Database Management and Performance Tuning

Concurrency Tuning

Pei Li

University of Zurich
Institute of Informatics

Unit 8

Acknowledgements: The slides are provided by Nikolaus Augsten
and adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

1. Concurrency Tuning
   - Lock Tuning

Concurrency Tuning Goals

- Performance goals:
  - reduce blocking (one transaction waits for another to release its locks)
  - avoid deadlocks and rollbacks
- Correctness goals:
  - serializability: each transaction appears to execute in isolation
  - note: correctness of serial execution must be ensured by the programmer!

Trade-off between performance and correctness!

Ideal Transaction

- Acquires few locks.
- Favors shared locks over exclusive locks.
  - only exclusive locks create conflicts
- Acquires locks with fine granularity.
  - granularities: table, page, row
  - reduces the scope of each conflict
- Holds locks for a short time.
  - reduce waiting time

Lock Tuning

1. Eliminate unnecessary locks
2. Control granularity of locking
3. Circumvent hot spots
4. Isolation guarantees and snapshot isolation
5. Split long transactions

1. Eliminate Unnecessary Locks

- Lock overhead:
  - memory: store lock control blocks
  - CPU: process lock requests
- Locks not necessary if
  - only one transaction runs at a time, e.g., while loading the database
  - all transactions are read-only, e.g., decision support queries on archival data

2. Control Granularity of Locking

- Locks can be defined at different granularities:
  - row-level locking (also: record-level locking)
  - page-level locking
  - table-level locking
- Fine-grained locking (row-level):
  - good for short online-transactions
  - each transaction accesses only a few records
- Coarse-grained locking (table-level):
  - avoid blocking long transactions
  - avoid deadlocks
  - reduced locking overhead

Lock Escalation

- Lock escalation: (SQL Server and DB2 UDB)
  - automatically upgrades row-level locks into table locks if number of row-level locks reaches predefined threshold
  - lock escalation can lead to deadlock
- Oracle does not implement lock escalation.

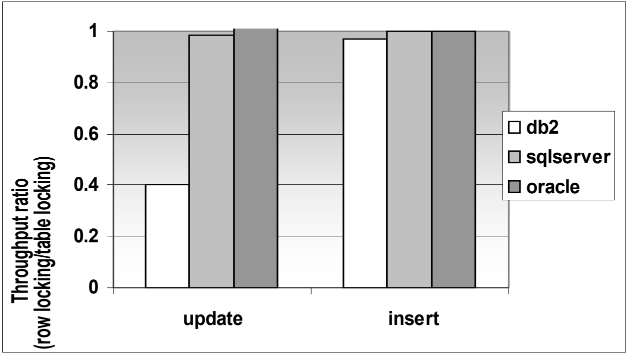
Granularity Tuning Parameters

1. Explicit control of the granularity:
   - within transaction: statement within transaction explicitly requests a table-level lock, shared or exclusive (Oracle, DB2)
   - across transactions: lock granularity is defined for each table; all transactions accessing this table use the same granularity (SQL Server)
2. Escalation point setting:
   - lock is escalated if number of row-level locks exceeds threshold (escalation point)
   - escalation point can be set by database administrator
   - rule of thumb: high enough to prevent escalation for short online transactions
3. Lock table size:
   - maximum overall number of locks can be limited
   - if the lock table is full, system will be forced to escalate

Overhead of Table vs. Row Locking

- Experimental setting:
  - `accounts(number,branchnum,balance)`
  - clustered index on account number
  - 100,000 rows
  - SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
  - lock escalation switched off
- Queries: (no concurrent transactions!)
  - 100,000 updates (1 query)
    example: `update accounts set balance=balance*1.05`
  - 100,000 inserts (100,000 queries)
    example: `insert into accounts values(713,15,2296.12)`

Overhead of Table vs. Row Locking

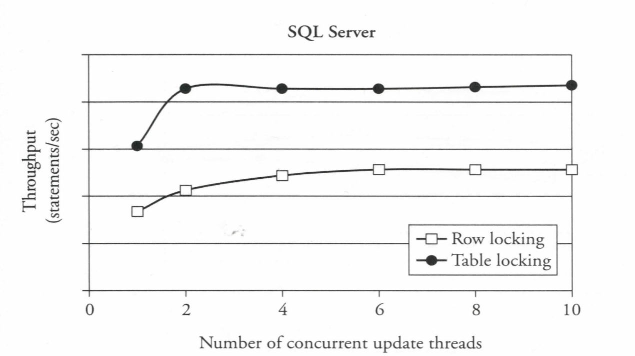


- Row locking (100k rows must be locked) should be more expensive than table locking (1 table must be locked).
- SQL Server, Oracle: recovery overhead (logging changes) hides difference in locking overhead
- DB2: low overhead due to logical logging of updates, difference in locking overhead visible

