Phantom Problem and its Solution

- Phantom Problem.
- Predicate locking, precision locking, granular locking.
- Multigranularity protocol.
- Key Range Locking.

Phantom Problem Motivation

Issue:
Does the CCM by ensuring CSR of schedules guarantee Isolation of Database applications (that is, serializability from the perspective of applications)?

- Recall that isolation means that the concurrent execution of applications is equivalent to some sequential execution of the same applications.
- To ensure isolation, the CCM:
  - Models the database as a collection of distinct objects.
  - Models a transaction as a sequence of begin, read, write, and commit/abort operations on the objects.
  - Ensures that the resulting schedules (which is the interleaving of the operations of the transactions) are CSR.
  - To ensure CSR, it may use any of the protocols like 2PL, TO, SGT, optimistic validation, etc.
- So does ensuring CSR of the schedules as we have studied so far guarantee isolation (that is, equivalence to some sequential execution of applications)?
- Unfortunately, the answer in the context of databases is NO due to the *Phantom Problem*.
- That is, ensuring schedules are CSR may not always ensure equivalence to the serial execution of the transaction programs.

Example of the Phantom Problem

- Relation *emp* = *(name, salary, title)*.
- Transactions:

T_1:	T_2 :
select title	**select** salary
from emp	**from** emp
where salary = 1000	**where** title = prof
insert t_4 in emp	**insert** t_5 in emp
where t_4 = (sal, 900, prof)	**where** t_5 = (tim, 1000, pres)

- Initial State:

tid	name	salary	title
t1	paul	1000	prof
t2	don	200	student
t3	sue	1000	prof

- Execution:

T_1:	r(t1)	r(t3)				w(t4)
T_2:			r(t1)	r(t3)	w(t5)	

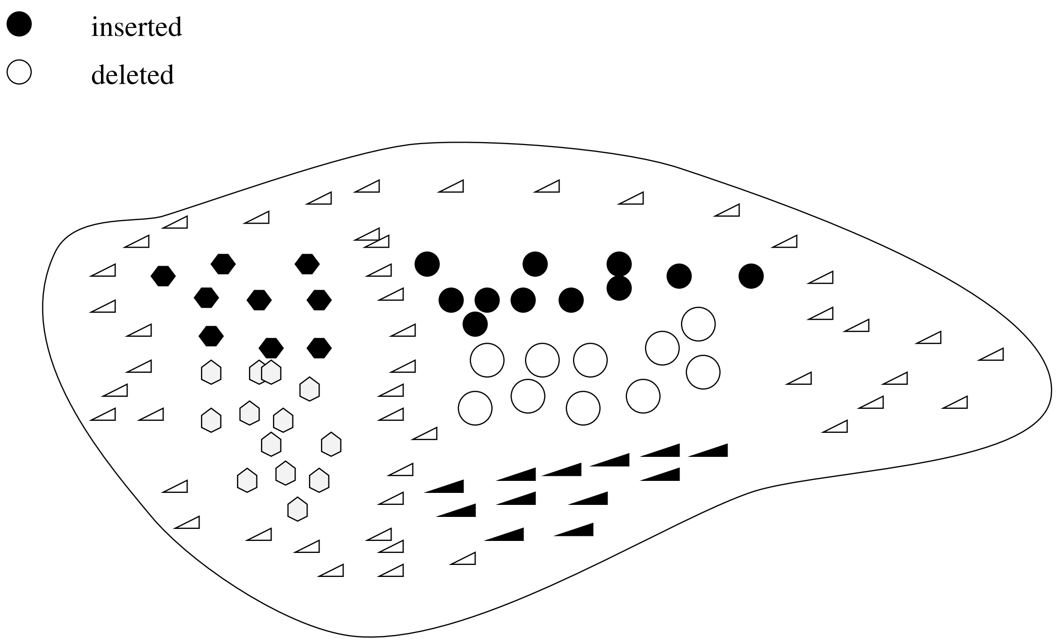
Example

- Final State:

tid	name	salary	title
t1	paul	1000	prof
t2	don	200	student
t3	sue	1000	prof
t4	sal	900	prof
t5	tim	1000	pres

- Note that even though the schedule is CSR (according to our definition), the execution is not equivalent to some serial execution of the two transaction programs.

What Went Wrong?



- View the database as consisting of all objects that could possibly exist.
- Objects that are currently present in the database are marked inserted.
- Objects that are currently not present are marked as deleted.
- When a transaction program requests to read the set of tuples that satisfy a predicate p, it reads all tuples that satisfy p (both marked inserted and marked deleted).

