

Using memcached in MySQL Deployments

Martin 'MC' Brown
Technical Writer, APS
Sun Microsystems
mc.brown@sun.com

sun.com/mysql

About memcached

- Memory cache
- Very fast (everything is in RAM!)
- Simple key/value interface
- Individual instances are autonomous
- No replication (not needed)
- Storage and distribution is client driven
- Linux/Unix only (unofficial Windows available)

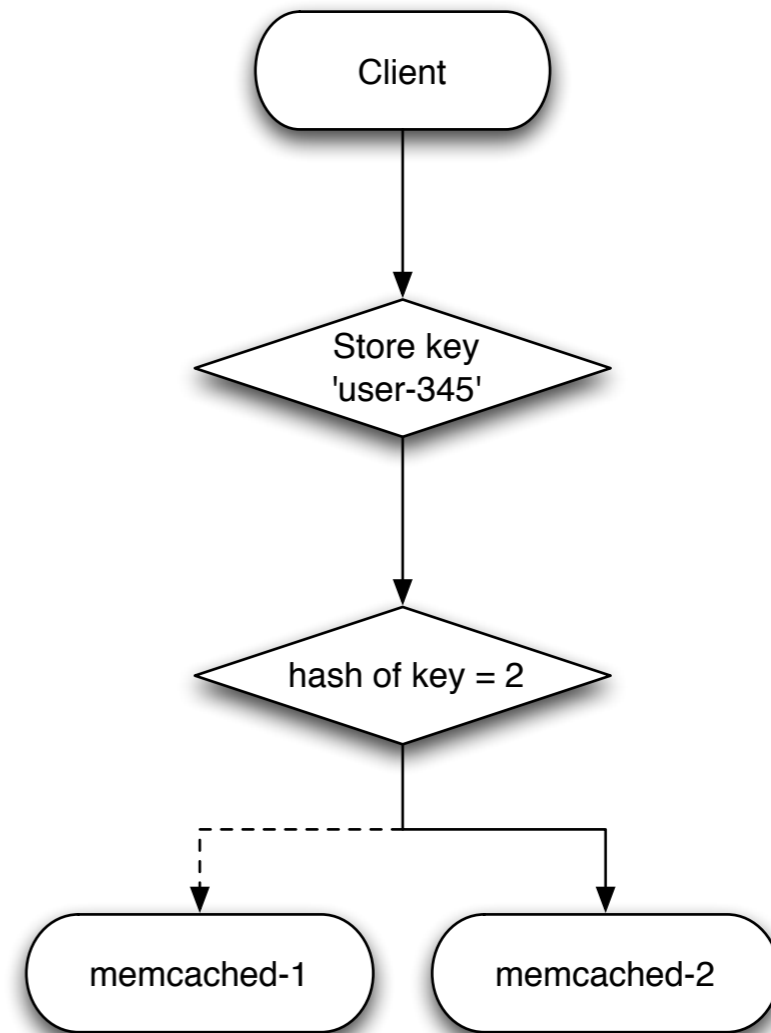
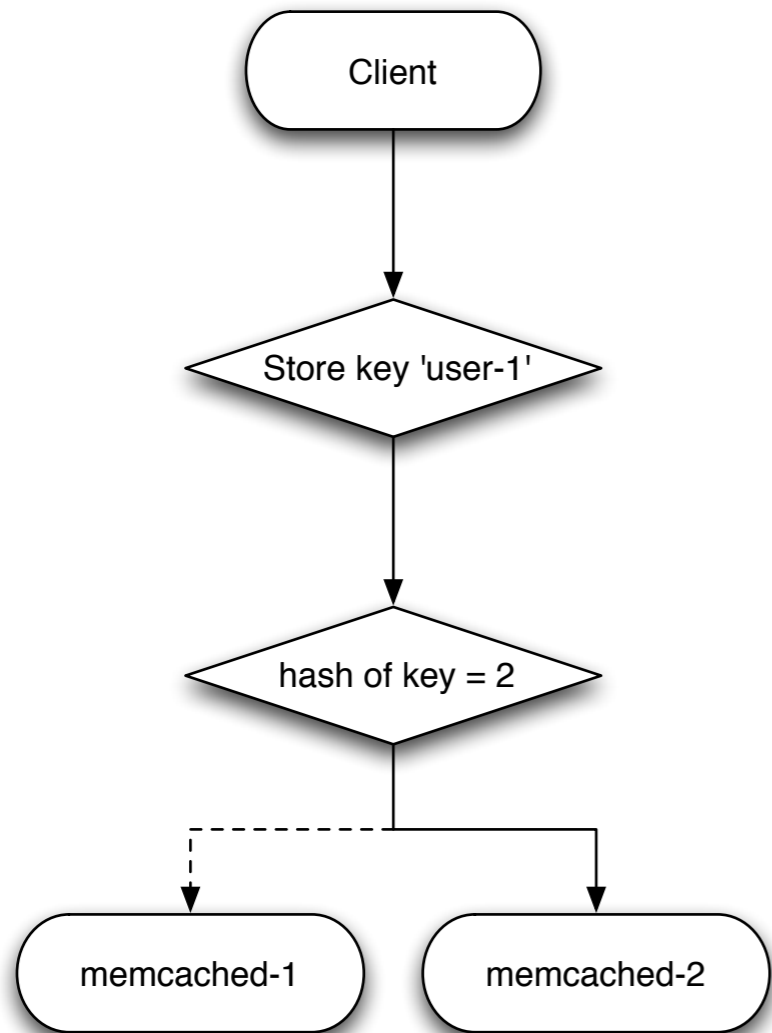
Memcached Interface

