

Hierarchical locking in B-tree indexes

Götz Graefe

HP Labs

Palo Alto, CA

Overview

- Problems
- Key range locking
 - Traditional & hierarchical techniques
- Locks on separator keys
 - Node splits, lock escalation, granularity changes
- Locks on key prefixes
 - Key insertion, lock escalation, granularity changes
- Conclusions

Problem 1: scalability

- Lock hierarchy today: index, page, key
 - 1 TB / 4-32 KB = 32-256 M pages
- Lock escalation after ~10-100 K locks
 - Each lock costs 64-96 B and 1-3 K cycles
 - Effectively to 32-256 M pages
 - Huge stepping, intermediate lock granularity
- One bad lock escalation ends concurrency

Problem 2: complexity

Traditional key range locking

- Non-intuitive lock modes, e.g., *RangeI_N*
- Violations of two-phase locking
- Missing lock modes, lack of concurrency
- Asymmetry of key insertion and deletion

Assumptions

- OLTP and DSS are merging
 - “Zero latency” operational decision support
 - Continuous updates as well as large queries
- Serializability must be supported
 - Read-committed etc. are “turbo options”
 - Used only due to poor concurrency
- Predicate locking is correct
 - Key range locking is a practical special case
- Locks & latches are different

Latches versus locks

	Latches	Locks
Separate ...	Threads	Transactions
Protect ...	In-memory data structures	Database contents
Modes	Shared, exclusive, (perhaps) update	Shared, exclusive, update, intention, escrow, etc.
Duration	Critical section	End of transaction
Deadlock ...	Avoidance	Detection and resolution
... by ...	Coding discipline, "lock leveling"	Lock manager, graph traversal, transaction abort, partial rollback, lock de-escalation

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X	
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C	C
RS-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C	C

Key

N	No Conflict	SIU	Share with Intent Update
I	Illegal	SIX	Shared with Intent Exclusive
C	Conflict	UIX	Update with Intent Exclusive
NL	No Lock	BU	Bulk Update
SCH-S	Schema Stability Locks	RS-S	Shared Range-Shared
SCH-M	Schema Modification Locks	RS-U	Shared Range-Update
S	Shared	RI-N	Insert Range-Null
U	Update	RI-S	Insert Range-Shared
X	Exclusive	RI-U	Insert Range-Update
IS	Intent Shared	RI-X	Insert Range-Exclusive
IU	Intent Update	RX-S	Exclusive Range-Shared
IX	Intent Exclusive	RX-U	Exclusive Range-Update
		RX-X	Exclusive Range-Exclusive

SQL Server lock modes

Issues with traditional key range locks

- Non-intuitive lock modes, no formal derivation
 - Education, maintenance, improvements, testing
- Non-intuitive lock duration
 - “Test & release” “insert” locks
- Missed opportunities for concurrency
 - Missing lock modes, lack of orthogonality
 - Extra penalty for serializable isolation level
- Next-key locking versus prior-key locking

Next-key versus prior-key locking

- Minor issue – mostly in merged indexes
- When inserting a component (e.g., line item), lock the same object (e.g., order) or the next?
- However, beware of “append” operations

Order 4711, Customer “Smith”, ...
Order 4711, Line 1, Quantity 3, ...
Order 4712, Customer “Jones”, ...

Missing lock modes

- Missing RangeX_N mode
 - Exclusive lock on gap without locking a key
 - One transaction inserts into the gap while another transaction reads or updates the boundary key
- Missing RangeS_N mode
 - Locking absence of keys without locking a key

Specific examples

- Clustered index on column A
 - Existing keys 10, 20, 30, 40, 50, ...
 - Additional columns B, C, D, E, ...
- Missing RangeS_N mode
 - T1: “select count (*) ... where A = 22”
 - T2: “update ... set D = ... where A in (20, 30)”
 - Shared (20,30] conflicts with exclusive [30,30]
- Missing RangeX_N mode
 - T1: “insert ... values (24, ...)”
 - T2 as above conflicts with exclusive (20,30]

Desired lock modes

- Locks on key values
 - $[123, 123]$ when updating columns B, C, D, E
- Locks on gaps (open interval) only
 - $(123, 125)$ to prevent phantom 124
- Locks on both key and gap
 - $[127, 129)$ as component of a large range scan

120 121 123 125 127 129 130

Strict hierarchal key range locking

- Apply traditional theory to the problem
 - Key value, gap (open interval), their combination
 - Automatic derivation of required new lock modes
- Traditional optimization to reduce lock calls
 - Same # of lock manager invocations
- Key insertion via ghost records
 - System transaction creates & releases locks
 - User transaction turns ghost into valid record
 - High concurrency due to short lock duration

