Chapter 2

Computational Models

"Between theory and practice, some talk as they were two. Between theory and practice, both can be gained." (Bhagavid-gita 5:4)

2.1 Goal and Overview of the Chapter

In the previous chapter we have made it clear why transactions are important, and what is relevant when we start discussing them from a technical point of view. In this chapter, we will go even further and introduce two precise computational models for transactional servers around which all our discussions in subsequent chapters of this book will center. In particular, the theory and algorithms of concurrency control and of recovery that we will present are based on these two models, from which most of our correctness and efficiency arguments can be carried over to real implementations and applications. Especially for correctness, it will be necessary to look at transactions even from a formal point of view, and we will see here to what extent this is possible, and what natural limits exist.

The computational models we will introduce below follow a common "recipe" for devising transaction models, which we first describe in Section 2.2. Then we present in detail a simple model — coined the *"page model"* — for ease of concise explanation in Section 2.3, and a more complex one — coined the *"object model"* — to capture advanced issues in Section 2.4. In Parts II and III, these models form the basis for a wide development of concurrency control as well as recovery theory and algorithms. Thus the material in this chapter together with that of the previous chapter sets the stage for laying out a more detailed roadmap of the book's contents and organization, given in Section 2.5, which ends the introductory part of the text.

2.2 Ingredients

Computational models for transactional servers can be made precise in a variety of ways, depending on the level of abstraction chosen, the programming environment, or the application at hand. However, the way in which such a model is devised is not completely arbitrary, but follows a kind of "methodology" that reflects how to specify the essential and relevant ingredients. The common approach by which our computational models are developed and will later be used is characterized by the following five-step "program":

1. First, *elementary operations* on data objects are defined, that is, operations that can be assumed to be indivisible so that each individual operation by itself appears atomic and isolated from other operations. (Note that we are not yet considering the embedding of such operations in the context of a transaction.) This first step leaves some leeway in the definition of what a data object is and what kind of elementary operations we consider, and this will later lead to the distinction between the page model and the object model.

2. Next, *transactions*, as executions of transaction programs, can be modeled as sequences or partial orders of such operations, with the intuitive interpretation that such a collection is to be executed in conjunction in order to guarantee its ACID properties.

3. When several transactions are "shuffled" into a large collection of operations (in such a way that the individual transactions can still be identified), we obtain *schedules* or *histories* as an abstract notion of concurrent (or parallel) executions.

4. From the set of all syntactically correct schedules we must identify those that can be considered "correct" in the sense of the ACID guarantees.

5. Finally, we need *algorithms* or *protocols* to create correct schedules in an online manner such that operations are scheduled for execution dynamically as they are submitted by the application programs.

These five steps are relevant for vastly every transaction model, in particular when such a model is developed all the way from basic operations to protocols. In the remainder of this chapter we will apply the steps to the two computational models which we want to study in this book. The *page model* (also known as the *read/write model*) is a simple model that can be nicely motivated from looking at the way data pages are accessed (i.e., read or written) at the storage layer of a database system. As will be seen shortly, this model is simple enough to capture the essence of the concurrency control and recovery problem in a concise and elegant way, yet it is general enough to describe many (but not all) important system implementation issues. The major limitation of the page model is that it captures *no* semantics of the data access operations, given the simple low-level nature of page reads and writes.

A richer model would therefore consider higher-level operations, for example, those of the access layer or the query processing layer of a database system. From here it is only a small step to consider even a limited form of application semantics by taking into account an application program's method invocations on "business objects", i.e., instances of abstract data types (ADTs), that reside in the data server or the application server (see the discussion of system architectures in Section 1.3). All these considerations give rise to our second computational model, the *object model*, in which transactions consist of ADT operations that may themselves invoke operations on other objects, all of which would eventually result in read or write operations on objects of type "page."

In summary, the two computational models we are about to introduce differ in their definitions of data objects. For the page model, these are *pages* with read and write operations; for the object model, these are general *objects* with an encapsulated set of operations applicable to them.

2.3 The Page Model

The simple page model to be defined next is motivated by the observation that all higher-level operations on data are eventually mapped into read and write operations on pages. We have seen this in our brief review of database system concepts in the previous chapter, and similar mappings onto pages are used by all other forms of data servers (e.g., mail or document servers). Thus, to study the effects of concurrent executions, in the sense of interleaving the operations of different transactions, it is, in principle, sufficient to inspect the interleavings of the resulting page operations. In doing so, each page read or write is assumed to be an indivisible operation, regardless of whether it takes place in a page cache in memory or on disk. It is clear that we need to revise this assumption once we start discussing recovery issues that arise after server failures, but the "indivisibility axiom" for page operations will take us a fairly long way into recovery considerations as well.

