Lock Your Database Applications

Chicago, May 17th 2007

php|tek

Lukas Kahwe Smith (lsmith@optaros.com)

http://pooteeweet.org/files/phptek07/lock_your_db_app.pdf

optaros

About Myself

- Lukas Kahwe Smith
- Started using PHP in the late PHP3 days
- Joined PEAR sometime in 2002 to work on MDB
- Began posting on internals@lists.php.net in 2003
- Started the unofficial official PHP todo list in 2005
- Makes his living as a consultant for Optaros
  - OSS consulting and system integration firm
  - We are always hiring
    - http://optaros.com/en/company/careers
- Can be reached
  - Via Email: lsmith@optaros.com
  - On IRC: “lsmith” on efnet, freenode, ircnet ..
  - In Person: Here or at Ultimate Frisbee tournaments

optaros

Terminology and Considerations

- Data change
  - INSERT, UPDATE, DELETE
- Transaction
  - Series of SQL statements processed as an atomic unit
- Ready-only transaction
  - Transaction that contains no data changes
- Concurrent transactions
  - Two or more transactions who's start or end times overlap
- Lock
  - Method to prevent concurrent transactions interference
- Throughput vs. Response time
  - How many requests per second vs. how long one request takes

optaros

Painful Reality of Trade Offs

Locking strategies are a **trade off** between optimizing data integrity, performance and usability

optaros

Painful Reality of Trade Offs

- Disallow any data changes
  - Very fast reads (no locking needed)
  - Not very useful (few systems need no data changes)
- Any data access locks given data until the transaction end
  - Great for data integrity
  - Very bad for concurrency
- All locks are stored as finely grained as possible
  - Nice for concurrency (*)
  - Very bad for performance (throughput)
    - (*) Low throughput also increases chances of waits

optaros

LOCK modes

- Shared LOCK
  - May coexist with any number of shared LOCKs and one update LOCK
  - Used when reading data
- Update LOCK
  - May coexist with any number of shared LOCKs, but not with any other update or exclusive LOCKs
  - Used for reading when updating is planned
  - Do not allow writing, contrary to what the name implies
  - Not supported by all RDBMS
- Exclusive LOCK
  - May not coexist with any other LOCKs
  - Used when writing data
  - Exclusive LOCKs are often upgrades from shared or update LOCKs

optaros

LOCK granularity

- Granularity types
  - Database LOCK
  - Extent LOCK
  - Table LOCK
  - Page LOCK
  - Row LOCK
  - Column LOCK
- Most granular LOCKs by database
  - MyISAM does table LOCKs
  - InnoDB does row LOCKs
  - BerkleyDB does page LOCKs
  - SQLite does database LOCKs
  - SQL Server does column LOCKs
  - Most others do row LOCKs

optaros

More on LOCKs

- A small grain LOCK implies a shared big grain LOCK
  - Page LOCK implies a shared table LOCK
    - Therefore no exclusive table LOCK is possible
  - This is called an “Intent LOCK”
- “Escalation” is used to handle high overhead when too many LOCKs of a certain small grain level are created
  - Escalate the LOCK to the next bigger granularity level
- Explicit “manual” locking is supported by most DBMS
  - LOCK [SHARED | EXCLUSIVE] TABLE[S] <TABLE NAME>

optaros

Transactions

- Group multiple statements into one atomic unit
- Isolation levels enable deciding what concurrency data integrity problems are tolerable to get better concurrency
  - READ UNCOMMITED
    - No locks are issues or read, very high concurrency
  - READ COMMITTED
    - Shared locks may be released before transaction end
    - Fairly high concurrency
  - REPEATABLE READS
    - Shared locks not released before transaction end
    - Much reduced concurrency
  - SERIALIZABLE
    - Execute transactions in a manner that produces the same results as executed in serial
    - Severely reduced concurrency

optaros

Lost Update

- Transaction #1
  1) ...
  2) READ Row #1
  3) Data change Row #1
  4) ...
  5) COMMIT
  6) ...

