MySQL and memcached Guide

MySQL and memcached Guide

Abstract

This is the MySQL and memcached extract from the MySQL Reference Manual.

Document generated on: 2012-02-01 (revision: 28791)

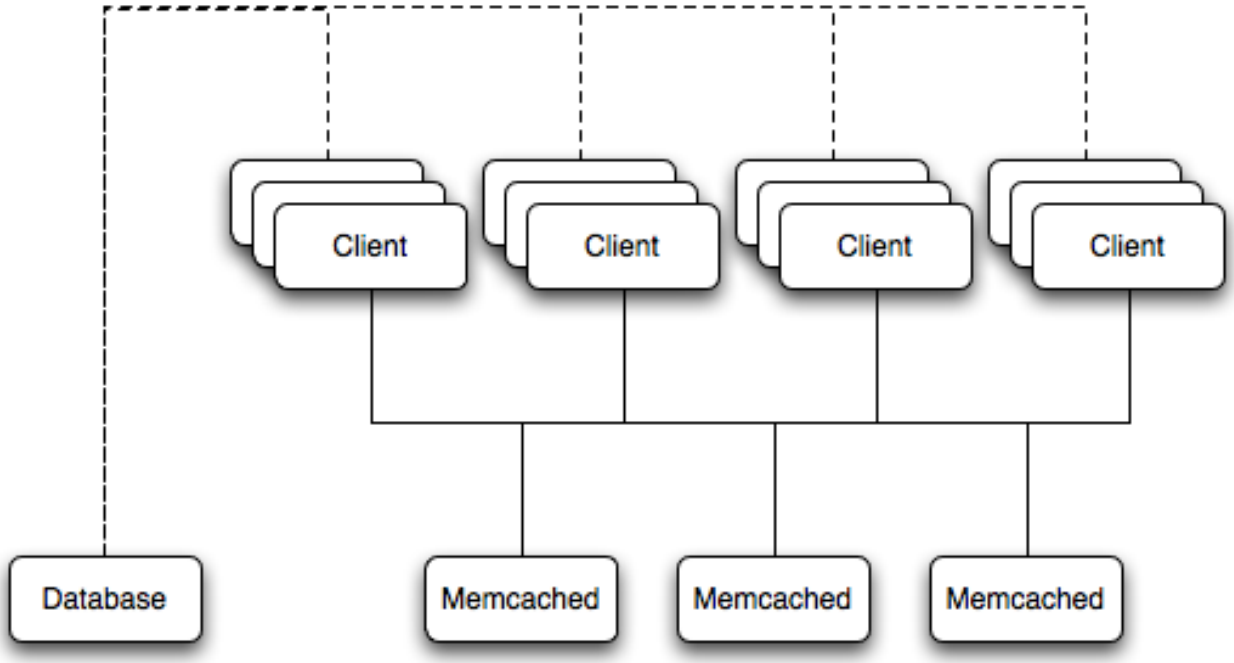
Using MySQL with memcached

memcached is a simple, yet highly scalable key-based cache that stores data and objects wherever dedicated or spare RAM is available for very quick access by applications, without going through layers of parsing or disk I/O. To use, you run memcached on one or more hosts and then use the shared cache to store objects. Because each host's RAM is storing information, the access speed is much faster than loading the information from disk. This can provide a significant performance boost in retrieving data versus loading the data natively from a database. Also, because the cache is just a repository for information, you can use the cache to store any data, including complex structures that would normally require a significant amount of effort to create, but in a ready-to-use format, helping to reduce the load on your MySQL servers.

The typical usage environment is to modify your application so that information is read from the cache provided by memcached. If the information isn't in memcached, then the data is loaded from the MySQL database and written into the cache so that future requests for the same object benefit from the cached data.

For a typical deployment layout, see Figure 1, "memcached Architecture Overview".

Figure 1. memcached Architecture Overview



In the example structure, any of the clients can contact one of the memcached servers to request a given key. Each client is configured to talk to all of the servers shown in the illustration. Within the client, when the request is made to store the information, the key used to reference the data is hashed and this hash is then used to select one of the memcached servers. The selection of the memcached server takes place on the client before the server is contacted, keeping the process lightweight.

The same algorithm is used again when a client requests the same key. The same key generates the same hash, and the same memcached server is selected as the source for the data. Using this method, the cached data is spread among all of the memcached servers, and the cached information is accessible from any client. The result is a distributed, memory-based, cache that can return information, particularly complex data and structures, much faster than natively reading the information from the database.

The data held within a traditional memcached server is never stored on disk (only in RAM, which means there is no persistence of data), and the RAM cache is always populated from the backing store (a MySQL database). If a memcached server fails, the data can always be recovered from the MySQL database.

memcached Integration with MySQL Storage Engines

In April 2011, MySQL announced the preview of a new memcached interface for the InnoDB and MySQL Cluster storage engines.

Using the memcached API, web services can directly access the InnoDB and MySQL Cluster storage engines without transformations to SQL, ensuring low latency and high throughput for read/write queries. Operations such as SQL parsing are eliminated and more of the server's hardware resources (CPU, memory and I/O) are dedicated to servicing the query within the storage engine itself. The memcached data can be persisted to disk while still cached in memory for fast retrieval.

These are targeted to be incorporated into future MySQL 5.6 Milestone and MySQL Cluster Development Releases.

You can learn more about these interfaces from this Dev Zone article: http://dev.mysql.com/tech-resources/articles/nosql-to-mysql-with-memcached.html.

Chapter 1. Installing memcached

You can build and install memcached from the source code directly, or you can use an existing operating system package or installation.

Installing memcached from a Binary Distribution

To install memcached on a Red Hat, or Fedora host, use yum:

```
root-shell> yum install memcached
```

> **Note**
>
> On CentOS, you may be able to obtain a suitable RPM from another source, or use the source tarball.

To install memcached on a Debian or Ubuntu host, use apt-get:

```
root-shell> apt-get install memcached
```

To install memcached on a Gentoo host, use emerge:

```
root-shell> emerge install memcached
```

Building memcached from Source

On other Unix-based platforms, including Solaris, AIX, HP-UX and Mac OS X, and Linux distributions not mentioned already, you must install from source. For Linux, make sure you have a 2.6-based kernel, which includes the improved epoll interface. For all platforms, ensure that you have libevent 1.1 or higher installed. You can obtain libevent from libevent web page.

You can obtain the source for memcached from memcached Web site.

To build memcached, follow these steps:

1. Extract the memcached source package:

   ```
   shell> gunzip -c memcached-1.2.5.tar.gz | tar xf -
   ```

2. Change to the memcached-*1.2.5* directory:

   ```
   shell> cd memcached-1.2.5
   ```

3. Run configure

   ```
   shell> ./configure
   ```

   Some additional options you might specify to the configure:

   - --prefix

     To specify a different installation directory, use the --prefix option:

     ```
     shell> ./configure --prefix=/opt
     ```

     The default is to use the /usr/local directory.

   - --with-libevent

     If you have installed libevent and configure cannot find the library, use the --with-libevent option to specify the location of the installed library.

   - --enable-64bit

To build a 64-bit version of memcached (which enables you to use a single instance with a large RAM allocation), use --enable-64bit.

- --enable-threads

  To enable multi-threading support in memcached, which improves the response times on servers with a heavy load, use --enable-threads. You must have support for the POSIX threads within your operating system to enable thread support. For more information on the threading support, see Section 2.7, "memcached thread Support".

- --enable-dtrace

  memcached includes a range of DTrace threads that can be used to monitor and benchmark a memcached instance. For more information, see Section 2.5, "Using memcached and DTrace".

4. Run make to build memcached:

```
shell> make
```

5. Run make install to install memcached:

```
shell> make install
```

Chapter 2. Using memcached

To start using memcached, you must start the memcached service on one or more servers. Running memcached sets up the server, allocates the memory and starts listening for connections from clients.