Hierarchical key range locking

- Locking a key for record update
 - IX on the combination, X on the key, \emptyset on the gap
- Locking only a gap for key absence
 - IS on the combination, \emptyset on the key, S on the gap
- All combinations possible and useful

	S	X	IS	IX
S	ok		ok	
X				
IS	ok		ok	ok
IX			ok	ok

Hierarchical locking in B-trees

Lock manager invocations

- Combine IS+S+Ø into SØ etc.
Cut lock manager invocations by factor 2-3x
- Strict application of standard techniques

No new semantics

Automatic derivation

	S	X	IS	IX
S	ok		ok	
X				
IS	ok		ok	ok
IX			ok	ok

	S	X	SØ	ØS	XØ	ØX	SX	XS
S	ok		ok	ok				
X								
SØ	ok		ok	ok		ok	ok	
ØS	ok		ok	ok	ok			ok
XØ				ok		ok		
ØX			ok		ok			
SX			ok					
XS				ok				

Key insertion and deletion

- Standard technique for deletion: ghost keys
 - Also known as “pseudo-deleted records”
 - Marked invalid, not seen by future queries
 - Lockable resources, including deletion operation
 - Simplifies recovery of deletion – locks, space
 - Space reclamation by asynchronous clean-up
- Also useful in key insertion
 - “System transaction” creates new ghost(s)
 - User transaction marks it valid
 - High concurrency among user insertions

Missing lock modes revisited

- Missing RangeX_N mode
 - Exclusive lock on gap without locking a key
 - One transaction inserts into the gap while another transaction reads or updates the boundary key
 - Possible by $\emptyset X$ mode – key free, gap exclusive – with another transaction using $S\emptyset$ or $X\emptyset$
- Missing RangeS_N mode
 - Locking absence of keys without locking a key
 - Possible by $\emptyset S$ mode – key free, gap shared

Specific examples revisited

- Clustered index on column A
 - Existing keys 10, 20, 30, 40, 50, ...
 - Additional columns B, C, D, E, ...
- Missing RangeS_N mode
 - T1: “select count (*) ... where A = 22”
 - T2: “update ... set D = ... where A in (20, 30)”
 - T1: $\emptyset S$ – key free, gap shared – on key 20
 - T2: $X\emptyset$ – key exclusive, gap free – on key 20
- Gap (open interval) can be locked without locking the key value itself

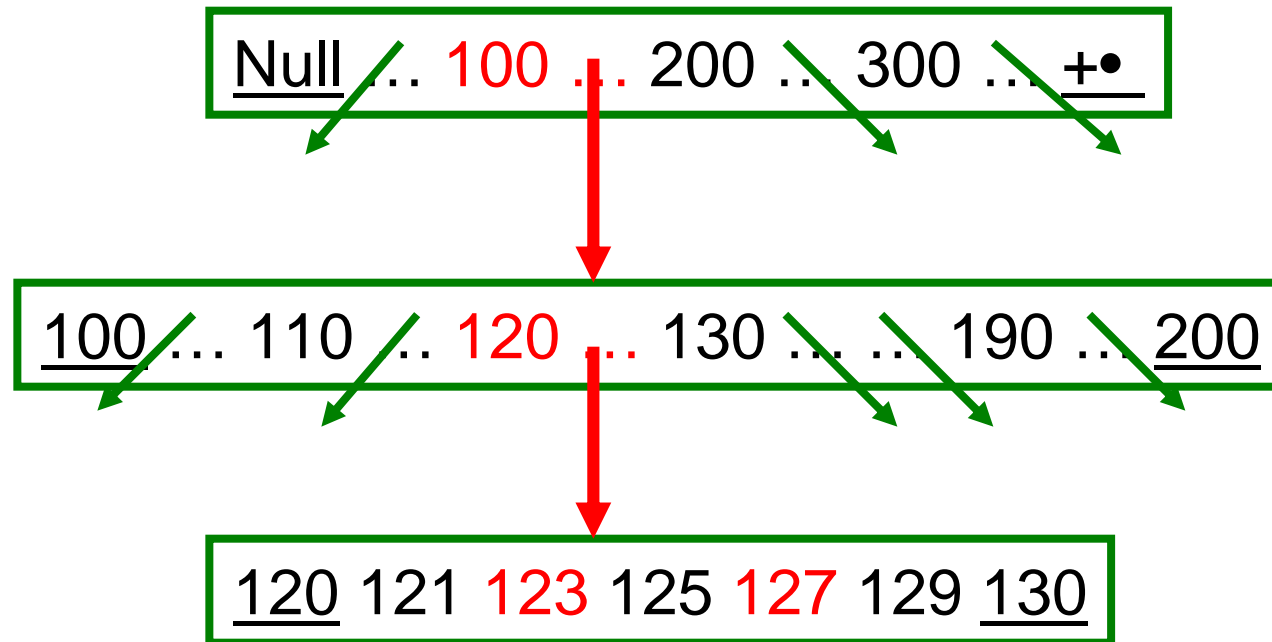
Summary: hierarchical locks in leaves

- “Radically old” design
- Sound theory – no “creative” lock modes
 - Strict application of multi-granularity locking
 - Automatic derivation of “macro” lock modes
 - Lock retention to transaction end
- More concurrency than traditional designs
 - Missing lock modes due to lack of orthogonality
- Key insertion & deletion via ghost records
 - Efficient system transactions
 - Symmetric design

