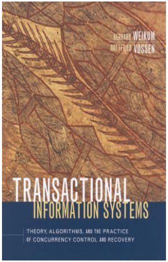
Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery



Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
ISBN 1-55860-508-8

"Teamwork is essential. It allows you to blame someone else."(Anonymous)

Part II: Concurrency Control

- 3 Concurrency Control: Notions of Correctness for the Page Model
- 4 Concurrency Control Algorithms
- 5 Multiversion Concurrency Control
- 6 Concurrency Control on Objects: Notions of Correctness
- 7 Concurrency Control Algorithms on Objects
- 8 Concurrency Control on Relational Databases
- 9 Concurrency Control on Search Structures
- 10 Implementation and Pragmatic Issues

Chapter 9: Concurrency Control on Search Structures

- 9.2 Implementation by B⁺-trees
- 9.3 Key-Range Locking at the Access Layer
- 9.4 Techniques for the Page Layer
- 9.5 Further Optimizations
- 9.6 Lessons Learned

“ As long as one keeps searching, the answers come. ”
(Joan Baez)

Example

t_2:

Select * From Persons
Where City = „Phoenix„

→ fetches records p, q

t_1:

Update Persons
Set City = „Phoenix„
Where Age $\geq$ 50
And City = „Dallas„

→ modifies record x

Select * From Persons
Where City = „Dallas„

→ fetches records d, e

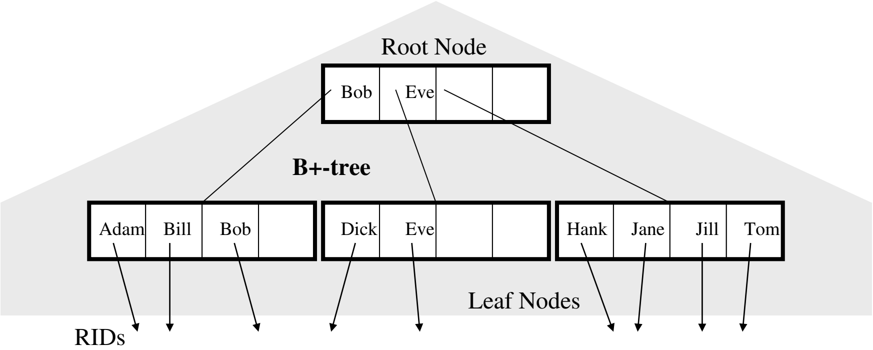
Observations:

- *page locking would prevent this phantom-problem execution*
- *locking the accessed records alone is insufficient*
- *need appropriate locks on (key, RID) pairs in City index*

Chapter 9: Concurrency Control on Search Structures

- **9.2 Implementation by B$^+$-trees**
- 9.3 Key-Range Locking at the Access Layer
- 9.4 Techniques for the Page Layer
- 9.5 Further Optimizations
- 9.6 Lessons Learned