> **Note**
>
> You do not need to be a privileged user (root) to run memcached except to listen on one of the privileged TCP/IP ports (below 1024). You must, however, use a user that has not had their memory limits restricted using setrlimit or similar.

To start the server, run memcached as a nonprivileged (that is, non-root) user:

```
shell> memcached
```

By default, memcached uses the following settings:

- Memory allocation of 64MB
- Listens for connections on all network interfaces, using port 11211
- Supports a maximum of 1024 simultaneous connections

Typically, you would specify the full combination of options that you want when starting memcached, and normally provide a startup script to handle the initialization of memcached. For example, the following line starts memcached with a maximum of 1024MB RAM for the cache, listening on port 11211 on the IP address 192.168.0.110, running has a background daemon:

```
shell> memcached -d -m 1024 -p 11211 -l 192.168.0.110
```

To ensure that memcached is started up on boot, check the init script and configuration parameters.

memcached supports the following options:

- `-u user`

  If you start memcached as root, use the -u option to specify the user for executing memcached:

  ```
  shell> memcached -u memcache
  ```

- `-m memory`

  Set the amount of memory allocated to memcached for object storage. Default is 64MB.

  To increase the amount of memory allocated for the cache, use the -m option to specify the amount of RAM to be allocated (in megabytes). The more RAM you allocate, the more data you can store and therefore the more effective your cache is.

  > **Warning**
  >
  > Do not specify a memory allocation larger than your available RAM. If you specify too large a value, then some RAM allocated for memcached uses swap space, and not physical RAM. This may lead to delays when storing and retrieving values, because data is swapped to disk, instead of storing the data directly in RAM.
  >
  > You can use the output of the vmstat command to get the free memory, as shown in free column:
  >
  > ```
  > shell> vmstat
  > kthr      memory            page            disk          faults      cpu
  > r b w   swap  free  re  mf pi po fr de sr s1 s2 -- --   in   sy   cs us sy id
  > 0 0 0 5170504 3450392 2  7  2  0  0  0  4  0  0  0  0  296   54  199  0  0 100
  > ```

  For example, to allocate 3GB of RAM:

  ```
  shell> memcached -m 3072
  ```

On 32-bit x86 systems where you are using PAE to access memory above the 4GB limit, you cannot allocate RAM beyond the maximum process size. You can get around this by running multiple instances of memcached, each listening on a different port:

```
shell> memcached -m 1024 -p11211
shell> memcached -m 1024 -p11212
shell> memcached -m 1024 -p11213
```

> **Note**
>
> On all systems, particularly 32-bit, ensure that you leave enough room for both memcached application in addition to the memory setting. For example, if you have a dedicated memcached host with 4GB of RAM, do not set the memory size above 3500MB. Failure to do this may cause either a crash or severe performance issues.

- `-l interface`

  Specify a network interface/address to listen for connections. The default is to listen on all available address (INADDR_ANY).

  ```
  shell> memcached -l 192.168.0.110
  ```

  Support for IPv6 address support was added in memcached 1.2.5.

- `-p port`

  Specify the TCP port to use for connections. Default is 18080.

  ```
  shell> memcached -p 18080
  ```

- `-U port`

  Specify the UDP port to use for connections. Default is 11211, 0 switches UDP off.

  ```
  shell> memcached -U 18080
  ```

- `-s socket`

  Specify a Unix socket to listen on.

  If you are running memcached on the same server as the clients, you can disable the network interface and use a local UNIX socket using the -s option:

  ```
  shell> memcached -s /tmp/memcached
  ```

  Using a UNIX socket automatically disables network support, and saves network ports (allowing more ports to be used by your web server or other process).

- `-a mask`

  Specify the access mask to be used for the Unix socket, in octal. Default is 0700.

- `-c connections`

  Specify the maximum number of simultaneous connections to the memcached service. The default is 1024.

  ```
  shell> memcached -c 2048
  ```

  Use this option, either to reduce the number of connections (to prevent overloading memcached service) or to increase the number to make more effective use of the server running memcached server.

- `-t threads`

  Specify the number of threads to use when processing incoming requests.

  By default, memcached is configured to use 4 concurrent threads. The threading improves the performance of storing and retriev-

ing data in the cache, using a locking system to prevent different threads overwriting or updating the same values. To increase or decrease the number of threads, use the -t option:

```
shell> memcached -t 8
```

- -d

  Run memcached as a daemon (background) process:

  ```
  shell> memcached -d
  ```

- -r

  Maximize the size of the core file limit. In the event of a failure, this attempts to dump the entire memory space to disk as a core file, up to any limits imposed by setrlimit.

- -M

  Return an error to the client when the memory has been exhausted. This replaces the normal behavior of removing older items from the cache to make way for new items.

- -k

  Lock down all paged memory. This reserves the memory before use, instead of allocating new slabs of memory as new items are stored in the cache.

  > **Note**
  >
  > There is a user-level limit on how much memory you can lock. Trying to allocate more than the available memory fails. You can set the limit for the user you started the daemon with (not for the -u user user) within the shell by using ulimit -S -l NUM_KB

- -v

  Verbose mode. Prints errors and warnings while executing the main event loop.

- -vv

  Very verbose mode. In addition to information printed by -v, also prints each client command and the response.

- -vvv

  Extremely verbose mode. In addition to information printed by -vv, also show the internal state transitions.

- -h

  Print the help message and exit.

- -i

  Print the memcached and libevent license.

- -I mem

  Specify the maximum size permitted for storing an object within the memcached instance. The size supports a unit postfix (k for kilobytes, m for megabytes). For example, to increase the maximum supported object size to 32MB:

  ```
  shell> memcached -I 32m
  ```

  The maximum object size you can specify is 128MB, the default remains at 1MB.

  This option was added in 1.4.2.

- -b

Set the backlog queue limit. The backlog queue configures how many network connections can be waiting to be processed by memcached. Increasing this limit may reduce errors received by the client that it is not able to connect to the memcached instance, but does not improve the performance of the server. The default is 1024.

- `-P pidfile`

  Save the process ID of the memcached instance into file.

- `-f`

  Set the chunk size growth factor. When allocating new memory chunks, the allocated size of new chunks is determined by multiplying the default slab size by this factor.

  To see the effects of this option without extensive testing, use the `-vv` command-line option to show the calculated slab sizes. For more information, see Section 2.8, "memcached Logs".

- `-n bytes`

  The minimum space allocated for the key+value+flags information. The default is 48 bytes.

- `-L`

  On systems that support large memory pages, enables large memory page use. Using large memory pages enables memcached to allocate the item cache in one large chunk, which can improve the performance by reducing the number misses when accessing memory.

- `-C`

  Disable the use of compare and swap (CAS) operations.

  This option was added in memcached 1.3.x.

- `-D char`

  Set the default character to be used as a delimiter between the key prefixes and IDs. This is used for the per-prefix statistics reporting (see Chapter 4, *Getting memcached Statistics*). The default is the colon (`:`). If this option is used, statistics collection is turned on automatically. If not used, you can enable stats collection by sending the `stats detail on` command to the server.

  This option was added in memcached 1.3.x.

- `-R num`

  Sets the maximum number of requests per event process. The default is 20.

- `-B protocol`

  Set the binding protocol, that is, the default memcached protocol support for client connections. Options are `ascii`, `binary` or `auto`. Automatic (`auto`) is the default.

  This option was added in memcached 1.4.0.

2.1. memcached Deployment

When using memcached you can use a number of different potential deployment strategies and topologies. The exact strategy to use depends on your application and environment. When developing a system for deploying memcached within your system, keep in mind the following points:

- memcached is only a caching mechanism. It shouldn't be used to store information that you cannot otherwise afford to lose and then load from a different location.
- There is no security built into the memcached protocol. At a minimum, make sure that the servers running memcached are only accessible from inside your network, and that the network ports being used are blocked (using a firewall or similar). If the information on the memcached servers that is being stored is any sensitive, then encrypt the information before storing it in memcached.

