sense:

correct chopping

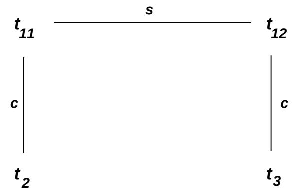

Figure 8.1: A chopping graph.

Definition 8.27 (Correct chopping)

A chopping of $T = \{t_1, \ldots, t_n\}$ is *correct* if every execution of the transaction pieces according to the following execution rules is conflict-equivalent to some serial history for the original transactions T.

Rule 1: When transaction pieces execute, they obey the precedence relationship defined by the original transaction program.

Rule 2: Each piece will be executed according to some concurrency-control protocol that ensures conflict serializability and will commit its changes when it ends.

The following can now be proven:

Theorem 8.28

A chopping is correct if the associated chopping graph does not contain an *sc* cycle.

The proof assumes that transaction pieces are executed, for example, according to the two-phase locking protocol and then exploits the fact that this protocol guarantees an acyclic conflict graph for the committed transactions.

Example 8.29

Let us return to the bank example we considered in the last two examples, where we already established the following transactions:

$$\begin{aligned}
t_1 &= r_1(A_1)w_1(A_1)r_1(B_1)w_1(B_1) \\
t_2 &= r_2(A_3)w_2(A_3)r_2(B_1)w_2(B_1) \\
t_3 &= r_3(A_4)w_3(A_4)r_3(B_2)w_3(B_2) \\
t_4 &= r_4(A_2) \\
t_5 &= r_5(A_4) \\
t_6 &= r_6(A_1)r_6(A_2)r_6(A_3)r_6(B_1)r_6(A_4)r_6(A_5)r_6(B_2)
\end{aligned}$$

We first chop t_6 into two pieces as follows:

$$\begin{aligned}
t_{61} &= r_{61}(A_1)r_{61}(A_2)r_{61}(A_3)r_{61}(B_1) \\
t_{62} &= r_{62}(A_4)r_{62}(A_5)r_{62}(B_2)
\end{aligned}$$

A look at the corresponding chopping graph, which is shown in Figure 8.2, indicates that the obtained chopping is correct.

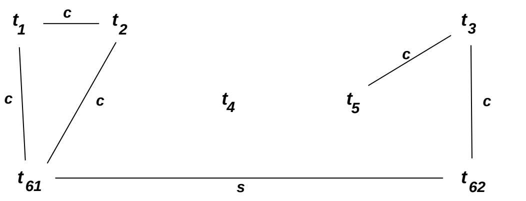

Figure 8.2: Chopping graph without *sc* cycle.

We next try to chop transaction t_1 into the following pieces:

$$\begin{aligned}
t_{11} &= r_{11}(A_1)w_{11}(A_1) \\
t_{12} &= r_{12}(B_1)w_{12}(B_1)
\end{aligned}$$

Now we obtain the chopping graph shown in Figure 8.3, which contains an *sc* cycle; hence, this chopping is not correct. □

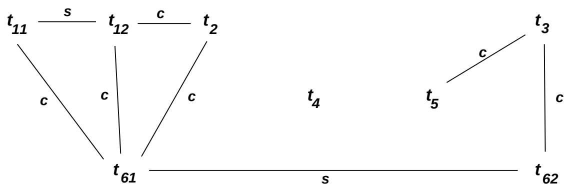

Figure 8.3: Chopping graph with *sc* cycle.

What the previous example indicates is the fact that making a chopping finer can introduce *sc* cycles, and it is clear that once such a cycle has occurred, it cannot be broken again by chopping further (since a cycle has at least one *c* edge, and further choppings always introduce additional *s* edges). On the other hand, an algorithm can be devised for finding the "finest" chopping possible among a given set of transactions. This algorithm, which can be found in the references cited in the bibliographical remarks, is based on the following observation.

From a given set $T = \{t_1, \ldots, t_n\}$, take any transaction t_i and chop it into $t_{i_1}, \ldots, t_{i_k}$. If the graph with nodes $\{t_1, \ldots, t_{i-1}, t_{i_1}, \ldots, t_{i_k}, t_{i+1}, \ldots, t_n\}$ does not contain an *sc* cycle, this chopping is a *private* chopping of t_i. If for each $t \in T$, a private chopping can be found, a chopping of T is simply obtained by taking the union of the n private choppings.