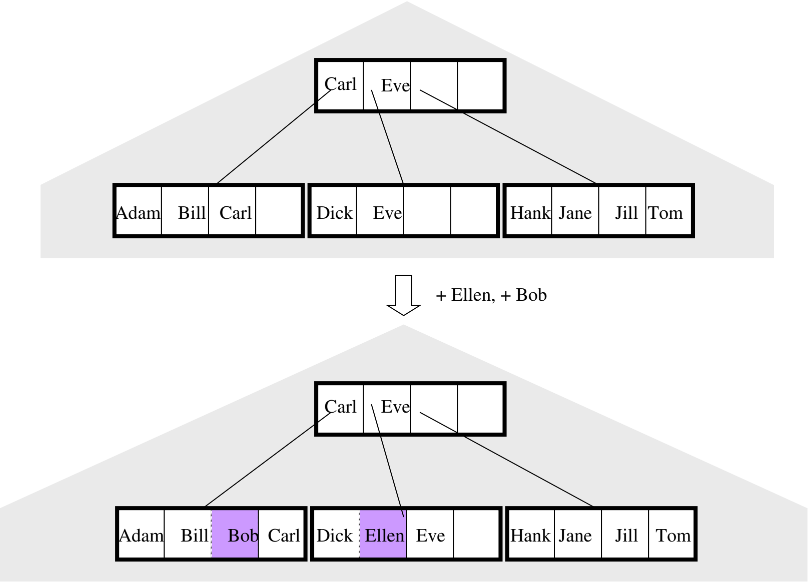
Implementation of Index by B$^+$-tree



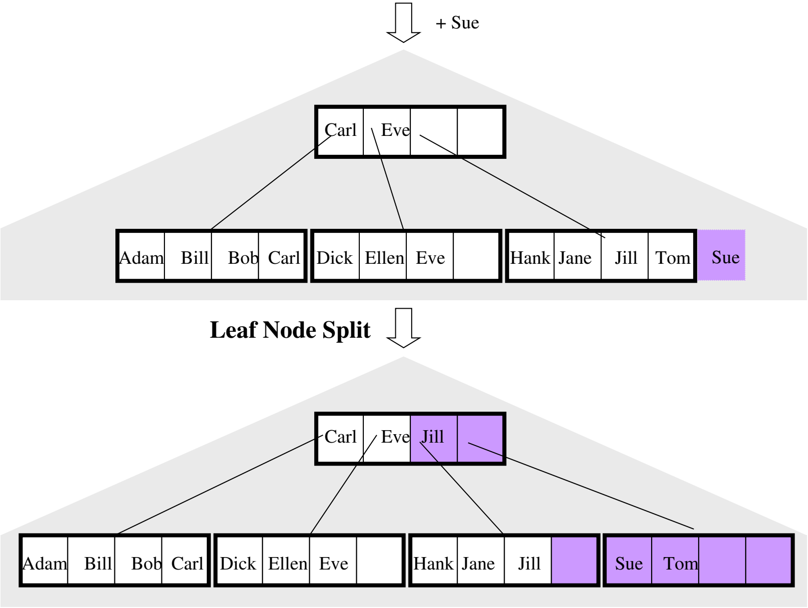
Search tree interface:

- lookup <index> where <indexed field> = <search key>
- lookup <index> where <indexed field> between <lower bound> and <higher bound>

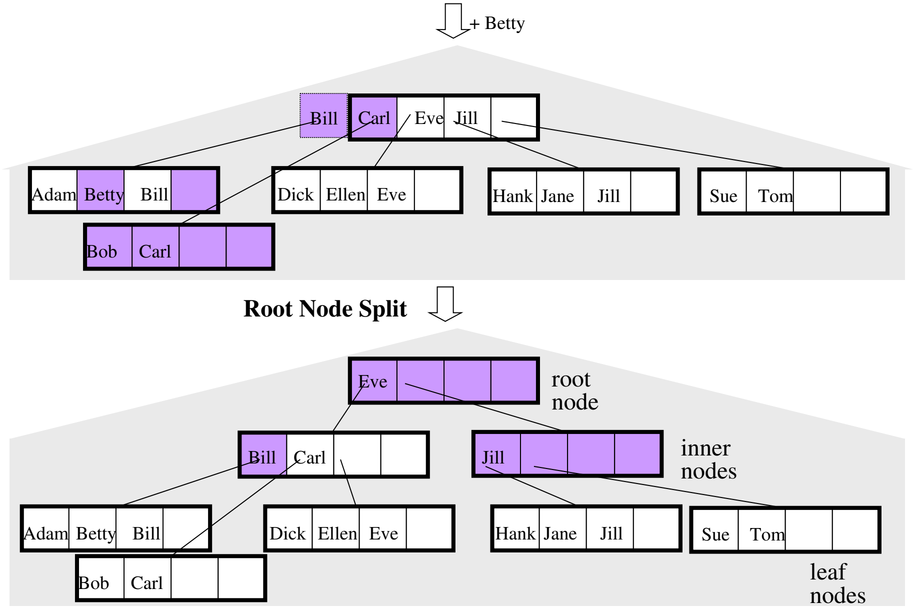
Simple Insertion into B+-tree Index



Insertion into B+-tree with Leaf Node Split



Insertion into B+-tree with Root Node Split



Chapter 9: Concurrency Control on Search Structures

- 9.2 Implementation by B$^+$-trees
- **9.3 Key-Range Locking at the Access Layer**
- 9.4 Techniques for the Page Layer
- 9.5 Further Optimizations
- 9.6 Lessons Learned

Simple Key-Range Locking

ADT interface for search structure:

insert (key, RID)
delete (key, RID)
search (key)
range_search (lowkey, highkey)

Protocol:

- insert, delete, and search lock single key (insert and delete in compatible modes)
- range_search locks interval [lowkey, highkey]
- table scan effectively locks interval [- ∞, + ∞]

\+ page locks acquired during subtransactions

→ lock manager needs "key in interval" test
→ range_search "preclaims" lock on entire interval

Incremental Key-Range Locking

refined ADT interface with
range_search(lowkey, highkey) replaced by:

search (lowkey) ↑key ↑page
next (currentkey, currentpage, highkey) ↑key ↑page
next ...

