Multi-Level Concurrency Control of a Database System

83-60-4

Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

Multi-Level Concurrency Control of a Database System

Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

November 1983

This material is based upon work supported by the National Science Foundation under Grant No. ECS-8307478.

ABSTRACT

A typical database system maintains target data, which contain information useful for users, and access path data, which facilitate faster accesses to target data. Further, most large database systems support concurrent processing of multiple transactions. For a static database system model, where units of concurrency control are not dynamically created or deleted, various concurrency control methods are known. Also, many methods that allow concurrent accesses to indexing structures without invalidating their integrity are known. However, a straightforward integration of these two kinds of concurrency control methods fails because of the phantom problem. In this paper, we introduce group locks in order to solve this problem and discuss their implementation. As one side benefit of introducing group locks, we show that if the lowest-level access path data as well as the target data are two-phase locked by transactions, consistency of the logical data will be preserved.

Key Words and Phrases: database system, concurrency control, phantom problem, access path data, B-tree, multi-level structure, data abstraction, time abstraction

1. Introduction

As well as storing information required by its users (target data**), a database system usually stores information that enables quick accesses to these target data (access path data) [ASTR-76, BACH-74, SCHE-74, STON-76, TSIC-75, WEDE-74, YAO-77]. Access path data are often organized into indexing structures like B-trees because of their flexibility and good performance [ALLC-80, BAYE-77b, COME-79, HAER-78, KEEH-74, MARU-76, MARU-77, NAKA-78, STON-80, WEDE-74].

The concurrency control problem for a database system has been attacked from the following two angles.

1. For a database system modeled as a static*** collection of database entities (typically, pages), extensive serializability theories have been developed, and various mechanisms for realizing serializable executions have been proposed [BAYE-75, BAYE-80, BERN-79, CHAM-74, ESWA-76, GRAY-76****, GRAY-78, HAWL-75, MURO-82, PAPA-79, RIES-77, RIES-79, ROSE-78, SCHL-78, STEA-76].

** In this section the term "target data" is used in a little ambiguous way. It means either the logical data or the physical data that represent those logical data.

*** A collection of database entities is static if database entities in the collection are not dynamically created or removed.

**** The scheme discussed in [GRAY-76] uses a hierarchy of locks, where a higher level lock covers multiple database entities covered by its lower level locks. However, the scheme is not a multi-level concurrency control scheme in the sense as the term is used in this paper.

2. The problem of concurrent operations on indexing structures (typically, B-trees) has been studied by many researchers, and a variety of techniques to enhance concurrency on them have been devised [BAYE-77a, ELLI-80a, ELLI-80b, GUIB-78, KUNG-80, KWON-82, LEHM-81, MILL-78, SAMA-76].

So far, these two approaches have been mostly isolated. A limitation of the first approach alone is that although it is possible to treat access path data in the same way as target data, access path data (especially, root nodes of indexing structures) may become bottlenecks for efficient concurrent operations. On the other hand, although the second approach provides consistency for accesses to indexing structures ("action-level consistency" [GRAY-81]), it does not provide consistency at the transaction level.

We refer to concurrency control that achieves transaction-level consistency as *long-term* concurrency control, and concurrency control that achieves action-level consistency as *short-term* concurrency control. For expository convenience, only simple locking methods are considered in this paper. Locking used at the transaction level will be referred to as *long-term locking*, (typically, two-phase locking [ESWA-76]) and that at the action level *short-term locking*.

The main objective of this paper is to present a scheme that integrates the results of the above two approaches. Manber and Ladner have addressed essentially the same problem for the case where each logical data item can be explicitly identified by a unique key value, by using a binary tree as the indexing structure [MANB-82]. The model in this paper allows multiple data items to be associated with each key value.**

When a data item cannot be designated by a unique key value, the phantom problem [ESWA-76] becomes more complex to handle. In this paper, "group locks" are introduced in order to handle phantoms, and their implementation method is presented. As one important side benefit of introducing the concept of group locks, we can prove the following fact. Assume that access path data are organized as search trees. Then, as long as the lowest-level tree nodes as well as the target data are two-phase locked by transactions, consistency at the logical data level will be preserved. Other search tree nodes (including root nodes) need be locked only for a period shorter than the duration of each tree access.

At the logical data level of the multi-level concurrency control scheme presented in this paper, transactions are assumed to operate on "logical objects", and access path data are completely hidden. At the physical data level, access path data are introduced, and operations on access path data are considered correct as long as target data are correctly reached. One interesting consequence of this weaker requirement for the opera-

** The model in the first version of this paper (Aug. 1981) assumed that each logical data item can be explicitly identified by a unique key value. According to the suggestion by referees, this restriction has been removed from the current paper.

tions on access path data is that those operations need not be serializable in terms of the transactions to which they belong. Further, it is possible to enhance the level of concurrency for access path data by using various techniques, since organizations of access path data are usually restricted, and the kinds of operations that manipulate them are limited (see [KWON-82, LEHM-81]).

In [ASTR-76] two kinds of locks, i.e., "(relational) tuple locks" and "page locks" are briefly explained. Tuple locks are applied at the target data level, and page locks are used at the virtual memory level. However, system R treats access path data in the same way as target data: "share locks must be maintained on all tuples and index values which are read, for the duration of the transaction" [ASTR-76, p. 126]. Cedar DBMS performs locking uniformly over pages, and it employs various techniques to reduce unnecessary conflicts over pages [BROW-81].

In the multi-level concurrency control scheme presented in this paper, separate concurrency control methods are used for the access path data and for the target data. The interaction of these concurrency control methods is the main subject of our analysis. In order to prove precisely that the multi-level concurrency control scheme works correctly, a technique that handles data abstraction in a concurrent system is used. The technique is called _time abstraction_, since it allows us to pinpoint the execution timings of logical operations according to our need.

When a multi-level concurrency control scheme is employed by a distributed database system, identical physical structures are not required for replicating logical data** at different sites. On the other hand, when pages, for example, are used as units of concurrency control, most concurrency control schemes currently proposed require that pages themselves be replicated in order to support replicated logical data. Then, it is almost impossible to replicate logical data on different file systems.

** When a data item is redundantly represented in a distributed database system, "logical data" at individual sites are "physical data" from the system's viewpoint.

2. Logical Database System Model

In this section we state the database system model that we support at the logical level. Although the model itself does not allow dynamic creations and deletions of entities constituting a database system, the model can easily support this feature by not keeping entities with default or irrelevant values. A well-known problem that occurs when dynamic creations and deletions of database entities are allowed is the phantom problem [ESWA-76]. A unique feature of our model is the concept of groups, which are introduced in order to handle the phantom problem within the framework of a serializability theory.

A logical database system consists of a set of data items, a set of groups, and a set of transactions. Data items and groups are collectively called logical objects. A data item can possess a data value. The data value possessed by a data item is called useless if it will never be accessed. Otherwise, it is useful. The set of data items may be countably infinite, although the number of data items with useful data values must be finite at any given time.

Further, a data item can belong to any finite number (possibly zero) of groups at each given time. When a data item X belongs to a group G, we say that X is a member of G. Associations of data items with groups can dynamically change. The set of groups may also be countably infinite, but the number of groups with at least one member must be finite at any given time.

A group can be defined in any conceivable way. A typical case is one where a set of data items possessing a common attribute value form a group. An atypical case is one where a set of data items sharing a page for their physical representations form a group. Groups are not mathematical sets since two different groups may contain the same set of members.

