Concurrency Control of Large Unstructured Data

Gopi Attaluri  
Dept. of Computer Science  
University of Waterloo  
Waterloo, Ontario  
Canada N2L 3G1  
gkattalu@bluebox.uwaterloo.ca

Jacob Slonim  
Dept. of Computer Science  
University of Waterloo  
Waterloo, Ontario  
Canada N2L 3G1  
jslonim@uwaterloo.ca

Paul Larson  
Dept. of Computer Science  
University of Waterloo  
Waterloo, Ontario  
Canada N2L 3G1  
palarson@uwaterloo.ca

Abstract

Current research projects and database product extensions reflect the importance of unconventional database applications (for example, document processing and computer-aided design). Unconventional applications differ from the conventional ones (for example, airline reservation) in the nature of transactions and data. The transactions may be complex, long and involve human interaction. The data includes small, structured objects as well as large, unstructured objects. The unstructured objects are stored in and managed by unstructured data base management systems (UDS). A UDS employs coarse-granularity locking which limit the concurrency and reduce the throughput of transactions.

This paper deals with this problem of coarse-granularity locking at a UDS. It proposes and investigates the logical concurrency control (LCC) concept as a solution. In LCC, a software module sits on top of the UDS and offers a higher-level service. This module employs application-domain-specific fine-granularity locking to achieve high concurrency.

1 Introduction

There is a growing number of unconventional applications making use of the database technology as illustrated by the following fictitious example. XYZ Associates is a civil engineering firm that designs factory buildings. The design of a factory is made up of three kinds of data: engineering drawings, alphanumeric data, and text documents. Engineering drawings depict the architecture of factory buildings and are represented, stored, and managed as two-dimensional images. The alphanumeric data describes the people designing the factory and the factory design itself. The text documents detail the design procedures followed and comments on the design by experts.

Engineers access the three kinds of data in designing a factory. To change a design, they modify the engineering drawings, carry out an analytical and simulation-based analysis of the new design, and incorporate the results and observations in the alphanumeric data and text documents.

The database requirements for this unconventional application differ from conventional applications, such as airline reservation and banking. We list four important differences which concern the nature of application programs and the conceptual unit of data (data object). First, a factory design application program involves human interaction; this leads to conversational transactions. Furthermore, there are several steps in an application program (interactive modification of engineering design image, analytic and simulation-based analysis); therefore, the duration of the application program (transaction) tends to be long (possibly minutes).

Second, data objects do not conform to a fixed format. Two drawings, representing different buildings in the factory, could not be expected to conform to a fixed format or structure; a text document may have structure, but the structure is not uniform across multiple documents. For example, in text documents with a chapter-section structure, the number of chapters could vary between two documents, and the number of sections could vary between two chapters within the same document. Throughout this paper the term unstructured data also includes non-uniformly structured data. For simplicity of exposition, we denote a database system supporting unstructured data as an unstructured database system (UDS) and a database system supporting structured data as a structured database system (SDS). A database system may support both structured and unstructured data, then it is an SDS as well as UDS.

1Unless mentioned otherwise, the term database system denotes database management systems, file systems, storage systems and any other software system storing and managing data.
Third, data objects vary in size and a typical data object could be much larger than a conventional application's. An engineering drawing image could take megabytes of storage. Small unstructured data objects pose no significant problems; however, large data objects raise concerns about managing the information such as storage, retrieval and concurrency control.

The final difference concerns the number of developed application programs. Unconventional applications such as this are relatively recent compared to conventional applications which means that there are fewer legacy applications. Therefore, supporting pre-existing applications, while employing new ideas and schemes in a database system, may not be as big a problem as it is in conventional applications.

1.1 Research Problem and Solution Investigated

Several DBMSs (database management systems) model and manage large, unstructured data as binary large objects (BLOBs) [1]. They employ coarse-granularity locking scheme, wherein the entire object is locked as a single granule. A “coarse-granularity” UDS achieves good performance when unstructured data objects are not shared much and when the access granularity (the amount of data accessed by a transaction per object) matches the object size. However, unconventional applications require transactions to share data. And, a transaction may usually access some parts, rather than an entire unstructured data object. Therefore, coarse granularity locks could increase the data contention and the response time (duration) of transactions. Since unconventional applications involve long-duration transactions, an increase in data contention is particularly harmful to their performance.