8.4.3 Applicability of Chopping

Chopping is a compile-time technique that transforms programs (by inserting additional `COMMIT WORK` statements such that upon execution, the program issues a sequence of short transactions (coined "pieces") rather than a single, longer, transaction. The semantic knowledge that is necessary for applying the chopping algorithm can in a real-world situation, such as SQL application programs, be derived from the predicates in queries and update statements (i.e., the `WHERE` clauses). This information allows us to reason about high-level conflicts between statements as discussed in the previous sections of this chapter. For example, if a database contains a relation *Accounts* with attributes such as *Balance*, *AccountType*, *City*, and a key *AccountNo*, then statements such as

```
SELECT AccountNo, Balance FROM Accounts
WHERE City = 'Austin'
```

and

```
UPDATE Accounts SET Balance = Balance * 1.05
WHERE City = 'San Francisco'
```

are not in conflict from a conceptual point of view, since disjoint records will be read (or, in the terminology used earlier, since the hyperplanes associated with the two conditions are disjoint). However, if the underlying relational database system actually employs page locking and accounts from the two different cities may possibly reside on the same page, or if the system uses record locking but there is a non-zero probability for a conflict on some auxiliary storage structures such as addressing or free space management tables, then the two SQL statements shown above would conservatively have to be assumed to be conflicting. Obviously this would lead to more c edges in a chopping graph, and may significantly limit chopping opportunities. So chopping would perform best if the underlying database system employed predicate-oriented concurrency control (which no commercial system does at this point).

Another form of conservativism is in order when we consider parameterized SQL statements, which contain program variables (see Chapter 1), denoted by prefixing the variable name with a colon. For example, consider the two statements

```
SELECT AccountNo, Balance FROM Accounts
WHERE AccountType = 'savings' AND City = :x
```

and

```
UPDATE Accounts SET Balance = Balance * 1.05
WHERE AccountType = 'checking' AND City = :y .
```

We could safely infer that these two statement do not have a conflict at the level of the SQL predicates. However, for two instantiations of the second statement (the `UPDATE` statement) nothing could be concluded anymore, as the two instances of the program variable `:y` could possibly be bound to the same city value.

In this way, chopping-relevant information can be extracted even from parameterized SQL programs, and this observation can be extended to entire application programs with embedded SQL statements, provided that these are "straight-line" programs without any control-flow branching. So in this limited setting the chopping method would be ready for practical use. For general application programs, however, where the statements are often part of loops or if-then-else constructs, the simple chopping algorithm does not work anymore. In this realistic setting the idea would be to conservatively construct a parameter-less straight-line program that *covers* all possible executions of the actual program. A chopping graph would then be established for this synthesized program, and the chopping algorithm could be applied.

For example, the program fragment

```
SELECT AccountNo, Balance FROM Accounts
WHERE AccountType = 'savings' AND City = :x;

if not found then
   SELECT AccountNo, Balance FROM Accounts
   WHERE AccountType = 'checking' AND City = :x;
fi;
```


would lead to the following straight-line, parameter-less fragment that *covers* the actual program:

```
SELECT AccountNo, Balance FROM Accounts
WHERE AccountType = 'savings';
SELECT AccountNo, Balance FROM Accounts
WHERE AccountType = 'checking';
```

Whether this conservative method is practically viable is an open issue. Skeptical readers may suspect that the covering straight-line, parameter-less program would often have to be so conservative that no real chopping is feasible anymore, but the true answer to such conjectures lies in more experimental research.

In Chapter 3, we discussed interleaving specifications as a way of generalizing the assumption underlying serializability that *all* steps of a transaction should appear to be executed indivisibly to *all* other transactions in a correct schedule. In particular, we discussed the following options: (i) to consider units of isolation which do not necessarily consist of *all* the steps of a transaction, but of certain subsets of the steps, and (ii) to consider *varying* units of isolation w.r.t. other transactions, i.e., some steps may constitute an indivisible unit to some transactions, but not to others. We formalized this by specifying for each pair of given transactions the indivisible units of one transaction *relative to the other*. The intuition was to think of the borderline between two consecutive indivisible units of the same transaction as a "breakpoint" at which a transaction may be interleaved with another transaction, and these breakpoints needed to be specified in advance. A similar intuition underlies transaction chopping. Indeed, if an application programmer specifying transactions has sufficient knowledge of the semantics of the application for which the transactions are written, she may be able to state which mutual interleavings or individual choppings are acceptable and which ones are not. In this sense, chopping can be considered as an approach to automate the derivation of an interleaving specification, which can basically be "read off" a chopping graph. Thus, when it comes to scheduling, it is not even necessary to develop a separate scheduler for handling interleaving specifications, but any scheduler yielding histories (for transaction pieces) in CSR will do.

8.5 Lessons Learned

The exposition in the chapter can be seen as a case study which reveals the opportunities that arise with respect to concurrency control from the semantics of a particular object model. As we have seen, there are multiple options for the important case of relational databases, ranging from exploiting predicates as they arise in query expressions or integrity constraints like functional dependencies to the systematic chopping of transactions into shorter pieces. The basic abstraction underlying all three approaches is the *predicate* or set of conditions that determines a set of tuples (those satisfying the predicate). These tuples may or may not be in the current database state, which could give rise to the phantom problem, but which is now elegantly covered by the uniform handling of predicates.