Scalability issues

- Many pages, many keys per B-tree index
 - $1 \text{ TB} / 8 \text{ KB} = 128 \text{ M}$, $1 \text{ TB} / 100 \text{ B} = 10 \text{ G}$
- Lock escalation after 10-100 K locks
 - Each lock costs 64-96 B and 1-3 K cycles
 - Increases “lock footprint” up to 1,000,000-fold
 - Multi-programming collapses, no concurrency
- Needed: intermediate granularities of locking
- Two approaches:
 - B-tree structure – key range locking on separators
 - Key structure – key range locking on key prefixes

Example: locks on separator keys



“IX-100-2” for [100, 200)

“S-200-2” for [200, 300)

“IX-120-1” for [120, 130)

“X-127-0” for [127, 129)

“X∅-123-0” for [123, 123]

Hierarchical locks on separator keys

- Exploit B-tree structure as locking hierarchy
 - Large and very large key ranges as appropriate
 - Natural adaptation to data skew
- Locks cover the gap to the next separator key
 - At the same level of the B-tree
 - Fence keys may be helpful
 - No locks on separator values only or on gaps only
- Lock identifier includes node level
 - In addition to index identifier and hash (key value)

Hierarchical locks on separator keys

- All key locks are independent of page location
 - Predicate locks permit structure modifications
 - Page splits, B-tree defragmentation, ...
- Intention locks and absolute locks
- Shared, exclusive, increment, etc. locks
- Large locks
 - Low overhead in query processing
 - Low probability of late conflict and deadlock
- High overhead during transaction processing?