Transactions use the following operations in order to manipulate the data values possessed by data items.

Read(X): The current data value of data item X is returned to the transaction issuing this operation.

Write(X): The data value of data item X is updated to the one** supplied by the transaction issuing this operation.

Further, transactions can add and delete data items to and from groups by using the following operations.

Insert(X, G): Logical object X is made a member of group G.

Remove(X, G): The membership of data item X with group G is resolved.

Sometimes Insert(X, G) and Remove(X, G) operations are implicitly performed. For example, if data items sharing a page p for their physical representations form group G_p, then adding to page p a record that reresents a data item X must be inter-

---

** Since the data value provided is not relevant for our discussions, we do not show it explicitly by an argument.

preted as an execution of Insert(X, G_p).

Although groups can be arbitrarily defined, groups are usually formed so that the members of each group possess some common property. Then, the set of data items possessing a certain property can be located only by locating the members of the groups associated with that property. The following operation is used for locating the members of a group.

MemLocate(G): The names of the data items that are currently members of group G are returned.

Various operations like splitting and merging groups can be implemented by combining MemLocate(G), Remove(X, G) and Insert(X, G) operations.

In Fig. 1(a), data items W and Y are members of group Gl, and data item Z is a member of group G2. However, data item X is not a member of any group. At this point, MemLocate(Gl) will return the names of W and Y. Now, assume that Insert(X, Gl) is issued. The resultant database state is shown in Fig. 1(b). Note that X has become a member of group Gl. If MemmLocate(Gl) is issued at this point, the names of W, X and Y will be returned.

Let us call the operations that manipulate data items and groups _logical_ operations, and let _GUpdate_(_G_) represent either Insert(X, G) or Remove(X, G) for some X. Note that neither an Insert(X, G) operation nor a Remove(X, G) operation affects data item X itself. Then, define relation _conflict_ over the set of logical operations as shown in Fig. 2. We leave to the reader the proof that the effects seen by transactions will not change even if the execution order of any pair of non-conflicting operations are changed. We can regard a GUpdate(G) operation as a write operation to logical object G and a MemLocate(G) operation as a read operation to logical object G. Further, a GUpdate(G) operation does not conflict with another GUpdate(G) operation since they are commutative.

An execution of transactions in which the net effects of transactions are as if they were executed one at a time is called serializable [ESWA-76, PAPA-79]. Let us call a database system that does not allow dynamic creations or deletions of database entities (in our model, data items and groups) a static database system. It is well known that two-phase locking [ESWA-76] can guarantee a serializable execution for a static database system. Although our model includes an unusual feature (i.e., groups), it still is a static database system, and hence two-phase locking can realize a serializable execution, if groups as well as data items are two-phase locked according to relation conflict.

Locking consistent with relation conflict can be achieved with three lock modes (Free, Share, and Exclusive) provided for each data item and with additional three lock modes (Free, Locate, and Update) provided for each group. When a data item or a group is accessed, it must be locked in the mode as indicated in Fig. 2. Since GUpdate(G) operations are commutative, more than one transaction can simultaneously hold the "UPdate" lock of a group.

We assume that the following operations are used for logical locking.

MemLock(X, m): Data itme X is locked in mode m, which is either Share or Exclusive.

MemUnlock(X): The lock set on data item X by the transaction issuing this operation is reset.

GLock(G, m): Group G is locked in mode m, which is either Locate or Update.

GUnlock(G): The lock set on group G by the transaction issuing this operation is reset.

If data item X or group G is already locked in a conflicting mode when a MemLock(X, m) or GLock(G, m) operation is issued, the operation must be blocked, or the transaction issuing the operation must be aborted.

In a typical database system, data items to be accessed are often designated by specifying their key values or the ranges of their key values.** We now consider a method of defining groups in order to support such value-based accessing. For expository convenience, we consider only one key attribute whose values are totally ordered. The model that satisfies the following rules

** In this paper, a key value is not required to identify a data item uniquely. It is simply a value of an attribute or a set of values of multiple attributes.

will be referred to as the single-key logical database model.

M1. The key value of each data item X is uniquely defined at any time as Key(X). The key value may vary according to time, and further it may be NUL.*** The set of all possible key values except for NUL form a domain D. The key values in domain D are totally ordered by <. Further, -inf < K < +inf for any key value K in D.

M2. Group G[K] is defined for every key value K in D. A data item X such that Key(X) = K must be a member of group G[K].

Now, we can locate the data items whose current key values are equal to K by locating the members of G[K]. Further, we assume that we can locate by an operation RLocate(K_i, K_j) the members of the groups whose key values are between K_i and K_j. If there is a countably infinite number of key values in this range, then we must theoretically check the countably infinite number of groups. In Section 3 we will discuss a method to handle this problem.

*** An undefined key value must be regarded as NUL.

3. Physical Implementation

In this section we present an implementation method for the single-key logical database model discussed in Section 2.** Since the model allows possibly countably infinite sets of data items and groups, we cannot permanently provide _physical objects_ for all of those logical objects. Hence, we dynamically assign physical objects to logical objects and maintain only those physical objects whose values are useful or are different from _default_*** values. More specifically, a physical object is not provided for a data item with a useless data value or for a group with no members. An important consequence of this rule is that the values of logical objects are always uniquely defined unless they are useless, even if their corresponding physical objects do not exist. Since we are assuming that the set of data items that contain useful data values and the set of groups that contain at least one member are both finite at any given time, the number of physical objects thus required is finite.****

In order to represent each data item, we use a _target object_. A target object is a physical object of the following

** If we want to support multiple access paths, a separate indexing structure must be provided for each key attribute. Note that a data item can be a member of multiple groups.

*** A default value may not be fixed. When a default value is not fixed, it must be computable from the values of existing physical objects.

**** We are also assuming that the set of transactions executed simultaneously and the set of operations issued by each transaction are finite.

format:

```
record
  PLockMode    :  (Free, Share, Exclusive);
  PLockCount   :  integer;
  RefCount     :  integer;
  LLockMode    :  (Free, Share, Exclusive);
  LLockCount   :  integer;
  Value        :  ValueType
end.
```

The data value of a data item X is stored in the Value field of the target object x that represents X. The LLockMode field of x shows the current lock mode of X. The LLockCount field of x indicates the number of transactions that currently hold locks on X. When X is locked in "Exclusive" mode, x.LLockCount must be one. The use of other fields will be explained later.

On the other hand, in order to handle a group, a group descriptor is provided. A group descripto is a physical object of the following format:

```
record
  PLockMode    :  (Free, Share, Exclusive);
  PLockCount   :  integer;
  RefCount     :  integer;
  Key          :  KeyType;
  GLockMode    :  (Free, Locate, Update);
  GLockCount   :  integer;
  ILockCount   :  integer;
  Members      :  set of TargetObjectPtr
end.
```

The Key field of the group descriptor g for a group G[K] contains the key value K. The identifiers of the target objects that represent the members of G[K] are kept in the Members field of g. The GLockMode field of g indicates the current lock mode of G. The GLockCount field of g shows the number of locks being applied to G.