In general, we emphasize once again that the restriction to read/write page operations represents a strong form of abstraction (this is exactly why we later need the richer object model). However, as will be seen in the sequel, the abstraction from higher-level data operations, such as SQL commands or method invocations on business objects, down to the view that a resulting transaction consists of reads

and writes *only* is an interesting one and suffices for many purposes. In fact, a comprehensive theory of concurrency control and of recovery can be built on it, which is directly applicable to practical systems, albeit with some performance limitations.

We now start going through the 5-step procedure discussed in Section 2.2. Formally, a data server is assumed to contain a (finite) set $D = \{x, y, z, \ldots\}$ of (indivisible and disjoint) items with indivisible read and write operations. The reader may think of these data items as *pages*, as these are the units onto which all higher-level operations are eventually mapped and for which reads and writes can indeed be regarded as indivisible. As an abstract model, this page model is not necessarily limited to the real notion of pages and rather provides very general insights into concurrency problems and their solutions. Nonetheless, page-oriented concurrency control and recovery at the storage layer of a database system or other type of server is the major application of the page model and the results derived from it. Throughout the book, we will denote data items by small letters (often taken from the end of the alphabet), where indices will be used if necessary (e.g., x_1, y_4).

Before we define transactions in this model in general, let us look at a simplified version first which we will use to introduce the (somewhat artificial) distinction between the *syntax* and the *semantics* of a transaction. The simplified version considers transactions as *total* orders of steps, while the general model will allow *partial* orders. Since we are not going to make intensive use of transaction semantics in what follows, we briefly explain that issue for total orders here.

"syntax": totally ordered transaction

Let us consider a *transaction* t (in the page model) to be a (finite) sequence of steps (actions) of the form $r(x)$ or $w(x)$, written

$$t = p_1 \ldots p_n,$$

where $n < \infty$, $p_i \in \{r(x), w(x)\}$ for $1 \leq i \leq n$, and $x \in D$. We use r as an abbreviation for "read" and w as an abbreviation for "write". Thus, we abstract from the details of a transaction as a program execution; instead, we concentrate on the sequence of read and write operations that results from the execution.

Next we look at how we could define some form of semantics for individual transactions. Each step occurring in a transaction can be uniquely identified so that two distinct transactions will not have steps in common (of course, the same step *types* can occur in more than one transaction). Within a single transaction, we often denote steps in the form p_j, that is, with a step number as a subscript (so p_j is the j-th step of the transaction.) In the presence of multiple transactions, we add a unique transaction number to each transaction, and also use this number as an additional subscript for the steps of the transactions. Then p_{ij} denotes the j-th step of transaction i. Sometimes, a given context does not require an explicit numbering of the steps; in such cases we may simply write p_i for a step of transaction i.

As we have seen, associated with each step is both an action (read or write) and a data item from D. Thus we use the following terminology when we consider only a single transaction:

$$p_j = r(x) \quad : \text{ step } j \text{ reads data item } x$$
$$p_j = w(x) \quad : \text{ step } j \text{ writes data item } x$$

In our discussion on semantics following next, there is always just a single transaction under consideration, so that single-level indexing suffices.

According to the above settings, a transaction is a purely syntactic entity whose semantics or interpretation is unknown. If we had more information on the semantics of the program that launches a transaction and the intended state transformation of the underlying data, this knowledge could be used to interpret a transaction and associate a formal semantics with it. In the absence of such information, however, the best that can be done is a "syntactic" interpretation of the steps of a transaction that is as general as possible:

- In case $p_j = r(x)$, i.e., the j-th step of a given transaction is a read step, the current value of x is assigned to a local variable v_j:

  $$v_j := x$$

- In case $p_j = w(x)$, i.e., the j-th step of a given transaction is a write step, a possibly new value, computed by the respective program, is written into x.

  "semantics": interpretation of a transaction

  Each value written by a transaction t potentially depends on the values of *all* data items that t has previously read, which is formally expressed as follows:

  $$x := f_j(v_{j_1}, \ldots, v_{j_k})$$

  (x is the return value of an arbitrary but unknown function f_j) s.t.

  $$\{j_1, \ldots, j_k\} = \{j_r \mid p_{j_r} \text{ is a read step } \wedge j_r < j\}$$

  (All values v_{j_r}, $1 \le r \le k$, that were read prior to the j-th step of t are used as parameters in function f_j.)

