Locking and Race Conditions in Web Applications

What is a Race Condition?

When the completion of one operation depends on the completion of another, which in turn depends on the first.

The Olive Garden Example:

The waiter brings one breadstick for each person and then one additional.

Each person eats one bread stick, leaving one breadstick in the basket. No one takes the last breadstick in fear of being labeled a scrooge.

Recipes for Race Conditions

1. Evil Forms
2. Multiple reads and writes as one “Unit of Work”
3. Operations over multiple systems
4. Relational models on distributed systems
5. Ajax and round robin load balancing web applications
6. High traffic or load

Other Problems:

- Hard to test for
- Can be difficult to replicate
- Error logs tend to be misleading
- Often run undetected

Evil Forms

1. Prevent double-clicking submit
2. Tokenize forms (passive/active)
3. Tokenize with Ajax frequency polling
4. Tokenize as a user action item
5. Auto-cancel non-update submits
6. In place editing
7. Detect changes (before/after values)

Multiple Operations in one “Unit of Work”

```php
<?php
function fetchOrCreate($id)
{
    if ($obj = $this->fetch($id)) {
        return $obj;
    } else {
        return $this->create($id);
    }
}
```

In a multi-threaded environment such as a web server, this is a candidate for a race condition:

1. Thread #1 fetch() = *empty*
2. Thread #2 fetch() = *empty*
3. Thread #1 create()
4. Thread #2 create() // Exception: duplicate key

The Solution

1) Use a database transaction.
2) Use a global lock or semaphore.
3) Use the UNIQUE / PRIMARY keys in your application logic:

```php
<?php
function fetchOrCreate($id)
{
    if ($obj = $this->fetch($id)) {
        return $obj;
    } else {
        try {
            return $this->create($id);
        } catch(UniqueKeyException $e) {
            return $this->fetch($id);
        }
    }
}
```

Best for frequent reads
(setting lookup table)

```php
<?php
function fetchOrCreate($id)
{
    try {
        $obj = $this->create($id);
    } catch(UniqueKeyException $e) {
        $obj = $this->fetch($id);
    }
    return $obj;
}
```

Best for frequent writes
(email table)

Operations over Multiple Systems

1. User record is created with login credentials **(LDAP, RDMS)**
2. Account record is created and linked to the user record **(RDMS)**
3. Billing record is created and linked to the account **(Payment Processor)**
4. Program settings and new user flags are set **(RDMS)**
5. Session and Caches are created **(Memcached)**
6. Signup analytics are logged **(Data Warehouse, ODS)**
7. Welcome email is sent **(Gearman, Email Processor)**

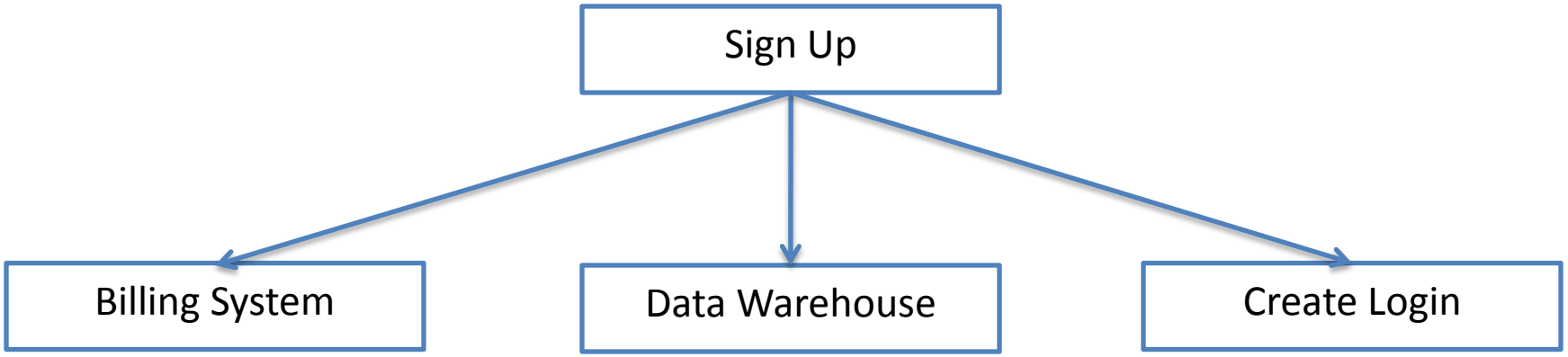
Partial Failures:

- Inconsistent data
- Orphaned rows
- Faulty reporting
- Unhappy customers

ACID Programming

Atomic, Consistent, Isolated and Durable.