This paper addresses the problem of data contention in coarse-granularity UDSs. We investigate logical concurrency control (LCC) solution of supporting a higher-level transaction service on top of the UDS without modifying the UDS and its preexisting applications. The higher-level service is identical to the UDS in functionality, except that it supports fine-granularity subobject locking to reduce the data contention.

2 Logical Concurrency Control

Figure 1 illustrates the concept of LCC. The physical system is an abstraction of coarse-granularity UDSs. The physical transaction manager (PTM) is a component that manages transactions and offers the physical transaction service, with ACID\(^2\) properties, to the physical system’s clients. The PTM employs a strict two-phase locking (2PL) scheme, the granularity of locks being a data granule. It allows a physical transaction to withhold locks and continue after commit. This feature, also known as lock retention, is equivalent to creating a new transaction with the locks released by the committed transaction.

A data object is an application-domain-specific conceptual unit of information. Subobjects are possibly overlapping components of a data object, a subobject being the unit of information accessed by an application. An application domain thus imposes a structure on data granules that comprises data objects and subobjects. The mapping between data granules and data objects, and between data objects and their subobjects is application-domain-dependent and is external to the physical system. The physical system chooses the size of data granule based on factors such as overhead of concurrency control, degree of concurrency, ease of implementation, and generality.

A logical concurrency control manager (LCCM) is a software module located on top of the physical system. Each LCCM offers a logical transaction service, with the ACID properties, to clients from a particular application domain. An LCCM’s clients access common data and share a common notion of data objects and subobjects. The logical transaction interface allows applications to lock and access subobjects. In essence, an LCCM supports the data-object abstraction over the data granules. The logical and physical transaction services are similar except that they employ different granularities of concurrency control.

Clients accessing the physical system directly are called local clients and those accessing it indirectly called local clients

\(^2\)ACID stands for Atomicity, Consistency, Isolation and Durability.
through an LCCM are known as logical clients. The physical transactions submitted by a local client are referred to as local transactions, while those from an LCCM are called guardian transactions (or simply guardians). An LCCM implements logical transactions by employing guardians. A logical concurrency control scheme (LCC scheme) describes how the LCCMs carry out this task. It guarantees the following properties. (a) A logical transaction is correct — it satisfies the ACID properties. It is isolated from other transactions, be they logical or local. (b) An application 'behaves' the same way whether it is processed as a local transaction (by the physical system) or a logical transaction (by the LCCM). (c) Two data subobjects of a data object can be simultaneously accessed by multiple logical transactions in a conflicting manner. If multiple local transactions were to access the same objects, they would have to sequentially lock the data granule that the data object is mapped onto. If a data object is mapped onto multiple data granules, the data granules are locked once on behalf of all logical transactions concurrently accessing it. When multiple local transactions access the same data object, they have to lock each data granule individually, resulting in a higher number of physical locks. (d) No restrictions are imposed on data accessed by local and logical clients. A local transaction can access the data accessible to a logical transaction and vice versa. (e) An LCCM is like any other local client of the physical system and enjoys no special privileges. (f) Data currently used by local applications can be accessed by logical applications. The LCCM's transparency ensures that local clients will not be affected. (g) Multiple LCCMs, where each LCCM deals with a domain of data and/or applications, can co-exist.

2.1 An LCC Scheme

For simplicity we assume that there is a single LCCM runs on top of the physical system. The LCCM assigns each data object (and the corresponding data granule(s)) to a guardian; the guardian remains active when the object is accessed by one or more logical transactions. Table 1 describes how the LCCM maps logical transaction operations on to physical transaction (guardian) operations. For simplicity of presentation, we assume that each data object maps to a single data granule. In our notation, O_i identifies data object i, and D_i, the data granule it is mapped to. S_{ij} identifies subobject j of object O_i. The guardian managing O_i is denoted as G_i.