As an example, consider the following transaction:

$$t = r(x)r(y)r(z)w(u)w(x)$$

Here we have $p_1 = r(x)$, $p_2 = r(y)$, $p_3 = r(z)$, $p_4 = w(u)$, and $p_5 = w(x)$. The first three steps assign values (those of data items x, y, and z) to variables v_1, v_2, and v_3, respectively. The values of u and x written by the last two steps depend on these variables, in principle as follows:

$$u = f_4(v_1, v_2, v_3)$$
$$x = f_5(v_1, v_2, v_3)$$

We mention here that the view of a transaction as a *sequence* of steps is a simplification, but not an essential part of the model. Indeed, from a conceptual point of view we do not even have to order *all* steps of a transaction in a straight-line fashion, and it will often be the case that the specific order in which two or more steps are executed does not matter, as long as the ACID principle applies. Also, we should strive to capture a parallelized transaction execution (i.e., on a multiprocessor) as well. Therefore, we can relax the *total ordering* requirement to the steps of a transaction to a *partial ordering* and generally modify the previous definition as shown below. For the sake of completeness, we define partial orders first:

partial order

Definition 2.1 (Partial Order)

Let A be an arbitrary set. A relation $R \subseteq A \times A$ is a *partial order* on A if the following conditions hold for all elements $a, b, c \in A$:

(i) $(a, a) \in R$ (reflexivity)

(ii) $(a, b) \in R \wedge (b, a) \in R \quad \Rightarrow \quad a = b$ (anti-symmetry)

(iii) $(a, b) \in R \wedge (b, c) \in R \quad \Rightarrow \quad (a, c) \in R$ (transitivity)

As is well-known from mathematics, any relation R over a finite set A (and hence a partial order) can be visualized through a graph in which the elements of A form the nodes, and in which a pair $(a, b) \in R$ is represented by a directed edge of the form $a \rightarrow b$. Examples of partial orders appear in Exercises 1 and 2 of this chapter. Note that a *total* order R of A is a partial order with the additional requirement that for any two distinct $a, b \in A$, either $(a, b) \in R$ or $(b, a) \in R$.

We now exploit the notion of a partial order in the context of transactions.

page-model (read/write) transaction

Definition 2.2 (Page-Model Transaction)

A *transaction* t is a partial order of steps (actions) of the form $r(x)$ or $w(x)$, where $x \in D$ and reads and writes as well as multiple writes applied to the same data item are ordered. More formally, a transaction is a pair

$$t = (\text{op}, <),$$

where op is a finite set of steps of the form $r(x)$ or $w(x)$, $x \in D$, and $< \subseteq \text{op} \times \text{op}$ is a partial order on set op for which the following holds: If $\{p, q\} \subseteq \text{op}$ s.t. p and q both access the same data item and at least one of them is a write step, then $p < q \vee q < p$.

So in the partial ordering of a transaction's steps, we disallow that a read and write operation on the same data item or two write operations on the same data item are unordered. Instead, for each of these two types of action pairs we require an ordering. The reason for this restriction is fairly obvious: with unordered steps of these kinds it would be impossible to tell the exact effect on the respective data item. For example, if we left a read and a write unordered, the value that is read would remain inherently ambiguous; it could be the one before the write or the new value that is written. So the constraints in the above definition serve to ensure an unambiguous interpretation.

In what follows, we will generally try to stick to the latter definition (partial orders), and it should be clear that, with "conflicting" steps inside a transaction being ordered, the "semantics" we have introduced for totally ordered transactions carries over to partially ordered ones as well. However, partial orders require considerable effort in some of our formal notation to be introduced in the next chapter (e.g., the so-called shuffle product, which "shuffles" the steps of multiple transactions into a concurrent execution). Therefore, we will simplify some of the discussion of correctness criteria in Chapter 3 (and elsewhere) by just sticking to total orders. Once we have arrived at a standard notion of correctness that we will use for the remaining exposition, we will generalize our notation to partial orders of steps.

Although we will in the sequel consider read/write transactions as syntactic entities only (with the pseudo-semantics just sketched), it may be considered as a certain advantage of this model and its theory to be discussed shortly that this theory can be developed in the absence of semantic information and hence can be used for *every possible interpretation* of the transactions. Thus the read/write page model is fairly general despite its simple structure.

The page model as described above allows a transaction to read or write the

same data item more than once, as it is the case in the example:

$$t = r(x)w(x)r(y)r(x)w(x).$$

Here t reads and writes x twice, although it is reasonable to assume that the value of x remains available, after having been read the first time, in the local variables of the underlying program for as long as it is needed by t, and that only the last write step determines the final value of x produced by this transaction. To exclude "redundancies" of this kind, we henceforth make the following assumptions:

1. In each transaction each data item is read or written at most once, and

2. no data item is read (again) after it has been written.