Experiment: Fine-Grained Locking

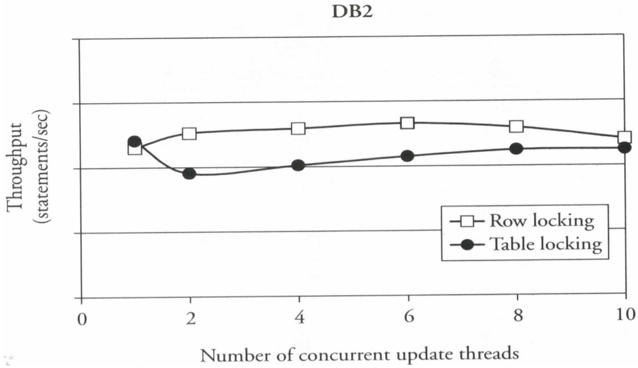
- Experimental setting:
  - table with bank accounts
  - clustered index on account number
  - long transaction (summation of account balances)
  - multiple short transactions (debit/credit transfers)
  - parameter: number of concurrent transactions
  - SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
  - lock escalation switched off

Experiment: Fine-Grained Locking



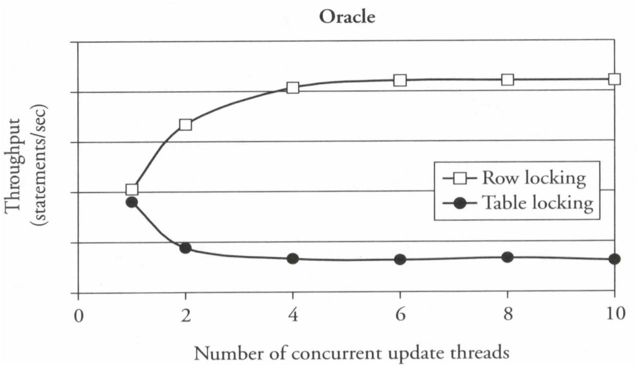
- Serializability with row locking forces key range locks.
- Key range locks are performed in clustered index.
- SQL Server: Clustered index is sparse, thus whole pages are locked.
- Row-level locking only slightly increases concurrency.
- Table-locking prevents rollback for summation query.

Experiment: Fine-Grained Locking



- Row locking slightly better than table locking.
- DB2 automatically selects locking granularity if not forced manually.
  - index scan in this experiment leads to row-level locking
  - table scan would lead to table-level locking

Experiment: Fine-Grained Locking



- Oracle uses snapshot isolation: summation query not in conflict with short transactions.
- Table locking: short transactions must wait.

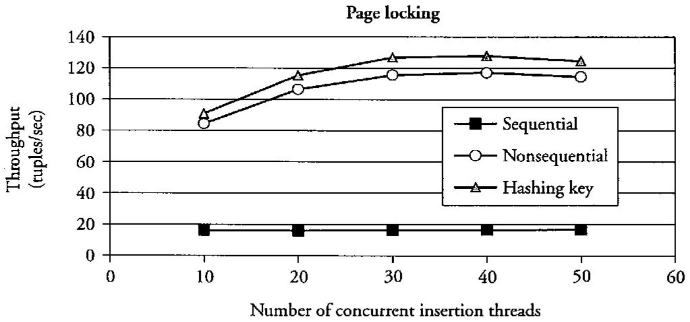
3. Circumvent Hot Spots

- Hot spot: items that are
  - accessed by many transactions
  - updated at least by some transactions
- Circumventing hot spots:
  - access hot spot as late as possible in transaction
    (reduces waiting time for other transactions since locks are kept to the end of a transactions)
  - use partitioning, e.g., multiple free lists
  - use special database facilities, e.g., latch on counter

Partitioning Example: Distributed Insertions

- Insert contention: last table page is bottleneck
  - appending data to heap file (e.g., log files)
  - insert records with sequential keys into table with B^+-tree
- Solutions:
  - use clustered hash index
  - if only B^+ tree available: use hashed insertion time as key
  - use row locking instead of page locking
  - if only reads are scans: define many insertion points (composite index on random integer ($1..k$) and key attribute)