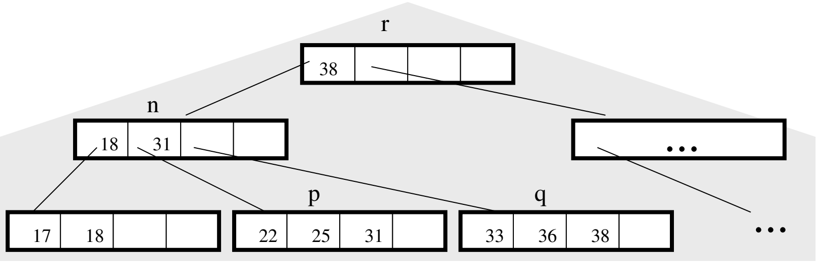
Approach:

operations lock intervals [found-key, next-existing-key)
identified by "found-key" (i.e., only keys that do exist in the index)
+ page locks during subtransactions

Incremental Key-Range (Previous-Key) Locking Protocol:

- search(x) requests read lock on x if x is present, or largest key < x if x is not found
- next(currentkey, ...) requests read lock on currentkey
- insert (y, RID) requests write locks on y and largest key < y
- delete (y, RID) requests write locks on y and largest key < y

Example: Incremental Key-Range Locking



range_search (23, 34):

```
search (23) ↑ 25 ↑ p
  lock page r, page n, page p
  lock key 22
  unlock pages r, n, p
next (25, p, 34) ↑ 31 ↑ p
  lock key 25
next (31, p, 34) ↑ 33 ↑ q
  lock key 31
next (33, q, 34) ↑ nil ↑ nil
  lock key 33
```

insert (27, ...):

```
lock page r, page n, page p
lock key 25
lock key 27
unlock pages r, n, p
```

Correctness of Incremental Key-Range Locking

Theorem 9.1:
Previous-key locking generates only conflict-serializable schedules as far as index operations are concerned.

Proof sketch:

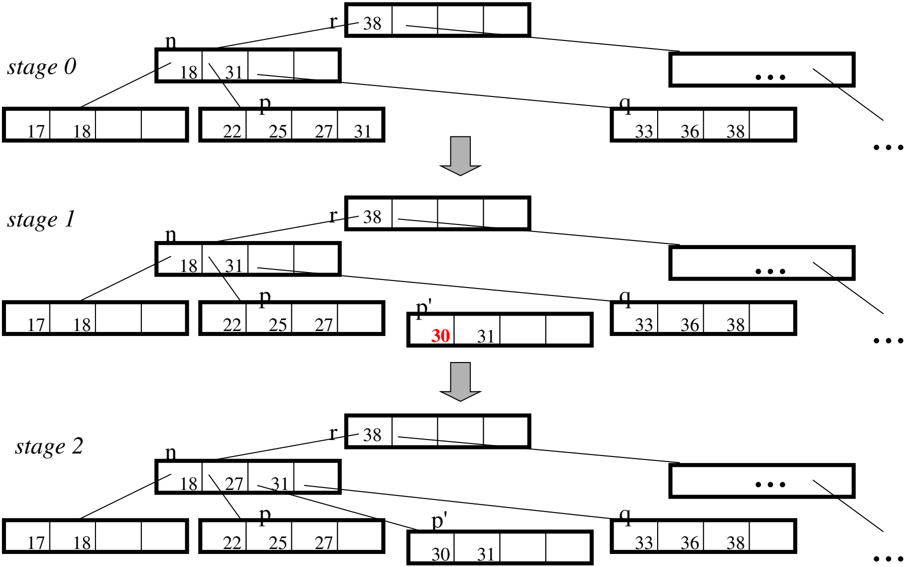
- search(x) is in conflict with insert(y, RID) or delete (y, RID) only for x=y
  - for successful search the conflict is detected by locks on x
  - for unsuccessful search the conflict is detected by locks on largest key < x
- range_search (low, high) is in conflict with insert (y, RID) or delete (y, RID) if y falls into [low, high]
  - this conflict is detected because range_search incrementally acquires locks on all keys from low or the largest key < low up to and including the largest key ≤ high, which must include the largest key < y
- insert (x, RID) and insert (y, RID) conflict only for x=y (and only for unique index)

…

Chapter 9: Concurrency Control on Search Structures

- 9.2 Implementation by B$^+$-trees
- 9.3 Key-Range Locking at the Access Layer
- **9.4 Techniques for the Page Layer**
- 9.5 Further Optimizations
- 9.6 Lessons Learned

Problem Scenario



Problem: *search(31) or insert(31) in between stage 1 and stage 2*

Solution 1: Lock Coupling (“Crabbing”)

Definition: A tree node is **split-safe** if it has enough free space to accommodate at least one additional routing key and child pointer

Protocol:

- Search operations need a read lock before accessing a node. Insert operations need a write lock before accessing a node.
- A lock can be granted only if there is no conflict and the requestor holds a lock (in the same mode) on the node's parent.
- Search operations can release a lock on a node once they have acquired a lock on a child of that node.
- Insert operations can release a lock on a node if
  - the node is split-safe and
  - they have acquired a lock on a child of that node

Theorem 9.2:
Lock coupling for search and insert operations generates only OCSR schedules.