Notice that the latter condition does not exclude the possibility of "blind writes," which is a write step on a data item that is not preceded by a read of that data item.

2.4 The Object Model

This section introduces the *object model* as an alternative to or generalization of the page model. The reason why we explicitly separate this model from the page model, although the latter is implicitly included here, is that many important foundational issues and also basic algorithms and implementation techniques can be more easily understood in the simpler page model first. The object model provides a framework for representing method invocations, or *operations* for short, on *arbitrary kinds of objects*. This will later allow us to exploit some "semantic" properties of the invoked operations for the sake of improved performance. In addition, the object model is also able to capture cases where an operation on an object invokes other operations on the same or other objects. Indeed, often the implementation of an object and its operations requires calling operations of some lower-level types of objects.

For example, operations at the access layer of a database system, such as index searches, need to invoke page-oriented operations at the storage layer underneath. Similar invocation hierarchies may exist among a collection of business objects that are made available as abstract-data-type (ADT) instances within a data server or an application server, e.g., a "bank account" object type along with operations like deposit, withdraw, get_balance, get_history, compute_interests, etc. Figures 2.1 and 2.2 depict two examples of transaction executions against such object-model scenarios.

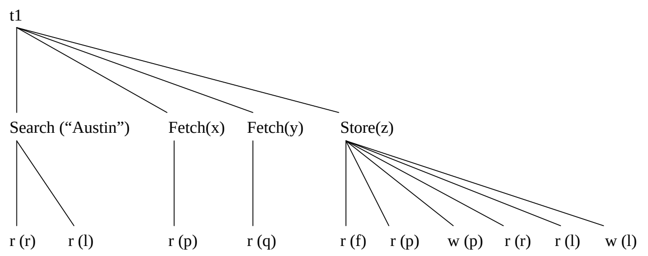

Figure 2.1: Sample scenario for the object model: database system internal layers.

database system layers example

Figure 2.1 shows a transaction, labeled t_1, which performs, during its execution, (i) an SQL Select command to retrieve, say, all records from a database of persons who live in Austin (i.e., whose City attribute has the value "Austin") and, after inspecting the result set, (ii) an SQL command to insert a record for a new person who happens to live in Austin, too. Since SQL commands are translated into operations at the query processing layer already at compile-time (see Section 1.5 for a short primer on such database system issues), the operations invoked at run-time are those of the query processing layer. Because of the extremely simple nature of the example's SQL commands, the resulting operations are also very simple, and already partly resemble operations at the access layer (i.e., operations of the layer onto which the query processing layer translates its operations). We assume that the Select command is executed by first issueing a "Search" operation on an index that returns the RIDs (i.e., addresses) of the result records. Next, these records, which we refer to as x and y in the figure, are fetched by dereferencing their RIDs. The "Search" operation in turn invokes operations at the underlying storage layer, that is, read and write operations on pages. Here we assume that first the root of a B^+-tree is read, labeled as page r in the figure, and then we are pointed to a leaf page, labeled l, that contains the relevant RID list for the search key "Austin". (So, for simplicity, we assume that the B^+-tree has height two.) The subsequent "Fetch" operations to access the two result records x and y by their RIDs require only one page access each to pages p and q, respectively. Finally, the SQL Insert command is executed at the query processing layer as a "Store" operation, storing

the new record and also maintaining the index on the City attribute (assuming no other indexes are defined for the underlying table). This involves first reading a storage-layer metadata page, labeled f, that holds free space information in order to find a page p with sufficient empty space. Then that page is read and subsequently written after the new record z has been placed in the page. The last thing to do on behalf of the "Store" operation is to add the RID of the new record z to the RID list of the key "Austin" in the City index; this requires reading the B^+-tree root page r, reading the proper leaf page l, and finally writing page l after the addition of the new RID.

In Figure 2.1, this entire execution is represented in a graphical, compact form, by connecting the calling operation and the called operation with an arc when operations invoke other operations. As a convention, the caller is always placed closer to the top of the picture than the callee. Furthermore, we represent the order in which operations are invoked by placing them in "chronological" order from left to right. When giving a formal definition shortly, this ordering convention needs to be revised a bit, but the above statement reflects the essence.

business object example

The same approach to representing transaction executions in an object model can also be applied to collections of business objects as shown in the second example in Figure 2.2. The example contains a transaction, labeled t_2, that withdraws money from one bank account x by invoking a "Withdraw" operation and subsequently deposits money in another account y by means of the operation "Deposit". Here x and y are instances of the object type (or ADT) "Account" which provides among its interface operations the mentioned "Withdraw" and "Deposit" methods. The implementation of these operations requires calling other operations. Assuming that the implementation is on top of a database system (e.g., inside a database server that supports ADTs), most of it is straightforward and could be directly translated into a sequence of "Search", "Fetch", and "Modify" operation invocations at the query processing layer. (The code for the ADT implementation would be precompiled and accessible by the database server, too.) For the underlying storage layer, we assume that the records can be searched by an index on their "AccountID" attribute and that no other indexes are involved in the example. Thus the resulting page operations are straightforward and very similar to those of Figure 2.1; to help intuition, we have even kept the labels of the previous example, e.g., r for a B^+-tree root page. To distinguish the account objects x and y from the corresponding stored records on which the "Fetch" operations are invoked, we label the records as $\hat{x}$ and $\hat{y}$, respectively.