According to rule M2, group G[K] is defined for every key value K in the domain D. Then, providing a group descriptor for every group is simply impossible since there can be a countably infinite number of key values. We handle this problem by not providing a group descriptor g such that g.RefCount = 0 and g.Members = NIL.**

We now discuss an additional locking mechanism for groups. Assume that group descriptors g_i and g_j such that g_i.Key = K_i, g_j.Key = K_j, and $K_i < K_j$ are provided, and that no group descriptor g such that K_i < g.Key < K_j is provided. When this assumption holds, we say that <u>interval</u> I(K_i, K_j) exists. Assume further that a RLocate(K_i, K_j) request is issued. Then, every group G[K] such that $K_i < K < K_j$ must be locked in "Locate" mode.

Let us now consider locking every group G[K] such that K_i < K < K_j, or interval I(K_i, K_j), in "Locate" mode, which is the only lock mode possible for an interval. Instead of creating and locking possibly infinite number of group descriptors that should exist between g_i and g_j, we maintain in the <u>ILockCount</u> field of g_i the number of locks being applied to these hypothetical group descriptors in interval I(K_i, K_j). When interval I(K_i, K_j) is locked in "Locate" mode, a data item cannot be inserted or removed from any group G[K] such that $K_i < K < K_j$. This locking

** The precise condition will be given after "interval locks" are introduced.

method will be called _interval locking_.

We now precisely define the dafault value for a non-existing group descriptor. Assume that there exist two group descriptors g_i and g_j such that g_i.Key = K_i, g_j.Key = K_j, and $K_i < K_j$ and that there exists no group descriptor g such that $K_i <$ g.Key $< K_j$. Then, if the group descriptor g' associated with any group G[K] such that $K_i < K < K_j$ would exist,

```
g'.RefCount   = 0,
g'.GLockMode  = Free   if g_i.ILockCount = 0,
                Locate if g_i.ILockCount ≠ 0,
g'.GLockCount = g_i.ILockcount,
g'.ILockCount = g_i.ILockcount, and
g'.Members = NIL.
```

As we stated, we assume that data items to be accessed are designated by specifying their key values. A common method for supporting such an access method is to provide an indexing structrue consisting of _access path objects_, which also are physical objects. We assume that access path objects are of the following format:

```
record
  PLockMode     :  (Free, Share, Exclusive);
  PLockCount    :  integer;
  NumberOfSons  :  1 .. MaxFanout;
  Son           :  array[1..MaxFanout] of ObjectPtr;
  Boundary      :  array[1..MaxFanout-1] of KeyType
end.
```

Access path objects and group descriptors are organized as a _multi-way search tree_.

A1. The search tree is empty or possesses one _root_ node.

A2. The root node, if it exists, is either an access path object or a group descriptor.

A3. When an access path object is the root node, it possesses at least two and at most _MaxFanout_ descendant nodes. Otherwise, it possesses at least _MinFanout_ and at most _MaxFanout_ descendant nodes. We assume that _MinFanout_ $\leq$ (_MaxFanout_ div 2). A descendant node is either an access path object or a group descriptor.

A4. All leaf nodes of the search tree are group descriptors, and all group descriptors are away from the root node by the same number of intervening access path objects.

A5 If a group descriptor g can be reached by following Son[i] of an access path object p, then p.Boundary[i-1] < g.Key $\leq$ p.Boundary[i] for p.Boundary[i-1] and p.Boundary[i] if they exist.

The condition that _MinFanout_ $\leq$ (_MaxFanout_ div 2) guarantees that an access path object with MaxFanout descendants can be split into two access path objects of legitimate sizes. This condition, which is slightly different from the one for an ordinary B-tree, is required by the "top-down algorithm" used for tracing the search tree.

When a physical object (an access path object, a group descriptor, or a target object) is accessed, it must be _physically locked_ in either Share or Exclusive mode depending on the mode of the access. The following two operations are used for physical locking.

plock(x, m): The physical object designated by x is physically locked in lock mode m. If it is already locked in a conflicting mode, then the transaction issuing this operation is blocked.

```
while ((x.PLockMode = Exclusive) or
       (x.PLockMode = Share) and (m = Exclusive)) do wait;
x.PLockMode := m;
x.PLockCount := x.PLockCount + 1.
```

punlock(x): The physical lock set by the transaction issuing this operation on physical object x is reset.

```
x.PLockCount := x.PLockCount - 1;
if x.PLockCount = 0 then x.PLockMode := Free.
```

In principle, a physical lock applied to a physical object can be released as soon as the access to the physical object is completed. That is, transaction-based two-phase locking, for example, is not necessary. If a physical object is accessed more than once by the same transaction, the physical object can be locked each time when it is accessed.

We are now ready to discuss implementations of various logical operations. Fig. 3 shows the correspondence between logical operations and their correponding physical operations.

In order to locate a target object for a data item X, we must first locate the group descriptor g such that g.Key = Key(X).

glocate(K): If the group descriptor g such that g.Key = K already exists, its RefCount is incremented by one, and its identifier is returned. If the group descriptor g such that g.Key = K does not exist, it must be created. Let m_I and c_I be the lock mode and the lock count, respectively, of the interval where g falls. That is, if K_i is the largest key value such that $K_i < K$ and for which a group descriptor g_i currently exists, then m_I = Free when g_i.ILockCount = 0, m_I = Locate when g_i.ILoclcount ≠ 0, and c_I = g_i.ILockCount. Now, g can be created with the following field values.

```
PLockMode   = Free,
PLockCount  = 0,
RefCount    = 1,
Key         = K,
GLockMode   = m_I,
GLockCount  = c_I,
ILockCount  = c_I,
Members     = NIL.
```

After g is inserted into the search tree, the identifier of g is returned.

Fuction glocate(K) must scan the search tree starting from the root node until a group descriptor is reached. The implementation of glocate(K) shown in Fig. 4 uses the top-down algorithm given in [GUIB-78].** If an access path object with too many (= MaxFanout) descendants is encountered, the access path object must be split. If an access path object with too few (= MinFanout) descendants is encountered, the access path object must

** Obviously, any algorithm that guarantees consistency for search tree accesses can be used. See [KWON-82, LEHM-81] for various algorithms that can be used for this purpose.

be merged with its neighbor or some descendant pointers in its neighbor must be moved to the access path object. (Although the root node must be treated differently, we do not discuss the details.) Further, we assume that physical locks on access path objects are seized and released according to the "tree protocol" of [SILB-80].

When group descriptor g such that g.Key = K is reached, the set of target objects that represent the data items belonging to G[K] can be located.

memlocate(g): The target object identifiers in g.Members are returned to the transaction issuing this operation. When the identifier of a target object x is returned to a transaction, x.RefCount must be incremented by one.

Also, we assume that if a target object x does not exist for a data item X, then x can be created as x :- new(TargetObjectType).**

Once the target object x for a data item X is known, the data value of X can be accessed.

read(x): x.Value is returned to the transaction issuing this operation.

write(x): x.Value is updated to the data value provided by the transaction issuing this operation.

** ":-" is the Simula notation for the assignment operator for a pointer value.

Assume that target object x represents a data item X, and group descriptor g represents a group G. Then, Insert (X, G) and Remove(X, G) operations can be implemented as follows.

```
insert(x, g):

        g.Members  := g.Members U {x};
        x.RefCount := x.RefCount + 1.

remove(x, g):

        g.Members  := g.Members - {x};
        x.RefCount := x.RefCount - 1.
```

A target object and a group descriptor must be released after their use. A target object can be released as follows.

memrelease(x): x.RefCount is decremented by one. If the resultant x.RefCount is zero, then x is deleted.