- Transaction #2
  1) READ Row #1
  2) ...
  3) ...
  4) Data change Row #1
  5) ...
  6) COMMIT

- When Transaction #1 commits, the changes in Transaction #2 are lost
- May be avoided with READ COMMITTED
  - Useful when you can live with some errors, like a search engine

optaros

Dirty Read

- Transaction #1
  1) READ Row #1
  2) Data change Row #1
  3) ...
  4) ROLLBACK
- Transaction #2
  1) ...
  2) ...
  3) Read Row #1
  4) ...
  5) COMMIT

- Transaction #2 reads a row that has not yet been committed
- May be avoided with READ COMMITTED
  - Often you will not read the same rows again in transaction
  - If you only have a single statement in your transaction

optaros

Non-repeatable Read

- Transaction #1
  1) ...
  2) READ Row #1
  3) ...
  4) ...
  5) READ Row #1
  6) COMMIT

- Transaction #2
  1) READ Row #1
  2) ...
  3) Data change Row #1
  4) COMMIT
  5) ...
  6) ...

- When Transaction #1 reads the modified row in the second read after Transaction #2 has committed
- May be avoided with REPEATABLE READ
  - Should be used to transfer money between accounts

optaros

Phantoms

- Transaction #1
  1) READ Rows id IN (1-30)
  2) ...
  3) ...
  4) READ Rows id IN (1-30)
  5) COMMIT

- Transaction #2
  1) ...
  2) INSERT Row id = 27
  3) COMMIT
  4) ...
  5) ...

- When Transaction #1 reads the inserted row in the second read after Transaction #2 has committed
- May be avoided with SERIALIZABLE
  - Useful when selecting details and aggregates in separate queries inside a banking application
  - Can often be emulated using LOCK TABLE <Table Name>

optaros

More on Isolation Levels

- Unsupported isolation levels may be upgraded automatically
- Expect lower levels to be prioritized over higher levels
- Transaction without data changes may be labeled READ ONLY
  - SQL spec allows ignoring this hint
- Transactions with data changes may be labeled FOR UPDATE
  - Label also works with SELECTs inside a transaction
  - Cause update LOCKs, which can avoid deadlocks
- Most DBMS default to READ COMMITED
- Most DBMS at least also support REPEATABLE READ
- DBMS will also lock INDEX pages which tend to be
  - More granular because index keys are usually smaller than full rows
  - As a result they are also more “shiftier” since data changes cause page shifting and splitting more often

optaros

SAVEPOINTS and Isolation Levels

- Give the ability to partially rollback a transaction
  - Set any number of uniquely identified SAVEPOINTs through out a transaction
  - Optionally rollback to a specific SAVEPOINT
- Can be used to read rows inside a REPEATABLE READ transaction, but release lock before the end of the transaction
  - SELECT * FROM Table1;
  - SAVEPOINT savepoint1;
  - SELECT * FROM Table2;
  - /* Release shared lock on Table2 */
  - ROLLBACK TO SAVEPOINT savepoint1;
  - ...
  - COMMIT;

optaros

Fundamental Looming Thread

Using LOCKs immediately opens the **risk of DEADLOCK**s:

Condition when two or more transactions are **waiting on each other** to release LOCKs

optaros

Anti-Deadlock and Concurrency Tipps

- If update LOCKs are not supported
  - Emulate them with dummy UPDATE queries
    - UPDATE table1 SET column1 = column1 WHERE ..
  - Escalate manually with LOCK TABLE statement
- Always process all tables in a fixed order
  - At least follow a specific order (by alphabet) if the order does not matter to reduce the risk
- Partition data that unlike operations are separated
  - Optionally replicate data for reporting applications, so that reporting queries do not cause locks on other queries
- Keep transactions as short as possible
  - Split transactions whenever possible
  - Move any heavy processing out side of the transaction
  - Carry as much information over to the next transaction

optaros

Hot Spots

- Concurrent access to same objects that case exclusive locks
  - All INSERTs usually get written to the last page and therefore cause exclusive LOCKs on that page
    - Row level locking can help here
    - CLUSTERED KEYS on unordered PRIMARY KEYS will cause INSERTs to hit different pages
      - Counter page shifting with PCTREEE/FILLFACTOR
- Split INDEX pages usually stored in the same index file
  - Sacrifice space by writing "at the end of extent" instead which may be optionally configured on some DBMS
- Rows accessed by SEQUENCES and AUTO INCREMENTED or other monotonic values will lead to reading same index page
  - Store monotonic value in reverse (123 => 321)
  - Use Global Unique Identifier (GUID)
  - Add the sequence number to some nearly unique non number value derived from the row (like a name)