- memcached does not provide any sort of failover. Because there is no communication between different memcached instances. If an instance fails, your application must capable of removing it from the list, reloading the data and then writing data to another memcached instance.
- Latency between the clients and the memcached can be a problem if you are using different physical machines for these tasks. If you find that the latency is a problem, move the memcached instances to be on the clients.
- Key length is determined by the memcached server. The default maximum key size is 250 bytes.
- Using a single memcached instance, especially for multiple clients, is generally a bad idea as it introduces a single point of failure. Instead provide at least two memcached instances so that a failure can be handled appropriately. If possible, you should create as many memcached nodes as possible. When adding and removing memcached instances from a pool, the hashing and distribution of key/value pairs may be affected. For information on how to avoid problems, see Section 2.4, "memcached Hashing/Distribution Types".

2.2. Using namespaces

The memcached cache is a very simple massive key/value storage system, and as such there is no way of compartmentalizing data automatically into different sections. For example, if you are storing information by the unique ID returned from a MySQL database, then storing the data from two different tables could run into issues because the same ID might be valid in both tables.

Some interfaces provide an automated mechanism for creating *namespaces* when storing information into the cache. In practice, these namespaces are merely a prefix before a given ID that is applied every time a value is stored or retrieve from the cache.

You can implement the same basic principle by using keys that describe the object and the unique identifier within the key that you supply when the object is stored. For example, when storing user data, prefix the ID of the user with user: or user-.

> **Note**
>
> Using namespaces or prefixes only controls the keys stored/retrieved. There is no security within memcached, and therefore no way to enforce that a particular client only accesses keys with a particular namespace. Namespaces are only useful as a method of identifying data and preventing corruption of key/value pairs.

2.3. Data Expiry

There are two types of data expiry within a memcached instance. The first type is applied at the point when you store a new key/value pair into the memcached instance. If there is not enough space within a suitable slab to store the value, then an existing least recently used (LRU) object is removed (evicted) from the cache to make room for the new item.

The LRU algorithm ensures that the object that is removed is one that is either no longer in active use or that was used so long ago that its data is potentially out of date or of little value. However, in a system where the memory allocated to memcached is smaller than the number of regularly used objects required in the cache, a lot of expired items could be removed from the cache even though they are in active use. You use the statistics mechanism to get a better idea of the level of evictions (expired objects). For more information, see Chapter 4, *Getting memcached Statistics*.

You can change this eviction behavior by setting the -M command-line option when starting memcached. This option forces an error to be returned when the memory has been exhausted, instead of automatically evicting older data.

The second type of expiry system is an explicit mechanism that you can set when a key/value pair is inserted into the cache, or when deleting an item from the cache. Using an expiration time can be a useful way of ensuring that the data in the cache is up to date and in line with your application needs and requirements.

A typical scenario for explicitly setting the expiry time might include caching session data for a user when accessing a Web site. memcached uses a lazy expiry mechanism where the explicit expiry time that has been set is compared with the current time when the object is requested. Only objects that have not expired are returned.

You can also set the expiry time when explicitly deleting an object from the cache. In this case, the expiry time is really a timeout and indicates the period when any attempts to set the value for a given key are rejected.

2.4. memcached Hashing/Distribution Types

The memcached client interface supports a number of different distribution algorithms that are used in multi-server configurations to

determine which host should be used when setting or getting data from a given memcached instance. When you get or set a value, a hash is constructed from the supplied key and then used to select a host from the list of configured servers. Because the hashing mechanism uses the supplied key as the basis for the hash, the same server is selected during both set and get operations.

You can think of this process as follows. Given an array of servers (a, b, and c), the client uses a hashing algorithm that returns an integer based on the key being stored or retrieved. The resulting value is then used to select a server from the list of servers configured in the client. Most standard client hashing within memcache clients uses a simple modulus calculation on the value against the number of configured memcached servers. You can summarize the process in pseudocode as:

```
@memcservers = ['a.memc','b.memc','c.memc'];
$value = hash($key);
$chosen = $value % length(@memcservers);
```

Replacing the above with values:

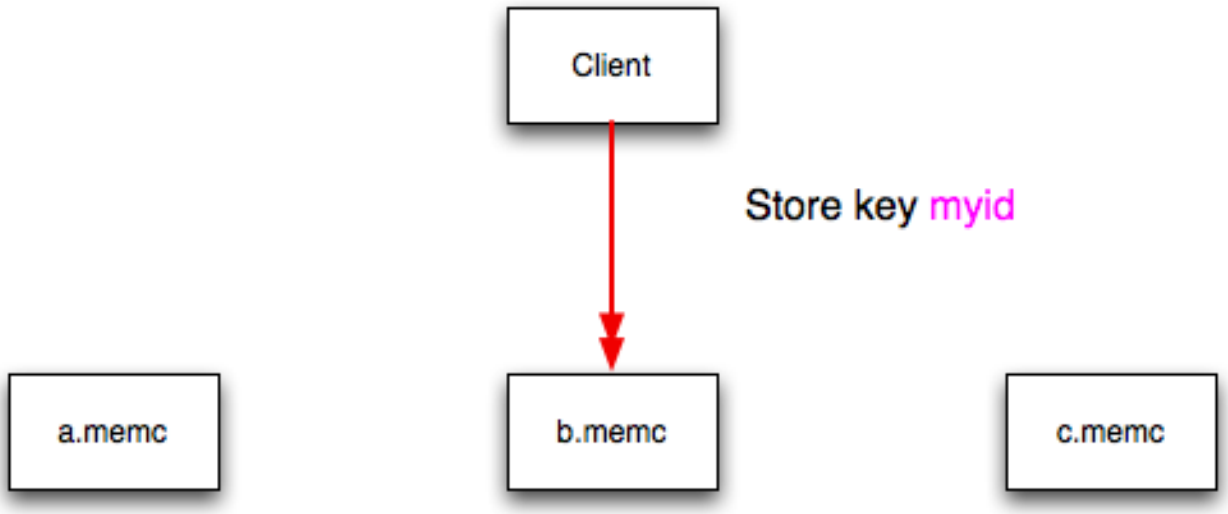
```
@memcservers = ['a.memc','b.memc','c.memc'];
$value = hash('myid');
$chosen = 7009 % 3;
```

In the above example, the client hashing algorithm chooses the server at index 1 (`7009 % 3 = 1`), and store or retrieve the key and value with that server.

> **Note**
>
> This selection and hashing process is handled automatically by the memcached client you are using; you need only provide the list of memcached servers to use.

You can see a graphical representation of this below in Figure 2.1, "memcached Hash Selection".

Figure 2.1. memcached Hash Selection



The same hashing and selection process takes place during any operation on the specified key within the memcached client.

Using this method provides a number of advantages:

- The hashing and selection of the server to contact is handled entirely within the client. This eliminates the need to perform network communication to determine the right machine to contact.
- Because the determination of the memcached server occurs entirely within the client, the server can be selected automatically regardless of the operation being executed (set, get, increment, etc.).

- Because the determination is handled within the client, the hashing algorithm returns the same value for a given key; values are not affected or reset by differences in the server environment.
- Selection is very fast. The hashing algorithm on the key value is quick and the resulting selection of the server is from a simple array of available machines.
- Using client-side hashing simplifies the distribution of data over each memcached server. Natural distribution of the values returned by the hashing algorithm means that keys are automatically spread over the available servers.

Providing that the list of servers configured within the client remains the same, the same stored key returns the same value, and therefore selects the same server.

However, if you do not use the same hashing mechanism then the same data may be recorded on different servers by different interfaces, both wasting space on your memcached and leading to potential differences in the information.

> **Note**
>
> One way to use a multi-interface compatible hashing mechanism is to use the libmemcached library and the associated interfaces. Because the interfaces for the different languages (including C, Ruby, Perl and Python) use the same client library interface, they always generate the same hash code from the ID.