The mapping of logical locks on to physical locks is further illustrated in Figure 2. It shows three logical transactions LT_1, LT_2, LT_3 and a two-dimensional data object O_1 with subobjects S_{11}, S_{12} and S_{13}. S_{11} and S_{12} are write- and read-locked by LT_1 and LT_2, respectively. When LT_3 requests a read lock on S_{13}, the lock cannot be granted because it conflicts with the write lock on S_{11}. So LT_3 must wait until S_{11} is released by LT_1. However, the lock request does not conflict with LT_1's lock on S_{12} and LT_3 need not wait for its release.

Figure 3 explains the relationships from the lock operations of Figure 2. It shows an additional data granule D_2, data object O_2 and guardian G_2. Guardians G_1 and G_2 own physical locks on D_1 and D_2. There are three logical transactions (LT_1, LT_2, LT_3). The logical transactions have locked or requested locks on subobjects (S_{11}, S_{12}, S_{13}, S_{21}, S_{22}) as indicated by the dotted lines.

For our logical concurrency control scheme to be correct, the "integrated history" consisting of local and logical transactions should satisfy the ACID properties. Our LCC scheme is described in detail and its
Table 1: Mapping Logical Transaction Operations

<table>
<thead>
<tr>
<th>Logical Transaction Operation</th>
<th>Physical Transaction Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin/End Transaction</td>
<td>—</td>
</tr>
<tr>
<td>Lock Subobject $S_{ij}$</td>
<td>Guard $G_i$ created. It locks the data granule corresponding data object $O_i$, namely data granule $D_i$. $G_i$ reads $D_i$ from the physical system. If $G_i$ exists and has already locked and read $D_i$, above steps are not needed.</td>
</tr>
<tr>
<td>Read Subobject $S_{ij}$</td>
<td>—</td>
</tr>
<tr>
<td>Write Subobject $S_{ij}$</td>
<td>—</td>
</tr>
<tr>
<td>Unlock Subobject $S_{ij}$</td>
<td>If guardian $G_i$ has no other subobject locks to manage, abort it.</td>
</tr>
<tr>
<td>Prepare Transaction</td>
<td>Apply logical transaction’s operations on subobjects to the corresponding data granules through guardians. Prepare those guardians for commit.</td>
</tr>
<tr>
<td>Commit Transaction</td>
<td>Commit guardians of data objects accessed by the logical transaction. Withhold guardians’ physical locks on data granules and restart after commit. Release logical transaction’s subobject locks using the Unlock operation.</td>
</tr>
<tr>
<td>Abort Transaction</td>
<td>If logical transaction is prepared, guardians of objects accessed by it must have been prepared. Abort them and release their physical locks. Release logical transaction’s subobject locks using the Unlock operation.</td>
</tr>
</tbody>
</table>

Correctness is proved in [1].

3 Issues in Implementing the LCC Scheme

Modeling Subobjects

The choice of “subobject” in LCC is application-domain-dependent. Text documents serve as our example for unstructured objects in order to illustrate the issues in subobject locking. In its simple structure, a text document is a sequence of words. In a more complex structure, it is a hierarchy of components such as chapters, sections, paragraphs, lines, words and characters. A component is non-uniform in size and may occur one or more times. For example, a chapter has one or more sections.

A person accessing a simple text document can retrieve and modify any contiguous sequence of words, and each contiguous sequence may be considered as a subobject. When two or more people simultaneously access the document, their subobjects may overlap. The subobject locking mechanism has to ensure the correctness of their transactions in such cases. In the case of complex text documents, a user can either access a component (chapter, section, paragraph, line, or word), or a contiguous sequence of words spanning chapters, sections, and paragraphs (for example, from the start of paragraph 4 of section 2 to the end of paragraph 10 of section 5). In either case, the semantics of subobject locks must be defined such that (a) two locks conflict one of them is a write lock and either the locked subobjects overlap even by a single word, and (b) a user is able to lock any subobject provided there are no conflicting locks.

We consider three methods to model subobjects to satisfy these constraints. The first and simplest method is to identify the smallest granularity component in the object, and make it the lock granule. A subobject is then modeled as a set of lock granules. When a subobject lock is requested, each member of the set is locked. In the case of text documents, the smallest data granule may be a word and subobjects may be modeled as sets of words. A subobject lock is implemented by locking the corresponding words.