If x.RefCount = 0, then neither x belongs to any group, nor its identifier is held by any transaction. Therefore, x will never be accessed, and hence it can be deleted. Note that when x.RefCount = 0, x.LLockCount must be zero. This requirement is natural since a lock on x cannot be released if x has been released.

Our implementation does not allow transactions to delete target objects explicitly. However, a target object deletion requested by a transaction can be supported as follows. Assume that NUL is the data value to be returned when a read(x) operation is applied to a non-existing target object x. Then, x is effectively deleted if NUL is assigned to x.Value. A target object containing such NUL value can be regarded as a "tombstone" [LOME-75].

A group descriptor g seized by a glocate(K) operation can be released by a grelease(g) operation.

grelease(g): g.RefCount is decremented by one. Let g_i be the group descriptor that immediately precedes g (i.e., g_i.Key < g.Key, and g_i.Key < g'.Key < g.Key for no existing group descriptor g'). Now, if the following condition holds, g can be deleted:

```
g.RefCount   = 0,
g.GLockMode  = Free   if g_i.ILockCount = 0,
               Locate if g_i.ILockCount ≠ 0,
g.GLockCount = g_i.ILockCount,
g.ILockCount = g_i.ILockCount, and
g.Members    = NIL.
```

In order for the top-down algorithm to work correctly, each grelease(g) must be preceded by a <u>glocate</u>'(<u>K</u>) operation such that g.Key = K if the deletion of g is expected. This operation must work like a gloccate(K) operation except that it does not increment g.RefCount. A glocate'(K) operation prevents access path objects from possessing too few descendants, as well as it can locate the immediate ancestor node of g.

Logical locking must be performed by regarding that each Read(X) or Write(X) operation occurs when its corresponding read(x) or write(x) operation occurs. The reason why this rule works correctly is discussed in Section 4. Let x be the target object representing a data item X. Then, logical operations MemLock(X, m) and MemUnlock(X) can be performed by physical operations memlock(x, m) and memunlock(x), respectively.

```
  memlock(x, m):

         while ((x.LLockMode = Exclusive) or
                (x.LLockMode = Share) and (m = Exclusive)) do wait;
         x.LLockMode  := m;
         x.LLockCount := x.LLockCount + 1.

  memunlock(x):

         x.LLockCount := x.LLockCount - 1;
         if x.LLockCount = 0 then x.LLockMode := Free.
```

Group locking can be performed similarly. Let g be the group descriptor representing a group G[K]. Then, logical operations GLock(G[K], m) and GUnlock(G[K]) can be performed by physical operations glock(g, m) and gunlock(g), respectively.

```
  glock(g, m):

         while ((g.GLockMode = Share) and (m = Update ) or
                (g.LGockMode = Update) and (m = Share)) do wait;
         g.GLockMode := m;
         g.GLockCount := g.GLockCount + 1.

  gunlock(g):

         g.GLockCount := g.GLockCount - 1;
         if g.GLockCount = 0 then g.GLockMode := Free.
```

Assume that for a pair of group descriptors g_i and g_j, g_i.Key = K_i, g_j.Key = K_j, and $K_i < K_j$, and further that there exists no group descriptor g such that $K_i <$ g.Key $<K_j$. Then,

interval $I(K_i, K_j)$ can be locked and unlocked as follows.

```
ilock($g_i$):   $g_i$.ILockCount := $g_i$.ILockCount + 1.
iunlock($g_i$): $g_i$.ILockCount := $g_i$.ILockCount - 1.
```

An example of a logical transaction and its physical counterpart is given in Fig. 5. The transaction is interested in every data item X such that Key(X) = K, and there currently exist only one such data item. Note that logical locks are applied according to the two-phase locking rule. In Fig. 6, the periods of the logical locks and the physical locks applied by a transaction whose logical representation is Read(X); Read(Y); Write(Z) is shown, where Key(X) = K_1, Key(Y) = K_2, Key(Z) = K_3. Note that physical locks are applied for far shorter periods than logical locks.

4. Correctness

In this section, we show that the implementation of the multi-level concurrency control scheme given in the preceding section is correct, following the ordinary approach [GUTT-78, HOAR-72, LISK-77, OWIC-77] for proving correctness of systems (or programs) with multi-level structures. First, it is proved that logical objects and logical operations are correctly implemented by physical objects and physical operations. Then, we show that logical operations are correctly scheduled. In our proof, execution timings of logical operations are explicitly defined. We call this technique _time abstraction_.

We first show that logical objects and logical operations applied to them are correctly implemented. For this purpose, we define the _data abstraction functions_ as follows.

A1. Each target object x represents a _separate_ data item X.**
If a target object x exists for a data item X, then the data value of X is defined by x.Value, and the lock status of X is defined by x.LLockMode and x.LLockCount. If such x does not exist, then the data value of X is undefined, and no locks are applied on X.

A2. If a group descriptor g such that g.Key = K exists, then g.Members contains the identifiers of the target objects

** If target object x_2 is not a continuation of target object x_1, they represent different data items even if they contain the same data value.

that represent the members of group G[K], and the lock status of G[K] is shown by g.GLockMode and g.GLockCount. If such g does not exist, then G[K] is empty. In this case, the lock status of G[K] can be known from g_i.ILockCount, where g_i is the group descriptor that would immediately precede g if g existed.

We now state the requirement for correct implementation of logical operations.

Definition (Correct Implementation of Logical Operations). An implementation of logical operations is correct if the following conditions are satisfied.

I1. A logical write operation Write(X) correctly updates the data value of X as defined by A1, and a logical read operation Read(X) returns the current data value of X as defined by A1.

I2. Each group G[K] as defined by A2 is properly accessed by Insert(X, G[K]), Remove(X, G[K]) and MemLocate(G[K]) operations as discussed in Section 2.

The following lemma is trivially true.

Lemma 1. Read(X) and Write(X) are properly implemented by read(x) and write(x), respectively, where x is the target object for data item X.

Note that a data item with an undefined data value will never be accessed, since its target object cannot be reached.

The following lemma is also trivial.

Lemma 2. Insert(X, G), Remove(X, G) and MemLocate(G) are correctly implemented by insert(x, g), remove(x, g) and memlocate(g), respectively, where x and g are the target object and the group descriptor that respectively represent X and G.

We say that glocate(K) and grelease(g) are correctly implemented, if they satisfy the following requirements.

G1. At any time at most one group descriptor exists in the system for each group G[K].

G2. For each pair of glocate(K) and grelease(g) issued by a transaction, group descriptor g returned by glocate(K) is the correct group descriptor for G[K] until the corresponding grelease(g) occurs.

G3. The state of G[K] is continuous when its group descriptor is created or deleted.

Although we do not show the detailed implementations of procedures glocate(K) and grelease(g), we assume that they satisfy the specifications given in Section 3. Then, we have the following lemma, which concerns correctness of sequential programs.

Lemma 3. Requirements G1, G2 and G3 are satisfied if glocate(K) (and glocate'(K)) and grelease(g) operations are executed one at a time.

Proof. Requirements G1 and G2 immediately follow from the specifications for glocate(K) and grelease(g). Requirements G3 follows from the fact that the value of a group descriptor created or deleted is identical to the default value defined for that group descriptor. []

Further, requirements G1, G2 and G3 are still satisfied even when glocate(K) and grelease(g) operations are executed concurrently.