A particularity of the example in Figure 2.2 is that the "Withdraw" operation in-

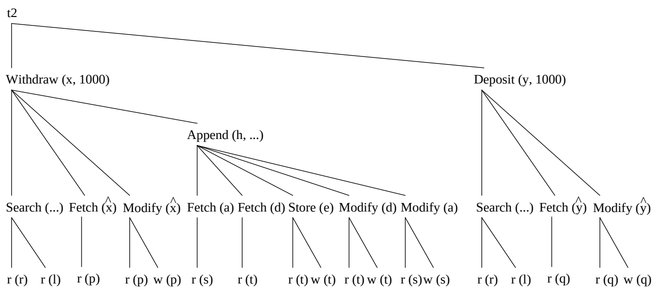

Figure 2.2: Sample scenario for the object model: business objects.

vokes also another business-object method, labeled "Append", in addition to the operations of the database system's query processing layer. This "Append" operation belongs to an object h of type "History", which can be viewed as a specialized way of capturing the recent cash-flow history of the bank account. For example, one could keep the last twenty cash flows, or all cash flows of the last thirty days, in a small history queue and make this information accessible by additional "Get_History" or "Browse_History" methods. The specific semantics of purging expired entries from the history queue would be built into the object, and this is why it makes sense to encapsulate the history data as an ADT object. In the transaction execution of Figure 2.2, the invoked "Append" operation is assumed to fetch a header record a (on page s) that contains pointers to the head and tail of the history queue, assuming that the implementation uses a doubly linked list of records. Adding an entry at the tail of the queue would then find the page t with the currently most recent entry, record d. If this page has sufficient empty space for the new record, then this is where the new entry could be placed as record e. Storing the new entry and changing the linked list to connect the previously last entry with the new one results in further write operations on page t. Finally, the queue's header record a in page s needs to be updated to point to the new tail.

The two examples and especially their graphical depiction already suggest a definition that essentially views transactions as *trees* with the invoked operations as tree nodes. However, to ensure that such trees are sufficiently self-contained in that

they reflect all relevant aspects of the transaction execution and allow us to reason rigorously about concurrency between transactions, we require that the leaf nodes of a transaction tree be elementary operations in the sense of the page model. So all tree leaves must be read and write operations on pages, or some other notion of abstract operations for which we can safely assume that each operation is indivisible. At this point, we only know about page operations having this property, at least with regard to concurrency effects. Thus, for the time being, the reader can always associate page operations with the leaves of a transaction tree. If all or some leaf nodes of a given transaction tree do not correspond to page operations, all we have to do is to further expand the operations at these leaves by making their invoked operations at lower implementation levels explicit until we end up with elementary read or write operations on pages. Note that this kind of expansion is always possible as all forms of data servers eventually store their (permanent) data items in pages (also known as blocks). Further note that the expansion is a dynamic one that tracks all operations that are invoked during the execution of a higher-level operation, as opposed to a hierarchy derived from the static structure of the operation's code.

transactions as trees

Analogously to the introduction of the page model, we still have to make some amendments to the above considerations, to capture the ordering of operations within a transaction. A simple definition would require a sequential ordering of all leaf nodes, matching their left-to-right order in the example figures. However, as in the page model, there are good reasons to require only a partial ordering among the leaf-node operations. An additional constraint that arises from the requirement that leaf nodes be elementary page-model operations is to enforce the ordering of operation pairs where one or both of the two operations is a write. As in the page model, the argument for this constraint is to ensure an unambiguous interpretation of the transaction.

object-model transaction

We are now ready to give the following formal definition of an object-model transaction.

Definition 2.3 (Object-Model Transaction)

A *transaction* t is a (finite) tree of labeled nodes with

- the transaction identifier as the label of the root node,
- the names and parameters of invoked operations as labels of inner (i.e., non-leaf, non-root) nodes, and
- page-model read/write operations as labels of leaf nodes,

along with a partial order "$<$" on the leaf nodes such that for all leaf-node operations p and q with p of the form $w(x)$ and q of the form $r(x)$ or $w(x)$ or vice versa, we have $p < q \vee q < p$.