This method allows for subobjects of arbitrary size (in terms of number of smallest granularity components) and a great degree of concurrency, but the management overhead of locking smallest granularity components is unacceptable in most cases.

The second method is to employ the MGL protocol for hierarchically structured objects. An unstructured object is decomposed into a hierarchy of finer components. The granularity of locks is refined as we move down from the root to leaves. The MGL protocol tries to reduce the lock management overhead by locking a parent node when many of its children are to be accessed. This method has the following limitations. (a) There may or may not be a natural hierarchical decomposition for an unstructured object. A com-
plex text document may lend itself naturally to such a decomposition as chapters, sections, paragraphs and words, but how does one handle simple text documents or multidimensional images? (b) The multi-granularity hierarchy is predetermined before processing the first lock request. The subobjects requested by users may not be compatible with any node of the hierarchy. (c) It is non-trivial to handle the smallest granularity components (leaves). If a leaf is to contain multiple components (for example, multiple paragraphs), it will limit the concurrency. Alternatively, if each component is made a leaf (that is, a lockable unit), the number of leaves and the height of the hierarchy will both increase. This will in turn increase the lock management overhead. (d) It is not clear how many children each parent node in the MGL hierarchy must be divided into. Ries and Stonebraker [2] argue that each parent node must be divided into a large number (for example, 1000) of fine granularity child nodes. Ng and Hung [3] report that a parent node in a MGL hierarchy introduces contention among transactions because the parent node must be locked to lock any of its children. They report that when a parent has more than 20 children, the contention on the parents nullifies the gains from increased concurrency.

Subobject locking is complicated because of two factors. First, the lock space is potentially large; it is the set of all possible subobjects. Its size is determined by the smallest granularity supported, but is significantly larger than the number of subobjects of the smallest granularity. Second, the lock conflict relationship is complex because subobjects are not mutually exclusive. A subobject can either partially or fully overlap another subobject. A subobject lock can conflict with the lock of any overlapping subobject. As a result, checking of lock conflicts is involved.

These factors can be addressed by a third method wherein subobjects are modeled as single- or multidimensional ranges. For example, in a document with chapter, section, and paragraph, the range (1.2.3, 4.5.6) indicates a subobject that starts at the paragraph 3, section 2, chapter 1, and ends before paragraph 6, section 5, chapter 4. Though each boundary of the range is made up of three parts (chapter, section, and paragraph), we are essentially dealing with a single dimension here. A lock on a subobject is in essence a lock on the corresponding range of words of the document.

Range-based subobject locking is applicable to complex and simple text documents and multidimensional images. Lock granularity corresponds, dynamically, to the subobjects accessed by applications.

Therefore, no incompatibility arises between the lock granularity and subobject granularity. Large objects are not pre-decomposed into a hierarchy of components. And objects need not be (hierarchically) structured. Multiple locks on smaller ranges can be escalated to a lock on a single enclosing range to reduce the number of locks. Alternatively, a range lock can be de-escalated into locks on smaller ranges to release part(s) of the original range. The cost of range locking is determined by the number of current subobjects that overlap with the subobject being locked. It is independent of the size (that is, number of words) of the document. In comparison, the cost with MGL depends on the depth of the object's granularity hierarchy, and is independent of the number of current subobject locks on the object.

In our research, subobjects are modeled as multidimensional ranges [1]. A subobject lock is implemented as a lock on the corresponding range. Range locking is applied to two types of data: complex structured text documents and multidimensional images.

Mapping Logical Transactions, Guardians, and Objects

Several choices exist in associating guardians with data objects. On one extreme, a separate guardian may be associated with each object. In this case, a logical transaction maps to multiple guardians, as many as the number of objects accessed. This approach has the following strengths and weaknesses. A logical-transaction commit requires multiple physical-transaction commits — as many as the number of objects accessed. The commit overhead may be reduced by read-only and group-commit optimizations. Guardian failures are isolated. If a guardian aborts and loses its locks, only the logical transactions accessing the corresponding object have their logical locks preempted. Two logical transactions are not affected by one another's failure if they do not access any common object(s). When a single threaded guardian blocks for a physical lock on a data granule, only the logical transactions requesting locks on subobjects mapping to that granule block.