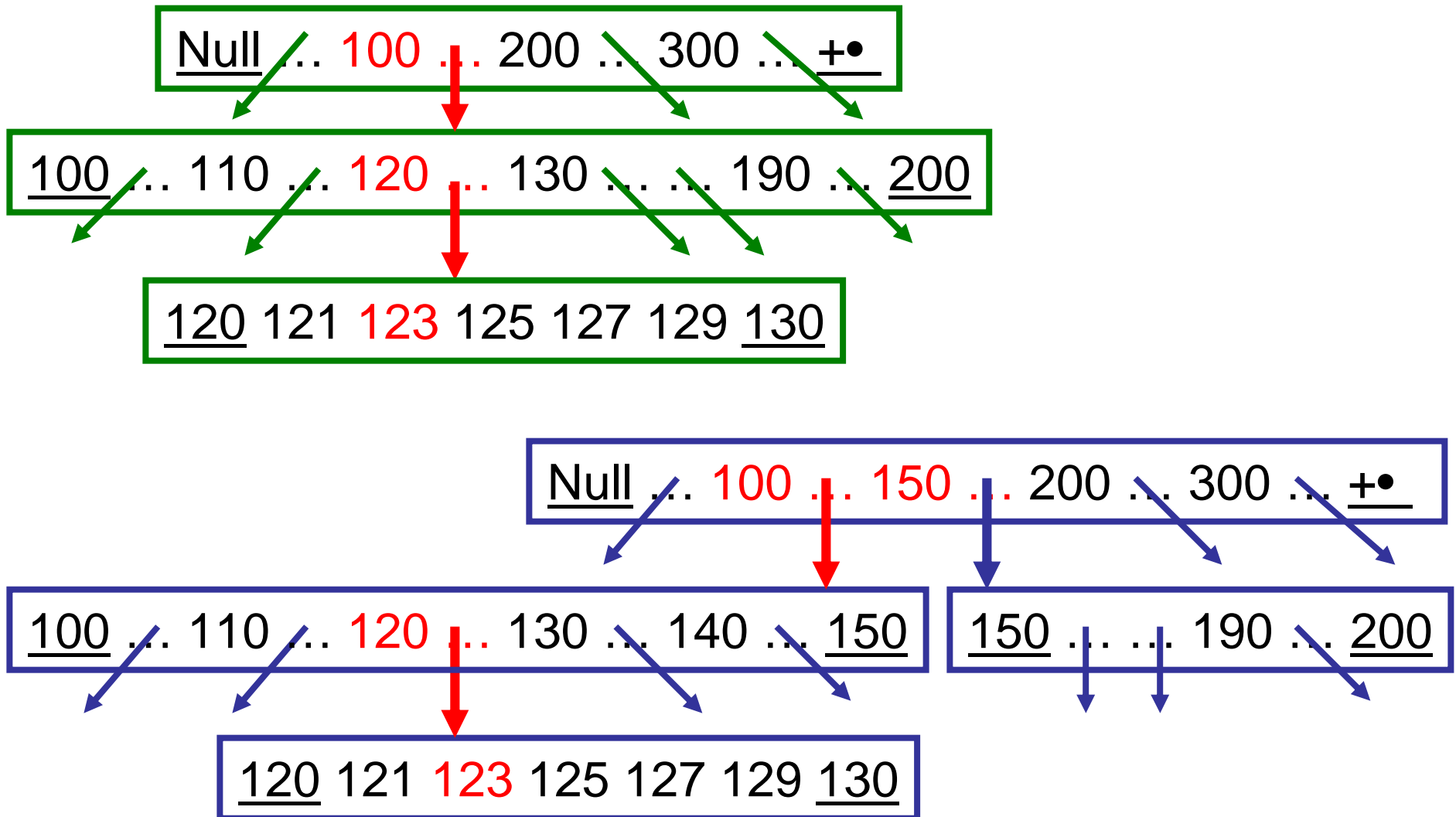
Node split & merge operations

- Node split also splits key range in parent
 - No disruption of active transactions
 - Compare to ghost insertion in a leaf
- Node merge requires merging key locks
 - Granted locks must be compatible
 - Wait lists must be merged
 - Compare to ghost deletion in a leaf
- Load balancing
 - Conceptually, merge and then split differently

Example: split at parent level

- Data & locks at leaf level not affected
- Separator keys at parent levels
 - Records move to a new page
 - Locks remain in place unchanged
- Separator keys at grandparent level
 - One new record is inserted
 - Locks are duplicated for the new separator key

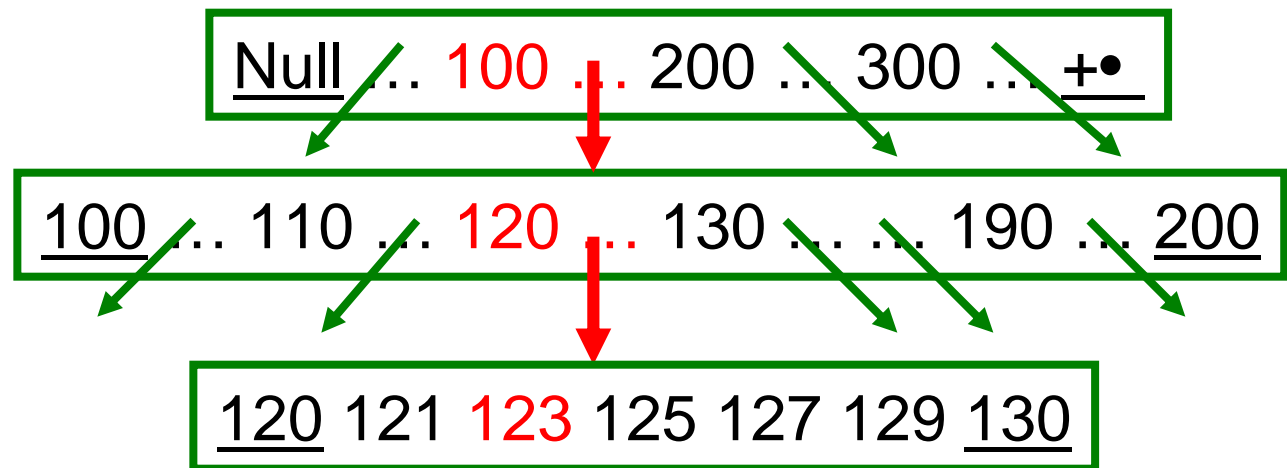
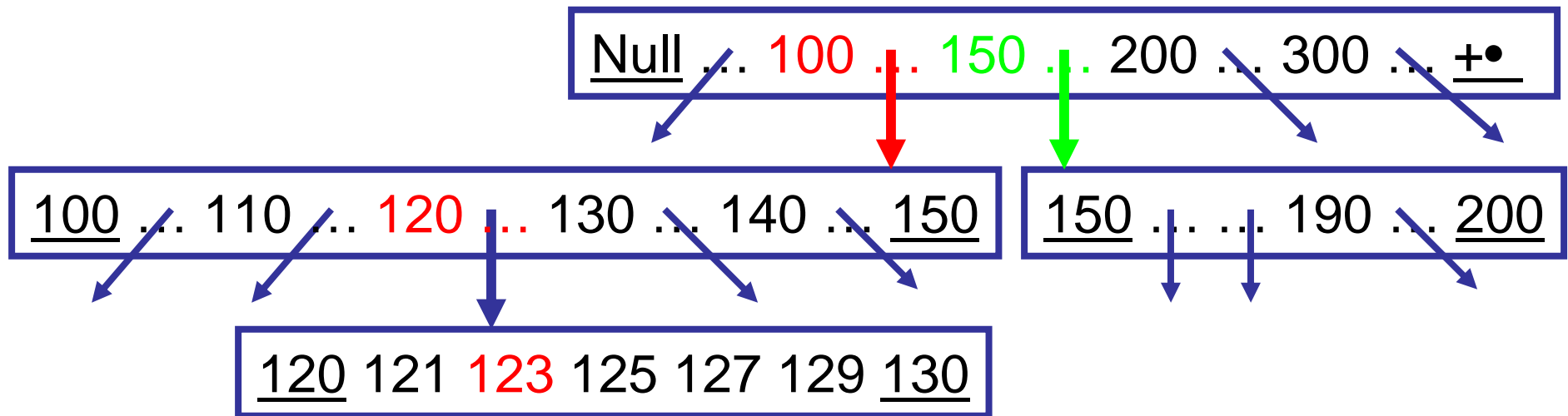
Example: split at parent level