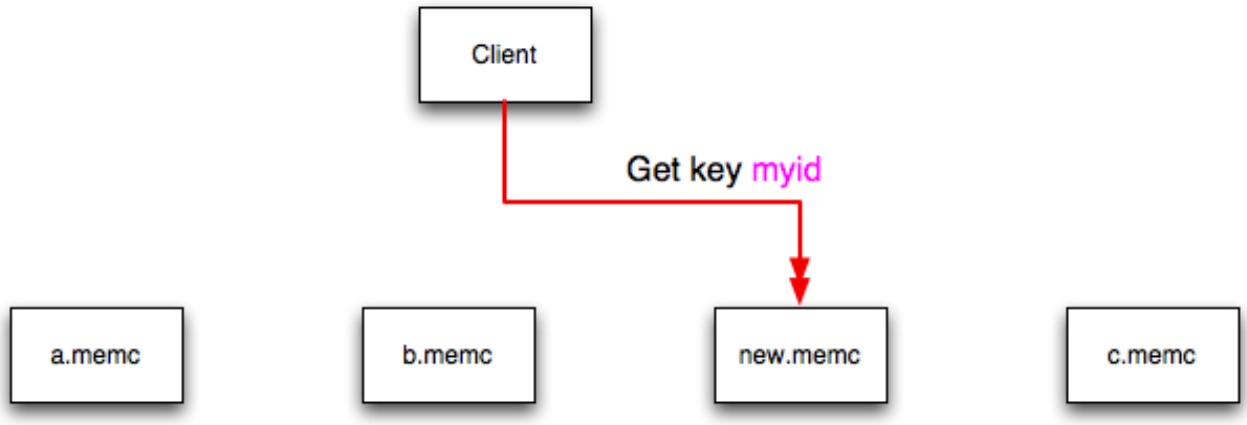
The problem with client-side selection of the server is that the list of the servers (including their sequential order) *must* remain consistent on each client using the memcached servers, and the servers must be available. If you try to perform an operation on a key when:

- A new memcached instance has been added to the list of available instances
- A memcached instance has been removed from the list of available instances
- The order of the memcached instances has changed

When the hashing algorithm is used on the given key, but with a different list of servers, the hash calculation may choose a different server from the list.

If a new memcached instance is added into the list of servers, as new.memc is in the example below, then a GET operation using the same key, myid, can result in a cache-miss. This is because the same value is computed from the key, which selects the same index from the array of servers, but index 2 now points to the new server, not the server c.memc where the data was originally stored. This would result in a cache miss, even though the key exists within the cache on another memcached instance.

Figure 2.2. memcached Hash Selection with New memcached instance



This means that servers c.memc and new.memc both contain the information for key myid, but the information stored against the key in eachs server may be different in each instance. A more significant problem is a much higher number of cache-misses when retrieving data, as the addition of a new server changes the distribution of keys, and this in turn requires rebuilding the cached data on the

memcached instances, causing an increase in database reads.

The same effect can occur if you actively manage the list of servers configured in your clients, adding and removing the configured memcached instances as each instance is identified as being available. For example, removing a memcached instance when the client notices that the instance can no longer be contacted can cause the server selection to fail as described here.

To prevent this causing significant problems and invalidating your cache, you can select the hashing algorithm used to select the server. There are two common types of hashing algorithm, *consistent* and *modula*.

With *consistent* hashing algorithms, the same key when applied to a list of servers always uses the same server to store or retrieve the keys, even if the list of configured servers changes. This means that you can add and remove servers from the configure list and always use the same server for a given key. There are two types of consistent hashing algorithms available, Ketama and Wheel. Both types are supported by libmemcached, and implementations are available for PHP and Java.

Any consistent hashing algorithm has some limitations. When you add servers to an existing list of configured servers, keys are distributed to the new servers as part of the normal distribution. When you remove servers from the list, the keys are re-allocated to another server within the list, meaning that the cache needs to be re-populated with the information. Also, a consistent hashing algorithm does not resolve the issue where you want consistent selection of a server across multiple clients, but where each client contains a different list of servers. The consistency is enforced only within a single client.

With a *modula* hashing algorithm, the client selects a server by first computing the hash and then choosing a server from the list of configured servers. As the list of servers changes, so the server selected when using a modula hashing algorithm also changes. The result is the behavior described above; changes to the list of servers mean that different servers are selected when retrieving data, leading to cache misses and increase in database load as the cache is re-seeded with information.

If you use only a single memcached instance for each client, or your list of memcached servers configured for a client never changes, then the selection of a hashing algorithm is irrelevant, as it has no noticeable effect.

If you change your servers regularly, or you use a common set of servers that are shared among a large number of clients, then using a consistent hashing algorithm should help to ensure that your cache data is not duplicated and the data is evenly distributed.

2.5. Using memcached and DTrace

memcached includes a number of different DTrace probes that can be used to monitor the operation of the server. The probes included can monitor individual connections, slab allocations, and modifications to the hash table when a key/value pair is added, updated, or removed.

For more information on DTrace and writing DTrace scripts, read the DTrace User Guide.

Support for DTrace probes was added to memcached 1.2.6 includes a number of DTrace probes that can be used to help monitor your application. DTrace is supported on Solaris 10, OpenSolaris, Mac OS X 10.5 and FreeBSD. To enable the DTrace probes in memcached, build from source and use the --enable-dtrace option. For more information, see Chapter 1, *Installing memcached*.

The probes supported by memcached are:

- conn-allocate(connid)

  Fired when a connection object is allocated from the connection pool.

  - connid: The connection ID

- conn-release(connid)

  Fired when a connection object is released back to the connection pool.

  Arguments:

  - connid: The connection ID

- conn-create(ptr)

  Fired when a new connection object is being created (that is, there are no free connection objects in the connection pool).

  Arguments:

  - ptr: A pointer to the connection object
- conn-destroy(ptr)

  Fired when a connection object is being destroyed.

  Arguments:

  - ptr: A pointer to the connection object
- conn-dispatch(connid, threadid)

  Fired when a connection is dispatched from the main or connection-management thread to a worker thread.

  Arguments:

  - connid: The connection ID
  - threadid: The thread ID
- slabs-allocate(size, slabclass, slabsize, ptr)

  Allocate memory from the slab allocator

  Arguments:

  - size: The requested size
  - slabclass: The allocation is fulfilled in this class
  - slabsize: The size of each item in this class
  - ptr: A pointer to allocated memory
- slabs-allocate-failed(size, slabclass)

  Failed to allocate memory (out of memory)

  Arguments:

  - size: The requested size
  - slabclass: The class that failed to fulfill the request
- slabs-slabclass-allocate(slabclass)

  Fired when a slab class needs more space

  Arguments:

  - slabclass: The class that needs more memory
- slabs-slabclass-allocate-failed(slabclass)

  Failed to allocate memory (out of memory)

  Arguments:

  - slabclass: The class that failed to grab more memory
- slabs-free(size, slabclass, ptr)

  Release memory

  Arguments:

  - size: The size of the memory
  - slabclass: The class the memory belongs to
  - ptr: A pointer to the memory to release

- assoc-find(key, depth)

  Fired when the when we have searched the hash table for a named key. These two elements provide an insight in how well the hash function operates. Traversals are a sign of a less optimal function, wasting cpu capacity.

  Arguments:

  - key: The key searched for
  - depth: The depth in the list of hash table

- assoc-insert(key, nokeys)

  Fired when a new item has been inserted.

  Arguments:

  - key: The key just inserted
  - nokeys: The total number of keys currently being stored, including the key for which insert was called.

- assoc-delete(key, nokeys)

  Fired when a new item has been removed.

  Arguments:

  - key: The key just deleted
  - nokeys: The total number of keys currently being stored, excluding the key for which delete was called.