- Small functions that perform an atomic data operation
- Predictable successes and failures
- Failures should leave the data source unaltered
- Can be nested hierarchically:



Example Code

```php
<?php
// @return bool
function signup($name, $password, $creditCard) {
    $inTransaction = DB::begin();

    $loginId      = $this->createLogin($name, $password);
    $billingId    = $this->createBilling($loginId, $creditCard);
    $hasAnalytics = $this->createAnalytics($loginId);
    $success      = $loginId && $billingId && $hasAnalytics;

    if (!$success || $inTransaction && !DB::commit()) {
        DB::rollback();
        return false;
    } else {
        return true;
    }
}

function createLogin($name, $password) {
    $inTransaction = DB::begin();
    $result = DB::query('INSERT INTO user (name, password) VALUES (?, ?)', array($name, $password));
    if (!$result || $inTransaction && !DB::commit()) {
        return false;
    } else {
        return DB::lastInsertId();
    }
}

// createBilling, createAnalytics with similar format ...

```

Multi-Version Concurrency Control (MVCC)

MVCC relies on snapshots to retain multiple versions of data in order to provide consistency. Available in most traditional databases (MySQL, MS-SQL, Oracle, PostgreSQL, Firebird, etc.) and elsewhere (svn, git, reiserfs).

Pros:

- Transactions
- Reads are never blocked
- Atomic operations
- Point-in-time data ensures consistency in models

Cons:

- Can be expensive retaining multiple copies of data
- Deadlocks
- Reduced performance

ANSI Isolation Levels

Degrees of consistency that deal with snapshot "phenomena" and lock collisions by balancing concurrency with throughput.

READ UNCOMMITTED

Allows "dirty reads" in which one transaction can access data from uncommitted changes in a second transaction.

READ COMMITTED

Committed changes in one transaction appear in another, which can cause two identical queries to return different results.

REPEATABLE READ (InnoDB default)

Within a transaction, two identical reads will always return the same result.

SERIALIZABLE

Transactions running reads lock updates from other transactions.

Note: In MySQL, some levels do not allow statement based replication.

Deadlocks – Another Race Condition

A deadlock is an (often unpredictable) situation where two or more competing operations endlessly wait for their counterpart to finish.

Common Causes:

- Transactions with too many operations
- Insufficient isolation level
- Lack of indexes
- Locking queries:

```
INSERT INTO table1
SELECT *
FROM table 2;
```

Note: SHOW INNODB STATUS is a great tool for debugging deadlocks.

Non-Database Locks

Locking in distributed systems is slow and is often not fully implemented in non-RDMS. Multi-system operations that require ACID need alternatives.

Alternative Locking Methods:

- Filesystem lock
- Database Semaphore
- Memcached
- MongoDB
- APC
- Redis

Filesystem Lock

```php
<?php
function signup($name, $password) {
    $fp = fopen("/var/spool/$name.lock", 'r+');
    if (!flock($fp, LOCK_EX)) {
        return false;
    }
    $id = $this->createSite($name, $password);
    fclose($fp);

    return $id;
}
```

flock() is used by PHP's native session handler and is more reliable than *fileexists()* because it's a single atomic operation.

Database Named Semaphore

```php
<?php
// CREATE TABLE SignupSemaphore (name VARCHAR(50), PRIMARY KEY(name));
function signup($name, $password) {
    if (!DB::query('INSERT INTO SignupSemaphore VALUES (?)', $name)) {
        return false;
    }

    $id = $this->createSite($name, $password);

    DB::query('DELETE FROM SignupSemaphore WHERE name = ?', $name);

    return $id;
}
```

A semaphore is different than a transaction because the lock is independent of any single data source or operation.

Memcached Named Lock

```php
<?php
function signup($name, $password) {
    $key = "signup-$name";
    if (!MyMemCacheUtility::set($key, true, time() + 3600)) {
        return false;
    }

    $id = $this->createSite($name, $password);

    MyMemCacheUtility::delete($key);

    return $id;
}
```

Memcached offers an additional advantage by natively providing a timeout.

JavaScript andAjax

Heavily interactive web applications often rely on large amounts of javascript and ajax which can wreak havoc on sessions as well as cause race conditions.

Suggestions:

- Lock ajax requests to the same web node
- Consider an ajax only web server
- Read-only sessions by default
- Variable isolation / separation
- Increment caches with routine flushing
- Use Comet as opposed to frequent timed checks

Counters and Trackers

Most web applications rely on counters and trackers for analytics and security. For example: tracking page views, login attempts, date of last actions, etc.

Suggestions:

- Horizontal partitioning
- Memcached/MongoDB increment()
- JavaScript to an analytics node (or GA)
- Scheduled access log processing (e.g.: ETL)
- MySQL blackhole engine with a replicated slave