# Locking Key Ranges with Unbundled Transaction Services

David Lomet
Microsoft Research, Redmond, WA, USA
lomet@microsoft.com

Mohamed F. Mokbel
University of Minnesota, Minneapolis, MN, USA
mokbel@cs.umn.edu

## ABSTRACT

To adapt database technology to new environments like cloud platforms or multi-core hardware, or to try anew to provide an extensible database platform, it is useful to separate transaction services from data management elements that need close physical proximity to data. With "generic" transactional services of concurrency control and recovery in a separate *transactional* component (TC), indexing, cache and disk management, now in a *data* component (DC), can be simplified and tailored more easily to the platform or to a data type extension with a special purpose index. This decomposition requires that details of the DC's management of data be hidden from the TC. Thus, locking and logging need to be "logical", which poses a number of problems. One problem is the handling of locking for ranges of keys. Locks need to be taken at the TC prior to the records and their keys being known to the TC. We describe generic two approaches for dealing with this. (1) Make a "speculative" visit" to the DC to learn key values. (2) Lock a "covering resource" first, then learn and lock key values and ultimately release the covering resource lock. The "table" is the only logical (and hence known to the TC) covering resourse in the traditional locking hierarchy, but using it limits concurrency. Concurrency is improved with the introduction of new partition resources. We show how partitions as covering resources combine high concurrency with low locking overhead. Using partitions is sufficiently effective to consider adapting it for a traditional database kernel.

## 1. INTRODUCTION

### 1.1 Overview

Evolving database systems to deal with the new environment of cloud computing and multi-core hardware requires carefully re-thinking their architectural decomposition. This was already an issue for extensible database systems where new types with their own unique forms of indexing are been added by means of "data cartridges" , but are not provided the same concurrency control and recovery properties as the natively supported types of data.

In a recent paper [11], an architecture is proposed in which the transactional services of concurrency control and recovery are separated from the data management services involving indexing, cache, and disk management. That paper suggested a decomposition of a database kernel in which concurrency control and recovery are encapsulated in a transactional component (TC) with data services then placed in a data component (DC). These components are configured into a system in which the DC is the inner component that acts as a server to the TC. All requests for transactional access to stored data proceed first to the TC. After the TC performs appropriate concurrency control and recovery actions, it passes requests to the DC where the desired data management functions are executed. Figure 1 illustrates this architecture.

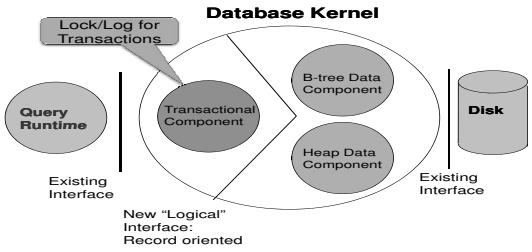



**Figure 1: Unbundled Services**

This architecture introduces a set of new problems in how one realizes transactional services. The TC has an "arms length" relationship to the DC, via a carefully constrained interface that is designed to hide the details of data indexing and how the data is mapped to physical devices. Because of this, the TC does not know how data is stored and so cannot exploit physical concepts such as pages when it performs concurrency control and recovery. Some of the recovery issues of this architecture were explored in [11], where a new recovery approach was described. There are analogous issues to be dealt with for concurrency control as well. This paper studies these issues, first carefully examining and circumscribing where new technology is needed, and then describing two generic ways of tackling the remaining problems. One of these approaches is then further elaborated and extended to produce a very effective solution with respect to both performance and concurrency. This is done within the context of lock based concurrency control, but many of the considerations touched on here apply to optimistic concurrency control as well.


*VLDB '09*, August 24-28, 2009, Lyon, France

## 1.2 Lock Based Concurrency Control

### 1.2.1 Simple Situations

Database management systems typically use *locking* for concurrency control, and support several transaction isolation levels. Providing lock based concurrency control for many of these isolation levels is straightforward. A request comes to the TC from the query processing (QP) component of the system. The request contains the identity of the record for which an operation is desired (read or update). For "read" operations, the TC takes a *share* mode lock on the identified record, and a read request is sent to the DC for the record. The lock is held before the requested operation is executed at the DC and is not released until sometime after the request is finished. Hence the resource is protected during access. For "write" operation, the TC takes an *exclusive* mode lock on the identified record, and requests to read and update the record. The lock is held before the requested operation is executed at the DC and is not released until after the record is read and updated. The DC returns to the TC the before and after versions of the record, which the TC uses to construct a redo/undo log record, which is then posted to the log while the lock is held.

Being provided with a record identifier in the data access request permits the TC to maintain its lock based on this record identifier, which is a "logical" entity that does not contain any physical location information. This is exactly what we need to effectively separate TC from DC. In conventional locking protocols, a lock is needed on a resource that protects the resource from conflicting accesses for the duration in which conflicts are possible. This is typically for the duration of the transaction, with strict two-phased locking usually used so as to guarantee that transactions are abortable. Of course, such an abort guarantee does not apply to read locks, and some isolation levels, e.g., read committed, which permits early dropping of share mode locks.

### 1.2.2 Locking Ranges

Separating transaction services from data management requires a more adventurous departure from existing technology in some cases. Relational databases permit a query to select a set of tuples based on a predicate. The predicate is not necessarily expressed in terms of a specific key or set of keys. Sometimes there is no substitute but to a full table scan for the predicate, in which case a table level lock is appropriate. However, database systems work hard to reduce the concurrency impact of query predicates. So such predicates are frequently mapped to key range scans of a table by finding a key range conjunct of the predicate.

Key ranges are locked by locking the records of the range and applicable to *primary and secondary* keys clustered by a B-tree [1, 2, 3, 4, 6, 13, 9]. But here a problem arises when we separate concurrency control from data management. Our separate TC may know the end points of the key range, but it does not know the keys of the specific records currently in the range prior to actually reading records by sending requests to the DC. And if the TC were to naively request records (with their keys) in the range prior to locking them, this would undermine the fundamental premise of lock based concurrency control, namely that records are locked prior to being accessed. If this premise is violated, race conditions exist that permit records of the range to change between an initial access and when they are locked.

Serializable transaction isolation level requires even more than that the records found in the range not be updated between when they are accessed and when they are locked. The selected key range set must not change until transaction completion. And this includes not just prohibiting changes to existing records, but also preventing new records from being added to the range for the duration of the transaction. Such inserted records are called "phantoms", in that they elude locking conflicts that only involve locks on the records of the range. For example, consider the following `UPDATE` transaction $T$ that aims to update the salary of all employees with IDs in the range [10,40].

`UPDATE` Employee `SET` Salary = Salary + 2000
`WHERE` ID $\geqslant$ 10 `AND` ID $\leqslant$ 40

In this case, the *lock manager* needs to grant transaction $T$ an exclusive X lock over all the records in the required range [10,40]. Consider the case where another transaction $T'$ inserts a new record within the range [10,40] before transaction $T$ has committed. If transaction $T$ runs under the `SERIALIZABLE` isolation level, then, it needs to lock not only the records with keys in the range but also the ranges between any two keys, e.g. using a *key range lock mode* [10, 12]. The main purpose of a key range lock mode is to prevent *phantoms*. Prior to a transaction inserting a new key and its record between two existing key identified records, the inserting transaction must lock one of these records in a mode which permits the insert to occur. An existing key range lock on the key prevents that insertion lock from being acquired until the range accessing transaction commits and releases its conflicting key range lock on the key.

Conventional *lock managers* rely on the traditional architecture of database management systems, which started with system R [7, 8], where the recovery, concurrency control, and access methods logic is tightly bound together in a monolithic storage engine. The lock manager, as part of the storage manager, has complete knowledge of the keys inside a given range because the point of execution when locks are posted is within the code accessing records of a page. And access to the page is protected (serialized), usually using a light weight lock called a latch.