Lemma 4. Even if procedures glocate(K) and grelease(g) are executed concurrently, they produce the same effects as when they are executed one at a time.

Proof. The implementation of glocate(K) and grelease(g) follows the tree protocol of [SILB-80], and hence their execution is serializable in terms of these operations. []

If logical operations are correctly implemented and if the execution sequence of logical operations is serializable in terms of transactions, then we can consider that the resultant system operation is correct. This condition is satisfied if a consistent locking scheme is employed at the logical object level. The point here is that accesses to physical objects (in particular, to access path objects) need not be serializable in terms of transactions.

In order to define the execution sequence of logical operations, execution timings must be specified for logical operations.

<u>Definition</u> (<u>Execution</u> <u>Timings</u> <u>of</u> <u>Logical</u> <u>Operations</u>). The execution timing of a logical operation Read(X) or Write(X) is defined to be the timing when the target object x for logical object X is accessed by the physical operation read(x) or write(x) that implements Read(X) or Write(X). Similarly, the execution timing of a logical operation MemLocate(G), Insert(X, G) or Remove(X, G) is defined to be the timing when the group descriptor g that represents G is accessed by the physical operation memlocate(g), insert(x, g) or remove(x, g) that implements the logical operation.

Note that if execution timings of logical operations are defined as above, then the values of logical objects as defined by A1 and A2 are accessed exactly at the points when they are supposedly accessed.

In Fig. 7, transactions T_1 and T_2 do not conflict at the logical object level, and hence their logical write operations may be interleaved in any way. Note that Write(B), for example, must be executed as

```
g_B :- glocate('B');
  b :- New(TargetObjectType);
write(b);
insert(b, g_B);
grelease(g_B).
```

Also, target objects are not shown in the figure.

The tree structure of Fig. 7(h), which results if T_1 and T_2 are executed in the order of Write(B), Write(D), Write(I) and Write(G), cannot occur if T_1 and T_2 are executed one at a time;

either Fig. 7(e) or Fig. 7(l) must result. Note that the tree structure in Fig. 7(h) represents the same logical database state as the tree structure in Fig. 7(e) or Fig. 7(l), and that it must be considered correct.

Another property that holds for the multi-level concurrency control algorithm given in Section 3 is that permanent blocking of transactions will not occur if logical locks will not cause deadlocks. This property follows from the following facts.

1. A transaction can release any physical lock when its logical lock request is blocked. Then, logical locks will never block physical lock requests.

2. Target objects need be physically locked only while they are accessed. Access path objects and group descriptors are physically locked according to the tree protocol [SILB-80]. Therefore, deadlocks that involve only physical locks will not occur.

Note that the same physical object can be physically locked more than once by the same transaction during its execution.

5. Discussions

In this section we compare group locking with other locking mechanisms devised for handling the phantom problem. Among the locking mechanisms compared are predicate locking [ESWA-76], precision locking [JORD-81], page locking, and "end-of-file marker" locking [BERN-81].

Before we proceed, we want to make the following observation. We usually say that when a data item X is accessed, a lock on X must be set, and we usually associate a separate lock X for each data item X. Actually, the name of the lock associated with a data item X is irrelevant, and an arbitrary lock f(X) can be associated with each data item X as long as it is consistently used in order to control accesses to X. Further, the lock f(X) and the lock g(Y) that are used to control accesses to data items X and Y, repectively, may coincide. In this case, the level of concurrency may be sacrificed, but correct system operation can still be realized.

The best known mechanism for handling the phantom problem is predicate locking. A lock in predicate locking is a pair (P, m), where P is a predicate over data item values, and m is a lock mode, either Share or Exclusive. Once a share predicate lock (P, Share) is set, the set of data items whose data values satisfy P are frozen. Data items whose current data values satisfy P cannot be updated. Further, the data values of other data items cannot be updated so that they will newly satisfy P. An exclusive predicate lock (P, Exclusive) also freezes the set of data items whose data values satisfy P except for the transaction that has set the exclusive predicate lock. Two predicate locks (P_1, m_1) and (P_2, m_2) conflict with each other if some conceivable data value satifies both P_1 and P_2 and if at least one of m_1 and m_2 is Exclusive.

Predicate locking can be simulated by group locking as follows. Each predicate must be represented by a group or by a set of groups. In order to simplify our discussion, we assume that each predicate P can be represented by group G_P. Then, a share predicate lock (P, Share) can be set by locking group G_P in mode Locate and by locking all members of group G_P in mode Share. An exclusive predicate lock (P, Exclusive) can be set by locking group G_P in mode Update and by locking all members of group G_P in mode Exclusive.

One problem with predicate locking is that testing whether two predicates conflict or not is often computationally hard. Another problem is that a conflict of two predicate lock requests (P_1, m_1) and (P_2, m_2) is often caused by a data value that currently does not exist and is irrelevant to the transactions issuing these requests.

For example**, consider a collection of records, each of which includes fields <u>State</u> and <u>Product</u>. That is, each record is of the form <state-name, product-name, other-information>. Further, assume that transaction T_1 is interested in modifying every record p such that p.State = Oregon, and that transaction T_2 is interested in modifying every record q such that q.Product = Orange. If T_1 and T_2 uses predicate locking, they cannot be executed concurrently, since the predicates <Oregon, *, *> and <*, Orange, *> can both be satisfied by a hypothetical record that satisfies the predicate <Oregon, Orange, *>. However, if

** This example was adapted from a referee's comment.

such a record does not actually occur while T_1 and T_2 are executed, and if the records satisfying <Oregon, *, *> and the records satisfying <*, Orange, *> do not share any page for their physical representations, T_1 and T_2 can be executed concurrently if page locking is used.

Precision locking solves these problems by using exclusive locks on individual data items instead of exclusive predicate locks. Since read operations do not conflict with each other, there is no need to test whether two share predicate lock requests conflict or not. Further, when a data value is updated, we can know the old and the new data values, and hence we need to test at most a data value against a predicate.

Precision locks are similar to group locks except for some minor differences. In group locking, for example, even if a group is locked in Locate mode, members of the group can be updated as long as their memberships with the group does not change.

The simplest approach to prevent the phantom problem is to apply long-term locking on pages. Since the set of pages are usually fixed, the phantom problem does not occur. However, the long-term locking of pages poses the following problems. When a data item is accessed, the page that contains its physical representation must be locked, then all of the data items whose physical representations share that page are also effectively locked. Further, replication of logical data in a distributed database system becomes difficult if different sites use different page sizes.

We now state an interesting scheme that allows short-term locking for the pages that contain only access path data. Assume that access path data are organized as search trees, and let us call the pages that contain only access path data _access path pages_. Then, the phantom problem can be prevented _as long as the lowest-level access path pages as well as the target data pages are two-phase locked by transactions_. Only short-term locking is required for the other access path pages.

We now explain why the above scheme works. We can consider that each group descriptor is contained in a lowest-level access path page. When target data are located, lowest-level access path pages are locked in Share mode, which is equivalent to Locate mode. When records for target data are added or deleted, lowest-level access path pages are locked in Exclusive mode, which is stronger that Update mode. We now know that the page locking performed in this way properly covers the required group locking.

