The Page Model

There are two transaction execution models: (a) Page model and (b) Object model. We will discuss the page model in this course. The simple page model is motivated by the observation that all higher-level operations on data are eventually mapped into read (r) and write (w) operations on pages. A page here is an abstraction of a relation, a file, a record, etc. 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 interleaving of the resulting page operations. In doing so, each read or write on a page is assumed to be an indivisible operation, regardless of whether it takes place in a page cache in memory or on disk. In general, the restriction to read/write operations represents a strong form of abstraction 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.

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. You 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. 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. The simplified page model version considers transactions as total orders of steps, while the general model will allow pa1tial 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.

Let us consider a transaction t (in the page model) to be a (finite) sequence of *r(x) or w(x) or both*, written as $t = p_1, \ldots, p_n$ where $n < \infty$, $p_i \in \{r(x), w(x)\}$ for $1 \le i \le n$, and $x \in D$. Thus, we abstract from the details of a transaction as a program execution, and concentrate only on the sequence of read and write operations that result 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 *t*.) 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. For example, 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*. Associated with each step is both an action (*r* or *w*) and a data item from *D*. Thus, we use the following terminology when we consider only a single transaction:

$p_j = r(x)$; step *j* reads data item *x* and $p_j = w(x)$; step *j* writes data item *x*

In our discussion on semantics that follows, there is always only 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 $p_j = r(x)$, in which 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), in which the jth step of a given transaction is a write step, a possibly new value, computed by the respective program, is written into *x*. 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:

  $x := f_j (v_{j1}, ..., v_{jk})$ (*x* is the return value of an arbitrary but unknown function f_j) such that:

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

(All values v_{jr}, $1 \leq r \leq k$, 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:

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

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.

Definition: 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:

1. $(a, a) \in R$ *(reflexivity)*
2. $(a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$ *(antisymmetry)*
3. $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ *(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 $a \rightarrow b$. 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.

DEFINITION: Page Model Transaction

A transaction t is a partial order of operations 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

$$t = (op, <)$$

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

Therefore, 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 as we will see. 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, 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 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:

- in each transaction each data item is read or written at most once,
- no data item is read (again) after it has been written.

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

Source book: Transactional Information Systems, Weikum and Vossen. Morgan Kaufmann, 2002.