- One big hash
- Store a key/value pair
- Retrieve a value by its key
 - > Data is just bytes, can be anything
 - > For compatibility, might want to consider JSON
- Additional commands/options are available
 - > Often you don't need them

Memcached distributed nature

- Can run one memcached instance for your application
- Gain more cache, failover, redundancy if you run more
- For multiple instances, run it multiple times on same host or different host
- Data distribution is handled by the client
 - > Client has list of memcached servers
 - > Client stores by key
 - > Hash on key

Memcached Server Selection



Memcached Data Distribution

- memcached-1 and memcached-2 dont know about each other
- They dont *need* to know about each other
- The client is responsible for deciding where to store
- The client is responsible for deciding whether to store
- The client is responsible for failover
 - > Delete a server from the list
 - > Ignore the issue

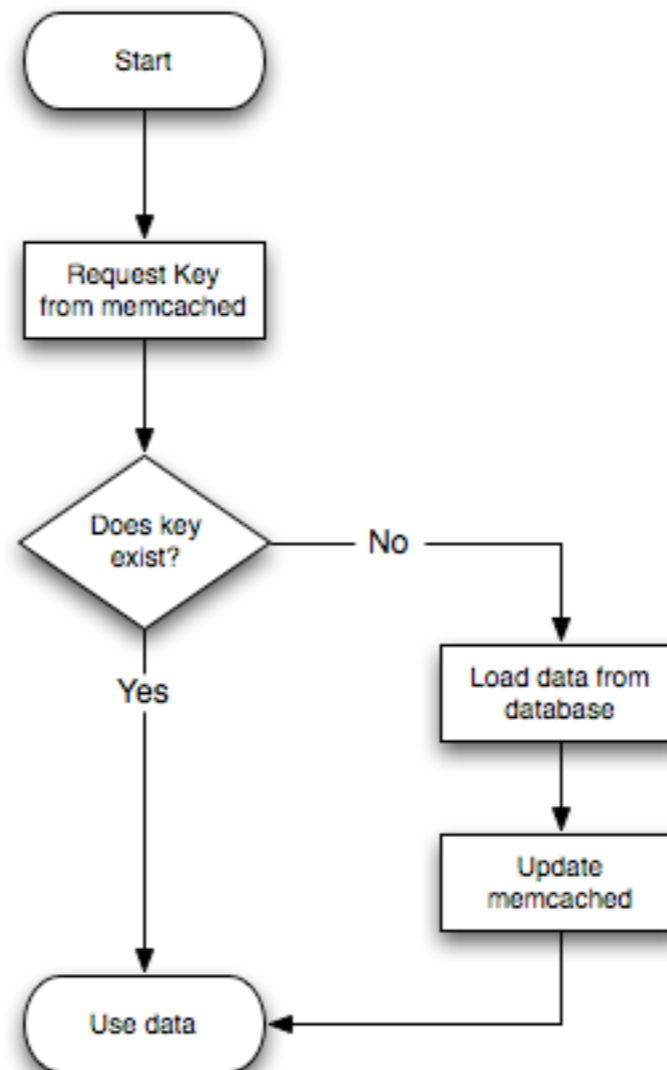
Cache Management

- Keys exist in cache until:
 - > Explicitly removed (delete)
 - > Removed through lack of use (Least Recently Used (LRU))
 - > Entry expires
- Specific Expiry
 - > Allows finer control over expiry
 - > Useful for sessions
 - > Specify an absolute time (epoch)
 - > Specify a relative time; object will expire within # seconds of store

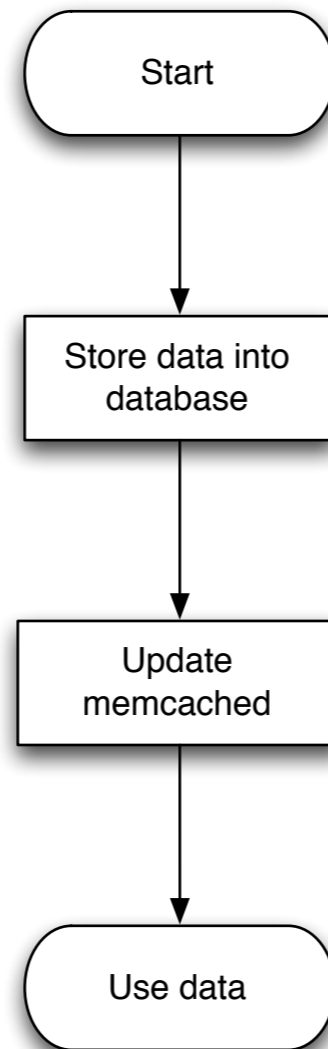
Starting memcached

- Run:
 - > memcached