Another kind of group locking, "end-of-file marker" locking, is discussed in [BERN-81]. In end-of-file marker locking, the records in each file form a group, and the end-of-file marker of that file effectively functions as the group descriptor.

6. Conclusion

A multi-level concurrency control scheme was presented, and its correctness was discussed. In order to handle the phantom problem, <u>groups locks</u> were introduced. Accesses to access path data were regarded correct as long as they could reach correct target data.

The scheme in this paper enables us to integrate the two major research results on concurrency control for database systems: long-term and short-term concurrency control. As one important side benefit, we pointed out that transactions need not apply long-term locking to upper-level nodes of indexing structures.

References

[ALLC-80] Allchin, J.E., Keller, A.M., and Wiederhold, G. FLASH: A language-independent, portable file access system. Proc. ACM-SIGMOD Conf. on Management of Data, 1980, pp. 151-156.

[ASTR-76] Astrahan, M.M., et al. System R: Relational approach to database management. ACM Tr. on Database Systems 1, 2 (June 1976), 97-137.

[BACH-74] Bachman, C.W. Implementation techniques for data structure sets. In Data Base Management Systems, Jardine, D.A. (ed.), North-Holland Publishing Company, 1974, pp. 147-160.

[BAYE-75] Bayer, R. On the integrity of data bases and resource locking. In Data Base Systems, Lecture Notes in Computer Science 39, Springer-Verlag, 1975, pp. 339-361.

[BAYE-77a] Bayer, R. and Schkolnick, M. Concurrency of operations on B-trees. Acta Informatica 9, 1977, 1-21.

[BAYE-77b] Bayer,Prefix B-trees. ACM Tr. on Database Systems 2, 1 (March 1977), 11-26.

[BAYE-80] Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. ACM Tr. on Database Systems 5, 2 (June 1980), 139-156.

[BERN-79] Bernstein, P., Shipman, D., and Wong, W. Formal aspects of serializability in database concurrency control. IEEE Tr. on Software Engineering SE-5, 3 (May 1979), 203-216.

[BERN-81] Bernstein, P., Goodman, N., and Lai, M.-Y. Laying phantoms to rest (by understanding the interactions between schedulers and translators in a database system). TR-03-81, Aiken Computation Laboratory, Harvard University, 1981.

[BROW-81] Brown, M.R., Gattell, R.G.G., and Suzuki, N. The Cedar DBMS: A preliminary report. Proc. ACM-SIGMOD Conf., 1981, pp. 205-211.

[CHAM-74] Chamberlin, D., Boyce, R., and Traiger, I. A deadlock-free scheme for resource locking in a data-base environment. Proc. IFIP Congress 1974, pp. 340-343.

[COME-79] Comer, D. The ubiquitous B-tree. Computing Surveys 11, 2 (June 1979), 121-137.

[ELLI-80a] Ellis, C.S. Concurrent search and insertion in AVL trees. IEEE Tr. on Computers C-29, 9 (Sept. 1980), 811-817.

[ELLI-80b] Ellis, C.S. Concurrent search and insertion in 2-3 trees. Acta Informatica 14 (1980), 63-86.

[ESWA-76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I. The notions of consistency and predicate locks in a database system. CACM 19, 11 (Nov. 1976), 624-633.

[GRAY-76] Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. In Modelling in Data Base Management Systems, Nijssen, G.M. (ed.), North Holland, 1976, pp. 365-394.

[GRAY-78] Gray, J. Notes on data base operating systems. In Lecture Notes in Computer Science 60, Springer-Verlag, 1978, pp. 393-481.

[GRAY-81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. The recovery manager of the System R database manager. Computing Surveys 13, 2 (June 1981), 223-242.

[GUIB-78] Guibas, L.J. and Sedgewick, R. A dichromatic framework for balanced trees. Proc. 19th Symp. on Foundations on Comp. Sci., 1978, pp. 8-21.

[GUTT-78] Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. CACM 21, 12 (Dec. 1978), 1048-1064.

[HAER-78] Haerder, T. Implementing a generalized access path structure for a relational database system. ACM Tr. on Database Systems 3, 3 (Sept. 1978), 285-298.

[HAWL-75] Hawley, D., Knowles, J., and Tozer, E. Database consistency and the CODASYL DBTG proposals. Comp. J. 18, 3 (Aug. 1975), 206-212.

[HOAR-72] Hoare, C.A.R. Proof of correctness of data representations. Acta Informatica 1 (1972), 271-281.

[JORD-81] Jordan, J.R., Banerjee, J., and Batman, R.B. Precision locks. Proc. SIGMOD-81, pp. 143-147.

[KEEH-74] Keehn, D.G. and Lacy, J.O. VSAM data set design parameters. IBM Systems Journal 13 (1974), 3, 186-212.

[KUNG-80] Kung, H.T. and Lehman, P.L. Concurrent manipulation of binary search trees. ACM Tr. on Database systems 5, 3 (Sept. 1980), 354-382.

[KWON-82] Kwong, Y.S. and Wood, D. A new method for concurrency in B-Trees. IEEE Tr. on Software Engineering SE-8, 3 (May 1982), 211-222.

[LEHM-81] Lehman, P.L. and Yao, S.B. Efficient locking for concurrent operations on B-trees. ACM Tr. on Database Systems 6, 4 (Dec. 1981), 650-670.

[LISK-77] Liskov, B. and Snyder, A. Abstraction mechanisms in CLU. CACM 20, 8 (Aug. 1977), 564-576.

[LOME-75] Lomet, D.B. Scheme for invalidating references to freed storage. IBM Journal of Research and Development 19, 1, (Jan. 1975), 26-35.

[MANB-82] Manber, U. and Ladner, R.E. Concurrency control in a dynamic search structure. Proc. ACM Symp. on Principles of Database Systems, 1982, pp. 268-282.

[MARU-76] Maruyama, K. and Smith, S.E. Optimal reorganization of distributed space disk files. CACM 19, 11 (Nov. 1976), 634-642.

[MARU-77] Maruyama, K. and Smith, S.E. Analysis of design alternatives for virtual memory indexes. CACM 20, 4 (April 1977), 245-254.

[MILL-78] Miller, R.E. and Snyder, L. Multiple access to B-trees (preliminary version). Proc. Conf. on Inf. Sciences and Systems, 1978, pp. 400-407.

[MURO-82] Muro, S., Kameda, T., and Minoura, T. Multi-version concurrency control scheme for a database system. To appear in JCSS. Also, TR 82-2, Dept. of Computing Science, Simon Fraser University, Feb. 1982.

[NAKA-78] Nakamura, T. and Mizoguchi, T. An analysis of storage utilization factor in block split data structuring scheme. Proc. 4th Int. Conf. on Very Large Databases, 1978, pp. 489-494.

[OWIC-77] Owicki, S.S. Specifications and proofs for abstract data types in concurrent programs. TR. 133, Computer Systems Lab., Stanford University, 1977.

[PAPA-79] Papadimitriou, C.H. The serializability of concurrent database updates. JACM 26, 4 (Oct. 1979), 631-653.

[RIES-77] Ries, D.R. and Stonebraker, M. Effects of locking granularity in a database management system. ACM Tr. on Database Syst. 2, 3 (Sept. 1977), 233-246.

[RIES-79] Ries, D.R. and Stonebraker, M.R. Locking granularity revisited. ACM Tr. on Database Syst. 4, 2 (June 1979), 210-227.