Given this tree-based definition, we will from now on use standard terminology for trees to refer to, for example, children, descendants, subtrees, ancestors, siblings, etc. Note that the definition allows for arbitrary types of operations as inner nodes, and that it contains Definition 2.2 as a special case, i.e., the object model generalizes the page model. These can be operations on an ADT object or some less stringent, but explicitly specified, form of operational interface (e.g., the interface operations of a database system's query processing layer). Further note that transaction trees do not have to be balanced in the sense that all leaf nodes have the same distance from the root. In Figure 2.2, for example, the leaf-node descendants under the "Append" operation are further away from the root than those of the "Modify" operation, although these two inner nodes are at the same tree level as seen from the root. For clarity, however, we will usually attempt to place operations that belong to the same object type(s) or interface within one level of the graphical picture of a transaction tree. This is why we have placed, in Figure 2.2, all operations of the query processing layer adjacently and have rather decided to let the "Append" operation stick out. When we later apply the object model to reason about concurrent executions, we will often deal with cases of perfectly balanced transaction trees anyway. The class of transaction trees where all leaf nodes have the same distance from the root is known by the name *multilevel transactions*, an important special case of our model.

inner-node ordering

A specific point in our formal definition that could, perhaps, strike the reader as an accidental omission is that a partial order is defined only for the leaf nodes of the tree. So one may wonder why no ordering has been defined for inner nodes. The answer is that such an ordering is already implicitly defined, and we wanted to keep the definition as concise as possible and avoid producing an overspecification. Namely, for two inner-node operations a and b, we say that a precedes b in the execution order if all leaf-node descendants of a precede all leaf-node descendants of b in the partial order $<$ of the leaf nodes. In other words, we consider a as

preceding b only if a terminates its execution before b begins; this is what ensures the required ordering between the two sets of leaf-node descendants. Otherwise, we would consider a and b as concurrent operations; and this is a feasible viewpoint as none of them is assumed to be indivisible. If we want to find out their resulting effects on the permanently stored data, we can resort to checking the interleaving of their children and further descendants until we arrive at the leaf nodes whose ordering ensures an unambiguous interpretation.

To conclude this section, we give a brief outlook on how we will later use the introduced object model in laying a foundation for and reasoning about advanced concurrency control algorithms. Once we consider multiple transactions that execute in a concurrent or parallel way, all we have to do is to form the union of the involved transaction trees, i.e., a forest of operations, and inspect the partial execution order of leaf node operations (and the implicitly derived order for higher-level operations). The difference to the single-transaction case is that the partial order will then be defined on the union of the leaf nodes of all involved trees. Because of the derivation of the inner-node ordering from the leaf-node order, this approach will allow us to study not only the interleavings at the leaf-node level but also concurrency or sequential orderings among higher-level operations. This capability to reason about concurrency at different tree levels is indeed needed to properly exploit the object model for the sake of better performance. The key to such performance enhancements lies in taking into consideration semantic properties of the higher-level, inner-node operations.

By far the most important one of such properties is the commutativity or non-commutativity of two operations. For example, two "Deposit" operations on the same bank account are commutative in that both of the two possible sequential execution orders lead to identical results in terms of the resulting state of the data and the return parameters of the invoked operations, assuming that a "Deposit" operation does not return the current balance of the account. Two "Withdraw" operations on the same account do, however, not generally commute because their ordering may be relevant if only the first withdrawal succeeds and the second one is refused due to overdraft protection; in this situation the two possible sequential orderings would result in different return values (i.e., indicators of the withdrawal's success) and could also lead to different final states of the account balance. The point about commutativity or non-commutativity is that these considerations hold for pairs of operations from the same as well as different transactions, so that we can apply commutativity-based reasoning also to concurrent executions of multiple transactions. It will be seen later that commutative operations allow us to tolerate interleaved access patterns that would have to be disallowed if we only knew that the operations read and write the same data item.

2.5 Roadmap of the Book

In this chapter and the previous one, we have set the stage for a discussion of numerous issues related to transactions, and we are now ready to lay out a roadmap of this book. Thus, in this section we explain how the book is organized to provide guidance to readers who are mostly interested in specific topics.