Example: merge at parent level

- Data & locks at leaf level not affected
- Separator keys at parent levels
 - Records move from the old page
 - Locks remain in place unchanged
- Separator keys at grandparent level
 - Granted locks on removal victim must be compatible with granted locks on its neighbor
 - Granted locks must be merged
 - Wait queues must be merged

Example: merge at parent level



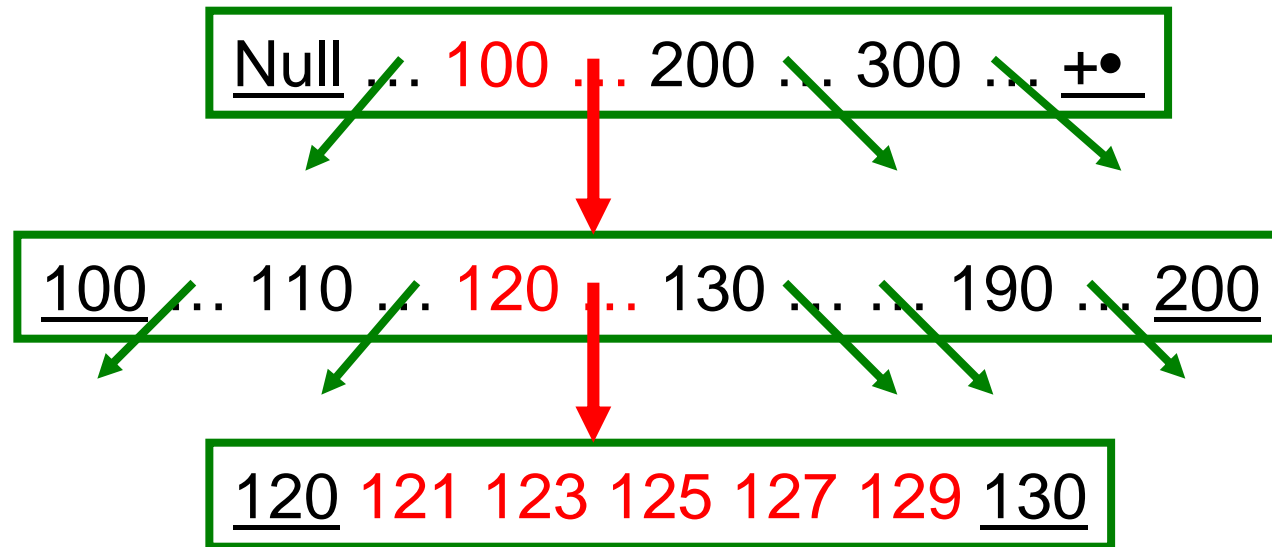
Lock escalation & de-escalation

- Optional but required for full benefit
- Escalation factor equals B-tree fan-out
 - 100-1,000x rather than up to 1,000,000x
 - Consistent for any and all B-trees
- De-escalation requires lock information
 - In local space – 12-16 B, 100s of cycles per lock
 - Many transactions use few lock manager calls
 - No conflicts during insertion of small locks
 - Lock large initially and de-escalate on demand?

