Lock Tuning

Concurrency Control Goals

- **Performance goals**
  - Reduce blocking
    - One transaction waits for another to release its locks
  - Avoid deadlocks
    - Transactions are waiting for each other to release their locks

- **Correctness goals**
  - Serializability: each transaction appears to execute in isolation
  - The programmer ensures that serial execution is correct.

Trade-off between correctness and performance
Ideal Transaction

- Acquires few locks and favors shared locks over exclusive locks
  - Reduce the number of conflicts -- conflicts are due to exclusive locks
- Acquires locks with fine granularity
  - Reduce the scope of each conflict
- Holds locks for a short time
  - Reduce waiting

Lock Tuning

- Use special facilities for long reads
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
- Select appropriate granularity of locking
- Avoid DDL statements
- Bottlenecks
  - Using system features to circumvent bottlenecks
- Transaction Chopping
  - Rewriting applications to obtain best locking performance
Lock Tuning

- **Use special facilities for long reads**
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
- Select appropriate granularity of locking
- Avoid DDL statements
- Bottlenecks
  - Using system features to circumvent bottlenecks
- Transaction Chopping
  - Rewriting applications to obtain best locking performance

Facilities for long reads

- In some systems (e.g., Oracle), read-only queries hold no locks yet appear to execute serializably
- **Method used** *(snapshot isolation in Oracle)*:
  - Re-create an old version of any data item that is changed after the read query begins
  - Gives the effect that a read-only transaction R reads the database as the database appeared when R began
Snapshot isolation

- Each transaction executes against the version of the data items that was committed when the transaction started:
  - No locks for read
  - Locks for writes
  - Costs space (old copy of data must be kept)
- Almost serializable level, when extended to read/write transactions
  - T1: x:=y
  - T2: y:=x
  - Initially x=3 and y =17
  - Serial execution: x,y=17 or x,y=3
  - Snapshot isolation: x=17, y=3 if both transactions start at the same time.

Recommendation (by default)

- To use snapshot isolation for read-only transactions, but ensure that read operations hold locks for transactions that perform updates.
Eliminate unnecessary locking

- Locking is not necessary in two situations:
  1. Only one transaction runs at a time
     - E.g. When loading the DB
  2. When all transactions are read-only
     - E.g., decision support queries on archival DB
- Reduce overhead by suppressing the acquisition of locks
- Memory consumption for lock control blocks + processor time to process lock requests

If object not found in hash table, it is unlocked
## Locking in SQL Server 7

### syslockinfo

<table>
<thead>
<tr>
<th>spid</th>
<th>dbid</th>
<th>objid</th>
<th>lock</th>
<th>granularity</th>
<th>lock owner</th>
<th>lock waiter</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>117</td>
<td>RID</td>
<td>X</td>
<td>LO1</td>
<td>LW1, LW4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>117</td>
<td>PAG</td>
<td>IX</td>
<td>LO1</td>
<td>LW3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>117</td>
<td>TAB</td>
<td>IX</td>
<td>LO1</td>
<td>LW2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>118</td>
<td>RID</td>
<td>S</td>
<td>LO2, LO3</td>
<td>LW2</td>
</tr>
</tbody>
</table>

- Lock – 32 bytes
- Lock owner block – 32 bytes
- Lock waiter block – 32 bytes

## Locking in Oracle 8i

- Enqueuer resource structure (fixed array – default 4 entries per transaction)
- Enqueued locks array
- Interested transaction list (fixed array – INITRANS - MAXTRANS)
- Data page
- Process
- Enqueue wait (time out ~ 3 sec)
- Deadlock detection
Lock Tuning

- Use special facilities for long reads
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
  - Select appropriate granularity of locking
- Avoid DDL statements
- Bottlenecks
  - Using system features to circumvent bottlenecks
- Transaction Chopping
  - Rewriting applications to obtain best locking performance

Isolation Levels

- **Read Uncommitted** *(No lost update)*
  - Exclusive locks for write operations are held for the duration of the transactions
  - Lock for writes until commit time. No locks for reads
  - Reads may read dirty data and will not be repeatable.
- **Read Committed** *(No inconsistent retrieval)*
  - Lock for writes until commit time.
  - Shared locks are released as soon as the read operation terminates.
  - Reads may access only committed data, still not repeatable.
- **Repeatable Read** *(no unrepeatable reads)*
  - Strict two phase locking: lock for writes and reads until commit time.
- **Serializable** *(no phantoms)*
  - Table locking or index locking to avoid phantoms
Sacrificing Isolation for Performance

A transaction that holds locks during a screen interaction is an invitation to bottlenecks

- Airline Reservation
  1. Retrieve list of seats available
  2. Talk with customer regarding availability
  3. Secure seat
    - Single transaction is intolerable, because each customer would hold lock on seats available.
    - Keep user interaction outside a transactional context

Correctness is sacrificed: ask for a seat but then find it’s unavailable. Tolerable in this application.