Example: Lock Coupling

insert (30):	search (31):
write lock r	
write lock n	
unlock r	
	read lock r
	request read lock on n
write lock p	
allocate new page p'	
write lock p'	
split contents of p onto p and p'	
adjust contents of n	
release locks on n, p, p'	
	acquire read lock on n
	release lock on r
	read lock p'
	release lock on n
	return RID for key 31
	release lock on p'

Lock Coupling with Range Searches, Next, and Delete Operations

- the initial *search (lowkey)* of a *range_search (lowkey, highkey)* operation applies the locking rules for exact-key search operations
- a *next (currentkey, currentpage, highkey)* operation needs to acquire a read lock on *currentpage*, and
  it can acquire a lock on another leaf node
  only if it holds a lock on the preceding leaf.

- delete operations do *not* trigger node merging
- an empty node can be deallocated only when all transactions that were active at the time when the node became empty have terminated *("drain technique")*

Correctness of Extended Lock Coupling

Theorem 9.3:
Lock coupling with next operations generates only OCSR schedules.

Theorem 9.4:
(Extended) Lock coupling at the page layer together with incremental key-range locking at the access layer ensure tree reducibility of all 2-level schedules.

Proof sketch:

- By Theorems 9.1 and 9.3, schedules with search, insert, delete, and next operations are tree reducible.
- So the remaining problem scenario is of the form:
  ... search_i (lowkey) ... insert_k (x, RID_1) ... next_i (currentkey_1, ..., highkey) ...
  ... insert_l (y, RID_2) ... next_i (currentkey_2, ..., highkey) ...
  with active transactions t_i, t_k, t_l
- x cannot fall into [lowkey, currentkey_1] and y cannot fall into [lowkey, currentkey_2] because of previous-key lock conflicts
- So both insert_k (x, ...) and insert_l (y, ...) can be commuted to the left of t_i

Example: Extended Lock Coupling

range_search (24, 35):	insert (30):
search (24)	
read lock r, read lock n, unlock r	
read lock p, unlock n	
read lock key 22, unlock p	
	write lock r, write lock n, unlock r
next (25, p, 35)	
read lock p	
read lock key 25, unlock p	
	write lock p
next (27, p, 35)	
request read lock on p	
	write lock key 30, write lock key 27
	release locks on p, p', n
acquire lock on p	
request read lock on key 27	
	commit transaction
acquire lock on key 27	
read lock p', unlock p, unlock p'	
next (30, p', 35)	
read lock p', read lock key 30, unlock p'	
...	

Solution 2: Link Technique

Link protocol:

- Search operations need only lock the currently accessed node (no need for holding two page locks simultaneously)
- Upon “not found”, search and next operations proceed to the right sibling node until they have seen a larger key

Solution 3: Giveup Technique

Giveup protocol:

- All operations need only lock the currently accessed node (no need for holding two page locks simultaneously)
- Each node contains a “range field” for its subtree, maintained by splits on a per node basis
- Upon seeing a node with a range field that does not contain the search key, the operation “gives up” and is retried, starting again from the root

Example: Link Technique

insert (30):	search (31):
	read lock r
	release lock on r
	read lock n
	release lock on n
write lock r	
write lock n	
unlock r	
write lock p	
	request read lock on p
allocate new page p'	
write lock p'	
split contents of p onto p and p'	
adjust contents of n	
release locks on n, p, p'	
	acquire lock on p
	release lock on p
	read lock p'
	return RID for key 31
	release lock on p'

Chapter 9: Concurrency Control on Search Structures

- 9.2 Implementation by B$^+$-trees
- 9.3 Key-Range Locking at the Access Layer
- 9.4 Techniques for the Page Layer
- **9.5 Further Optimizations**
- 9.6 Lessons Learned

Further Optimizations

- Index traversal can use deadlock-free page latching rather than full-fledged locks
- Insert operations for the same key interval are commutative
  → insert lock mode compatible with itself,
  but incompatible with read
- Insert operations merely need instant-duration lock on previous key
- Delete operations that leave a “ghost key” for deferred garbage collection need to lock only the deleted key
- Fewer locks (but possibly less concurrency) by locking (key, RID) pairs or only RIDs

Chapter 9: Concurrency Control on Search Structures

- 9.2 Implementation by B⁺-trees
- 9.3 Key-Range Locking at the Access Layer
- 9.4 Techniques for the Page Layer
- 9.5 Further Optimizations
- **9.6 Lessons Learned**

Lessons Learned

- Index concurrency control is a perfect example
  for **layered schedules**
- At the **access layer**, a primitive form of predicate locking is used,
  namely, key-range locking, and
  optimized for incremental, low-overhead lock acquisition
- At the **page layer**, short-term locks or latches are used
  to isolate index operations,
  with protocols ranging from S2PL for subtransactions to
  lock coupling, link techniques, or give-up protocols
- Locking rules at the two levels are **integrated** with each other