Guardians and objects can be mapped using other approaches.

- Assign a single guardian for all objects: A single multi-threaded guardian can manage all the objects, and fulfill lock requests from logical transactions by obtaining the necessary physical locks. It prepares and terminates logical transactions sequentially. At commit time, it withholds physical locks on objects that are currently accessed by other logical transactions. It may employ opti-
mizations such as read-only and group-commit to reduce the physical transaction commit processing overhead.

This approach has the following advantages and disadvantages: A logical transaction commit requires a single physical transaction commit. The physical transaction commit overhead is thus the same as that of implementing a logical transaction simply as a physical transaction. Guardian failures are exposed. If the guardian aborts and loses its locks, all current logical transactions are affected. Two logical transactions are affected by one other’s failure even if they have no objects in common. The guardian must be implemented by a multi-threaded process, where each thread can request and block for a physical lock independently. Otherwise, when the guardian waits for a physical lock, any logical lock requests on objects not currently locked by the guardian have to be blocked. Sybase libraries are thread-aware. Sybase allows multiple threads to work on a single transaction.

- **Assign multiple guardians to each object:** This approach would introduce contention and blocking for physical locks among the guardians.

### 2PC Support for Physical Transactions

In the LCC scheme, guardian transactions associated with a logical transaction need to be committed with the 2PC protocol to guarantee the atomicity of the logical transaction. For this, the PTM must support the 2PC protocol. This requirement is not unreasonable.

Even if 2PC is not supported by the physical system, LCC can be employed. However, the atomicity of a logical transaction associated with multiple guardians cannot be guaranteed; some of these guardians may abort whereas others commit successfully. The atomicity can be guaranteed if the logical transaction is associated with a single guardian.

### Logical Transaction Deadlocks

Logical transactions may be blocked for logical locks. This may cause a deadlock — a cycle of logical transactions, wherein each awaits a lock from its predecessor. Such deadlocks can be handled by prevention (through partial-ordering of subobjects, for example), avoidance or detection.

4. A message by Andrew Hastings of Transarc Corporation to the info-engrns mailing list on October 19th, 1995.

5. The X/Open standard for distributed transaction processing specifies that DBMSs support 2PC [5].

### Logical Transaction Dependencies

When multiple logical transactions access an object O_i simultaneously, they effectively share the guardian’s (G_i’s) physical lock on the corresponding data granule. This introduces some dependencies among the logical transactions. For example, if two logical transactions write to the object, their prepare and commit operations have to be executed sequentially. [1]

### Support for Physical Lock Withhold

The LCC scheme assumes that a physical transaction can withhold its locks after commit. Lock withhold is supported by Rdb [6] and ONTOS[7].

VERSAN also supports lock withhold. A restricted form of lock withhold is supported by System R [8] and DB2/6000. The lock withhold feature is important in a situation where several logical transactions modify a data object O_i simultaneously. Without this feature, the guardian loses its physical lock when one of the transactions completes and has to be committed. The logical locks of the remaining logical transactions are preempted as a result and the transactions have to be aborted unless the locks can be restored.

### Optimizing Read-Only Logical Transactions

The guardians need only hold their physical locks on the data granules until all the logical transactions accessing the corresponding objects either commit or abort. Guardian commit or abort is not required when a read-only logical transaction commits or aborts.

### Impact of Guardian (Physical Transaction) Aborts

If a guardian is aborted (either by the LCCM or the PTM) and loses its physical lock on a data granule, the current logical locks on subobjects of the corresponding data object are lost (preempted) and need to be restored when the physical lock is re-acquired later on. Fortunately, guardian aborts may be rare as we explain below.