Value of Serializability -- Data

Settings:

```sql
accounts (number, branchnum, balance);
create clustered index c on accounts(number);
```

- 100000 rows
- Cold buffer; same buffer size on all systems.
- Row level locking
- Isolation level (SERIALIZABLE or READ COMMITTED)
- SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
- Dual Xeon (550MHz,512Kb), 1Gb RAM, Internal RAID controller from Adaptec (80Mb), 4x18Gb drives (10000RPM), Windows 2000.
Value of Serializability -- transactions

Concurrent Transactions:

- **T1**: summation query [1 thread]
  
  ```sql
  select sum(balance) from accounts;
  ```

- **T2**: swap balance between two account numbers (in order of scan to avoid deadlocks) [N threads]
  
  ```sql
  valX:=select balance from accounts where number=X;
  valY:=select balance from accounts where number=Y;
  update accounts set balance=valX where number=Y;
  update accounts set balance=valY where number=X;
  ```

Value of Serializability -- results

- With SQL Server and DB2 the scan returns incorrect answers if the read committed isolation level is used (default setting).
- With Oracle correct answers are returned (snapshot isolation), but beware of swapping.
Cost of Serializability

Because the update conflicts with the scan, correct answers are obtained at the cost of decreased concurrency and thus decreased throughput.

Recommendation

- Begin with the highest degree of isolation (serializable)
  - If a transaction either suffers extensive deadlock or causes significant blocking, consider weakening the degree of isolation
  - Be aware that the answers may not be correct.
Lock Tuning

- Use special facilities for long reads
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
- Select appropriate granularity of locking
- Avoid DDL statements
- Bottlenecks
  - Using system features to circumvent bottlenecks
- Transaction Chopping
  - Rewriting applications to obtain best locking performance

Select appropriate granularity of locking

- RDBMSs support different granularities of locking: row or record-level (default), page-level, table-level
- Record-level locking is the best for online transaction environments where each transaction accesses only a few records spread on different pages
Locking Overhead -- data

Settings:

accounts(number, branchnum, balance);
create clustered index c on accounts(number);

- 100000 rows
- Cold buffer
- SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
- No lock escalation on Oracle; Parameter set so that there is no lock escalation on DB2; no control on SQL Server.
- Dual Xeon (550MHz,512Kb), 1Gb RAM, Internal RAID controller from Adaptec (80Mb), 4x18Gb drives (10000RPM), Windows 2000.

Locking Overhead -- transactions

Nb Concurrent Transactions:

- Update [10 000 updates]
  update accounts set balance = Val;
- Insert [10 000 inserts], e.g. typical one:
  insert into accounts values(664366,72255,2296.12);
Locking Overhead

Row locking is barely more expensive than table locking because recovery overhead is higher than locking overhead.
- Exception is updates on DB2 where table locking is distinctly less expensive than row locking.

Why choosing table-level locking?
- Used to avoid blocking long transactions
- Can be used to avoid deadlocks
- Reduce locking overhead in the case that there is no concurrency

Recommendation:
Long transactions (accesses almost all pages of the table) should use table locks mostly to avoid deadlocks
Short transactions should use record locks to enhance concurrency
How the user can control the granule size?

- **Explicit control of the granularity**
  - Within a transaction (Oracle, DB2): statement within a transaction explicitly requests a table-level lock
  - Across transactions (SQL Server): command defines the lock granularity for a table or group of tables. All transactions accessing these tables use the same lock granularity

- **Setting the escalation point** (SQL Server or DB2): systems acquire the default (finest) granularity lock until the nb of acquired locks exceeds some threshold set by the DBA. At that point, the next coarser granularity lock will be acquired.
  - The rule of thumb is to set the threshold high enough so that escalation never takes place in an online environment of relatively short transactions

- **Size of the lock table**: If the administrator selects a small lock table size, the system will be forced to escalate the lock granularity even if all transactions are short.

---

Avoid DDL statements

- **Data Definition Language**: language used to access and manipulate information about table names, column widths, ... that is metadata stored in the system catalog
- Metadata must be accessed by every transaction that:
  - Performs a compilation
  - Adds or removes a table or index
  - Changes an attribute description
- Can easily become a bottleneck

**Recommendation**: avoid updates to the system catalog during heavy system activity
Lock Tuning

- Use special facilities for long reads
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
- Select appropriate granularity of locking
- Avoid DDL statements
- **Circumventing bottlenecks (due to hot spots)**
- Transaction Chopping
  - Rewriting applications to obtain best locking performance

Circumventing bottlenecks

- **Hot spot**: piece of data that is accessed by many transactions and is updated by some.
  - Causes bottlenecks because each updating transaction must complete before any other transaction can obtain a lock on the hot data item
- **Techniques to circumvent:**
  - Use partitioning to eliminate it
  - Access the hot spot as late as possible in the transaction
  - Use special database management facilities
Example of hot spot: Sequential key generation

- Consider an application in which one needs a sequential number to act as a key in a table, e.g. invoice numbers for bills.
  - Transactions that insert data associate a unique identifier with each new data item
  - Concurrent transactions must coordinate to avoid giving the same identifier to different data items
- Ad hoc approach: a separate table holding the last invoice number. Fetch and update that number on each insert transaction.
  - Problem: This table becomes a bottleneck, because a transaction will release its lock on the counter only when the transaction commits
- Counter approach: use facility such as Sequence (Oracle)/Identity(SQL Server), that enables transactions to hold a latch on the counter.