What Went Wrong?

- T_1 reads all records in emp relation such that emp.salary = 1000, those that are marked inserted as well as those that are marked deleted.

- Thus, T_1 reads t1, t3, and t5.

- Similarly, T_2 read all records in emp (both marked inserted as well as marked deleted) such that emp.title = prof.

- That is, it reads records t1 and t3, and t4.

- If we model transactions as above, the schedule in the example is not serializable according to our definition.

Solving Phantom Problem

Approach 1:

Each transaction locks both data items that are marked inserted as well as those marked deleted.

> *not implementable directly since it requires transactions to acquire an unbounded number of locks.*

Approach 2:

- The database supports a *representative* object to represent all the objects that are marked 'deleted'.

- A transaction that needs to read a data item marked 'deleted', locks the representative object.

- Thus, transaction T_1 in the example locks the representatives object.

- A transaction that inserts/deletes an object also acquires a lock on the representative object.

  The representative object will become a hot spot.

Predicate Locking

- Transactions do not lock records but instead lock ***predicates***.

- So T_1 acquires a S-lock on the following predicate to perform its read operation:

  emp.salary = 1000

- Similarly, T_2 acquires a X lock on the following predicate to perform its insert operation:

  emp.title = pres **and** emp.name = tim **and** emp.sal = 1000

- Locks on two predicates p_1 and p_2 conflict if:
  - they are lock requests of different transactions.
  - at least one of them is an X mode lock request.
  - the predicate (p_1 **and** p_2) is satisfiable (some object satisfies both predicates).

- Predicate locking provides protection to transactions from phantom insertions and deletions.

Predicate Locking

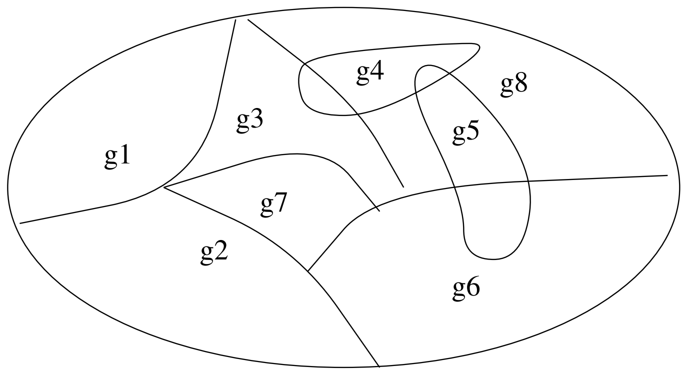
- *Advantages:*
  - Prevents phantoms from occurring.
  - Allows transactions to lock a single record, a small set of records, or a very large set of records (e.g., the entire database) using a single predicate lock.

- *Disadvantages:*
  1. Execution cost: need to check for predicate satisfiability.
  2. What are the predicates.
  3. Is pessimistic:
     - T_1 updates all green eyed entities.
     - T_2 updates all entities that are monsters.
     - Even though there are no green eyed monsters, the two predicates will conflict.

Precision Locking

- Similar to predicate locking, transactions ask request locks on predicates.
- The lock request of a transaction is always granted.
- When a transaction accesses a record, the record is compared to all the outstanding predicates p' of other transactions.
- An access is denied if the record satisfies p' and
  - the record is being read and p' is locked in X mode.
  - the record is being updated.
- Precision locks are cheaper than predicate locks.
- Precision locks are not as pessimistic as predicate locks.
- Unfortunately, they tend to convert the waits resulting from predicate locks into deadlocks.
  - T_1 acquires X lock on predicate emp.title = ‘secretary’.
  - T_2 acquires X lock on predicate emp.title = ‘secretary’.
  - T_1 tries to read a record. It will have to wait. T_2 tries to read a record. It will have to wait.

Granular Locks



Set of Data Items

- Granular locks are an engineering approach to the predicate locks.
- The key idea is to pick a finite fixed set of predicates, say $g_1, g_2, \ldots, g_n$.
- These predicates satisfy the property that $g_1 \vee g_2 \vee \ldots \vee g_n =$ TRUE.
- Predicates $g_1, g_2, \ldots, g_n$ define the lockable entities or *granules* of locking in the system.

Granular Locks

- A transaction T wishing to obtain a lock in mode M on the predicate p, acquires a M mode lock on a subset $lset(p)$ of the set of granules $\{g_1, g_2, \ldots, g_n\}$.