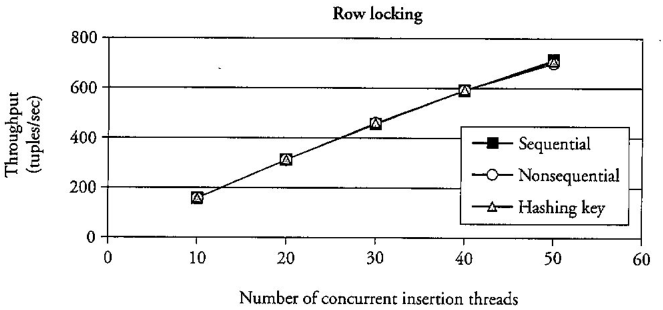
Experiment: Multiple Insertion Points and Page Locking



- Sequential: clustered B^+-tree index and key in insert order
- Non-sequential: clustered B^+-tree, key independent of insert order
- Hashing: composite index on random integer $(1..k)$ and key attribute
- Page locking and sequential keys: insert contention!

SQL Server 7 on Windows 2000

Experiment: Multiple Insertion Points and Row Locking



- No insert contention with row locking.

SQL Server 7 on Windows 2000

Partitioning Example: DDL Statements and Catalog

- Catalog: information about tables, e.g., names, column widths
- Data definition language (DDL) statements must access catalog
- Catalog can become hot spot
- Partition in time: avoid DDL statements during heavy system activity

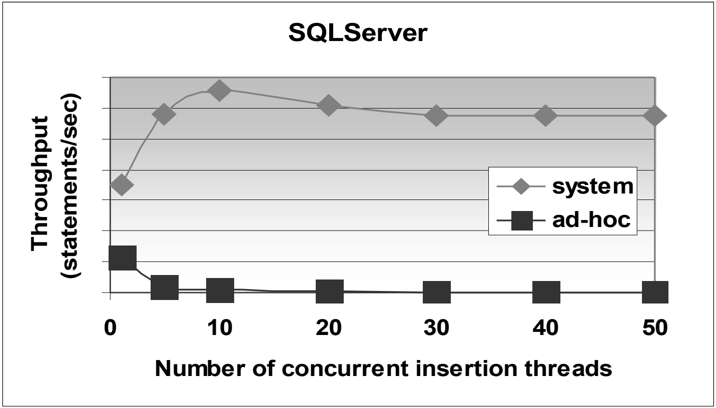
Partitioning Example: Free Lists

- Lock contention on free list:
  - free list: list of unused database buffer pages
  - a thread that needs a free page locks the free list
  - during the lock no other thread can get a free page
- Solution: Logical partitioning
  - create several free lists
  - each free list contains pointers to a portion of free pages
  - a thread that needs a free page randomly selects a list
  - with n free list the load per list is reduced by factor $1/n$

System Facilities: Latch on Counter

- Example: concurrent inserts with unique identifier
  - identifier is created by a counter
  - 2-phase locking: lock on counter is held until transaction ends
  - counter becomes hot spot
- Databases allow to hold a latch on the counter.
  - latch: exclusive lock that is held only during access
  - eliminates bottleneck but may introduce gaps in counter values
- Counter gaps with latches:
  - transaction T_1 increments counter to i
  - transaction T_2 increments counter to $i + 1$
  - if T_1 aborts now, then no data item has identifier i

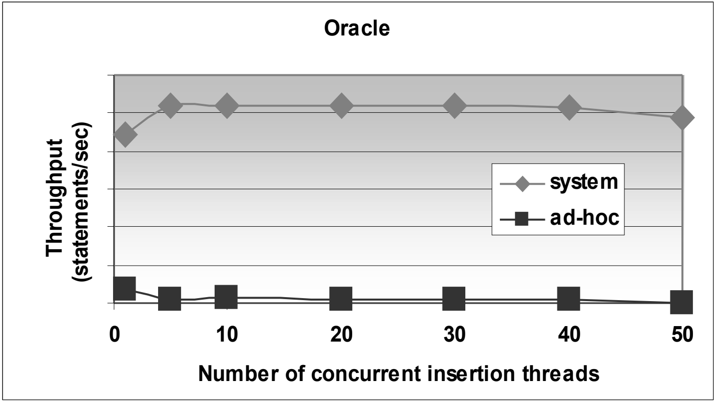
Experiment: Latch vs. Lock on Counter



- System (=latch): use system facility for generating counter values ("identity" in SQL Server)
- Ad hoc (=lock): increment a counter value in an ancillary table

SQL Server 7 on Windows 2000

Experiment: Latch vs. Lock on Counter



- System (=latch): use system facility for generating counter values ("sequence" in Oracle)
- Ad hoc (=lock): increment a counter value in an ancillary table

Oracle 8i EE on Windows 2000