Splitting the storage engine into TC and DC is a challenge for conventional lock managers when locking a range of keys is required. Consider the above example of transaction $T$ that needs to put range locks on all the records in the range $R = [10, 40]$. The TC, if it is to do the locking, needs to hold the necessary lock(s) before sending its request for the records in $R$ to the DC. But the TC does not know what keys are inside the range $R$. So, we face a dilemma where the TC has no idea about keys and their records that it should lock while the DC knows about the keys and records but has no idea about concurrency control issues.

One might propose that the TC request the keys of $R$ from the DC, one at a time. Then the TC can lock these records and subsequently submit the requested operation. However, that protocol does not work as the keys in the range may have changed between the time it were determined by the DC and the time that the locks were taken by the TC. An insertion might have occurred that changed the set of records in the range or an update might have occurred to one of the records. To be an effective locking protocol, the lock on a resource needs to be from a time before the resource is examined continuing until after all use of the resource is completed. This protocol simply does not do that.

## 1.3 Our Contribution

The problem we solve is to provide lock protocols that use a lock manager correctly in the context of the new architecture in which the transactional services of a TC are separated from the data services of a DC. We explored three workable alternatives that have very different performance versus concurrency trade-offs. Our preferred alternative, the [partition covering locks, provides an effective balance of performance and concurrency.

**Speculative Visit:** The main idea of the speculative visit protocol is that the TC makes two requests to the DC to retrieve each record in a range $R$. The first request, termed a *speculative* visit is an effort to get the next key of the range, which is then locked. The second request for the *same* next record, termed a *certification* visit is used to verify that the initial visit successfully identified the next key and the data of its record. If both visits return the same key, that is a success, otherwise, the process repeats. The *speculative* visit protocol has fine concurrency but it has the disadvantages of high execution overhead (two visits) and complexity.

**Table Covering Lock:** The main idea of the table locking protocol is to have the TC post a lock on a table prior to accessing records of the table via the DC. Under the umbrella of the locked table, which sits above records in the multi-granularity locking hierarchy, the TC can safely request records of $R$ from the DC. This has excellent performance as only a single lock is required. It has, however, a serious concurrency problem. An exclusive ($X$ *mode*) table lock prevents any other transaction $T'$ from accessing this table even if $T'$ is concerned with a totally different part of the table that does not overlap with $R$. To increase concurrency, we expand this protocol to include taking locks in an appropriate mode on the records returned, dropping the table lock as soon as we have finished locking the range. Thus, after we have finished with our locking phase, only the precise range that we need is protected.

**Partition Covering Lock:** We call our preferred approach *key partition* locking where we introduce a set of resources in the multi-granularity locking hierarchy called partitions, which are higher in the granularity hierarchy than records but lower than tables. The set of partition resources constitute a mathematical partition of the table, i.e., members of the set are disjoint and the union of all members covers the entire table. This gives us the opportunity to lock only the partitions of the table that we need, which provides much finer granularity than table locking provides, and hence better concurrency. *Partition* locking is not only relevant for unbundled transaction services but also for traditional database architectures, where it may provide better performance than conventional key range locking since the number of locks is greatly reduced. As with the table covering lock approach, we do record level locking, here with boundary partitions whose locks we then drop, in order to minimize the lock protected range.

**Paper Organization:** The rest of this paper is organized as follows: Section 2 gives needed background. Section 3 presents *table lock* and *speculative visit* techniques. The *partition* locking technique is described in Section 4. Section 5 compares performance of the proposed techniques. Finally, Section 6 provides a discussion and summary.

# 2. BACKGROUND

This section gives a background for the rest of the paper. The interface that we assume between transactional component and data component, the operations involved and how requests are executed are described. We then review transaction isolation levels, multi-granularity locking hierarchies, and locking modes.

## 2.1 TC:DC Interface

The TC:DC interface is sketched in [11],. Here we focus on the normal database requests, which are invoked in that interface using the **perform-operation** operation. This general purpose operation takes as an argument a description of the specific function desired. Among these functions are "update-record" and "read-record", both of which take a table name and key value to identify the record desired.

Since we are interested in range queries, the operation of interest is "read-next", which takes a table name, start key value, and perhaps an end key value and predicate used to filter records of the range. This reflects that the TC passes on to the DC operations that the SQL Server storage engine would have received in the SQL Server unified storage engine architecture. SQL Server permits an additional filter predicate to be attached to a range search request. This filter, executed at the lowest level of the storage engine, reduces (via the filter) the number of records that need to be passed up from the storage engine to the query processor, a valuable performance enhancement.

The "read-next" operation returns a record (perhaps a set of records) that are greater then the start key, less than the end key if provided, and that are not filtered out by the filter predicate. These records are passed through the TC back to the query processor runtime. The query processor runtime issues a sequence of these "read-next" calls as it scans through records of the range of interest to the query. Thus, this call is executed repeatedly as part of a query processor iterator examining a base table or index.

## 2.2 Isolation Levels

The *isolation level* for a transaction determines the transaction locking behavior. The SQL-92 standard provides four isolation levels. Ordered from the least to the most restrictive isolation levels, these are (1) `READ UNCOMMITTED`, (2) `READ COMMITTED`, (3) `REPEATABLE READ`, and (4) `SERIALIZABLE`.

Transactions executing under the `READ UNCOMMITTED` isolation level can read modified data values that have not yet been committed by other transactions. This is known as *dirty reads*. The `READ COMMITTED` isolation level prevents *dirty reads* by preventing transactions from reading modified data items that are not committed yet. However, it may result in a transaction seeing a different values for a data item read more than once. This is known as *non-repeatable read*. The `REPEATABLE READ` isolation level avoids the *non-repeatable read* problem by holding shared locks for the duration of the transaction. However, it does not prevent the *phantom* problem [5] described in Section 1.2.2. The `SERIALIZABLE` isolation level is the most restrictive isolation level that avoids *phantoms*. It must lock in such a way as to ensure that the predicates used in reading data remain correct for the duration of the transaction, hence ensuring serializable transactions. Our proposed locking protocols are applicable to all isolation levels.

## 2.3 Locking Hierarchy and Modes

Conventional locking exploits multi-granularity locking that allows a transaction to request and hold locks in a hierarchy of resources. Lockable resources are organized in a hierarchy where lower level resources are completely contained inside higher level resources in the hierarchy. A common resource hierarchy might, from the highest to the lowest, include *database*, *table*, *page*, and *record*. Locking a resource at a smaller granularity, e.g., *record*, increases concurrency but has a higher overhead because more locks must be held if many records need to be locked. On the other hand, locking at a larger granularity, e.g., *table*, may greatly reduce concurrency as locking an entire table restricts access to any part of the table by other transactions. However, it has a lower overhead because fewer locks are needed. A conventional lock manager deals with multi-granularity locking by supporting two distinct forms of lock modes, intention modes and explicit modes.

**Explicit locks.** Explicit locks can be placed on any resource in the lock hierarchy. The most common types of explicit locks are shared locks (S) and exclusive locks (X). A shared lock allows concurrent transactions to read the same resource while no other transaction can modify it. An exclusive lock prevents other transactions from reading or modifying the locked resource. An exception is in the `READ UNCOMMITTED` isolation level where a read operation is permitted even if there is an exclusive lock because the "dirty read" takes no locks.

**Intention locks.** An intention lock is placed on a resource higher in the resource hierarchy to signal an intent to place an explicit lock on a lower level resource that is contained within the higher level resource. The purpose is to block other transactions from using a lock of a higher level resource to access the whole resource in a way that would invalidate the lock at the lower level which protects part of that resource. For example, a transaction with an "intention" to read a record of a table places an intention shared (IS) lock on the table. The IS lock conflicts only with the X mode explicit lock and hence prevents a table from being locked for writing while another transaction is reading its records. Examples of intention locks include intention shared (IS), intention exclusive (IX), and shared with intention exclusive (SIX).