[ROSE-78] Rosenkrantz, D., Stearns, R., and Lewis, P. System level concurrency control for distributed database systems. ACM Tr. on Database Systems 3, 2 (June 1978), 178-198.

[SAMA-76] Samadi, B. B-trees in a system with multiple users. Information Processing Letters 5, 4 (Oct. 1976), 107-112.

[SCHE-74] Schenk, H. Implementational aspects of the CODASYL DBTG proposal. In Database Management, Klimbie, J.W. and Koffeman, K.L. (eds.), North-Holland Publishing Company, 1974, pp. 399-412.

[SCHL-78] Schlageter, G. Process synchronization in database systems. ACM Tr. on Database Systems 3, 3 (Sept. 1978), 248-271.

[SILB-80] Silberschatz, A. and Kedem, Z. Consistency in hierarchical database systems. JACM 27, 1 (Jan. 1980), 72-80.

[STEA-76] Stearns, R., Lewis, P., and Rosenkrantz, D. Concurrency control for database systems. Proc. IEEE Symp. on Foundations of Comp. Sci., Oct. 1976, pp. 19-32.

[STON-76] Stonebraker, M, Wong, E., and Kreps, P. The design and implementation of INGRES. ACM Tr. on Database Systems 1, 3 (Sept. 1976), 189-222.

[STON-80] Stonebraker, M. Retrospection on a database system. ACM Tr. on Database Systems 5, 2 (June 1980), 225-240.

[TSIC-75] Tsichritzis, D. A network framework for relation implementation. In Data Base Description, Douque, B.C.M. and Nijssen, G.M. (eds.), North-Holland, 1975, pp. 269-282.

[WEDE-74] Wedekind, H. On the selection of access paths in a data base system. In Data Base Management, Klimbie, J.W. and Koffeman, K.L. (eds.), North-Holland, 1974, pp. 385-397.

[YAO-77] Yao, S.B. An attribute based model for database access cost analysis. ACM Tr. on Database Systems 2, 1 (March 1977), 45-67.

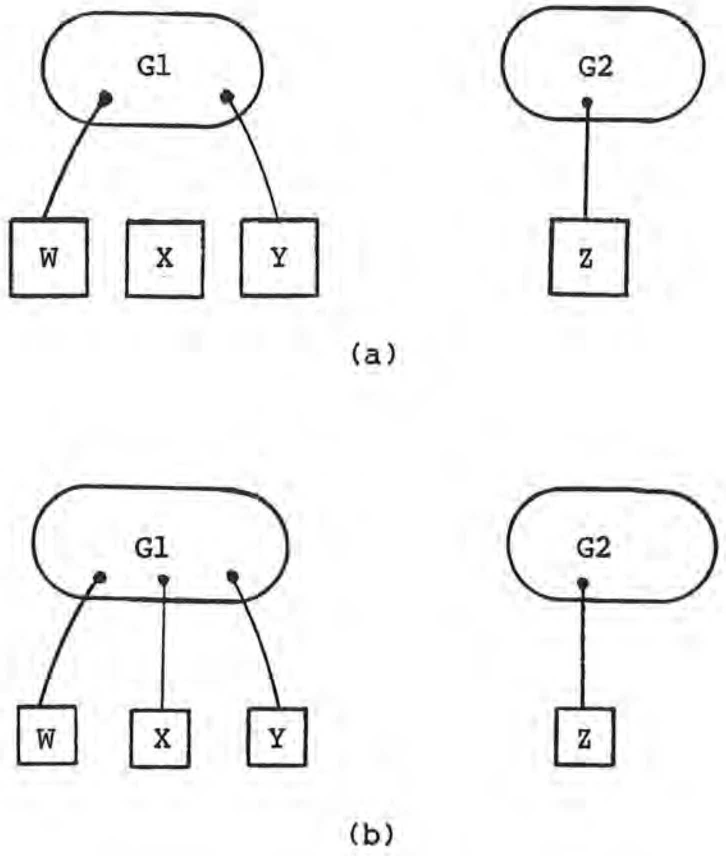


Fig. 1. Logical database states.

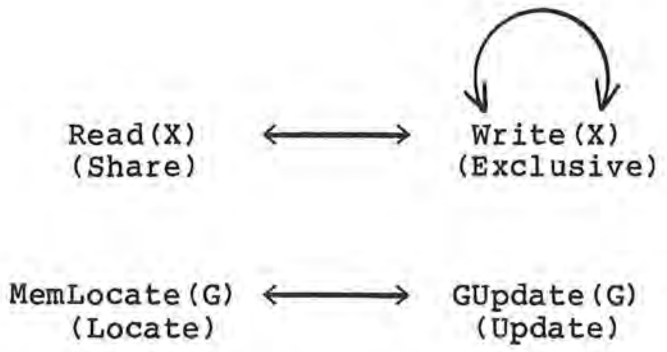


Fig. 2. Conflicting operations and locks used by them.

var X	<----->	x :- new(TargetObjectType)
{ X } :- MemLocate(G[K])	<----->	g :- glocate(K); { x } :- memlocate(g)
(no counterpart)	<----->	grelease(g)
(no counterpart)	<----->	memrelease(x)
Read(X)	<----->	read(x)
Write(X)	<----->	write(x)
Insert(X, G)	<----->	insert(x, g)
Remove(X, G)	<----->	remove(x, g)
MemLock(X, m)	<----->	memlock(x, m)
MemUnlock(X)	<----->	memunlock(x)
GLock(G, m)	<----->	glock(g, m)
GUnlock(G)	<----->	gunlock(g)

Fig. 3. Logical operations and their corresponding physical operations.

```
function glocate(K: KeyType): GroupDescriptorType;

  begin
    LastNodePtr :- nil;
    plock(LastNodePtr↑);      (* lock RootNodePtr *)
    NodePtr :- RootNodePtr;   (* start with the root node *)
    plock(NodePtr↑);          (* lock the root node *)
    while NodePtr↑ is an access path object do
      begin
        case
          NodePtr↑ is too big:
            begin
              split NodePtr↑;
              punlock(NodePtr↑);
              adjust NodePtr; (* put NodePtr on the right path *)
              plock(NodePtr↑);
            end;
          NodePtr↑ is too small:
            begin
              plock(neighbor of NodePtr↑);
              merge NodePtr↑ with its neighbor or move
              some son pointers of the neighbor to NodePtr↑;
              punlock(neighbor of NodePtr↑);
              punlock(NodePtr↑);
              adjust NodePtr; (* put NodePtr on the right path *)
              plock(NodePtr↑);
            end;
        end;
        punlock(LastNodePtr↑);
        LastNodePtr :- NodePtr;
        NodePtr :- NodePtr↑.Son[Rank] where
          Boundary[Rank-1] < K < Boundary[Rank] for NodePtr↑;
        plock(NodePtr↑);
      end;

    (* group descriptor is reached *)
    if NodePtr↑.Key = K then
        begin     (* group descriptor for K already exists *)
            punlock(LastNodePtr↑);
            NodePtr↑.RefCount := NodePtr↑.RefCount + 1;
            glocate :- NodePtr;
        end
    else
        begin     (* group descriptor for K does not exist *)
            create a group descriptor NewLeaf↑
              with Key = K, RefCount = 1, ... ;
            insert the pointer to NewLeaf↑ into LastNodePtr↑;
            punlock(LastNodePtr↑);
            glocate :- NewLeaf;
        end;
    punlock(NodePtr↑);
  end
```

Fig. 4. Function glocate(K).

```
                            g := glocate(K);
GLock(G[K], Locate);        glock(g, Locate);
                            { x } :- memlocate(g);
MemLock(X, Exclusive);      memlock(x, Exclusive);
GUnlock(G[K]);              gunlock(g);
                            grelease(g);
Read(X);                    read(x);
  .                           .
  .                           .
Write(X);                   write(x);
MemUnlock(X);               memunlock(x);
                            memrelease(x);
```

Fig. 5. A logical transaction and its physical counterpart.