- Similarly a transaction T' wishing to obtain a M' mode lock on predicate p' acquires a M' mode lock on a subset $lset(p')$ of the set of granules $\{g_1, g_2, \ldots, g_n\}$.

- **Granule Locking Correctness Criterion:** The sets $lset(p)$ and $lset(p')$ must satisfy the property:

  if $p \wedge p'$ is satisfiable, then $lset(p) \cap lset(p') \neq \emptyset$.

- Intuitively, if lock requests of transactions T and T' on predicates p and p' conflict, then their requests will also conflict on atleast one granule.

- Similar to predicate locks, granular locks also prevent phantom insertion and deletions.

Granular Locking Issues

- *Partitioning database into granules:*
  - fine granules: high concurrency, high lock overhead.
  - coarse granules: lower concurrency, lower lock overhead.
- *Mapping predicates to granules:* a transaction should be efficiently able to determine the set of granules it needs to lock.

A Simple Granular Locking Protocol

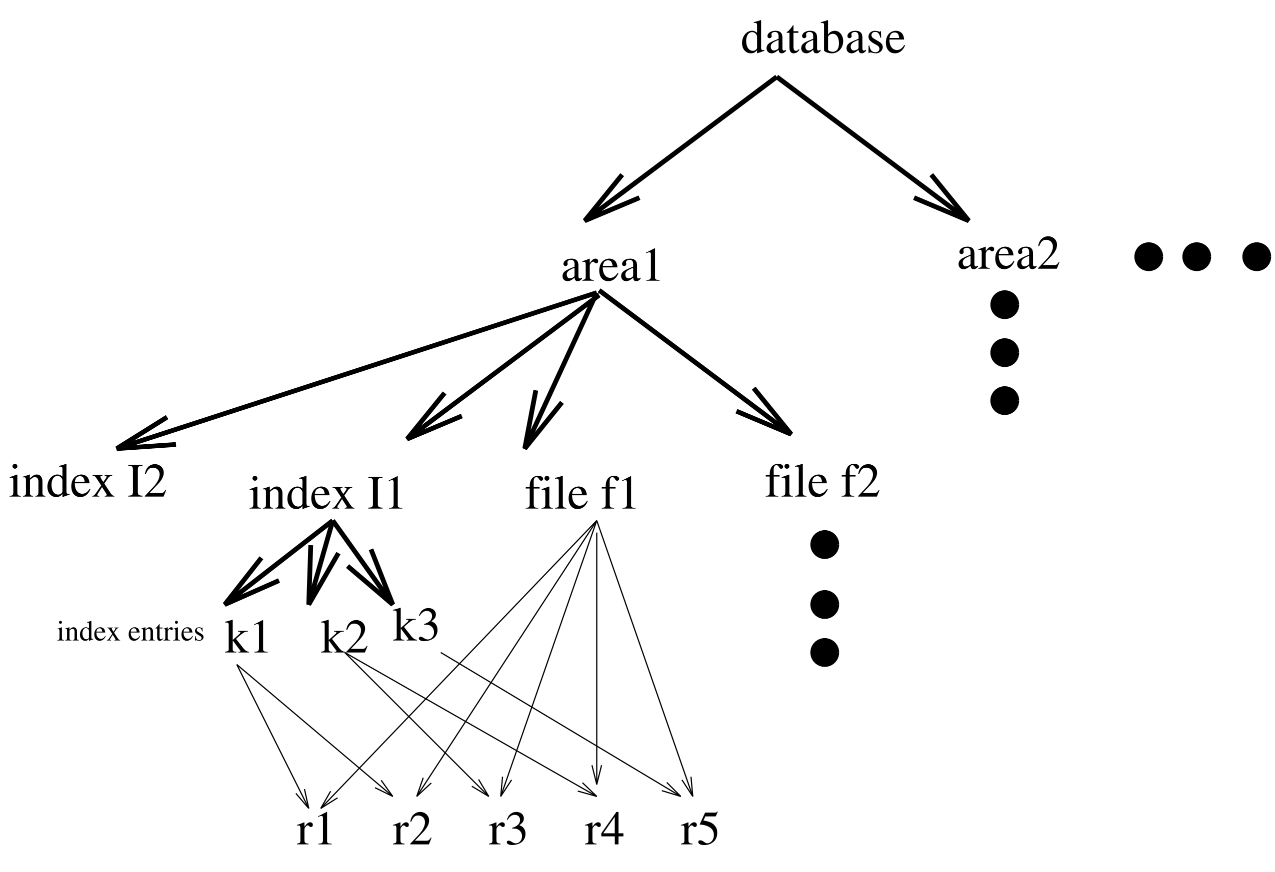
- Let the set of granules $G = \{g_1, g_2, \ldots, g_n\}$.
- Let T_i be a transaction accessing predicate p^i.
- $lset(p^i) = \{g_1^i, g_2^i, \ldots, g_r^i\}$, where for all $g_j \notin lset(p^i)$, $p^i \Rightarrow \neg g_j$.
- That is, the transaction locks all granules that overlap with the predicate p^i.