High concurrency requirements have led some systems to lock key ranges instead of entire tables when providing the `SERIALIZABLE` isolation level. A way of doing this without introducing additional resources into the resource hierarchy is by modifying the locking protocol used with the lock manager. Now, instead of a lock on a record resource locking just the record, a range lock locks both the record and the range between it and the next record (or between it and its predecessor). This is usually in the context of records ordered by a key, and when this is the case, it has been called "key range" locking. When a key range is a conjunct of a predicate used to select records read by a transaction, interpreting the lock on each key of the range as a key range lock prevents phantoms. Hence key range locks are used to provide the `SERIALIZABLE` isolation level. Note again that no new resource is introduced. Rather, an existing resource is expanded to include an adjacent range.

One can use existing lock modes to lock key ranges [12]. However, it is also possible, via new lock mode that pair the effects of prior modes, to enable additional concurrency while still preventing phantoms [10]. In this case, the lock mode is used to indicate whether the record, the adjacent range, or the record and the range are to be protected.

Systems sometimes introduce a page resource into the multi-granularity hierarchy. A page resource is between a table and a record in the resource hierarchy. A page lock protects all the records on the page, i.e. locks them. It provides a useful tradeoff of less locking overhead via fewer locks than record locking would require, while enabling much higher concurrency than a table lock. Page locks are not without difficulty however. Records may sometimes be reorganized so that their page changes, and this can add substantial complexity, e.g., when dealing with a B-tree page split. Further, with our unbundled architecture, the locking protocol is executed in a context (the TC) where it does not have any information about the page on which a record resides.

In this paper, we do not introduce new locking modes or replace existing ones. Instead, we show how to use conventional multi-granularity locking to correctly lock records when we split transaction services from data management. Further, we introduce a new lockable resource, termed a *partition*, that lies between the *record* and *table* resources in the locking hierarchy. Unlike page resources, our partition resource is strictly a logical resource. Then, we show how to efficiently use explicit, intention, and key range locking modes to lock the partitions so that the TC most of the time holds locks only on records in a desired very specific range.

# 3. LOCKING EXISTING RESOURCES

In a traditional integrated storage engine, the code that will submit lock requests to the lock manager has already accessed the page containing the records, and hence has knowledge of the records and their identifiers. Our unbundled TC executes without that prior knowledge of what resources it needs to lock and hence protect in a key range. Thus, it cannot directly request *record* granularity locks.

In this section, we present the details of two locking protocols that can be used to lock key ranges when transaction services are unbundled into a separate TC. Importantly, neither of these protocols requires that we introduce new lockable resources into the resource hierarchy. The first protocol, the *speculative visit* protocol uses existing record key locking, and has high concurrency, but it has not only locking cost but also additional overhead to discover the key values to lock. The second protocol, the *table lock* protocol, takes out a covering lock on a table and is very simple. It should have adequate performance, but potentially may have problematic concurrency.

## 3.1 The Speculative Visit Protocol

The main idea of the *speculative* visit protocol is to have the TC make a *speculative* visit to the DC to find the keys upon which to take out locks. The *speculative* visit is a request to retrieve a next record in the range $R$. After this *speculative* visit, the TC becomes aware of the record that it needs to lock; which was the missing information it needed. Now, the TC can lock the record read with the appropriate lock mode , which mainly depends on the isolation level. If the isolation level is `SERIALIZABLE`, the appropriate lock

mode is a key-range lock, otherwise, it is either an S or X lock depends on the required operation. However, the protocol thus far is not correct as the TC locks a record after it reads it and it may be the case that this record is modified between the time it is read and the time it is locked, or even that a record with a lower key value has been inserted between the prior record and the returned record.

To avoid such behavior, the TC follows its *speculative* visit and record locking with a *certification* visit. The *certification* visit is a second request from the TC to the DC to retrieve again the next record in key sequence that was retrieved in the previous *speculative* visit. If the retrieved record has the same key and did not change between the *speculative* and the *certification* visits, the TC considers reading this record as a success and proceeds to read and lock the next record in the requested range. If, on the other hand, the *speculative* and *certification* visits give different results, the speculative read is considered to fail. In this case, the TC treats the *certification* visit as a new *speculative* request and takes a lock on the new returned record followed by another *certification* visit. The TC repeats these alternating requests until the returned record from both the *speculative* and *certification* visits is the same. It then goes on to process and lock the successor records of the range in the same way.

As in all multi-granularity protocols, before locking a lower level resource, intention locks must first be taken on the higher level resources of the resource hierarchy. So the speculative locking protocol involves first taking the appropriate intention lock on the table resource (and higher resources if they exist). But in this case the lock is a pure intention lock (either IS or IX) which are compatible with other pure intention locks. These intention locks are required in any use of a multi-granularity resource hierarch.

Finally, there is no need to restrict requests to acquiring only a single record. One might speculatively request say the next ten records, lock them all in the desired mode and follow that with a certification request on these ten records. One can advance the scan through this batch of records up until a record, if one is present, where the speculative and certification visits produce different results. Finally, one can combine the speculative visit for a new batch of records with a certification visit for the prior batch. Such optimizations greatly reduce the number of times in which the DC needs to be called in executing this protocol. However, there is no way to avoid the transfer of all keys in the range across the TC:DC interface.

The operator used to access a range of records is an adaptation of "read-next". We need to extend "read-next" to return certified records together with speculative visit records. One way to do this is for "read-next" to include an additional argument that is the set of records to be certified. This is in addition to a start key and end key. (Note that we do not include a filter predicate as we need to lock all records in the range.) "read-next" returns with a "yes" (certified) or "no" (failed certification) indication, and a set of records following the last certified record returned. One then uses the highest certified record as the start record in the next invocation of "read-next". One can imagine variations on this that return, e.g., the number of certified records instead of "yes/no".

The *speculative visit* protocol has two main advantages: (1) It can be applied to resources already present in the multi-granularity resource hierarchy, and (2) It provides high concurrency as only the requested records are locked. However, a major disadvantage of the *speculative visit* protocol is its extra overhead where each record needs to be accessed at least twice before a read operation is confirmed. One can anticipate that speculative and certification visits will rarely differ as to their outcome. They are needed to protect against the infrequent case when a change occurs. An important negative, however, is that every record of the range needs to be fetched from the DC even when only some small number of the records might satisfy the full predicate of interest to the transaction. This can explode the number of times in which the TC:DC interface needs to be crossed.

## 3.2 The Table Lock Protocol

To ensure that a lock is held on a record before we access it, we look for a resource higher in the multi-granularity resource hierarchy of which the TC is aware and lock that instead. One resource above a *record* in the resource hierarchy is the *page*. However, the TC knows nothing about the pages involved in the requested range. Hence, even if it wanted to lock pages and deal with the complexity of physical data placement changing, the TC cannot exploit pages. The next resource in the hierarchy is the whole table, and that is a logical resource known to the TC. Thus, the main idea of the *Table Lock* protocol is to let the TC request to "temporarily" lock the whole table. Locking the whole table gives the TC a covering lock in which it can safely talk to the DC while it discovers the records inside the requested range. Then, the TC can read the records inside the requested range one by one. After reading each record, the TC would know that this record is inside the requested range. It can then, redundantly but temporarily, lock this record. Notice that locking this record is the goal of the TC, but could not be achieved until we had the table lock covering the records. Once the TC reads and locks all the required records, using record locks, it demotes the table lock into a weaker intention lock while retaining locks on the records in the range.

The *Table Lock* protocol is simple to implement with a very modest overhead. However, a major disadvantage of the *Table Lock* protocol is its low concurrency as it locks the whole table while it is initially accessing and locking records, even when it may need to access very few records. Other transactions that need to modify other parts of the table would be blocked during this period. The concurrency limitations of the *Table Lock* protocol suggest that we need smaller grained resources for our covering lock(s), which leads us to introduce our *Partition Lock* protocol in the next section.