A guardian is aborted by the LCCM only in one case: when a prepared logical transaction LT_i aborts, all the guardians that modified data on its behalf are aborted. The PTM may abort a physical transaction (which may be a guardian) for several reasons: (1) it is involved in a deadlock, (2) site failure, (3) the transaction has requested an abort, (4) the transaction has violated a database constraint, and (5) software failure. A guardian is never deadlocked as it accesses only one object, so it cannot be aborted by a PTM using the detection approach. Site failures cause an

6. Lock withhold is called lock carry-over in Rdb.
abort of all active (neither prepared nor committed) physical transactions and are disastrous (from a performance point of view) for any DBMS. In practice, site failures are infrequent (compared to the transaction rate). The LCCM can control reasons (3) and (4). Aborts from reasons (3) and (4) can be minimized by carefully mapping the logical transactions’ operations onto the guardians. Software failures are uncommon, akin to site failures.

One-phase Termination of Logical Transactions

A majority of logical transactions may require only one-phase termination, that is, they commit without first preparing. Such transactions can reduce the number of guardian (physical) transaction commits and improve performance. One-phase terminating logical transactions that access a particular set of data objects can be grouped and committed together [4].

When the entire group is completed (finished reads/writes and ready to be committed), it can be committed together through the guardians. However, if a site failure occurs before the group completes, all members are aborted and restarted. The work of transactions that were completed and were waiting for others is lost. Alternatively, a completed subgroup can be committed through the guardians. If the commit succeeds, the rest of the group continues normally. If it fails, the guardians are aborted and lose their physical locks. The currently incomplete logical transactions have to be processed accordingly.

Logical Transaction Aborts

In the LCC scheme described, a logical transaction can be aborted by the LCCM only if it is either in a deadlock or suspected to be in one. So the work wasted because of a logical transaction abort is indicated by the frequency of deadlocks. With an efficient deadlock handling approach (detection or avoidance), the number of logical transaction aborts can be minimized.

Logical Lock Preemption and Restoration

A guardian may occasionally lose its physical lock on a data granule, and effectively preempt all the logical locks on that data granule. The following are a few cases where a guardian may lose its physical locks. (i) Guardian may be involved in a deadlock. (ii) Guardian may timeout. (iii) A prepared logical transaction aborts because it is part of a larger distributed transaction and the two-phase commit coordinator has resolved to abort. When the prepared logical transaction aborts, every guardian that has modified data and prepared on behalf of that logical transaction is aborted. (iv) Guardian’s attempt to withhold a physical lock at commit may fail. If so, the physical lock is lost (released) when the guardian commits.

In each of the above cases, when a physical lock is lost, the logical locks on the corresponding data granule are preempted.

There are at least two approaches to handle logical lock preemption. In the first approach, the LCCM aborts the victim and reexecutes it later. In the second approach, the victim is continued provided the (after being restarted) the guardian can reacquire the physical lock, and the preempted locks can be reallocated. For this approach to work, the reallocation must be transparent to the victim to maintain the correctness of the transaction history. Let the term restoration denote such a reallocation. In [1], we present a simple restoration scheme wherein preempted locks are restored based on the victim’s read/write operations prior to the preemption.

The abort approach is simpler and easier to implement. However, in an environment where guardians lose physical locks frequently, aborting the victim may waste significant work. In contrast, the victim’s previous work is salvaged in the restoration approach. A drawback is that restoration may not always succeed, and the abort approach may have to be followed.

4 Experimental Validation of LCC

We developed a prototype system to validate our LCC scheme. Two versions of the prototype system have been implemented: centralized and distributed. In the centralized version, the clients and server are implemented as a single process with multiple threads. In the distributed version, clients and server (physical system and the LCCM) are located on different machines on a local area network.

Owing to space constraints, we discuss only the distributed version of the prototype (Figure 4) and its results here. The prototype consists of several components such as ObjectStore, the physical system, the LCCM, DCE and Encina. ObjectStore was chosen because it resembles numerous OODBMSs in the transactional access to unstructured data. ObjectStore was also easily and immediately available for experiments. DCE mainly provides remote procedure calls (RPC) in the distributed environment, and multithreading of processes through its Pthreads. Encina is a transaction management toolkit; it manages transactions over the distributed environment.