- Start without parameters
 - > Defaults to 64MB
 - > Listens on port 11211
 - > Max 1024 simultaneous connections
 - > Four threads
- Everything configurable
 - > memcached -m 8192
 - > 8GB RAM
 - > Dont exceed available physical RAM (defeats the object)

Memcached Application Flow (Load)



Memcached Application Flow (Save)



Choosing What to Cache

- Cache things you need quick access to
 - > Session data
 - > Display Data
 - > Cache the entire page or object
 - > Cache individual items
 - Still often quicker to load 20 items from memcached than one item from DB
 - > Regular objects
 - > Shared objects
 - > Max object size is 1MB (configurable at build time)
 - > Namespaces

Hashing algorithms

- Client hashing determines which server is chosen
- Two Types
 - > Modula hashing
 - > Simple array of servers
 - > $\text{Hash_number} \% \text{number_of_servers} = \text{server number}$
 - > Chooses server by hashing against list
 - > Change server list, changes the hash, changes the server
 - > Consistent Hashing
 - > Array of servers
 - > But hash value relates to exact server
 - > Change server list, selection remains consistent

Memcached interfaces

- Variety of interfaces for different languages and environments
- You can use Telnet (but not recommended)
- APIs for Perl, Python, etc
- APIs for some environments (Ruby on Rails)
- libmemcached provides UDFs for MySQL
- Most handle serialization for you