# 4. LOCKING WITH PARTITIONS

In this section, we present our main locking protocol, termed the *partition* lock protocol, for locking a range of values $R$ with a lock mode $M$. The main idea is to introduce a new "logical" resource into the multi-granularity locking hierarchy between table and record. We call this new resource a *partition*. A *partition* can be considered as a logical replacement for *page* resources where a *partition* may include several pages and vice versa. Similar to tables and in contrast to both pages and records, the TC has full knowledge of the partitions of a given table as it is the TC that does the partitioning. Thus, the TC can request to lock a partition without having to contact the DC first.

## 4.1 The Partition Resource

Without loss of generality and for the simplicity of the discussion, we assume that a table is pre-partitioned into a set of $N$ equal ranges based on its key domain. The number $N$ of partitions can be used as a database parameter that achieves a trade-off between concurrency and locking overhead. The smallest value of $N$ is one where the whole table is considered as the only *partition* resource. In this case, the *partition* locking protocol degenerates to the *table* locking protocol described in Section 3.2. The largest possible value for $N$ is the cardinality of the key domain, though this is usually impossible to support in practice as there would then usually be many more partition resources than there are records, which makes no sense. The TC can map a key to its containing partition resource. Perhaps most simply, it can divide the key domain into a number of equal size subranges, where each subrange serves as a partition resource. We note that the locking protocol that will be described in Section 4.2 is orthogonal to the way that the table is partitioned. More sophisticated techniques for partitioning a table into a set of ranges to assign to partition resources will be described in Section 4.3.

## 4.2 The Locking Protocol

The main idea of the *partition* locking protocol is that the TC first maps the range $R = [k_l, k_u]$ requested by a query into the set of partition resources on which the TC can request locks. The TC can request a lock on the partition resource prior to accessing the records included in the partition resource. By first requesting a lock on a partition resource, the TC obtains a cover in which it can talk safely with the DC to discover the records inside this part of the range. This is the same idea behind the *table lock* approach described in Section 3.2 in which the TC obtained a table lock as a cover to be able to safely access the records in the requested range. The difference here is that we do this for only a small partition rather than the whole table. In case that the requested range $R$ is included in only one partition, this protocol ends up to be very similar to the *table lock* protocol. But the common case will be that $N$ is not one, and that it will require multiple partitions to cover the range.

When multiple partitions are required to cover a range, we lock them one at a time as we access records sequentially within $R$. We expect the common case to be that $R$ and the partition resources that cover it will frequently be only a small part of the entire key range in the table. Thus we expect that concurrency will commonly be much better than the concurrency provided by the table lock protocol as we only need to lock the partitions covering $R$ rather than the whole table. There are also additional opportunities to improve the locking protocol performance by reducing the number of locks required.

The TC distinguishes between two types of *partition* resources with respect to a requested range $R$, *boundary* partitions and *internal* partitions. A *boundary* partition is a partition resource that includes either the start key, $k_l$, or the end key, $k_u$, of the requested range $R$ and hence, for which the range $R$ only partially overlaps the partition resource and in particular, where the partition resource covers keys that are not in $R$. An *internal* partition resource is completely included within the requested range $R$, and hence all keys covered by an internal partition are in $R$. Considering the `SERIALIZABLE` isolation level as the most strict one, our protocol treats *boundary* partition resources and *internal* partition resources differently.

**Boundary Partition:** A boundary partition $B$ is first locked in the requested lock mode $M$. As we access records in $B$, we lock them using record locking, as we did using the *table lock* protocol. When we are finished with the records of $B$, we demote the lock on $B$ to the appropriate intention lock mode, leaving the records locked, usually until transaction end.

**Internal Partition:** An internal partition $I$ is locked in the requested lock mode $M$. We access the records of $I$ but do not lock these records. Rather, the $M$ mode lock on $I$ is retained until end of transaction. Since the entire set of records in $I$ need to be locked, the lock on $I$ already exactly matches the range of records we want to lock.

Intuitively, partition locking provides much higher concurrency than table locking and can reduce the number of record locks needed as well. There are, however, additional potential complications with which we need to deal.

**Preventing phantoms at the end points of the range.** When using key range locking, we need to ensure that the keys for the records in a range provide protection for an adjacent range. We do need to choose, however, which range a key range lock protects, the range between it and the next higher key, or the range between it and the next lower key. Given that a range search ordinarily returns with the first key that is in the range, it is convenient to have a key range lock on that key protect the range between it and the preceding key. In this way, we guarantee that a key range lock on the first key of the range will cover the start of $R$ that precedes this initial key. At the end of the range, we need the DC to return to us not only the keys of records in the range, but the first key past the end of the range, so that we can use a key range lock on that key to cover the part of $R$ that follows the last key.

**Indicating the end of the partition.** In the "likely" case where the requested range $R$ spans more than one partition, the TC needs to know when the record being accessed is within the next partition so that it can proceed to lock the next partition prior to accessing records whose keys are within it. One way (among a number) of dealing with this is to submit requests to the DC, each request being for a closed range of records of the single partition resource. When there are no more records within the partition, the DC indicates this by providing an end record (or other indication). At that point, the TC locks the next partition resource and proceeds to request records from the DC that are within the range covered by the next partition.

**Preventing phantoms between partitions.** We need our locks for the range $R$ to be "seamless", i.e., there is never a place in the range where a phantom insertion can be made. To accomplish that, we need to be careful about the interaction between boundary partition $B$ and the first internal partition $I_1$. When we partition $R$ so as to request a sequence of ranges

from the DC, we need to place a key range lock on the first key value in $I_1$ before demoting the partition lock on $B$. This prevents a second transaction from inserting a record after the current last record of $B$ but below the boundary of $I_1$.

**Dealing with empty partitions.** If any partition resource has no records from our range $R$, we need to hold the lock on this partition resource as long as we would need locks on the underlying records. For internal partitions, our locking protocol is unchanged. We simply lock the partition as usual, whether empty or populated with records. However, we must adapt our protocol for empty end partitions where we normally demote the partition resource lock to an intention lock because we have taken out record locks. An empty partition has no records for us to lock. Hence, for a final boundary partition, we retain the partition lock, treating an empty end partition like an internal partition. For an initial boundary partition, we continue to demote the partition lock to an intention lock, but need to ensure that the first record in the range has a key range lock to protect this early part of the range.

### 4.2.1 Pseudo Code

In this section, we present the pseudo code for the *partition* locking protocol code for locking a range of values $R = [k_l, k_u]$ with lock mode $M$ assuming the `SERIALIZABLE` isolation level as the most restrictive isolation level where phantoms are avoided. In general, the *partition* locking protocol goes through four main steps: (1) processing the first boundary partition, (2) processing all internal partitions, (3) processing the last boundary partition, and (4) preventing phantoms at the end of the range.

In the first step, the TC locates the first boundary partition resource $B$ that includes the lower range boundary $k_l$. Then, Algorithm 1 gives the detailed pseudo code for locking a boundary partition. The input to this part of the protocol is the partition number to lock $B$, the requested lock mode $M$, and the requested range $R$. Based on whether the requested lock mode $M$ is exclusive or shared, the TC locks the boundary partition resource with either an SIX or S lock mode, respectively. This is similar to the case of locking the whole table in the *Table lock* protocol, however, we do it here only for a partition resource. Once the TC has such an appropriate lock for the partition resource, the TC keeps requesting the records from the DC one by one. The TC locks each retrieved record with a key range lock that prevents phantoms. If the DC returns null instead of the retrieved record, this means that there are no more records in the DC in the current partition. With this null value returned, the TC knows that it needs to move to the next partition.