Problems with Basic Granular Locking Protocol

- Example of Granules: a tuple, objects satisfying a range of key values, a relation, a file, a set of relations, the entire database.
- Assume that g_i and g_j are two gralunes (predicates) where—
  - g_i = records in the emp relation.
  - g_j = records in the emp relation such that emp.title = president.
- Note that the predicate g_j implies predicate g_i.
- In the granular locking strategy a transaction T wishing to acquire a lock on g_i will also have to lock on g_j.
- In contrast, if predicate locks were used, then a single lock on the predicate g_i covers the entire emp relation.
- It is desirable to come up with a granular locking strategy that behaves similar to the predicate locking strategy— that is, if a transaction acquires a lock on g_i it does not need to acquire a lock on the finer granule g_j.
- Multi granularity locking (MGL) is such a strategy.
- To describe MGL we first need to define the notion of a *Granule Graph* and *intention mode locking*.

Granule Graph

- For a given set of granules (predicates) $g_1, g_2, \ldots g_n$ consider a directed graph G – referred to as the *granule graph* or the *lock instance graph*.
- Nodes in G correspond to the granules $g_1, g_2, \ldots, g_n$.
- There is an edge from g_i to g_j if g_j implies g_i.
- By definition G is acyclic.
- Example of G:



Intention Mode Locks

- Besides the (S)hare and e(X)clusive locks, *intention* mode locks on granules are supported.
- There are two types of intention modes– (I)ntention (S)hare (IS), and (I)ntention e(X)clusive (IX).
- Lock Compatibility Matrix:

	IS	IX	S	SIX	X
IS	+	+	+	+	
IX	+	+			
S	+		+		
SIX	+				
X					

Multigranularity Locking

- Having defined lock instance graph G and intention lock modes we can define the MGL protocol.
- Transactions acquire locks from root to leaf order according to G.
- A transaction can acquire a S or a IS mode lock at a granule, if it has ***at least*** one parent of the granule in G locked in either of IS, IX, or SIX modes.
- A transaction can acquire a X or a SIX or a IX mode lock at a granule, if it has ***all*** parents of the granule in G locked in either of IX, or SIX modes.
- Locks are released in a leaf to root order.

Multi Granularity Locking

- Before a transaction can access an object it must have an appropriate *implicit* lock on the objects.
- A transaction T has an implicit S lock on a node in G if for each root node, there exists a path such that T has a S or an X lock on some node on the path.
- T has an implicit X lock on a node n in G if for each root node r, for all paths from the root to the node n, T has an X lock on some node on the path.
- The MGL protocol prevents two different transactions from acquiring conflicting implicit locks at the same time on any object.

Basic Granular Locking vrs Multi Granularity Locking

- Contrast MGL with our original granular locking scheme.
- In the original granular locking scheme, a transaction T needing to lock granule g_i, will acquire lock on g_j as well, where g_j implies g_i (that is, g_j is a finer granule than g_i).
- On the other hand in MGL, T can acquire an implicit lock on a finer granule by acquiring lock at the coarser granules. So it does not need to lock explicitly at the finer granule.
- However, in MGL transactions acquiring locks at the finer granule *need* to acquire intention locks at the coarser granules.
- So under what conditions is MGL better than basic granular locking?

Basic Granular Locking Vrs MGL

- Let the granule graph consist of a node for relation R, and records in R.
- Let k be the number of records in R.
- Let m be the number of transactions.
- Let pm transactions acquire locks at file granule.
- Let each transaction acquiring locks at record granule only access a single record.
- Let each transaction acquiring locks at relation granule acquire access each record in the relation.
- If basic granular locking is used, then total number of locks acquired by all transactions:

$$pm(k+1) + (1-p)m = (kp+1)m.$$

- If MGL is used, then total number of locks acquired by all transactions:

$$pm + (1-p)m(1+1) = (2-p)m.$$

- Thus, MGL is better than basic granular locking if:

$$k > \tfrac{1}{p} - 1$$

- Let $p = 0.001$ (that is, one in a thousand transactions acquire coarse granularity locks).
- MGL is better than basic granular locking if the number of tuples in relation R is greater than 999.