Example: escalation to parent lock

- Before
 - Intention lock on separator key in parent
 - Absolute locks on keys in leaves
- After
 - Absolute lock on separator key in parent
 - Locks on keys in leaves may be removed
 - List of removed & avoided locks in leaves
- Future de-escalation remains possible
 - Even during two-phase commit

Example: escalation to parent lock



“IS-100-2” for [100, 200)

“IS-120-1” for [120, 130)

“S-121-0” for [121, 123)

“S-123-0” for [123, 125)

...

“IS-100-2” for [100, 200)

“S-120-1” for [120, 130)

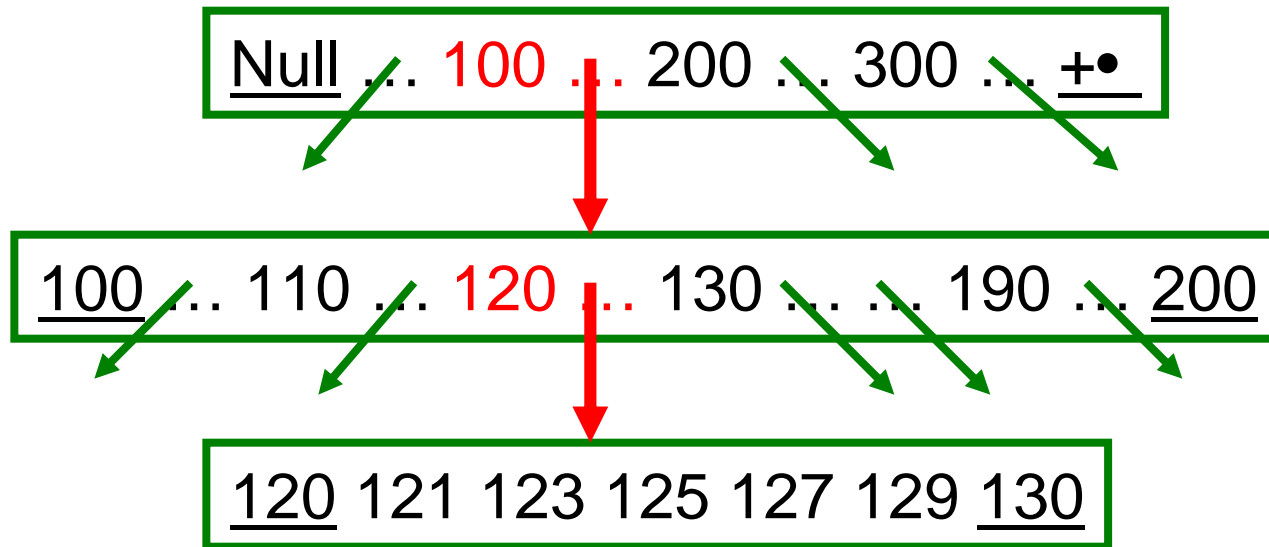
No leaf-level locks

Cached for possible future
de-escalation

Example: de-escalation to parent lock

- Before
 - Absolute lock on separator key in grandparent
 - Lists of avoided locks in parents and leaves
- After
 - Intention lock on separator key in grandparent
 - Absolute locks on separator keys in parent
 - List of avoided locks in leaves
- Efficiency
 - Small entries in lists of avoided locks
 - No contention during lock acquisition in parent

Example: de-escalation to parent lock



“S-100-2” for [100, 200)

No parent-level locks

No leaf-level locks

“IS-100-2” for [100, 200)

“S-120-1” for [120, 130)