Following this first part on background and motivation, the core of the book is divided into three main parts. As discussed in the previous sections, the two subsystems of a data server that are central to a transactional information system are the *concurrency control* and the *recovery* components. The second and third parts of the book are devoted to these two components, mostly from a single server's point of view. The book's fourth part then broadens the view and addresses the additional issues of *distributed transaction coordination* across multiple servers, based on the foundation laid by the two parts on single-server techniques. Together these three main parts provide a comprehensive in-depth understanding of how transactional information systems work, covering formal models, practical algorithms, and performance considerations for the various components and also the interdependencies among components. After all this technical material, the book is finally concluded with an outlook on future technology and application trends. In the following we briefly outline the contents of the chapters that constitute the three main parts.

Part II: concurrency control

Part II of the book, on concurrency control, starts out in *Chapter 3* with a thorough look at what transactional isolation means. This chapter introduces formal correctness criteria for the concurrent execution of transactions, most importantly the notion of *serializability*, and thus defines precisely the requirements that we pose on a concurrency control algorithm. It is based on the simplifying assumption that transactions are executed in a failure-free environment, so that there is not (yet) a need to handle transaction aborts. For faster progress on the algorithmic material, this chapter is based on the simpler page model and postpones the analogous discussion for the richer object model until a later chapter. *Chapter 4* then presents the most important classes of concurrency control algorithms and shows their correctness in terms of satisfying the serializability criterion of the previous chapter. *Chapter 5* extends the algorithmic discussion to a class of algorithms known as multi-version concurrency control, which is more powerful in that it enhances the concurrency among transactions but is also more complex and requires more system resources.

Both of the two algorithmic Chapters 4 and 5 are based on the page model. The thorough discussion of page-oriented concurrency control prepares the reader for the next step, where transactions are reconsidered within the object model. In fact, we will see that many considerations of the page model, both of correctness criteria and algorithms, are extremely useful for object-based concurrency control techniques as well. *Chapter 6* is the analog of Chapter 3, discussing formal correctness criteria that

can be seen as generalizations of those of the page model. *Chapter 7* then presents object-based concurrency control algorithms, largely building on basic techniques already introduced for the page model. Chapter 7 also explains in great detail the major advantage of object-based concurrency control, namely, the enhanced concurrency and resulting performance gains that one obtains from considering the semantics of the transactions' invoked operations. The presented techniques up to this point are as general as possible and applicable to all kinds of objects and object methods. The following two chapters specialize the object-based approach to two specific settings that are of high practical importance, deriving specific solutions that offer additional performance advantages. *Chapter 8* develops techniques for various forms of high-level methods provided by the relational data model, particularly predicate-oriented retrieval and update operations. *Chapter 9* is further specialized to the interface and internal operations provided by index structures, most notably, B^+-trees and similar structures. Although these considerations may appear to be of extremely special nature, they are among the most important ones for the efficiency of a database system. The entire part on concurrency control is concluded in *Chapter 10* with a closer look into implementation issues and various pragmatic concerns such as tuning options.

Part III: recovery

Part III of the book, on recovery, is organized by addressing different classes of failures, mostly in increasing order of potential damage and difficulty to handle. *Chapter 11* considers transaction failures, where only a single process or thread behaves abnormally (at the client or the server) and thus at most one active transaction is affected by the failure. The major problem to be addressed in this case is that such an abnormal, to-be-aborted transaction does not affect any of the concurrently executing transactions that should make regular progress. Chapter 11 provides the solutions to this problem, in terms of defining correctness criteria and developing practical algorithms, for both the page model and the object model. *Chapter 12* then considers a more severe class of failures, namely, “crashes” of the entire server caused, for example, by failures of the underlying database-system or operating-system software or by transient hardware failures. This case, which requires “crash recovery” actions for all in-progress transactions and the server's data, is the most challenging as far as efficient algorithms are concerned. Therefore, after defining the appropriate notion of correctness in Chapter 12, we first limit the discussion to the page model, and develop detailed algorithms for page-oriented crash recovery in *Chapter 13*. *Chapter 14* then extends our algorithmic and performance considerations to the object model. Similar to the concurrency control part, we can reuse many basic techniques from the page model here, but still need a number of additional steps towards a practically viable solution. Also, a number of specific settings, for example, the recovery of index structures, require an additional *Chapter 15* on special issues of crash recovery. At the next stage, *Chapter 16* considers failure cases where permanent data becomes corrupted, for example, by a disk failure, and presents "media recovery" techniques for guarding the data and its consistency against such failures by means of appropriate redundancy. These techniques build directly upon the algorithms for crash recovery, and will also be extended to provide a form of disaster recovery for cases such as fire or flood that may affect entire buildings but leave remote computer sites intact. The focus all these Chapters 11 through 16 is to ensure that the server's data remains consistent, but they disregard the possible inconvenience of manually restarting the applications after a failure. *Chapter 17* addresses this "missing link" towards zero-outage, highly available information systems in that it presents advanced techniques for application recovery. This final stage of recovery methods aims to mask all sorts of failures to the human users and requires additional considerations to recover the state of application processes and communication sessions.