In the second step, the TC aims to process all internal partitions. The TC goes through this step only if there is at least one internal partition. Algorithm 2 gives the detailed pseudo code for locking $N$ consecutive internal partitions. The input to this procedure is the first boundary partition $B$, the number of internal partitions $N$, and the requested lock mode $M$. This step goes through all the internal partitions and locks them one by one using the requested lock mode $M$. For each locked internal partition, the TC reads all the records from the DC without locking these records. The only exception and complication in this step is for the TC to lock the first record in the internal partitions with a key range mode and then demotes the lock on the first boundary partition to an intentional lock. In this case, a key range lock is needed on this retrieved record to prevent phantoms between the last record in the first boundary partition and the end of the first boundary partition. Regardless of whether the first boundary partition is empty, or $R$ within this partition is empty, or not, this key range lock will protect the range starting at the immediately earlier key, regardless of where that key is. If there are records within $R$ that are in the boundary partition, this key will then block access to the records between the last key of the boundary partition and the first internal partition. The part of the table that is blocked will be a smaller than would be the case by retaining the partition lock except in the case where the entire partition is empty.

**Algorithm 1** Locking a Boundary Partition

```
1: Procedure ProcessBoundary (Partition B, Mode M,
   Range R)
2: if M is an exclusive lock (X) then
3:    Obtain an SIX lock on B
4: else
5:    Obtain an S lock on B
6: end if
7: r ← Read the first record in B that overlaps with R
8: if r = NULL then
9:    return
10: end if
11: repeat
12:    Lock r with key range lock
13:    r ← Read next record in B that overlaps with R
14: until r = NULL
15: return
```

In the third step, the TC locates the last boundary partition that includes the end boundary of the requested range $k_u$. If this partition is different from the first partition, i.e., the requested range $R$ spans more than one partition resource, then the TC performs the same exact procedure it did for the first boundary partition (Algorithm 1), i.e., locking the partition resource with either SIX or S mode, locking the keys inside the partition resource with a key range mode, and reports a flag that indicates whether this end range partition is empty or not.

Finally, in the fourth step, the TC aims to find and lock the first record after the requested range with a key range lock. The objective is to prevent phantoms at the end of the requested range $R$. This step would take place only if there is at least one record in the last boundary partition, otherwise, the TC would deal with this partition as an internal one, i.e., there is no need to demote its lock, hence, there is no need to look for the next key. In this step, the TC distinguishes between two cases: (a) The first record after the requested range is located in the final partition. In this case, the TC locks this record with a key range lock that covers the end of the requested range $R$. Then, the TC demotes the lock on the final partition to an intention mode lock. (b) The first record after the requested range is *not* located in the final partition. In this case, the TC does nothing as it will retain the explicit lock on the last partition. This lock already covers the end of the range $R$, and prevents phantoms at the end of the range. It is important to note that we could simply retain the partition lock in both cases, and that would work fine. However, case (a) permits us to provide more concurrency in the last boundary partition by

**Algorithm 2** Locking Internal Partitions

```
1: Procedure ProcessInternalPartitions (Partition B,
   Number N, Mode M)
2: FirstInternalRecord ← TRUE
3: for i =1 to N do
4:    Lock partition B + i with lock mode M
5:    r ← Read the first record from partition B_L + i
6:    if r ≠ NULL AND FirstInernalRecord = TRUE then
7:       Lock r with key range lock
8:       Demote the lock on B to an intentional lock
9:       FirstInternalRecord ← FALSE
10:   end if
11:   repeat
12:      r ← Read next record from partition B + i
13:   until r = NULL
14: end for
```

using key range locks that block access to only part of this partition instead of all of it.

#### 4.2.2 Example

Figure 2 gives a detailed example of the *partition locking* protocol where a shard lock (S) is requested on a range of values $R = [k_l, k_u]$ under the `SERIALIZABLE` isolation level. Figure 2a shows that the requested range includes five partition resources; two *boundary* partitions and three *internal* partitions. Figure 2b depicts the case of dealing with the first *boundary* partition where we initially hold the requested shared lock on the partitions resource, followed by reading and locking all the records inside this range with key range locking mode. A locked partition resource is presented by bold line while a record that is read and locked is represented by a black circle. As we are working in the `SERIALIZABLE` isolation level, ranges between consecutive records are also locked. Also, the range from the first record in the partition and the record before it, which is outside $R$, is locked (depicted as bold line). Figure 2c depicts the case of dealing with the first *internal* partition where we: (a) lock this partition using a shared lock mode, (b) read and lock the first record in this partition by the a key range lock mode (depicted by a black circle and solid line between it and the last record in the *boundary* partition), (c) demote the lock on the boundary partition to an intent lock, and (d) read the records inside the internal partitions without locking them (depicted by gray circles).

Figure 2d depicts the case where (a) all *internal* ranges are locked while reading their records, (b) we start to process the last *boundary* partition by holding a shared lock over it. Finally, Figure 2e gives the final locking result after processing the last *boundary* partition where (a) The records inside this partition are locked with a key range mode, (b) The first record that is outside $R$ is also locked with a key range mode, and (c) the lock mode on the final *partition* resource is demoted to an intent lock.

Figure 2f gives an example of the final result when there are special case partitions. In particular, both the first *boundary* partition and one of the *internal* partitions do not contain any record from the requested range $R$. Also, the final *boundary* partition does not include any records after the requested range. For these three special cases, we can notice that: (a) there is no difference in dealing with the empty *internal* partition as it will be treated as if it is a non-empty one, (b) the lock on the first empty *boundary* partition is demoted to an intent lock, while the first record in $R$ is locked with a key range mode that covers all the empty range of the first *boundary* partition, (c) the lock on the final *boundary* partition is not demoted to an intent lock as it does not include any record after the last record in $R$.

#### 4.2.3 Discussion

We have implemented the *partition* locking protocol inside Microsoft SQL Server Storage Engine. This requires that we add the "partition" resource to the set of resources that can be locked by a transaction. In the mean time, we did not make any change in the locking modes, the locking compatibility matrix, or the locking manager algorithms, i.e., we use the same available locking modes, the compatibility matrix between various locking modes still hold, and the way that a transaction locks a resource or waits for a lock to be released did not change. With that change, the *partition* locking protocol can also be applied to the case of conventional lock managers where everything is tightly integrated within the storage engine. In general, the newly introduced *partition* granule along with the *partition* protocol results in much higher concurrency than conventional lock managers where it provides a lockable resource whose size is between a single record and the whole table.

### 4.3 Range Partitioning

In this section, we discuss how to partition a table into a set of partition resources. This is an orthogonal issue from the locking protocol itself as the locking protocol is applied to any set of partitions. A table could be either partitioned into a set of *fixed* or *dynamic* partitions as follows:

#### 4.3.1 Fixed Partitioning

In this case, the whole space is divided into a fixed set of partitions based on the record space domain. For example, if the domain record key is from 1 to 1024, and we can accommodate 128 range resources, then all record keys between 1 and 128 lie on the first *partition* resource, records with keys between 129 to 256 lie on the second *partition* resource and so on. The fixed partitioning scheme has the advantage of being simple and easy to implement with no overhead. All is needed is a simple mapping function that maps a record key to its corresponding *partition* source. However, fixed partitioning may be very inefficient for skewed data distributions. A way to compromise this skewed data is to increase the number of partition resources, i.e., a partition would have less number of keys, however, this would result in maintaining more lock resources.