Lock Escalation

- Different transactions acquire locks at different levels of granularity.
- Locking at finer granularity provides higher concurrency but has higher overhead.
- For small transactions, fine granularity record locks are acquired. For large transactions coarse granularity (e.g., relation level) locks are acquired.
- If transactions acquire too many locks, system probably guessed wrong about the lock needs of the transaction.
- Convert fine granularity locks into one course lock. This is referred to as lock escalation.
- Example. 1000 record locks on table T becomes one lock on table T.
- To escalate a lock, the transaction needs to ***convert*** its intention lock at a courser granule into a stronger lock.
- Lock conversion may result in a deadlock.
- Update mode locks can be used to prevent such deadlocks [Korth 81].

Summary of Phantom Problem

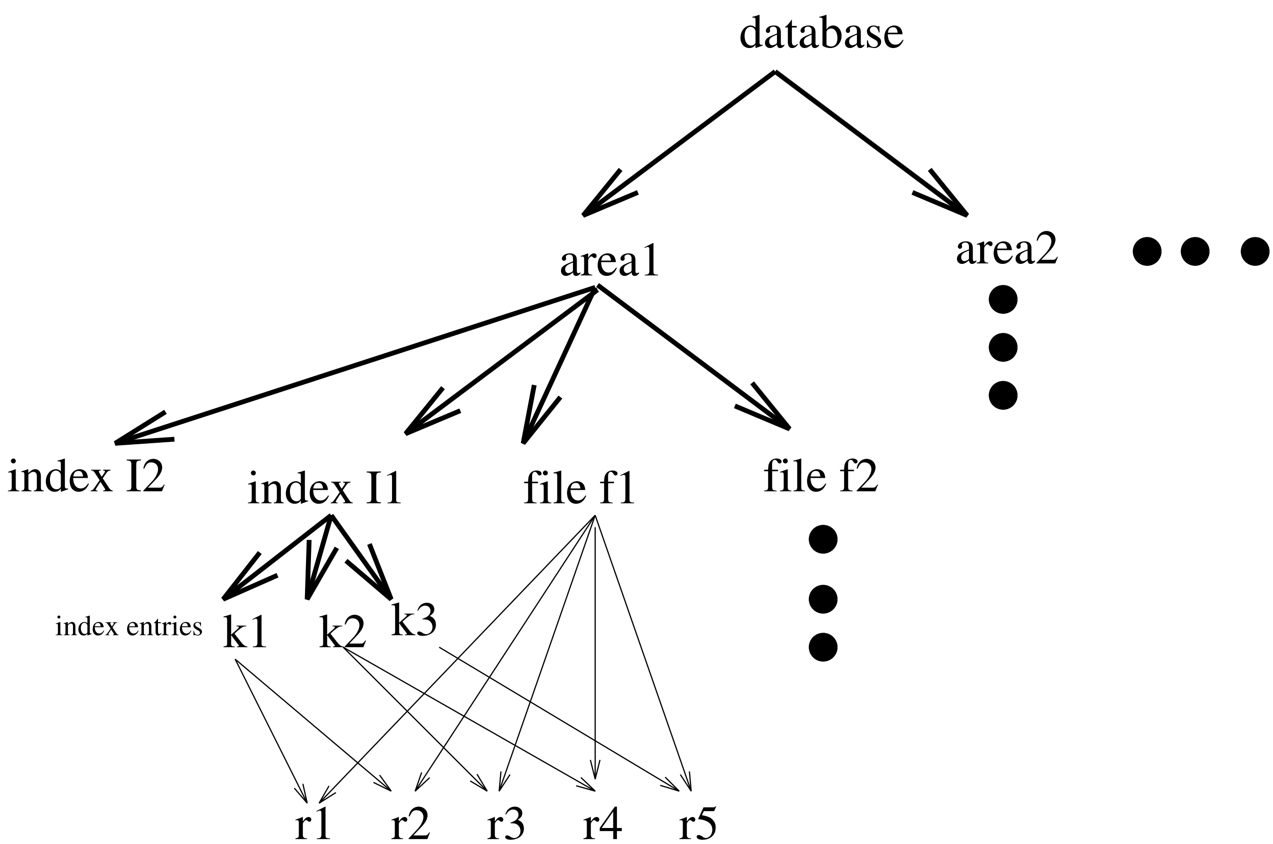
- Phantom problem arises if we model a transaction as consisting of a sequence of read and write operations on objects in the presence of insertion and deletion operations.