Part IV: coordination of distributed transactions

The techniques derived in Parts II and III allow us to build and operate a single server with transactional guarantees. With multiple servers in a distributed or federated system, an obvious thing to do is to simply instantiate the single-server concurrency-control and recovery components in each of the servers. Indeed, we will see in *Part IV*, on distributed transaction coordination, that the single-server techniques still do most of the job for distributed transactions as well. However, we do need some additional techniques to coordinate and possibly reconcile the effects and outcome of distributed transactions on the involved servers. The theoretical foundation and algorithmic solutions for the required kind of coordination are presented in *Chapter 18* for the concurrency control aspects, establishing a notion of global serializability based on the local serializability and other properties of the executions in the various servers. In *Chapter 19* we will then discuss the extensions needed for the recovery of one or multiple servers after various kinds of failures in a distributed environment. The key addition to the local recovery algorithms is the introduction of a distributed commit protocol to guarantee global atomicity.

2.6 Lessons Learned

This book is largely about algorithms for the two key building blocks of transactional information systems, concurrency control and recovery. The challenge lies in the *performance* of concurrency control and recovery components: providing high concurrency, high throughput and fast response times, and ensuring fast recovery from failures and thus high availability with low overhead during normal operation. To address these issues in a systematic manner that accompanies algorithms with correctness arguments and efficiency considerations, we have introduced two computational models: the page model as the most simple basis, and the richer, but

also more complex object model. These models allow us to develop a theory and the truly fundamental concepts and techniques of our subject in an abstract and concise manner, without getting lost in the plethora of details of real-life systems. However, following Einstein's famous quote that "nothing is as practical as a good theory", we will also exploit our abstract basis for deriving concrete algorithms and practically viable, systems-oriented solutions.

Exercises

1. Consider a finite alphabet Σ and the set Σ^* of all words over Σ. For $v, w \in \Sigma^*$ define the *prefix order* "$\leq$" by $v \leq w$ if v is a prefix of w. Show that "$\leq$" is a partial order on Σ^*.

2. Consider an arbitrary (finite) set M as well as its powerset $\mathcal{P}(M)$, i.e., the set of all subsets of M. Show that $subs \subseteq \mathcal{P}(M) \times \mathcal{P}(M)$, defined by $(M_1, M_2) \in subs$ if $M_1 \subseteq M_2$, is a partial order on $\mathcal{P}(M)$.

3. Consider the following read/write page-model transaction:

$$t = r(x)r(y)r(z)w(u)w(x)r(v)w(y)w(v)$$

   Describe the interpretation of t from an informal as well as from a formal point of view.

4. Redo Exercise 3 for

$$t = r(x)r(y)r(z)r(u)r(v).$$

5. Redo Exercise 3 for

$$t = w(x)w(y)w(z)w(u)w(v).$$

6. In the literature, the following variations of the read/write page model can be found:

   - "No Blind Writes" model:
     Each transaction writing a data item x must have read x previously.
   - "Two–Step" model:
     All read steps of a transaction occur prior to all write steps.

- "Action" model:
  Each step of a transaction represents an atomic execution of a read step followed by a write step of the same data item; thus, distinguishing reads from writes is no longer necessary.

Discuss the interpretation of a transaction along the lines of what has been introduced in this chapter for each of these models. What, if any, are the differences in interpretation to the standard read-write model?

Bibliographical and Other Remarks

The description of the "methodology" for devising a transaction model as sketched in Section 2.2 essentially follows Vossen (1995). The major credit for the (read-write) page model of transactions belongs to the 1998 Turing-Award winner Jim Gray; see, especially, Gray (1978, 1981) and Eswaran et al. (1976). Closely related notions of atomic actions have been discussed by other authors at around the same time, including Lampson (1981) and Lomet (1977). The page-model transaction concept became the subject of intensive theoretical studies, in particular in the work of Christos Papadimitriou and, independently, Philip Bernstein et al. around 1980; see, for example, Papadimitriou (1979) and Papadimitriou (1986) as well as Bernstein et al. (1979) and Bernstein et al. (1987). The object model of transactions, on the other hand, has been around for almost two decades, albeit in a rather vague, implicit form, going back to the work of Bjork and Davies (1973) on "spheres of control", which could be viewed as a form of nested method invocations and thus invocation trees. The first work that made the object model explicit and gave formal definitions has been by Beeri, Bernstein, and Goodman (1983, 1989). Parallel work on the special case of multilevel transactions has been done by Moss et al. (1986) and Weikum (1984, 1986, 1991).