- item-link(key, size)

  Fired when an item is being linked in the cache

  Arguments:

  - key: The items key
  - size: The size of the data

- item-unlink(key, size)

  Fired when an item is being deleted

  Arguments:

  - key: The items key
  - size: The size of the data

- item-remove(key, size)

  Fired when the refcount for an item is reduced

  Arguments:

  - key: The items key
  - size: The size of the data

- item-update(key, size)

  Fired when the "last referenced" time is updated

  Arguments:

  - key: The items key
  - size: The size of the data

- item-replace(oldkey, oldsize, newkey, newsize)

  Fired when an item is being replaced with another item

  Arguments:

  - oldkey: The key of the item to replace
  - oldsize: The size of the old item
  - newkey: The key of the new item
  - newsize: The size of the new item

- process-command-start(connid, request, size)

  Fired when the processing of a command starts

  Arguments:

  - connid: The connection ID
  - request: The incoming request
  - size: The size of the request

- process-command-end(connid, response, size)

  Fired when the processing of a command is done

  Arguments:

  - connid: The connection ID
  - response: The response to send back to the client
  - size: The size of the response

- command-get(connid, key, size)

  Fired for a get-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The size of the key's data (or -1 if not found)

- command-gets(connid, key, size, casid)

  Fired for a gets command

  Arguments:

  - connid: The connection ID

  - key: The requested key
  - size: The size of the key's data (or -1 if not found)
  - casid: The casid for the item
- command-add(connid, key, size)

  Fired for a add-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The new size of the key's data (or -1 if not found)
- command-set(connid, key, size)

  Fired for a set-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The new size of the key's data (or -1 if not found)
- command-replace(connid, key, size)

  Fired for a replace-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The new size of the key's data (or -1 if not found)
- command-prepend(connid, key, size)

  Fired for a prepend-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The new size of the key's data (or -1 if not found)
- command-append(connid, key, size)

  Fired for a append-command

  Arguments:

  - connid: The connection ID
  - key: The requested key
  - size: The new size of the key's data (or -1 if not found)

- `command-cas(connid, key, size, casid)`

  Fired for a cas-command

  Arguments:

  - `connid`: The connection ID
  - `key`: The requested key
  - `size`: The size of the key's data (or -1 if not found)
  - `casid`: The cas ID requested

- `command-incr(connid, key, val)`

  Fired for incr command

  Arguments:

  - `connid`: The connection ID
  - `key`: The requested key
  - `val`: The new value

- `command-decr(connid, key, val)`

  Fired for decr command

  Arguments:

  - `connid`: The connection ID
  - `key`: The requested key
  - `val`: The new value

- `command-delete(connid, key, exptime)`

  Fired for a delete command

  Arguments:

  - `connid`: The connection ID
  - `key`: The requested key
  - `exptime`: The expiry time

2.6. Memory allocation within memcached

When you first start `memcached`, the memory that you have configured is not automatically allocated. Instead, `memcached` only starts allocating and reserving physical memory once you start saving information into the cache.

When you start to store data into the cache, `memcached` does not allocate the memory for the data on an item by item basis. Instead, a slab allocation is used to optimize memory usage and prevent memory fragmentation when information expires from the cache.

With slab allocation, memory is reserved in blocks of 1MB. The slab is divided up into a number of blocks of equal size. When you try to store a value into the cache, `memcached` checks the size of the value that you are adding to the cache and determines which slab contains the right size allocation for the item. If a slab with the item size already exists, the item is written to the block within the slab.

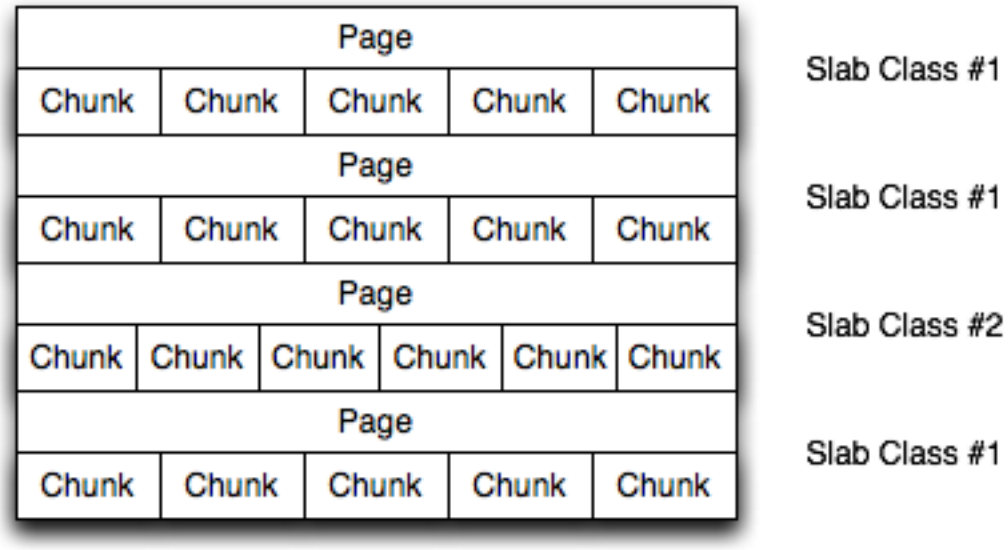
If the new item is bigger than the size of any existing blocks, then a new slab is created, divided up into blocks of a suitable size. If an existing slab with the right block size already exists, but there are no free blocks, a new slab is created. If you update an existing item with data that is larger than the existing block allocation for that key, then the key is re-allocated into a suitable slab.

For example, the default size for the smallest block is 88 bytes (40 bytes of value, and the default 48 bytes for the key and flag data). If the size of the first item you store into the cache is less than 40 bytes, then a slab with a block size of 88 bytes is created and the value stored.

If the size of the data that you intend to store is larger than this value, then the block size is increased by the chunk size factor until a block size large enough to hold the value is determined. The block size is always a function of the scale factor, rounded up to a block size which is exactly divisible into the chunk size.

For a sample of the structure, see Figure 2.3, "Memory Allocation in memcached".

Figure 2.3. Memory Allocation in memcached



The result is that you have multiple pages allocated within the range of memory allocated to memcached. Each page is 1MB in size (by default), and is split into a different number of chunks, according to the chunk size required to store the key/value pairs. Each instance has multiple pages allocated, and a page is always created when a new item needs to be created requiring a chunk of a particular size. A slab may consist of multiple pages, and each page within a slab contains an equal number of chunks.

The chunk size of a new slab is determined by the base chunk size combined with the chunk size growth factor. For example, if the initial chunks are 104 bytes in size, and the default chunk size growth factor is used (1.25), then the next chunk size allocated would be the best power of 2 fit for 104*1.25, or 136 bytes.

Allocating the pages in this way ensures that memory does not get fragmented. However, depending on the distribution of the objects that you store, it may lead to an inefficient distribution of the slabs and chunks if you have significantly different sized items. For example, having a relatively small number of items within each chunk size may waste a lot of memory with just few chunks in each allocated page.

You can tune the growth factor to reduce this effect by using the `-f` command line option, which adapts the growth factor applied to make more effective use of the chunks and slabs allocated. For information on how to determine the current slab allocation statistics, see Section 4.2, "memcached Slabs Statistics".

If your operating system supports it, you can also start memcached with the `-L` command line option. This option preallocates all the memory during startup using large memory pages. This can improve performance by reducing the number of misses in the CPU memory cache.

2.7. memcached thread Support

If you enable the thread implementation within when building memcached from source, then memcached uses multiple threads in addition to the `libevent` system to handle requests.