- Predicate locking (transactions acquire locks on predicates instead of data items) solves the phantom problem.

- Predicate locking permits a transaction to lock a large set of data using a single lock.

- Implementing predicate locking directly is too expensive.
- Granular locks are an engineering approach to implementing predicate locks.
- The key idea is to pick a finite fixed set of predicates which specify the lockable entities or the granules in the system.
- Predicate locking besides providing Phantom protection, also allows transactions to lock a single record, a small set of records, or a very large set of records (e.g., entire database) using a single predicate lock.
- Multi granularity locking (MGL) protocol mimics predicate locking providing phantom protection as well as allowing applications to lock at multiple granularity.

Motivation for Key Range Locking



- Consider the lock instance graph of the previous slide.
- Assume that file f_1 stores relation R where records in R have fields $(k1, k2, k3)$.
- Consider a transaction T_1 that uses index I1 to scan all records in the file such that $2 \leq k1 \leq 20$.
- Consider another transaction T_2 that insert a record (25, 3, 20) in R.
- The scan needs a S lock on the predicate $2 \leq k1 \leq 20$.
- According to MGL protocol T_1 will acquire a IS lock on the database and area1 and a S lock on either the index I1 or the file f1.

- T_2 to insert the record needs to acquire IX locks on the database, area a1, index I1, and the file f1.

Motivation for Key Range Locking

- Since IX conflicts with S, T_2 will be unable to execute concurrently with the scan of T_1 even though T_2's insertion does not conflict with T_1's scan.

- Thus, any scan over a file will conflict with insertion/deletion anywhere in the file resulting in low concurrency.

- One approach to increasing concurrency is to support key ranges as a granule.

- We will assume that there is a single index on a file and it is a unique index.

Static Key Range Locking

- Assume that the entire key space of key K1 is partitioned statically into 3 parts (-α, 10], (10, 20], (20, α) and the three ranges are supported as lockable granules.

- To scan the predicate $2 \leq k1 \leq 20$, T_1 needs to acquire S locks on the granules (-α, 10] and (10, 20].

- To insert (25,3,20), T_1 needs to acquire an IX locks on the file, the index I1, and the range (20, α) and an X lock on the record.

- Note that the two transactions can go on concurrently since they do not conflict.

Static Key Range Locking

- For most applications static key range locking with fixed key ranges is too restrictive in terms of concurrency.

  Consider that most updates and scans are within the range (10,20] and rarely in other ranges.

- Fixed ranges do not adapt to the dynamically changing database.

  Since transactions' access pattern may change with time it is difficult to come up with a fixed set of key ranges that provide high concurrency irrespective of the dynamic nature of the transactions.

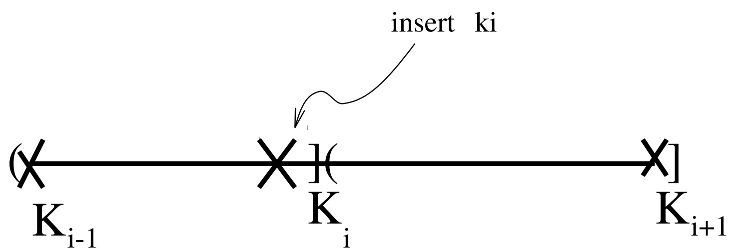
- The solution is to have dynamically changing key ranges that adapt to the keys present in the database.

Dynamic Key Range Locking

- Let $k_1, k_2, \ldots, k_n$ be the key values present in the relation R.
- The set of key ranges are (-α, k_1], (k_1, k_2], (k_2, k_3], ..., (k_n, α).
- The range $(k_i, k_{i+1}]$ is 'identified' by the key k_{i+1}.
- Let us denote the range $(k_i, k_{i+1}]$ by $range(k_{i+1})$. The basic

  idea of dynamic KRL is still as it was in static KRL– that is, ranges are lockable granules.
- However, insertion of a new record splits a key range.
- Furthermore, deletion of a record consolidates the key range.
- Splitting and consolidation complicates the locking protocol.

Insertion of a Key



- Assume that transaction T_1 wishes to insert a key k_i into the range $(k_{i-1}, k_{i+1}]$– that is, $range(k_{i+1})$.

- To insert k_i according to the MGL protocol, T_1 needs to acquire a lock on the range $range(k_{i+1}$ in the IX mode.

- However, inserting key k_i will split the old range $(k_{i-1}, k_{i+1}]$ into two ranges $(k_{i-1}, k_i]$ and $(k_i, k_{i+1}]$.