#### 4.3.2 Dynamic Partitioning

The main idea is to avoid skewed data distributions by adaptively changing the sizes of the partition resources so that each partition has similar number of records. This can be done in either a *space-dependent* or *data-dependent* way. With *space-dependent* partitioning, the main idea is to start by having only one partition resource that spans the whole space domain. Then, once the number of entries exceeds a certain number $m$, we split the partition resource into two halves so that each half includes less than $m$ entries. If it ends up that all entries are on one half, we split this half into another two equal halves recursively. Similarly, if two sibling partition resources have a total of less than $m$ entries, they can be merged to one range resource. With *data-dependent* partitioning, the main idea is to have the range resource

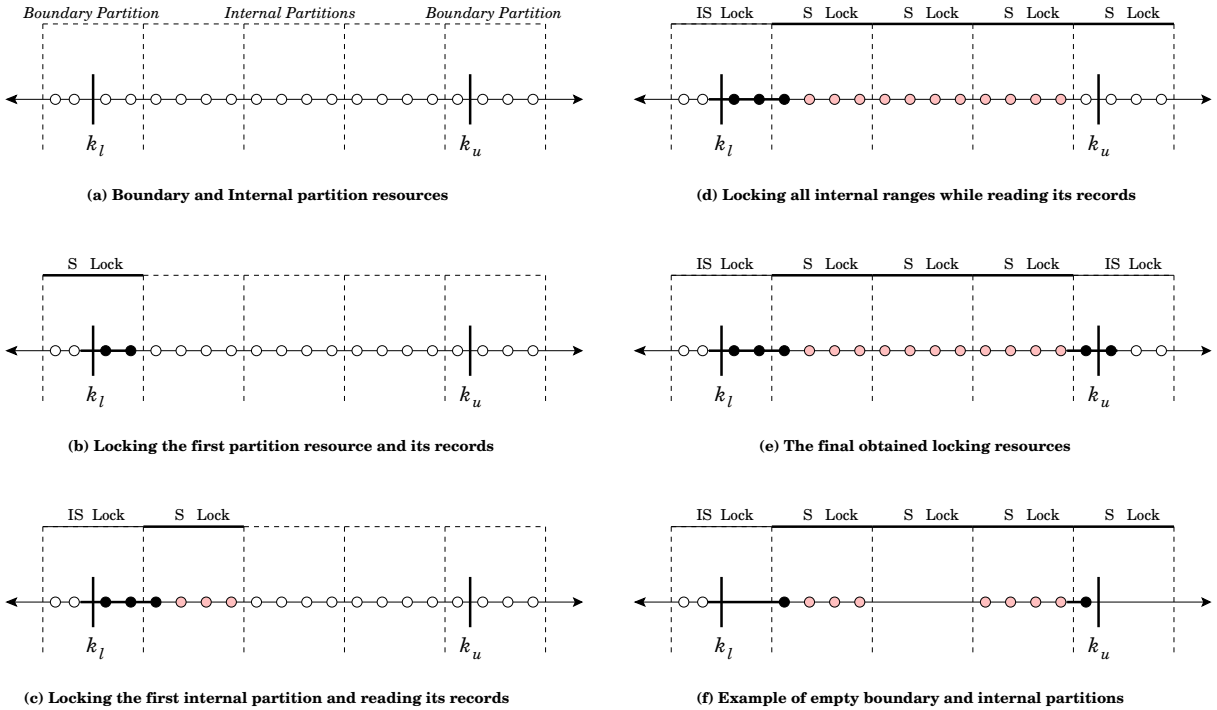



**Figure 2: Example of the *partition locking* protocol for `SERIALIZABLE` isolation level with S lock request.**

boundaries determined by the current data records. Data-dependent partitioning dynamically assigns the ranges and their sizes based on the density of the key values so that all records in the table are assigned *almost evenly* to partition resources. This can be achieved by utilizing the histogram information maintained by the query optimizer to evenly distribute the key record values on a set of range resources.

In both cases, dynamic partitioning avoids the drawbacks of fixed partitioning by tolerating data skewness where the size of a partition resource is proportional to the number of key records that are contained in this resource. However, a major disadvantage in the dynamic partitioning methods is the overhead of splitting and merging partition resources. For example, consider a partition resource $P$ that needs to be split into two partitions $P_1$ and $P_2$. In this case, the partition resource $P$ will disappear from the lock manager table along with its waiting queue. Then, all transactions that either hold a lock on $P$ or in the waiting list of locking $P$ need to be investigated to check if they need to hold a lock on $P_1$, $P_2$, or both of them. The merging case is also as complicated as the splitting case. For example, assume a transaction $T_1$ that holds a shared lock over resource $P_1$ and another transaction $T_2$ that holds an exclusive lock over resource $P_2$. In case that both resources are to be merged together to a resource $P$, then either $T_1$ or $T_2$ can hold a lock on $P$ while the other transaction will be inserted in $P$ waiting list. To reduce the overhead of splitting and merging, we can employ one of two techniques: (a) bulk a set of required changes and do them at once, thus, amortizing the cost of merging and splitting, or (2) restrict the splitting and merging of range resources to only the ones that are not locked by any transaction, where in this case, there will be no overhead of splitting or merging.

Another disadvantage of data-dependent partitioning scheme is that, unlike the case of *space-dependent* partitioning, there is no straightforward mapping function that can be used by the TC to map a certain record $r$ to its partition granular. Instead, the DC should explicitly pass to the TC all boundaries for all partition resources. This results in passing information between the two components which, to some extent, violates the main objective of unbundling transaction services.

## 4.4 Pages as Partitions

Many DBMSs already have a resource in the multi-granularity hierarchy that is between a table and a record. It is the page. A page is, indeed, a very good partiton resource, as it typically covers anywhere from 20 to 100 records, and pages form a disjoint partition of a table. One has to exercise care when executing B-tree structure modification operations, as those can change the mapping between pages and the records that they cover, though that should be managable. Thus, in a database system with a unified storage engine that has access to page information when locking is done, using pages as the partitions in the protocols described below works fine. In that case, our protocols, where key range locks are used in boundary partitions, improve the concurrency of traditional page based locking. In our splitting of database kernel into TC and DC, page information is not available in the TC when it requests locks from the lock manager. Hence, we cannot lock page based resources.

## 4.5 Our Prototype

In our implementation inside Microsoft SQL Server storage engine, we opt to use the fixed partitioning method as it is the simplest one to validate our ideas, and we cannot use page based resources. Further research may be dedicated to experimentally analyze the effect of other partitioning methods. It is important to emphasize that our proposed *partition locking* protocol is applicable regardless of the underlying partitioning scheme.

| | Description |
|---|---|
| $R$ | Requested range for the interval $[k_l, k_u]$ |
| $N_T$ | Number of records in the table |
| $N_R$ | Number of records in the requested range $R$ |
| $P_R$ | Number of partitions in $R$ |
| $t_r$ | Time to read a record from the DC |
| $t_l$ | Time to lock a record by the TC |
| $T$ | Transaction execution time without the locking part |
| $v$ | Average number of lock attempts for validating a record in the speculative visit protocol |

**Table 1: Used Symbols in Performance Analysis**

# 5. PERFORMANCE ANALYSIS

In this section, we analyze the performance of our proposed three locking protocols, *speculative visit*, *table lock*, and *partition lock* protocols. As a base of comparison, we will also include the conventional locking protocol, termed the *key lock* protocol, with the understanding that this locking protocol is not feasible when locking is done by an unbundled transaction service. Our performance analysis will be based on two main factors: (a) Concurrency control overhead due to locking and reading the data in the requested range, and (b) Loss of concurrency incurred by holding locks on resources for some period of time.

Table 1 contains the symbols used in our analysis. The number of records in the table is $N_T$ while the requested range $R$ has $N_R$ records and $P_R$ partitions. The time it takes for the TC to read a record is $t_r$ and the time to lock a record is $t_l$. The transaction execution time not covered by reading and locking the records in the requested range and which we assume follows the locking activity is $T$. Finally, for the speculative visit protocol, the average number of attempts (visits) to read and lock a record is $v$. So the number of failed attempts is $v - 1$.

## 5.1 Locking Overhead

The locking overhead is measured by the time it takes from the transaction to acquire locks and read the data in the requested range. Since all the four locking protocols will need to have either an intent or explicit lock on the table $T$, we will not include this minor cost in our analysis.