Latches and Locks

- **Locks** are used for concurrency control
  - Requests for locks are queued
    - Priority queue
  - Lock data structure
    - Locking mode, lock granularity, transaction id.
    - Lock table
- **Latches** are used for mutual exclusion
  - Requests for latch succeeds or fails
  - Released immediately after access rather than being held till the end of the transaction
  - Does not allow shared mode
  - Does not provide support for queuing waiting threads
  - Single location in memory
    - Test and set for latch manipulation
Counter Facility -- data

Settings:

- accounts (number, branchnum, balance);
  create clustered index c on accounts(number);

- counter (nextkey);
  insert into counter values (1);

- default isolation level: READ COMMITTED; Empty tables
- Dual Xeon (550MHz,512Kb), 1Gb RAM, Internal RAID controller from Adaptec (80Mb), 4x18Gb drives (10000RPM), Windows 2000.

Counter Facility -- transactions

Concurrent Transactions:

- System [100 000 inserts, N threads]
  - SQL Server 7 (uses Identity column)
    insert into accounts values (94496,2789);
  - Oracle 8i
    insert into accounts values (seq.nextval,94496,2789);

- Ad-hoc [100 000 inserts, N threads]
  begin transaction
  NextKey:=select nextkey from counter;
  update counter set nextkey = NextKey+1;
  commit transaction
  begin transaction
  insert into accounts values(NextKey,?,?,?)
  commit transaction
Avoid Bottlenecks: Counters

- System generated counter (system) much better than a counter managed as an attribute value within a table (ad hoc).
- The Oracle counter can become a bottleneck if every update is logged to disk, but caching many counter numbers is possible.
- Counters may miss ids.

Lock Tuning

- Use special facilities for long reads
- Eliminate unnecessary locking
- Weaken isolation levels when application allows
  - Relaxing correctness to improve performance
- Select appropriate granularity of locking
- Avoid DDL statements
- Bottlenecks
  - Using system features to circumvent bottlenecks
- Transaction Chopping
  - Rewriting applications to obtain best locking performance
Transaction chopping

- How long should a transaction be?

Transaction length influence performance:

- The more locks a transaction requests, the more likely it will have to wait for some other transaction to release a lock.
- The longer a transaction T executes, the more time another transaction will have to wait if it is blocked by T.

In situations in which blocking occurs, short transactions are better than longer ones

- Sometimes, we can “chop” transactions in shorter ones without losing isolation guarantees.

Example: Simple Purchases

- Purchase item I for price P
  1. If cash < P then roll back transaction (constraint)
  2. Inventory(I) := inventory(I)+P
  3. Cash := Cash – P

- Two purchase transaction P1 and P2
  - P1 has item I for price 50
  - P2 has item I for price 75
  - Cash is 100
Example: Simple Purchases

- If 1-2-3 as one transaction then one of P1, P2 rolls back.
- If 1, 2, 3 as three distinct transactions:
  - P1 checks that cash > 50. It is.
  - P2 checks that cash > 75. It is.
  - P1 completes. Cash = 50.
  - P2 completes. Cash = -25.

Example: Simple Purchases

- Orthodox solution
  - Make whole program a single transaction
    - Cash becomes a bottleneck!
- Chopping solution
  - Find a way to rearrange and then chop up the transactions without violating serializable isolation level.
Example: Simple Purchases

- Chopping solution:
  1. If $\text{Cash} < P$ then roll back.
     $\text{Cash} := \text{Cash} – P$.
  2. $\text{Inventory}(I) := \text{inventory}(I) + P$

- Chopping execution:
  - $P_{11}$: $100 > 50$. Cash := 50.
  - $P_{21}$: $75 > 50$. Rollback.
  - $P_{12}$: inventory := inventory + 50.

Transaction Chopping

- Execution rules:
  - When pieces execute, they follow the partial order defined by the transactions.
  - If a piece is aborted because of a conflict, it will be resubmitted until it commits.
  - If a piece is aborted because of an abort, no other pieces for that transaction will execute.
When is transaction chopping possible?

- Two important questions must be answered:
  - Will the transactions that are concurrent with T cause T to produce an inconsistent state or to observe an inconsistent value if T is broken up?
  - Will the transactions that are concurrent with T be made inconsistent if T is broken up?
- **Rule of thumb:** Suppose T accesses data X and Y, but any other transaction T’ accesses at most one of X or Y and nothing else. Then, T can be divided into two transactions, one of which accesses X and the other accesses Y.
- **Caution:** Adding a new transaction to a set of existing transactions may invalidate all previously established choppings.