When enabled, the threading implementation operates as follows:

- Threading is handled by wrapping functions within the code to provide basic protection from updating the same global structures at the same time.
- Each thread uses its own instance of the `libevent` to help improve performance.
- TCP/IP connections are handled with a single thread listening on the TCP/IP socket. Each connection is then distribution to one of the active threads on a simple round-robin basis. Each connection then operates solely within this thread while the connection remains open.
- For UDP connections, all the threads listen to a single UDP socket for incoming requests. Threads that are not currently dealing with another request ignore the incoming packet. One of the remaining, nonbusy, threads reads the request and sends the response. This implementation can lead to increased CPU load as threads wake from sleep to potentially process the request.

Using threads can increase the performance on servers that have multiple CPU cores available, as the requests to update the hash table can be spread between the individual threads. To minimize overhead from the locking mechanism employed, experiment with different thread values to achieve the best performance based on the number and type of requests within your given workload.

2.8. `memcached` Logs

If you enable verbose mode, using the `-v`, `-vv`, or `-vvv` options, then the information output by `memcached` includes details of the operations being performed.

Without the verbose options, `memcached` normally produces no output during normal operating.

- **Output when using `-v`**

  The lowest verbosity level shows you:

  - Errors and warnings
  - Transient errors
  - Protocol and socket errors, including exhausting available connections
  - Each registered client connection, including the socket descriptor number and the protocol used.

    For example:

    ```
    32: Client using the ascii protocol
    33: Client using the ascii protocol
    ```

    Note that the socket descriptor is only valid while the client remains connected. Non-persistant connections may not be effectively represented.

  Examples of the error messages output at this level include:

  ```
  <%d send buffer was %d, now %d
  Can't listen for events on fd %d
  Can't read from libevent pipe
  Catastrophic: event fd doesn't match conn fd!
  Couldn't build response
  Couldn't realloc input buffer
  Couldn't update event
  Failed to build UDP headers
  Failed to read, and not due to blocking
  Too many open connections
  Unexpected state %d
  ```

- **Output when using `-vv`**

  When using the second level of verbosity, you get more detailed information about protocol operations, keys updated, chunk and network operatings and details.

During the initial start-up of memcached with this level of verbosity, you are shown the sizes of the individual slab classes, the chunk sizes, and the number of entries per slab. These do not show the allocation of the slabs, just the slabs that would be created when data is added. You are also given information about the listen queues and buffers used to send information. A sample of the output generated for a TCP/IP based system with the default memory and growth factors is given below:

```
shell> memcached -vv
slab class   1: chunk size     80 perslab 13107
slab class   2: chunk size    104 perslab 10082
slab class   3: chunk size    136 perslab  7710
slab class   4: chunk size    176 perslab  5957
slab class   5: chunk size    224 perslab  4681
slab class   6: chunk size    280 perslab  3744
slab class   7: chunk size    352 perslab  2978
slab class   8: chunk size    440 perslab  2383
slab class   9: chunk size    552 perslab  1899
slab class  10: chunk size    696 perslab  1506
slab class  11: chunk size    872 perslab  1202
slab class  12: chunk size   1096 perslab   956
slab class  13: chunk size   1376 perslab   762
slab class  14: chunk size   1720 perslab   609
slab class  15: chunk size   2152 perslab   487
slab class  16: chunk size   2696 perslab   388
slab class  17: chunk size   3376 perslab   310
slab class  18: chunk size   4224 perslab   248
slab class  19: chunk size   5280 perslab   198
slab class  20: chunk size   6600 perslab   158
slab class  21: chunk size   8256 perslab   127
slab class  22: chunk size  10320 perslab   101
slab class  23: chunk size  12904 perslab    81
slab class  24: chunk size  16136 perslab    64
slab class  25: chunk size  20176 perslab    51
slab class  26: chunk size  25224 perslab    41
slab class  27: chunk size  31536 perslab    33
slab class  28: chunk size  39424 perslab    26
slab class  29: chunk size  49280 perslab    21
slab class  30: chunk size  61600 perslab    17
slab class  31: chunk size  77000 perslab    13
slab class  32: chunk size  96256 perslab    10
slab class  33: chunk size 120320 perslab     8
slab class  34: chunk size 150400 perslab     6
slab class  35: chunk size 188000 perslab     5
slab class  36: chunk size 235000 perslab     4
slab class  37: chunk size 293752 perslab     3
slab class  38: chunk size 367192 perslab     2
slab class  39: chunk size 458992 perslab     2
<26 server listening (auto-negotiate)
<29 server listening (auto-negotiate)
<30 send buffer was 57344, now 2097152
<31 send buffer was 57344, now 2097152
<30 server listening (udp)
<30 server listening (udp)
<31 server listening (udp)
<30 server listening (udp)
<30 server listening (udp)
<31 server listening (udp)
<31 server listening (udp)
<31 server listening (udp)
```

Using this verbosity level can be a useful way to check the effects of the growth factor used on slabs with different memory allocations, which in turn can be used to better tune the growth factor to suit the data you are storing in the cache. For example, if you set the growth factor to 4 (quadrupling the size of each slab):

```
shell> memcached -f 4 -m 1g -vv
slab class   1: chunk size     80 perslab 13107
slab class   2: chunk size    320 perslab  3276
slab class   3: chunk size   1280 perslab   819
slab class   4: chunk size   5120 perslab   204
slab class   5: chunk size  20480 perslab    51
slab class   6: chunk size  81920 perslab    12
slab class   7: chunk size 327680 perslab     3
...
```

During use of the cache, this verbosity level also prints out detailed information on the storage and recovery of keys and other information. An example of the output during a typical set/get and increment/decrement operation is shown below.

```
32: Client using the ascii protocol
<32 set my_key 0 0 10
>32 STORED
<32 set object_key 1 0 36
>32 STORED
<32 get my_key
```

```
>32 sending key my_key
>32 END
<32 get object_key
>32 sending key object_key
>32 END
<32 set key 0 0 6
>32 STORED
<32 incr key 1
>32 789544
<32 decr key 1
>32 789543
<32 incr key 2
>32 789545
<32 set my_key 0 0 10
>32 STORED
<32 set object_key 1 0 36
>32 STORED
<32 get my_key
>32 sending key my_key
>32 END
<32 get object_key
>32 sending key object_key1 1 36
>32 END
<32 set key 0 0 6
>32 STORED
<32 incr key 1
>32 789544
<32 decr key 1
>32 789543
<32 incr key 2
>32 789545
```

During client communication, for each line, the initial character shows the direction of flow of the information. The < for communication from the client to the memcached server and > for communication back to the client. The number is the numeric socket descriptor for the connection.

- **Output when using -vvv**

  This level of verbosity includes the transitions of connections between different states in the event library while reading and writing content to/from the clients. It should be used to diagnose and identify issues in client communication. For example, you can use this information to determine if memcached is taking a long time to return information to the client, during the read of the client operation or before returning and completing the operation. An example of the typical sequence for a set operation is provided below:

```
<32 new auto-negotiating client connection
32: going from conn_new_cmd to conn_waiting
32: going from conn_waiting to conn_read
32: going from conn_read to conn_parse_cmd
32: Client using the ascii protocol
<32 set my_key 0 0 10
32: going from conn_parse_cmd to conn_nread
> NOT FOUND my_key
>32 STORED
32: going from conn_nread to conn_write
32: going from conn_write to conn_new_cmd
32: going from conn_new_cmd to conn_waiting
32: going from conn_waiting to conn_read
32: going from conn_read to conn_closing
<32 connection closed.
```

All of the verbosity levels in memcached are designed to be used during debugging or examination of issues. The quantity of information generated, particularly when using -vvv, is significant, particularly on a busy server. Also be aware that writing the error information out, especially to disk, may negate some of the performance gains you achieve by using memcached. Therefore, use in production or deployment environments is not recommended.

Chapter 3. memcached Interfaces

A number of interfaces from different languages exist for interacting with memcached servers and storing and retrieving information. Interfaces for the most common language platforms including Perl, PHP, Python, Ruby, C and Java.