Memcached interface consistency

- All APIs use the same base interface
 - > get() gets a value by the specified key
 - > get_multi() gets multiple values by the specified keys
 - > set() sets the key/value (new or existing)
 - > add() adds a key/value (fails if it already exists)
 - > replace() updates an existing key/value
 - > delete() deletes a value
 - > incr() increments
 - > decr() decrements
- All support an expiry
- Not all APIs are consistent, but usually similar

Client Compatibility

- Use JSON for data storage
 - > Simplified for handling structured data
 - > Compatible with lots of environments
- Use a common key hashing algorithm
 - > Different APIs may use different hashing mechanisms
 - > This may lead to the same data being stored on different caches
- Use consistent server lists

Memcached and Perl Example

- Cache::Memcached module is popular

```

use Cache::Memcached;
my $cache = new Cache::Memcached {
    'servers' => [
        '192.168.0.100:11211',
        '192.168.0.101:11211',
        '192.168.0.102:11211',
    ],
};
$cache->set($filmkey, $filmdata)
$cache->get($filmkey);

```

Memcached and MySQL

- You can use memcached servers directly within MySQL
- The libmemcached UDFs provide complete suite of memcached API as MySQL functions
- Build the UDFs
- Install the plugin
- Use the supplied script (install.pl) to load the functions from the plugin

MySQL UDFs Basic Usage

- Store a value
 - > `SELECT memc_set('myid', 'myvalue');`
- Get a value
 - > `SELECT memc_get('myid');`
- Can use it as part of a bigger expression
- Seed rows automatically into cache
- Could be seen as inefficient

Use Triggers on any table (1)

- Triggers allow us to update the cache regardless
- Set the table to be used:

```
CREATE TABLE session_data (  
    userid VARCHAR( 32 ) NOT NULL,  
    sessiondata VARCHAR( 255 ),  
    PRIMARY KEY ( userid )  
    ) ENGINE=MyISAM;
```

- Just a standard table

Use Triggers on any table (2)

- Create the INSERT trigger:

```
DELIMITER |
```

```
CREATE TRIGGER session_cache_insert
```

```
AFTER INSERT ON session_data
```

```
FOR EACH ROW BEGIN
```

```
    SET @memc_rc:= memc_set( NEW.userid, NEW.sessiondata );
```

```
END; |
```

```
DELIMITER ;
```

- Now do an INSERT:
 - > INSERT INTO session_data values ('mc','Martin MC Brown');
- It's in the cache!

Use Triggers on any table (3)

- Create the UPDATE trigger:

```
CREATE TRIGGER session_cache_update
  AFTER UPDATE ON session_data
  FOR EACH ROW BEGIN
    SET @memc_rc:= memc_set( NEW.userid, NEW.sessiondata );
  END;
```

- Now updates will update the cache
 - > UPDATE session_data SET sessiondata = 'MC' WHERE userid = 'mc';

Using Triggers Without Storage

- Instead of using a standard table, use BLACKHOLE storage engine
- Data is never written anywhere
- Trigger writes to memcache anyway
- Keeps cache in SQL logic

Triggers/BLACKHOLE Points

- Downsides:
 - > Caching simple row data can be inefficient
 - > Generally it's better to cache information that requires multiple statements to retrieve, rather than DB data that could be cached elsewhere
- Upsides:
 - > Removes the need to do explicit updates in your application
 - > Can be efficient for frequently used data (sessions, summary fragments)

Some things to ponder

- It's a cache
- Cache what you need; not everything
- Don't worry about 'filling it up'
- Don't worry about 'seeding it'
- Don't worry about replication

Some more things to ponder

- Don't panic about server failures
 - > But do consider the consequences
- Don't panic about cache misses
 - > You can load it from the DB (it's a cache!)
 - > But do investigate the reasons if they are excessive
- Don't cache things you don't need
 - > Images
 - > Files that can be accessed directly through Apache

Questions

Martin 'MC' Brown
Technical Writer, APS
Sun Microsystems
mc.brown@sun.com

sun.com/mysql