- After the insertion, the T_1's IX lock on $range(K_{i+1})$ only protects the range $(k_i, k_{i+1}]$.

- It is possible that another transaction acquires a S lock on $range(k_i)$ and reads the the range $(k_{i-1}, k_i]$ – phantom probem!

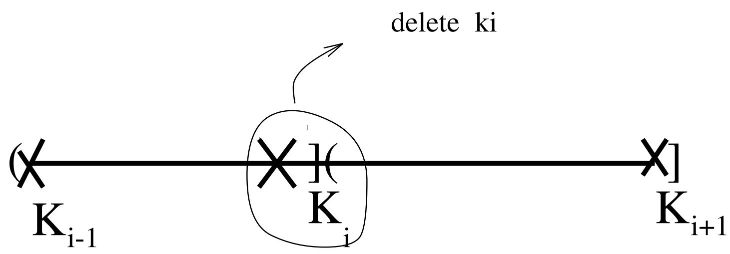
Insertion of a Key

- To protect the range $(k_{i-1}, k_i]$, transaction T_1 needs to acquire an IX lock on $range(k_i)$.

- Note that after the insertion, the lock on the $range(k_{i+1})$ can be released since there is no need to protect the range $(k_i, k_{i+1}]$ from scans.

- Locking protocol for insertion of k_i:
  - Instant duration IX lock $range(k_{i+1})$.
  - X lock key k_i.
  - IX lock $range(k_i)$.

Deletion of a Key



- Assume that transaction T_1 wishes to delete a key k_i.

- To delete k_i, according to the MGL protocol, it needs to acquire an IX lock on $range(k_i)$.

- However, deleting the key k_i consolidates the old ranges $(k_{i-1}, k_i]$ and $(k_i, k_{i+1}]$ into a single range $(k_{i-1}, k_{i+1}]$.

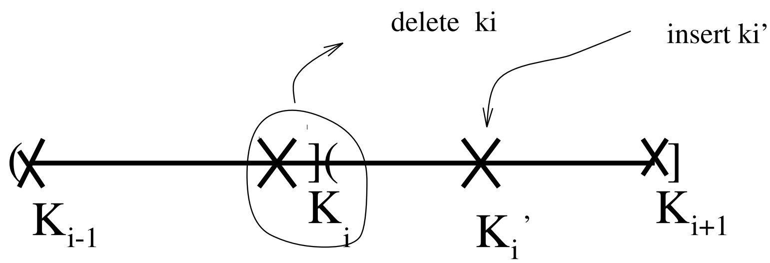
- After the deletion, since the key k_i dissapears, T_1's lock on $range(k_i)$ disappears.

- It is possible that another transaction acquires a S lock on $range(k_{i+1})$ before the deletion commits!

- Thus, to protect the deletion T_1 accoring to MGL transaction T_1 needs to acquire an IX lock on $range(k_{i+1})$.

- Will the above scheme work?

Deletion of a key



- Assume that to insert k_i transaction T_1 acquired an IX lock on $range(k_{i+1})$.
- This will not prevent T_2 from inserting say k_i' within the delete range.
- However, after the insertion, T_1's IX lock on $range(k_{i+1})$ is protecting the range $(k_i', k_{i+1}]$ and not the range in $(k_{i-1}, k_i']$ where the deletion took place!
- Another problem may be that an IX lock on $range(k_{i+1})$ will not prevent deletion of the key k_{i+1}.
- However, deletion of key k_{i+1} results in the loss of the IX lock of T_1 on the delete range.
- One solution is to acquire a more restrictive lock that prevents insertion and deletion in the delete range from occurring.

Deletion of a Key

- To delete key value k_i acquire the following locks:
  - Instant IX lock on $range(k_i)$.
  - X lock on key k_i.
  - SIX lock on $range(k_{i+1})$.

- Key range locking:

operation	range(k_i)	k_i	range(k_{i+1})
read key	IS	S	none
update key	IX	X	none
read scan	S	none	none
update scan	SIX	X	none
insert	IX	X	IX (instant)
delete	IX (instant)	X	SIX

Summary of KRL

- Recall that we introduced key ranges as lockable granules in order to permit concurrent access to the different key ranges within a file.
- Static key range locking is simpler but is not adaptive to dynamically changing key values and may result in low concurrency.
- Dynamic key ranges where a range is associated with each key present in the file adapts well to the changing key values present in the database and provides high degree of concurrency.
- However, while key range locking increases concurrency, it also increases locking overhead per transaction– since the transactions need to acquire locks at one extra granule.
- Is it possible to overcome this extra locking overhead?