There is an obvious catch, however, to such case studies, which lies in the fact that the results obtained, though elegant and interesting, are hardly of immediate practical relevance. For example, testing the satisfiability of predicates is an NP-complete problem, as is testing of final-state serializability in the IDM transaction model. Nevertheless, there are "approximations" derived from the general scenarios we have discussed in this chapter which *are* practically relevant. For example, the ideas underlying predicate-oriented concurrency control are used in *index key locking* as will be described in the following chapter, and they are also relevant to federated database systems which will be the subject of Chapter 18.

Exercises

1. Discuss how predicate locking can be extended to disjunctive conditions such as queries of the form

```
SELECT Name FROM Emp
WHERE Position = 'Manager' OR Department = 'Research' .
```

Also discuss how join queries such as

```
SELECT Emp.Name, Emp.Department FROM Emp, Dept
WHERE Emp.Position = 'Manager' AND Dept.City = 'Toronto'
AND Emp.Department = Dept.Department
```

could be (conservatively) handled by predicate locking.

2. The sample histories we studied for IDM transactions in this chapter mostly referred to a database with one relation and one attribute. Show the following: If the database has more than one relation, the serializability of each projection of a given history onto an individual relation does *not* imply serializability of that history in general. Use the following scenario as a counterexample: Let database schema D contain relation schemata R and S, where R has attributes

A and B, and where S has attributes B and C, and consider the following transactions and history:

$$t_1 = m^R(A=1; A=2)m^S(C=5; C=6)$$
$$t_2 = m^R(A=2; A=3)m^S(C=4; C=5)$$
$$s = m_1^R(A=1; A=2)m_2^R(A=2; A=3)m_2^S(C=4; C=5)m_1^S(C=5; C=6)$$

3. Consider a restricted IDM transaction model without modify operations (i.e., a model of "ID transactions"). Show the following:

   (a) Not every modification can be simulated by deletions followed by insertions; more precisely, show that $t_1 = m(0;1) \not\approx t_2 = d(0)i(1)$ (by exhibiting a counterexample state).

   (b) Show that $s = i_1(1)d_2(1)i_2(2)d_1(2)i_3(1)i_3(2) \in \text{FSR}_{IDM} - \text{CSR}_{IDM}$.

   (c) Show that testing membership in FSR_{IDM} can be tested in polynomial time for histories of ID transactions.

4. The following approach is intended to "approximate" exact testing of membership in FSR_{IDM} by combining local and global testing in various degrees. Consider the following family of algorithms, which is based on an "exactness parameter" k determining the amount of exact testing a concrete algorithm will perform; the higher k, the more precisely FSR_{IDM} is approximated and the higher the particular algorithm's complexity. In the following description, let "coalesce(T_k, $G(s)$)" denote the graph obtained from the conflict graph $G(s)$ of a history s by collapsing all nodes whose transactions belong to $T_k \subseteq T$ into one node and by correcting the edge relation correspondingly (a node outside T_k is connected to T_k if it is connected to any node in T_k in $G(s)$).

Algorithm SR(k)
Input: A strict history s for a set T of n transactions;
Output: "yes" or "no"
Method:

```
{ for each T_k ⊆ T, |T_k| = k do
    if coalesce(T_k, G(s)) acyclic then
      if π_{T_k}(s) ∈ FSR then
        write("yes"); stop fi fi od;
  write("no")
```

}.

Notice that $s \overset{*}{\leftrightarrow} s'\pi_{T_k}(s)s''$ holds if coalesce(T_k, $G(s)$) is acyclic, where $s's''$ is a serial history for $T - T_k$ and s' only contains complete transactions. Thus, s is serializable if $\pi_{T_k}(s)$ is; therefore, the algorithm performs an exact test in Step 3 for $\pi_{T_k}(s)$.

(a) Show that the SR(k) algorithms are of polynomial time complexity for any given k.

(b) Let SR_k denote the set of histories for which algorithm SR(k) returns "yes." show the following:

(i) $\mathrm{SR}_0 = \mathrm{SR}_1 = \mathrm{CSR}_{IDM}$,

(ii) $\mathrm{SR}_k \subset \mathrm{SR}_{k+1} \subset \mathrm{FSR}_{IDM}$ for each $k \geq 1$,

(iii) for each strict and final-state serializable history s there exists an integer $k > 0$ s.t. $s \in \mathrm{SR}_k$.