No leaf-level locks

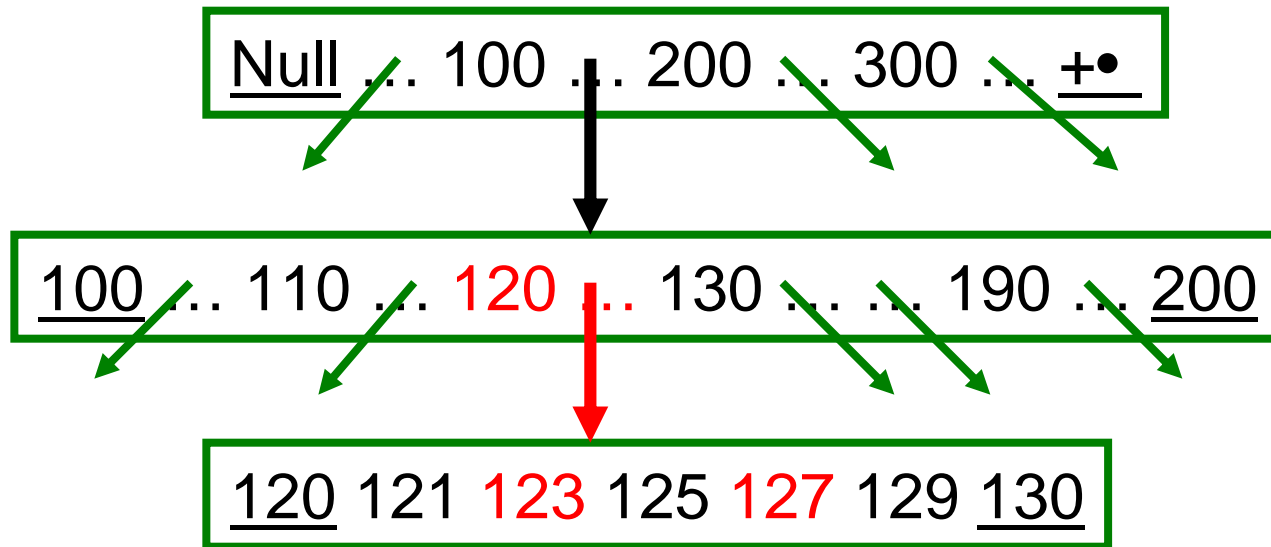
Hierarchy changes

- A new technique – not used anywhere to-date
- Optional but required for full benefit & utility
 - Minimal overhead for OLTP – single key lock
 - Large locks for query processing
 - Large locks when, where, and while needed
 - Label each B-tree node whether to lock its keys
- Change the lock hierarchy on-the-fly
 - Abandoning a granularity of locking
 - Delayed or immediate introduction

Example: abandon locks in a parent

- Actually abandon a granularity of locking
- Force de-escalation to intention locks
 - Cannot create new conflicts
 - No waiting or delay
- Simply remove those intention locks
 - From lock manager's hash table
 - From transactions' internal data structures
 - Tricky during partial transaction rollback

Example: abandon locks in a parent



“IS-120-1” for [120, 130)

“S-123-0” for [123, 125)

“S-127-0” for [127, 129)

No parent-level locks

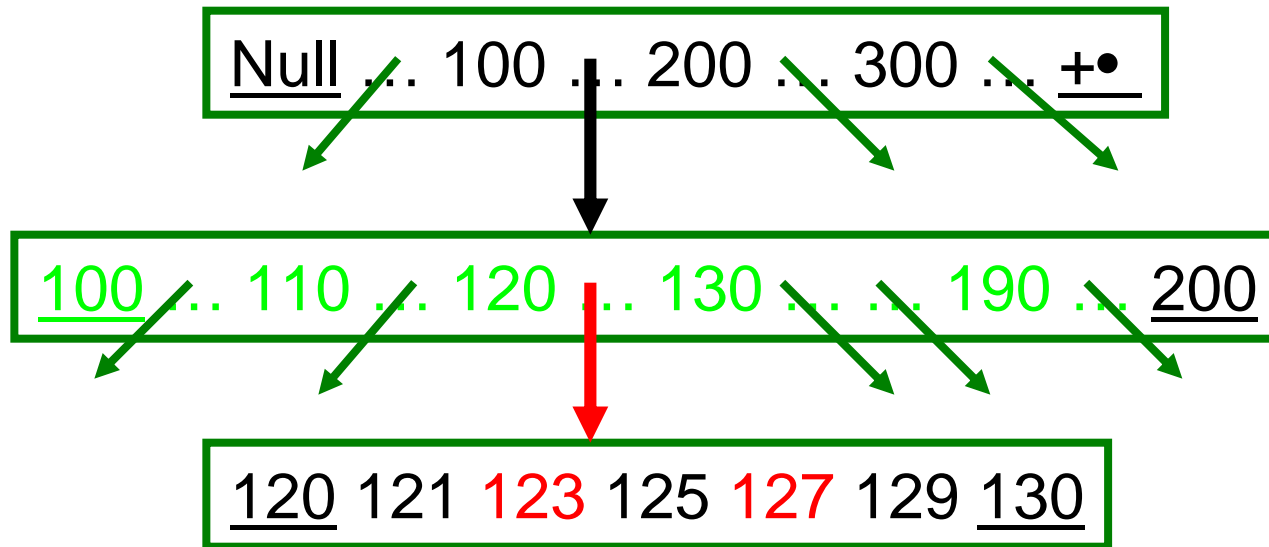
“S-123-0” for [123, 125)

“S-127-0” for [127, 129)

Example: locks in a parent – deferred

- Label the node to require locking keys
 - Applies to all new transactions
- Permit intention locks only – no absolute locks
 - Retain a transaction holding all intention locks
- Wait until all active transactions have finished
 - No disruptions of active or new transactions
- Permit intention and absolute locks
 - Commit transaction holding all intention locks

Example: locks in a parent – deferred



No parent-level locks

“S-123-0” for [123, 125)

“S-127-0” for [127, 129)

System transaction:

“IS-100-1” for [100, 110)

“IS-110-1” for [110, 120)