We introduced a thin layer of software over ObjectStore to add three key extensions. They are not part of the LCC scheme; they merely fill the gaps between ObjectStore and the physical system model described in
Section 2. These extensions facilitate multi-threading, data object locking, and two-phase commit of physical transactions.

The operation of the LCCM follows the LCC scheme described in Section 2. Guardians implement logical transactions' operations on the physical system. They are dynamically started and terminated by the LCCM based on logical transaction’s operations.

The number of guardians is a system parameter and can be configured anywhere from one to the number of database objects. Database objects are statically and uniformly partitioned among the guardians. The number of objects managed by a guardian equals the number of database objects divided by the number of guardians. In the results reported below, we configure a single guardian for all objects.

The experiments deal with two types of data: binary images and text. Binary images are unstructured two-dimensional byte arrays. Text data is non-uniformly structured. It consists of objects with one- and two-dimensional structure. A one-dimensional text object is made up of a list of paragraphs. A two-dimensional object is divided into sections. Each section is then divided into paragraphs. The two-dimensional text data has been taken from Unix man pages. One-dimensional text and binary images were artificially generated for these experiments.

Image data objects are stored and managed as one-dimensional binary large objects or BLOBs. A subobject of an image object is an arbitrary two-dimensional segment in the object. A subobject is stored and managed as a one-dimensional binary large object (BLOB).

4.1 Summary of Experimental Results

Table 2 summarizes the performance of our LCC scheme with logical transaction response time as the performance metric. Each row of the table belongs to a different experimental parameter and compares the performance of the LCC session (where the LCCM is present and serves logical clients) and non-LCC session (where the LCCM is absent and the logical clients go directly to the physical system) in the corresponding experiment. The first two columns describe the parameter and the range of values it takes in the experiment. The third column indicates whether the LCC session is better/worse than the non-LCC session initially (that is, at the initial value of the parameter). The fourth column specifies whether the LCC session gains or loses over the non-LCC session as the parameter’s value increases. The fifth column specifies a break-even value of the parameter, at which the LCC and non-LCC sessions perform equally; values in this column are rounded to integers. A “-” value in this column means that the experiment does not indicate the existence of a break-even value. A value such as “> 36” means that a break-even value may exist and is larger than the highest value considered in the experiment, 36.

Deciding Factors in Deploying LCC

We also conducted experiments to study other aspects of performance such as data contention and frequency of aborts (from deadlocks and otherwise). In summary, the our results show that LCC is increasingly beneficial as the following “positive” factors dominate.

- **Data objects processed**: Data contention increases with the number of data objects processed by a logical transaction. It is better controlled by employing LCC.
- **Percentage of write operations**: Data contention also increases with the percentage of write operations of a transaction and necessitates LCC.

---

7In a majority of our experiments, local transaction response time is only marginally increased by the introduction of LCC. Hence, we here focus on logical transaction response time instead of an overall (logical and local) transaction response time.
Table 2: Logical Transaction Response Time Summary

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Parameter Value Range</th>
<th>LCC vs. Non-LCC Initially</th>
<th>LCC gains/losses vs. Non-LCC</th>
<th>Break-even Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subobject size (Object size 24 units)</td>
<td>1-8</td>
<td>better</td>
<td>loses</td>
<td>8</td>
</tr>
<tr>
<td>Percent write operations</td>
<td>0-45</td>
<td>equal</td>
<td>gains</td>
<td>0</td>
</tr>
<tr>
<td>Subobjects pro. per object</td>
<td>1-10</td>
<td>better</td>
<td>loses</td>
<td>8</td>
</tr>
<tr>
<td>Percent induced aborts</td>
<td>0-20</td>
<td>better</td>
<td>loses</td>
<td>19</td>
</tr>
<tr>
<td>Logical transaction thinktime (sec)</td>
<td>0-70</td>
<td>better</td>
<td>gains</td>
<td>-</td>
</tr>
<tr>
<td>Objects processed by logical transaction</td>
<td>1-18</td>
<td>equal</td>
<td>gains</td>
<td>1</td>
</tr>
<tr>
<td>Logical data sharing</td>
<td>0-27</td>
<td>worse</td>
<td>gains</td>
<td>1</td>
</tr>
<tr>
<td>Num logical clients</td>
<td>1-10</td>
<td>worse</td>
<td>gains</td>
<td>2</td>
</tr>
<tr>
<td>Local-logical data sharing component (Preempt policy)</td>
<td>0-100</td>
<td>better</td>
<td>loses</td>
<td>&gt; 100</td>
</tr>
<tr>
<td>Num local clients</td>
<td>4-36</td>
<td>better</td>
<td>loses</td>
<td>&gt; 36</td>
</tr>
</tbody>
</table>