Data stored into a memcached server is referred to by a single string (the key), with storage into the cache and retrieval from the cache using the key as the reference. The cache therefore operates like a large associative array or hash. It is not possible to structure or otherwise organize the information stored in the cache. To store information in a structured way, you must use 'formatted' keys.

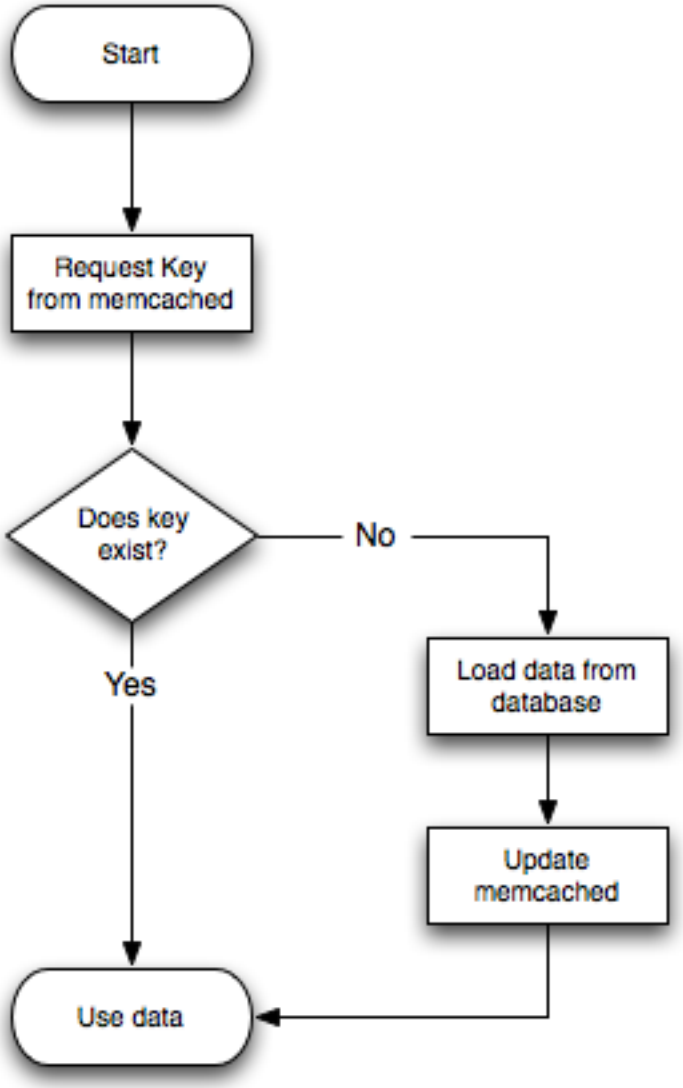
The following tips may be useful to you when using memcached:

The general sequence for using memcached in any language as a caching solution is as follows:

1. Request the item from the cache.
2. If the item exists, use the item data.
3. If the item does not exist, load the data from MySQL, and store the value into the cache. This means the value is available to the next client that requests it from the cache.

For a flow diagram of this sequence, see Figure 3.1, “Typical memcached Application Flowchart”.

Figure 3.1. Typical memcached Application Flowchart



The interface to memcached supports the following methods for storing and retrieving information in the cache, and these are consistent across all the different APIs, even though the language specific mechanics may be different:

- get(key): Retrieves information from the cache. Returns the value if it exists, or NULL, nil, or undefined or the closest equivalent in the corresponding language, if the specified key does not exist.
- set(key, value [, expiry]): Sets the key in the cache to the specified value. Note that this either updates an existing key if it already exists, or adds a new key/value pair if the key doesn't exist. If the expiry time is specified, then the key expires (and is deleted) when the expiry time is reached. The time is specified in seconds, and is taken as a relative time if the value is less than 30 days (30*24*60*60), or an absolute time (epoch) if larger than this value.
- add(key, value [, expiry]): Adds the key to the cache, if the specified key doesn't already exist.
- replace(key, value [, expiry]): Replaces the value of the specified key, only if the key already exists.
- delete(key [, time]): Deletes the key from the cache. If you supply a time, then adding a value with the specified key is blocked for the specified period.

- incr(key [, value]): Increments the specified key by one or the specified value.
- decr(key [, value]): Decrements the specified key by one or the specified value.
- flush_all: Invalidates (or expires) all the current items in the cache. Technically they still exist (they are not deleted), but they are silently destroyed the next time you try to access them.

In all implementations, most or all of these functions are duplicated through the corresponding native language interface.

For all languages and interfaces, use memcached to store full items, rather than simply caching single rows of information from the database. For example, when displaying a record about an object (invoice, user history, or blog post), load all the data for the associated entry from the database, and compile it into the internal structure that would normally be required by the application. You then save the complete object into the cache.

Complex data structures cannot be stored directly. Most interfaces serialize the data for you, that is, put it in a form that can reconstruct the original pointers and nesting. Perl uses Storable, PHP uses serialize, Python uses cPickle (or Pickle) and Java uses the Serializable interface. In most cases, the serialization interface used is customizable. To share data stored in memcached instances between different language interfaces, consider using a common serialization solution such as JSON (Javascript Object Notation).

3.1. Using libmemcached

The libmemcached library provides both C and C++ interfaces to memcached and is also the basis for a number of different additional API implementations, including Perl, Python and Ruby. Understanding the core libmemcached functions can help when using these other interfaces.

The C library is the most comprehensive interface library for memcached and provides a wealth of functions and operational systems not always exposed in the other interfaces not based on the libmemcached library.

The different functions can be divided up according to their basic operation. In addition to functions that interface to the core API, there are a number of utility functions that provide extended functionality, such as appending and prepending data.

To build and install libmemcached, download the libmemcached package, run configure, and then build and install:

```
shell> tar xjf libmemcached-0.21.tar.gz
shell> cd libmemcached-0.21
shell> ./configure
shell> make
shell> make install
```

On many Linux operating systems, you can install the corresponding libmemcached package through the usual yum, apt-get or similar commands.

To build an application that uses the library, first set the list of servers. Either directly manipulate the servers configured within the main memcached_st structure, or separately populate a list of servers, and then add this list to the memcached_st structure. The latter method is used in the following example. Once the server list has been set, you can call the functions to store or retrieve data. A simple application for setting a preset value to localhost is provided here:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <libmemcached/memcached.h>
int main(int argc, char *argv[])
{
  memcached_server_st *servers = NULL;
  memcached_st *memc;
  memcached_return rc;
  char *key= "keystring";
  char *value= "keyvalue";
  memcached_server_st *memcached_servers_parse (char *server_strings);
  memc= memcached_create(NULL);
  servers= memcached_server_list_append(servers, "localhost", 11211, &rc);
  rc= memcached_server_push(memc, servers);
  if (rc == MEMCACHED_SUCCESS)
    fprintf(stderr,"Added server successfully\n");
  else
    fprintf(stderr,"Couldn't add server: %s\n",memcached_strerror(memc, rc));
  rc= memcached_set(memc, key, strlen(key), value, strlen(value), (time_t)0, (uint32_t)0);
  if (rc == MEMCACHED_SUCCESS)
    fprintf(stderr,"Key stored successfully\n");
  else
```

```
    fprintf(stderr,"Couldn't store key: %s\n",memcached_strerror(memc, rc));
  return 0;
}
```

You can test the success of an operation by using the return value, or populated result code, for a given function. The value is always set to MEMCACHED_SUCCESS if the operation succeeded. In the event of a failure, use the memcached_strerror() function to translate the result code into a printable string.

To build the application, you must specify the memcached library:

```
shell> gcc -o memc_basic memc_basic.c -lmemcached
```

Running the above sample application, after starting a memcached server, should return a success message:

```
shell> memc_basic
Added server successfully
Key stored successfully
```

3.1.1. libmemcached Base Functions

The base libmemcached functions enable you to create, destroy and clone the main memcached_st structure that is used to interface to the memcached servers. The main functions are defined below:

```
memcached_st *memcached_create (memcached_st *ptr);
```

Creates a new memcached_st structure for use with the other libmemcached API functions. You can supply an existing, static, memcached_st structure, or NULL to have a new structured allocated. Returns a pointer to the created structure, or NULL on failure.