**Key Lock Protocol.** In this conventional locking protocol, the TC needs time lock ($t_l$) to lock each record of the $N_R$ records in the requested range $R$. So, the number of locks is $N_R$ and the locking overhead is $N_R(t_l + t_r)$ as we also read the $N_R$ records.

**Speculative Visit Protocol.** In this protocol, the TC needs time to read a record with a speculative visit ($t_r$), followed by time to lock this record ($t_l$), followed by time to read the same record again through a certification visit ($t_r$). The TC needs to do this procedure, on average, $v$ times for each single record $N_R$ until the certification visit validates the speculative visit. So, the locking overhead is $vN_R(t_l + 2t_r)$

**Table Lock Protocol.** This protocol will have the same overhead as the conventional key locking protocol where the TC reads and locks the $N_R$ records. So, the locking overhead is $N_R(t_l + t_r)$.

**Partition Lock Protocol.** The locking overhead in this protocol has three components: (a) The TC needs to read all the $N_R$ records in the requested range, $N_R t_r$, (b) The TC needs to lock all the *internal* and *boundary* partitions, $P_R$, in the requested range, $P_R t_l$, and (c) The TC needs to lock all the records in the *boundary* partitions that are contained within the requested range $R$. On average, half of the records in a *boundary* partition will be in $R$. For two *boundary* partitions, the TC needs to lock, on average, the number of records per partition, $\frac{N_R}{P_R} t_l$. So, the total number locking overhead is: $(P_R + \frac{N_R}{P_R})t_l + N_R t_r$

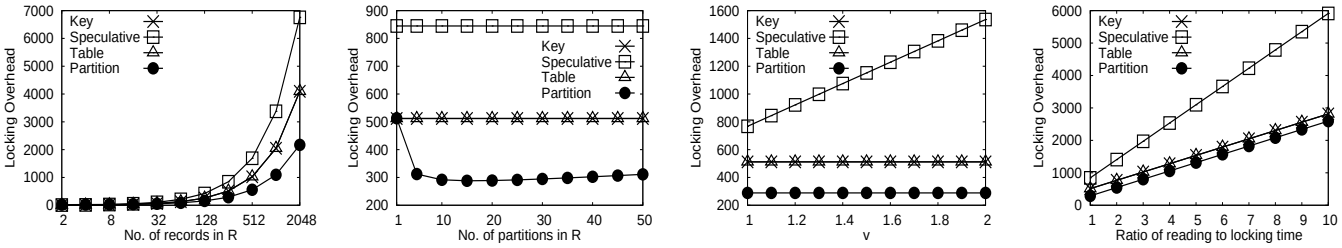
Figure 3 gives the comparison between the four locking protocols with respect to locking overhead. Unless mentioned otherwise, the number of records and of partitions in the requested range, $N_R$ and $P_R$, are set to 256 and 20, respectively. The locking time $t_l$ is set equal to the reading time $t_r$ as one time unit. This is mainly to penalize large span locks held to end of transaction. The factor $v$ for the speculative visit protocol is set to 1.1, i.e., on average 10% of the records will be locked twice before they are certified.

In Figure 3a, $N_R$ is varied from 2 to 2048 in a log scale. The partition locking protocol is superior to the other protocols in our "average case" analysis. The conventional key locking protocol has similar performance to that of the table locking protocol while the speculative visit protocol exhibits the worst performance. The main reason for the partition locking protocol advantage is that it does not need to lock all the records $N_R$. Instead, it needs to only lock those records in $N_R$ that lie in *boundary* partitions. On the other hand, both key locking and table locking protocols need to lock every record in $N_R$. Finally, for the speculative visit protocol, it always needs to read every record at least twice (speculative and certification reads), so it must have worse performance, even if every speculative visit is followed by successful certification. And, occasionally, it needs to visit and lock multiple times before certification succeeds.

Figure 3b gives the effect of increasing the number of partitions per range $P_R$ on the partition lock protocol. As other locking protocols are not affected by $P_R$, they are plotted only for comparison. If the range includes only one partition, the partition locking protocol degenerates to the performance of both the key and table locking protocols. The speculative visit protocol is, as before, always the worst. Increasing the number of partitions $P_R$ first results in significantly reducing the locking overhead as the partition locking protocol needs only to lock the partition resources. In our setting, having 15 partitions in the range gives the lowest locking overhead. However, increasing $P_R$ past 15 results in a very slight increase of the locking overhead as the increase in number of partitions becomes larger than the reduction in the number of records in the boundary partitions.

Figure 3c gives the effect of varying the number of visits $v$ in the speculative visit protocol from 1 to 2 compared to other locking protocols. Even in the best case scenario for the speculative visit protocol, i.e., $v = 1$, this protocol still exhibits the worst performance. And its performance dramatically worsens with the increase of $v$.

Finally, Figure 3d shows the effect of increasing the ratio of reading time $t_r$ to locking time $t_l$ form 1 to 10. All the locking protocols exhibit a similar linear increase in time and, except for the speculative visit protocol, the percent differences narrow as the locking overhead becomes a smaller part of the overall time. However, the speculative visit protocol gets even worse trend as its need to read each record at least twice becomes the dominant factor.




(a) $N_R$ (b) $P_R$ (c) $v$ (d) $t_r$

**Figure 3: Locking Overhead**

## 5.2 Loss of Concurrency

To measure the loss of concurrency for each locking protocol, we multiply the fraction of the table that is locked, and hence is blocked from being accessed by other transactions by the time period in which it is locked. This represents the hold time for locks integrated over time. This is our proxy for lost concurrency. We understand that it is not the same, but it does determine lost concurrency. To perform our analysis, we need to consider the total execution time for the underlying transaction. This consists of the time spent doing the locking plus the time $T$ introduced before. The time $T$ would penalize any protocol that retained large grained locks until end of transaction. None of our protocols retain those large grained locks. Indeed, they all hold exactly the locks that they need once the locking activity is completed. So this part of the lost concurrency is constant for all protocols and is $\frac{N_R}{N_T}(T)$. We include this as part of the lost concurrency for each of the protocols below.

**Key Lock Protocol.** In the conventional protocol, the TC only locks the part of the table that it needs. The time to perform this locking is $N_R(t_l + t_r)$. Only the fraction of the table $N_R/N_T$ will be locked, and on average, over the time in which locking is being done, only half these records will be locked. So, the total loss of concurrency will be $.5\frac{N_R}{N_T}(N_R(t_l + t_r)) + \frac{N_R}{N_T}(T)$.

**Speculative Visit Protocol.** Similar to conventional locking, the speculative visit protocol needs to lock only the range it is interested in. And again, only the fraction of the table $N_R/N_T$ will be locked, and on average over the time in which locking is being done, only half these records will be locked. As a result, the total loss of concurrency will be $.5\frac{N_R}{N_T}(vN_R(t_l + 2t_r)) + \frac{N_R}{N_T}(T)$.

**Table Lock Protocol.** The table lock protocol locks the entire table, while it reads and acquires locks on the records inside the requested range. This time is the locking overhead time. Thus, the loss of concurrency is $\frac{N_T}{N_T}(N_R(t_l + t_r) + \frac{N_R}{N_T}T$ or $N_R(t_l + t_r) + \frac{N_R}{N_T}T$.

**Partition Lock Protocol.** Even though this protocol uses partition locks during the locking phase, it remains the case that, on average, during the period when locks are being acquired, only about half of the range is locked. This protocol's superior concurrency is mostly due to the shortening of the time required to do this locking. As partition locks are taken, however, before the records are accessed within the partition, we have a sequence of partition locks that lock $1/P_R, ...i/P_R, ...P_R/P_R$ of the range. Summing and dividing by $P_R$ (the number of steps) gives us the average part of the range that is locked as $.5\frac{(P_R+1)}{P_R}$. Thus, the lost concurrency for this protocol is $.5\frac{(P_R+1)}{P_R}\frac{N_R}{N_T}((P_R + \frac{N_R}{P_R})t_l + N_R t_r) + \frac{N_R}{N_T}(T)$.