- **Logical data sharing**: Logical data contention increases with this, making LCC more attractive.
- **Logical clients**: The processing overhead of LCC per logical transaction decreases as the number of logical clients increases. Furthermore, the number of instances where LCC pays off (that is, facilitates multiple logical transactions to access a data object concurrently through subobject locking) increases with this number.
- **Logical transaction thinktime**: As logical transaction thinktime (idle time between successive read/write operations) increases, the duration that a logical transaction is blocked, in case of a lock conflict, also increases. Therefore, the importance of concurrency increases with logical transaction thinktime.

LCC increasingly suffers as the “negative” factors listed below dominate.
- **Amount of data object processed**: Fine-granularity subobject locking is useless when a transaction accesses 30% or more of a data object. Any gain in concurrency is overshadowed by the subobject lock management overhead in such cases.
- **Size of subobject**: Subobject locking does not significantly increase concurrency when subobjects are coarse (say, a subobject is 30% or more of the object). The overhead of LCC is unjustifiable with such subobjects.
- **Local-logical data sharing**: When logical and local clients share and access the same data, local-guardian physical lock conflicts may arise. As this data sharing increases, the frequency of logical lock preemption (because guardians lose physical locks) and logical transaction aborts increases.
- **Local clients**: They increase the number of local-guardian conflicts.
- **2PC protocol failures**: When a prepared logical transaction aborts, each guardian prepared on its behalf needs to be aborted (unless it is read-only). This means that the logical locks of logical transactions associated with the guardian are preempted. The locks have to be restored, failing which the logical transactions need to be aborted.

5 Related Work

Significant work has been done to improve concurrency for complex data and applications. In the nested transaction model ([19]), a transaction is divided into several subtransactions, each of which may further be
divided. The subtransactions may be executed concurrently. Top-level transactions are isolated from one another. A subtransaction is isolated with respect to other subtransactions at the same level of nesting in its parent. Moss proposed a locking scheme to implement this isolation. Hadzilacos and Hadzilacos ([10]) applied the nested transactions to object-oriented databases.

In the multi-level transaction model of [11], a transaction consists of a set of actions. An action is further divided into lower-level actions, until the lowest level (say, reads and writes) is reached. A conflict predicate associated with each level determines whether any two actions of that level conflict. It is assumed that if two actions of level i conflict, there must be two conflicting level i−1 actions that they are made of. He described a concurrent control scheme to support a correctness criteria called level-by-level serializability. The above assumption is absent in the transaction model of ([12]). Instead, it is assumed that the serialization order of two higher-level actions must be consistent with the order of their sequential execution.

Our work is novel for the following reasons. We support ACID transactions. We accept the existence of several relational databases that use row or page locking (even if a row contains a large object) and assume in our model a coarse-granularity locking physical system, and pre-existing local transactions that use this locking. Finally, we focus on decomposing a large object improve concurrency, not exploiting the semantics of the operations in transactions.

6 Conclusions

We considered a coarse-granularity locking physical system storing large unstructured data. We proposed the LCC concept to facilitate improved concurrency without modifying either this physical system or the existing applications. We described an LCC scheme and discussed some of the important issues in implementing the scheme. We analyzed the scheme empirically to address the important question of whether it could improve the concurrency at acceptable processing overhead. Our results showed that the scheme improves both the concurrency and response time significantly, indicating that the processing overhead is relatively small. Owing to space limitation, we left out the proof of correctness of our scheme; it may be found in the thesis report ([11]). The thesis report describes the LCC concept in detail and discusses the LCC scheme elaborately and describes the experimental environment and procedure that we used to study the LCC scheme.

References