Thus, the classes SR(k) of "degree-k serializable" histories form an infinite hierarchy of classes of serializable histories, and "converge" towards FSR_{IDM}.

5. Show for the IDM model of transactions:

   (a) Final-state serializability is not monotone.

   (b) Final-state serializability is not prefix-commit closed.

   (c) Conflict serializability is monotone.

   (d) Extended conflict serializability is monotone.

   (e) Both conflict and extended conflict serializability are prefix-commit closed.

6. Extend the IDM transaction model by a suitably chosen `SELECT` operation and adapt the formalism, in particular the equivalence of schedules, appropriately. `SELECT` operations read values from a relational database, and two `SELECT` operations over the same relation must yield identical values in equivalent histories. Which of the theorems on IDM transactions can be generalized to such an extended model of "IDMS" transactions?

7. Consider the transactions from Example 8.26 again. Determine whether decomposing t_{11} any further still results in a correct chopping.

8. Consider the following transactions:

$$t_1 = r_1(x)w_1(x)r_1(y)w_1(y)$$
$$t_2 = r_2(x)$$
$$t_3 = r_3(y)w_3(y)$$

   Try to decompose t_1 into three pieces s.t. the result is a correct chopping.

9. Prove Theorem 8.28.

10. Show that if a set of chopped transactions contains an *sc* cycle, any further chopping of any of the transactions will not render the graph acyclic.

11. Suppose that a *purchase* program processes the purchases made by a company by adding the value of an item purchased to the *inventory* and by subtracting the money paid for it from *cash*. The application specification requires that the value of *cash* never becomes negative, so that a transaction recording a purchase has abort if subtracting the money from *cash* would make that value negative. The structure of such a program is given by a parameterized procedure *purchase(i, p)* recording the purchase of item i for price p, whose body can be written as follows:

```
if (p > cash)    then rollback
                 else inventory[i] := inventory[i] + p;
cash := cash - p;
```

    Discuss under which conditions and to what extent chopping can be applied to this scenario.

Bibliographical and Other Remarks

Precidate locking is an old idea in concurrency control that goes back to Eswaran et al. (1976). A complexity analysis showing that testing satisfiability of predicates is NP-complete even for simple predicates (i.e., Boolean combinations of atomic comparisons) was done by Hunt and Rosenkrantz (1979). The phantom problem, which was already brought up by Eswaran et al. (1976), was systematically investigated by Bernstein et al. (1981). Predicate-oriented concurrency control techniques both locking-based and optimistic variants, were later studied by many others, including Dadam et al. (1983), Reimer (1983), and Böttcher et al. (1986). Precision locking is from Jordan et al. (1981). A prototype implementation of predicate locking, based on techniques from Dadam et al. (1983), is described by Schaad et al. (1995). Reuter et al. (1992, 1997) discuss the use of predicate-oriented concurrency control synchronizing long-lived workflows with activity-specific entry and exit predicates as workflow invariants.

The IDM transaction model for relational databases goes back to the work of Abiteboul and Vianu (1988), who define the basic notions and give a host of results. Karabeg and Vianu (1988) provide a sound and complete axiomatization for IDM transactions. Concurrency control for IDM transactions was studied in depth by Vianu and Vossen (1992); in particular, they studied the various notions and classes of histories for IDM transactions as well as the complexity results for recognizing FSR_{IDM} (for IDM as well as for ID transactions) and its restrictions. Moreover, the exploitation of FDs in this context is from that paper, and proofs for Theorems 8.8, 8.12, 8.17, 8.21, and others can be found there. Vossen (1992a) shows that dynamic aspects such as considering prefixes and taking closures of history properties can also be respected in the IDM transaction model.

Vianu and Vossen (1993) investigate another foundation for concurrency control in the context of IDM transactions by describing an alternative notion of history correctness for histories that turns out to be orthogonal to serializability. While serializability is based on the assumption that *every* serial execution of given transactions is acceptable to all users, *goal correctness* tries to reflect the intended changes of a transaction to a database and whether these are compatible among the transactions that are ready for execution. Goal correctness is shown to be orthogonal to (final-state) serializability, but is easier to test. On the other hand, it has its drawbacks when it comes to dynamic scheduling, essentially as it requires complete knowledge about transactions, since otherwise nothing can be said about their goals. Laurent et al. (1995) investigate a different transaction model for relational databases in which goal-correctness turns out to be subsumed by serializability.

Transaction chopping is from Shasha (1992) as well as Shasha et al. (1992, 1995); a proof of Theorem 8.28 can be found in these references. Recently, Bernstein et al. (1998, 1999) as well as Ammann et al. (1997) have continued the research that studies transaction interleavings as well as choppings and proposed a technique called *assertional concurrency control*.