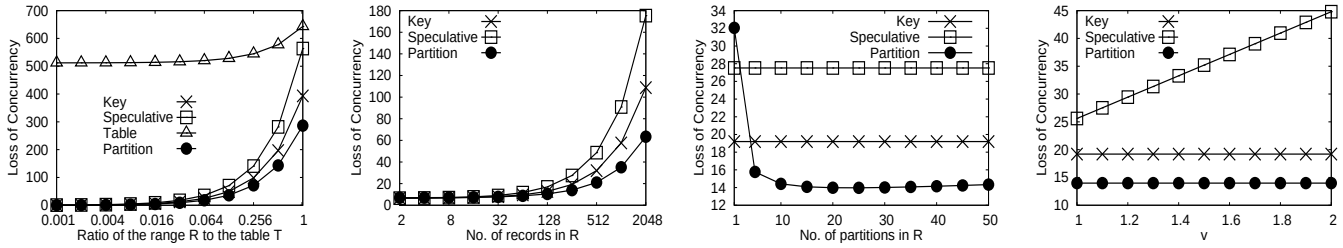
| | TC | Overhead | Loss of concurrency |
|---|---|---|---|
| Key | × | $N_R(t_l+t_r)$ | $.5\frac{N_R}{N_T}(N_R(t_l+t_r)) + \frac{N_R}{N_T}(T)$ |
| Speculative | √ | $vN_R(t_l+2t_r)$ | $.5\frac{N_R}{N_T}(vN_R(t_l+2t_r)) + \frac{N_R}{N_T}(T)$ |
| Table | √ | $N_R(t_l+t_r)$ | $N_R(t_l+t_r)+\frac{N_R}{N_T}T$ |
| Partition | √ | $N_R t_r +$ $(P_R + \frac{N_R}{P_R})t_l$ | $.5\frac{(P_R+1)}{P_R}\frac{N_R}{N_T}((P_R + \frac{N_R}{P_R})t_l$ $+ N_R t_r) + \frac{N_R}{N_T}(T)$ |

**Table 2: Performance Analysis**

Figures 4 gives the comparison between the four locking protocols with respect to loss of concurrency. Unless mentioned otherwise, the number of records and number of partitions in the requested range, $N_R$ and $P_R$, are set to 256 and 20, respectively. Also, the ratio of the number of records in range to the table, $N_R/N_T$, is set to 0.05. The locking time $t_l$ is set equal to the reading time $t_r$ as one time unit while the transaction execution time that does not include reading and locking, $T$, is set to 128 time units. Finally, the factor $v$ for the speculative visit protocol is set to 1.1.

In Figure 4a, $N_R/N_T$ is varied from 0.001 to 1 in a log scale. The table lock protocol has an extremely bad performance in all cases as it simply locks the whole table. In the extreme case where the requested range covers the whole table, the partition locking protocol is still better than the key locking and speculative visit protocols. This is basically due to the low overhead time required by partition locking. So, even if partition locking will lock some unrequested records in boundary partitions, it will does so only for a very short period of time. Due to the extreme bad performance for the table locking protocol, we will not include the table locking protocol in further comparisons.

Figures 4b-d give similar performance trend to Figures 3a-c, respectively. However, a surprising result is that the partition locking protocol consistently gives better concurrency than both the conventional key locking and speculative visit protocols. This is surprising as the partition locking protocol needs to lock the *boundary* partitions which results in locking extra records that are not covered by the requested range while both the conventional key locking and speculative visit protocols lock only the records inside the requested range. However, locking extra records is compromised by the partition locking protocol by holding the locks over these records and the records inside the requested range for a very short time period as it has the lowest locking overhead. So, partition locking protocol locks more records for a shorter time periods while both key locking and speculative visit protocols lock less records for longer time periods. As a total, it ends up that the partition locking protocol gives better




(a) $N_R/N_T$ (b) $N_R$ (c) $P_R$ (d) $v$

**Figure 4: Loss of Concurrency**

concurrency than all other locking protocols.

## 5.3 Worst Case Analysis

In this section, we discuss some worst-case scenarios for the proposed locking protocols. The worst case scenario for the *speculative visit* protocol, as depicted by Figures 3c and 4d, can go linearly was the increase of $v$. On the other side, the *table lock* protocol does not suffer from special worst case performance as in all cases, we need to lock the whole table. The performance may go worse only if the number of the records that need to be locked is increased.

The *partition locking* protocol could have several scenarios for worst case performance: (a) The entire requested range $R$ is in only one partition resource. This case is depicted as the first point in Figures 3b and 4c. In this case, the locking overhead of partition locking is similar to that of the table locking and key locking, yet still much better than that of the speculative visit. For concurrency, partition locking will be worse than key locking and speculative visit protocols, as on average half the partition resource is locked unnecessarily. (b) The entire range $R$ is empty. In this case, the partition locking protocol will unnecessarily lock the partition resources in $R$ while there are no records there. This would be in the advantage of speculative visit and key lock protocols as they do not need to lock any records in the entire range. However, the difference is not really great as can be seen by the first point in Figure 4a which gives the case that the number of records in the range is 0.001 of the number of records in the table.

## 5.4 Discussion

Table 2 summarizes the performance analysis for the four locking protocols. The second column in the table has a check mark ($\surd$) only for the locking protocols that are feasible for the unbundled transaction services environment. We can conclude that: (a) The table lock protocol is not practical as it encounters extremes loss of concurrency, (b) The speculative visit protocol is also not practical due to its very bad performance in the locking overhead, which leads to bad performance in terms of loss of concurrency, (c) Even if we disregard the fact that the conventional locking protocol is not feasible to our underlying environment of unbundled transaction services, it still provides worse performance than our proposed partition locking protocol. This indicates that our proposed partition locking protocol is not only superior than the table lock and speculative visit protocols for the case of unbundled transactions services, but it also results in better performance than the conventional key locking protocol for conventional environments.

# 6. CONCLUSION

This paper presented three locking protocols for locking key ranges in unbundled transaction services where the transaction services are partitioned to a transaction component (TC) and a data component (DC). The first two protocols, *speculative visit* and *table lock*, utilize existing lockable resources. The third protocol, *partition locking*, introduces a new lockable resource in the locking hierarchy, a *partition* resource, that lies between the key and table in the locking hierarchy where the TC has a full knowledge of the partition resource. Performance analysis shows that the partition locking protocol has lower locking overhead and higher concurrency than other protocols. We also compared the partition locking protocols with the conventional key locking protocol, though it cannot be applied to unbundled services, and showed that partition locking has lower overhead and better concurrency. This shows that partition locking protocol is also suitable for conventional locking managers.

# 7. REFERENCES


[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann, 1997.

[3] R. F. Chong, X. Wang, M. Dang, and D. R. Snow. *Understanding DB2: Learning Visually with Examples*. IBM Press, 2007.

[4] K. Delaney. *Inside Microsoft SQL Server 2005: The Storage Engine*. Microsoft Press, 2006.

[5] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. of the ACM*, 19(11):624–633, 1976.

[6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[7] J. Gray. et al. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–243, 1981.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] R. Greenwald, R. Stackowiak, and J. Stern. *Oracle Essentials: Oracle Database 10g*. O'Reilly, 2004.

[10] D. B. Lomet. Key Range Locking Strategies for Improved Concurrency. In *VLDB*, pages 655–664, 1993.

[11] D. B. Lomet, A. Fekete, G. Weikum, and M. Zwiling. Unbundling Transaction Services in the Cloud. In *CIDR*, 2009.

[12] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*, 1990.

[13] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.