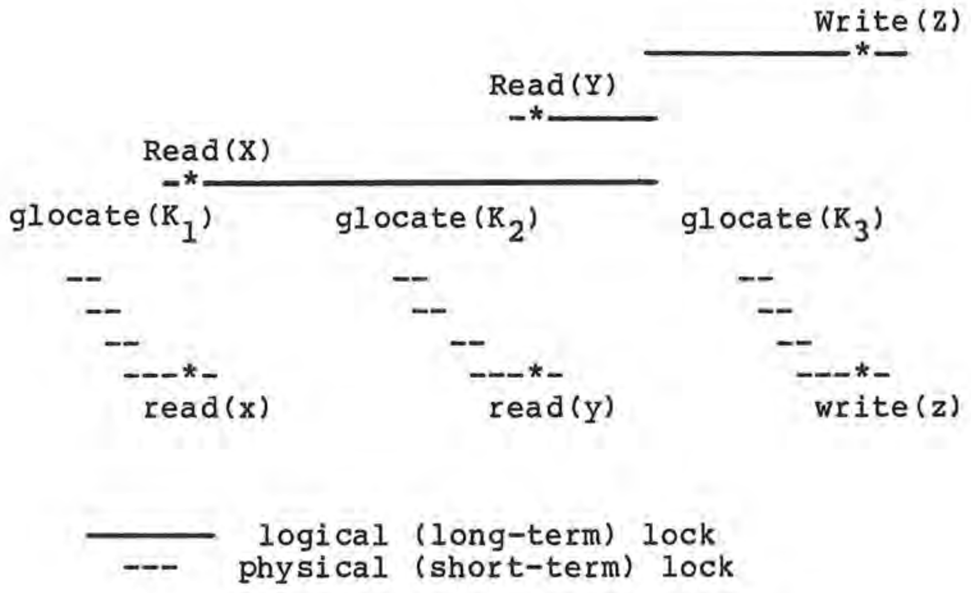
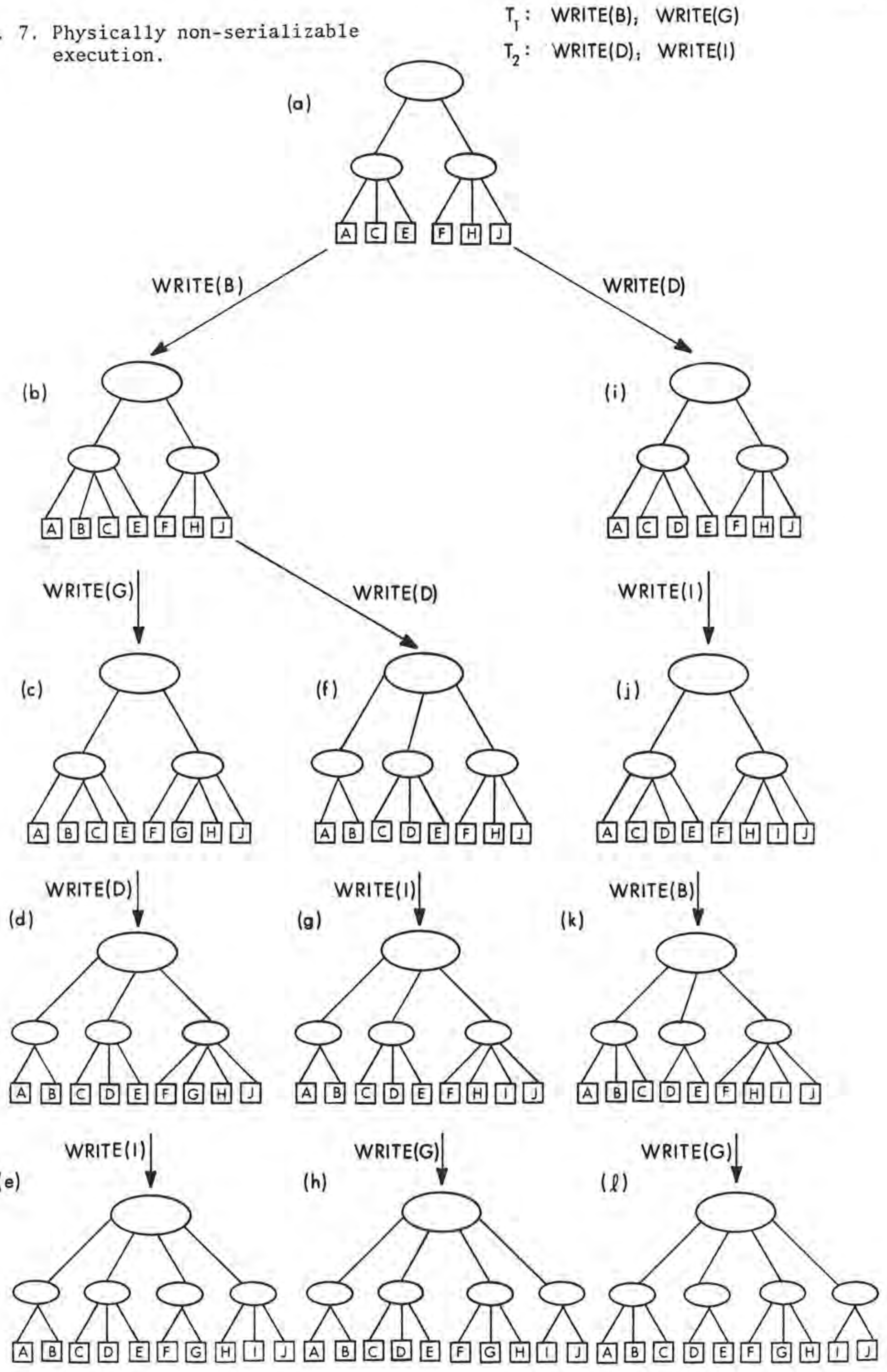


Fig. 6. Long-term and short-term locking.

Fig. 7. Physically non-serializable execution.


T_1: WRITE(B); WRITE(G)
T_2: WRITE(D); WRITE(I)
(a)
A C E F H J
WRITE(B)
WRITE(D)
(b)
A B C E F H J
(i)
A C D E F H J
WRITE(G)
WRITE(D)
WRITE(I)
(c)
A B C E F G H J
(f)
A B C D E F H J
(j)
A C D E F H I J
WRITE(D)
WRITE(I)
WRITE(B)
(d)
A B C D E F G H J
(g)
A B C D E F H I J
(k)
A B C D E F H I J
WRITE(I)
WRITE(G)
WRITE(G)
(e)
A B C D E F G H I J
(h)
A B C D E F G H I J
(ℓ)
A B C D E F G H I J