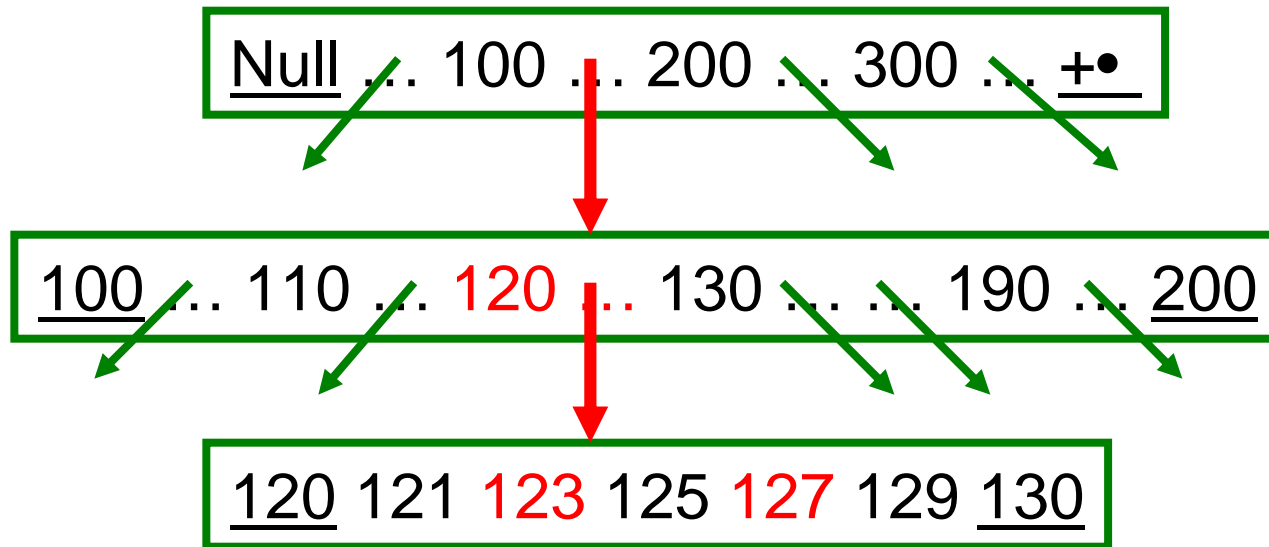
...

“IS-190-1” for [190, 200)

Example: locks in a parent – immediate

- Insert locks for active transactions
 - Find all locks in the parent's children (leaves)
 - Insert appropriate intention locks for separators
- Alternative design: label per key, not per node
 - More labels, faster immediate coarse locks
- Also: organize locks like B-tree hierarchy
 - Integrate into the buffer pool manager?

Example: locks in a parent – immediate



No parent-level locks

“S-123-0” for [123, 125)

“S-127-0” for [127, 129)

“IS-120-1” for [120, 130)

“S-123-0” for [123, 125)

“S-127-0” for [127, 129)

Summary: locks on separator keys

- Predicate locks exploit the B-tree structure
 - Range size, fan-out, skew
- Structural changes during user transactions
 - Split & merge & load balancing
- Adaptive granularities of locking
 - Information per node whether to lock or not
 - Minimal lock overhead for large queries
 - Maximal concurrency among small transactions
 - Minimal lock overhead if all transactions are small

Hierarchical locks on key prefixes

- Generalization of Tandem's "generic locking"
 - Predicate locking using the leading key bytes
- Leading columns in multi-column B-tree
 - Also: bytes within columns
 - Also: bytes in record identifiers
- Lock escalation & de-escalation
- Hierarchy changes
 - Uniform for an entire B-tree
 - Probably not practical

Summary

- Key range locking
 - Simplified by strictly using hierarchal techniques
 - Automatic derivation of lock modes
 - Increased concurrency with equal overhead
- Locks on separator keys
 - Lock escalation and de-escalation
 - Granularity changes, dynamic locking hierarchy
 - Serve combined OLTP and data warehousing
- Locks on key prefixes

Orthogonal techniques

- Cache optimizations
- Partitioned B-trees
 - Sort, index creation, bulk insertion & deletion
- Write-optimized B-trees
 - Defragmentation, RAID, flash memory
- Increment (“escrow”) locking
 - High concurrency in indexed summary views
- Merged B-tree indexes
 - Master-detail clustering based on sort order

Conclusions

- B-trees are proven and robust
 - Operational abilities better than other index forms
- B-trees need further research & improvement
 - Must adapt to new hardware – many-core, flash
 - May exploit new software – transactional memory
- OLTP and DSS are merging
 - Revisit concurrency control and recovery, versioning, online index operations, etc.
- Availability by MTTF and MTTR
 - Faster restart recovery for databases with B-trees