Commoning Resources to Reduce Locking Overhead

- In our discussion so far a key value k_i and the range associated with it – that is, $range(k_i)$ are treated as separate resources.
- To reduce locking overhead we can view the key and the range associated with the key as a *common* resource.
- Thus, if a transaction acquires an X lock on the key k_i, it also ends up acquiring an X lock on $range(k_i)$ as a side effect.

An Approach to Commoning Range and Key Resources

- Say T_1 wishes to acquire a lock in mode M on the $range(k_i)$.
- Furthermore, T_1 wishes to acquire a lock in mode M' on key k_i.
- T_1 acquires a lock on k_i in mode M" which is the least upper bound of the modes M and M'.
- A M" mode lock on key value k_i signifies a lock in mode M" on both the key k_i and a lock in mode M" on the $range(k_i)$.

Commoning the Key and Ranges Resources

- Key range locking with the range and key commoned will be as follows:

- Original Key Range Locking:

operation	range(k_i)	k_i	range(k_{i+1})
read key	IS	S	none
update key	IX	X	none
read scan	S	none	none
update scan	SIX	X	none
insert	IX	X	IX (instant)
delete	IX (instant)	X	SIX

- Key Range Locking with resources commoned.

operation	k_i	k_{i+1}
read key	S	none
update key	X	none
read scan	S	none
update scan	X	none
insert	X	IX (instant)
delete	X	SIX

Reduction of Concurrency by Commoning Resources

- Commoning resources using the above approach causes reduction in the concurrency.

- Consider that T_1 wishes to insert a key value k_i.

- T_1 needs to acquire an instant IX lock on $range(k_{i+1})$, a X lock on key k_i, and an IX lock on key $range(k_i)$.

- In the above resource commoning strategy, T_1 will acquire an X lock on k_i thereby preventing any other inserts in front of k_i to progress concurrently.

Another Approach to Commoning Resources

- Another approach to commoning the key and the range resources is to increase the number of lock modes.

- The new lock modes consist of two parts– (range mode, key mode).

- Thus, a lock in mode (IS, S) on key k_i denotes a lock in IS mode on $range(k_i)$ and a S mode lock on the key k_i.

- The lock in mode (M_1, M_2) is compatible with a lock in mode (M_1', M_2') iff
  - M_1 and M_1' are compatible.
  - M_2 and M_2' are compatible.

- This approach to commoning resources does not force an insertion that needs an IX lock on range and an X lock on key to acquire an X lock on the range as well.

Commoning the Key and Ranges Resources

- Key range locking with the second strategy of commoning range and key works as follows:

- Key Range Locking with resources commoned using old Strategy:

operation	k_i	k_{i+1}
read key	S	none
update key	X	none
read scan	S	none
update scan	X	none
insert	X	IX (instant)
delete	X	SIX

- Key Range Locking with resources commoned using new Strategy:

operation	k_i	k_{i+1}
read key	(IS,S)	none
update key	(IX,X)	none
read scan	(S,none)	none
update scan	(SIX,X)	none
insert	(IX,X)	(IX,none) (instant)
delete	(IX,X)	(SIX,none)

New Strategy Increases Concurrency

- Note that with the new resource commoning strategy, the degree of concurrency is increased.

- In particular, an insertion is permitted in the insert range which was not permitted by the previous strategy of resource commoning.

- See [Lomet 93] for how concurrency can be further increased by introducing more lock modes.

Key Range Locking Summary

- Key ranges are introduced as lockable granules to enhance concurrency.
- KRL can either be static or dynamic.
- Static KRL is not adaptive to the changes in the database.
- Dynamic KRL introduces some problems in locking since the granules change dynamically with the changes in the database.
- Having key ranges as lockable granules provides high concurrency but it also increases the locking overhead of the transactions.
- By merging resources (e.g., keys and ranges) the lock overhead can be reduced.
- Merging resources, in general, will result in a loss of some concurrency.
- Some amount of concurrency can be regained by inventing new lock modes.

Further Reducing Overhead

- Recall that even though we are doing locking at the range (or key), we still need to lock at the record level to protect records from being accessed using other access paths.
- We can combine the three – key range locks, key locks, and record locks into one to reduce overhead. However, this will results in even lower concurrency since obtaining a lock on record will lock the range in each access path via which the record can be reached.
- Inventing new lock modes will probably not be able to resolve this loss of concurrency. Thus, extra lock overhead of obtaining locks on keys will need to be incurred if maximal concurrency is desired.