optaros

MVCC – Multi Version Concurrency Control

- Never lock readers by keeping a copy of all uncommitted changes in open transactions
- "Overwriting" or "In-Line" MVCC
  - Keep a rollback log of all versions in memory (*)
  - On commit clean up the rollback log and update
  - Oracle, Interbase, InnoDB etc.
  - Can run into issues when rollback log grows too big or lots of expanding data changes cause page shifting
- "Appending" or "Out-of-Line" MVCC
  - Inserts all updates as new versioned tuples
  - One of the reasons why PostgreSQL needs VACUUM
  - Inefficient for heavy update loads, especially when the tuples sizes are more or less unchanged
- "Hybrid" MVCC
  - Only overwrite for non expanding changes, else append

optaros

MVCC Example

- Transaction #1
  1) BEGIN TRANSACTION
  2)
  3) 23F on LH11 is available
  4) Reserve 23F on LH111
  5)
  6) COMMIT

- Transaction #2
  1)
  2) BEGIN TRANSACTION
  3)
  4)
  5) 23F on LH11 is available
  6)
  7) Reserve 23F on LH111
  8) COMMIT => 23F on LH111 is reserved twice

- Without MVCC Transaction #2 would have to WAIT for Transaction #1 to COMMIT in 5)
- Good implementation ROLLBACK in 7) to prevent 8)
- Great if concurrency of writers on one tuple is low
- Prevents using an index for COUNT(*)

optaros

MVCC Example Test

```
// force WAIT by appending "FOR UPDATE"
$select = "SELECT 1 FROM flights WHERE flight = 'LH11' AND
   seat = '23F' AND customer IS NULL";

$update = "UPDATE flights SET customer = 'Smith' WHERE flight
   = 'LH11' AND seat = '23F' AND customer IS NULL";

$db1->beginTransaction(); $db2->beginTransaction();

$result = $db1->queryRow($select, 'boolean'); // true
$result = (bool)$db1->exec($update); // true

$result = $db2->queryRow($select, 'boolean'); // true
$db1->commit();
$result = (bool)$db2->exec($update); // false
```

optaros

Optimistic Locking

- Locking at COMMIT time, rather than during the transaction
  - Assume that no other transaction modifies the tuple between the independent read and the write transaction
  - Fail instead of wait on concurrent transactions
- Add a unique "counter" to the unique row identifier
  - PK + Timestamp/Sequence
    - INSERT INTO addr (id, cntr, street, city) VALUES (nextval('addr'), unix_t(), 'Foo Street', 'Bar Town');
      - unix_t() = 1179145598
    - SET TRANS ISOLATION LEVEL *READ UNCOMMITTED*; SELECT id, cntr, street, city FROM addr; *COMMIT*;
    - UPDATE addr SET street = 'Foo Ave', cntr = unix_t() WHERE id = :id AND cntr = 1179145598;
      - Affected rows is 1
    - UPDATE addr SET street = 'Foo Ave', cntr = unix_t() WHERE id = :id AND cntr = 1179145598;
      - Affected rows is 0 => Failure!

optaros

References

- SQL Performance Tuning
  - Book by Peter Gulutzan and Trudy Pelzer
- Issues with MVCC
  - http://www.mysqlperformanceblog.com/2007/02/25/pitfalls-of-converting-to-innodb/
- InnoDB WAIT timeout
  - http://www.ibphoenix.com/main.nfs?page=ibp_mvcc_roman
- FOR UPDATE
  - http://dev.mysql.com/doc/refman/5.0/en/innodb-locking-reads.html
- Isolation Level
  - http://en.wikipedia.org/wiki/Isolation_%28computer_science%29
- These slides
  - http://pooteeweet.org/files/phptek07/lock_your_db_app.pdf