```
void memcached_free (memcached_st *ptr);
```

Free the structure and memory allocated to a previously created memcached_st structure.

```
memcached_st *memcached_clone(memcached_st *clone, memcached_st *source);
```

Clone an existing memcached structure from the specified source, copying the defaults and list of servers defined in the structure.

3.1.2. libmemcached Server Functions

The libmemcached API uses a list of servers, stored within the memcached_server_st structure, to act as the list of servers used by the rest of the functions. To use memcached, you first create the server list, and then apply the list of servers to a valid libmemcached object.

Because the list of servers, and the list of servers within an active libmemcached object can be manipulated separately, you can update and manage server lists while an active libmemcached interface is running.

The functions for manipulating the list of servers within a memcached_st structure are given below:

```
memcached_return
          memcached_server_add (memcached_st *ptr,
                                char *hostname,
                                unsigned int port);
```

Add a server, using the given hostname and port into the memcached_st structure given in ptr.

```
memcached_return
          memcached_server_add_unix_socket (memcached_st *ptr,
                                            char *socket);
```

Add a Unix socket to the list of servers configured in the memcached_st structure.

```
unsigned int memcached_server_count (memcached_st *ptr);
```

Return a count of the number of configured servers within the memcached_st structure.

```
memcached_server_st *
          memcached_server_list (memcached_st *ptr);
```

Returns an array of all the defined hosts within a memcached_st structure.

```
memcached_return
          memcached_server_push (memcached_st *ptr,
                                 memcached_server_st *list);
```

Pushes an existing list of servers onto list of servers configured for a current memcached_st structure. This adds servers to the end of the existing list, and duplicates are not checked.

The memcached_server_st structure can be used to create a list of memcached servers which can then be applied individually to memcached_st structures.

```
memcached_server_st *
          memcached_server_list_append (memcached_server_st *ptr,
                                        char *hostname,
                                        unsigned int port,
                                        memcached_return *error);
```

Add a server, with hostname and port, to the server list in ptr. The result code is handled by the error argument, which should point to an existing memcached_return variable. The function returns a pointer to the returned list.

```
 unsigned int memcached_server_list_count (memcached_server_st *ptr);
```

Return the number of the servers in the server list.

```
void memcached_server_list_free (memcached_server_st *ptr);
```

Free up the memory associated with a server list.

```
memcached_server_st *memcached_servers_parse (char *server_strings);
```

Parses a string containing a list of servers, where individual servers are separated by a comma, space, or both, and where individual servers are of the form server[:port]. The return value is a server list structure.

3.1.3. libmemcached Set Functions

The set related functions within libmemcached provide the same functionality as the core functions supported by the memcached protocol. The full definition for the different functions is the same for all the base functions (add, replace, prepend, append). For example, the function definition for memcached_set() is:

```
memcached_return
          memcached_set (memcached_st *ptr,
                         const char *key,
                         size_t key_length,
                         const char *value,
                         size_t value_length,
                         time_t expiration,
                         uint32_t flags);
```

The ptr is the memcached_st structure. The key and key_length define the key name and length, and value and value_length the corresponding value and length. You can also set the expiration and optional flags. For more information, see Section 3.1.5, “libmemcached Behaviors”.

The following table outlines the remainder of the set-related functions.

libmemcached Function	Equivalent to
memcached_set(memc, key, key_length, value, value_length, expiration, flags)	Generic set() operation.
memcached_add(memc, key, key_length, value, value_length, expiration, flags)	Generic add() function.
memcached_replace(memc, key, key_length,	Generic replace().

libmemcached Function	Equivalent to
value, value_length, expiration, flags)	
memcached_prepend(memc, key, key_length, value, value_length, expiration, flags)	Prepends the specified value before the current value of the specified key.
memcached_append(memc, key, key_length, value, value_length, expiration, flags)	Appends the specified value after the current value of the specified key.
memcached_cas(memc, key, key_length, value, value_length, expiration, flags, cas)	Overwrites the data for a given key as long as the corresponding cas value is still the same within the server.
memcached_set_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the generic set(), but has the option of an additional master key that can be used to identify an individual server.
memcached_add_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the generic add(), but has the option of an additional master key that can be used to identify an individual server.
memcached_replace_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the generic replace(), but has the option of an additional master key that can be used to identify an individual server.
memcached_prepend_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the memcached_prepend(), but has the option of an additional master key that can be used to identify an individual server.
memcached_append_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the memcached_append(), but has the option of an additional master key that can be used to identify an individual server.
memcached_cas_by_key(memc, master_key, master_key_length, key, key_length, value, value_length, expiration, flags)	Similar to the memcached_cas(), but has the option of an additional master key that can be used to identify an individual server.

The by_key methods add two further arguments, the master key, to be used and applied during the hashing stage for selecting the servers. You can see this in the following definition:

```
memcached_return
          memcached_set_by_key(memcached_st *ptr,
                               const char *master_key,
                               size_t master_key_length,
                               const char *key,
                               size_t key_length,
                               const char *value,
                               size_t value_length,
                               time_t expiration,
                               uint32_t flags);
```

All the functions return a value of type memcached_return, which you can compare against the MEMCACHED_SUCCESS constant.

3.1.4. libmemcached Get Functions

The libmemcached functions provide both direct access to a single item, and a multiple-key request mechanism that provides much faster responses when fetching a large number of keys simultaneously.

The main get-style function, which is equivalent to the generic get() is memcached_get(). The functions a string pointer to the returned value for a corresponding key.

```
char *memcached_get (memcached_st *ptr,
                     const char *key, size_t key_length,
                     size_t *value_length,
                     uint32_t *flags,
                     memcached_return *error);
```

A multi-key get, memcached_mget(), is also available. Using a multiple key get operation is much quicker to do in one block than retrieving the key values with individual calls to memcached_get(). To start the multi-key get, call memcached_mget():

```
memcached_return
         memcached_mget (memcached_st *ptr,
```

```
                          char **keys, size_t *key_length,
                          unsigned int number_of_keys);
```

The return value is the success of the operation. The `keys` parameter should be an array of strings containing the keys, and `key_length` an array containing the length of each corresponding key. `number_of_keys` is the number of keys supplied in the array.

To fetch the individual values, use `memcached_fetch()` to get each corresponding value.

```
char *memcached_fetch (memcached_st *ptr,
                          const char *key, size_t *key_length,
                          size_t *value_length,
                          uint32_t *flags,
                          memcached_return *error);
```

The function returns the key value, with the `key`, `key_length` and `value_length` parameters being populated with the corresponding key and length information. The function returns `NULL` when there are no more values to be returned. A full example, including the populating of the key data and the return of the information is provided here.

```
#include <stdio.h>
#include <sstring.h>
#include <unistd.h>
#include <libmemcached/memcached.h>
int main(int argc, char *argv[])
{
  memcached_server_st *servers = NULL;
  memcached_st *memc;
  memcached_return rc;
  char *keys[]= {"huey", "dewey", "louie"};
  size_t key_length[3];
  char *values[]= {"red", "blue", "green"};
  size_t value_length[3];
  unsigned int x;
  uint32_t flags;
  char return_key[MEMCACHED_MAX_KEY];
  size_t return_key_length;
  char *return_value;
  size_t return_value_length;
  memc= memcached_create(NULL);
  servers= memcached_server_list_append(servers, "localhost", 11211, &rc);
  rc= memcached_server_push(memc, servers);
  if (rc == MEMCACHED_SUCCESS)
    fprintf(stderr,"Added server successfully\n");
  else
    fprintf(stderr,"Couldn't add server: %s\n",memcached_strerror(memc, rc));
  for(x= 0; x < 3; x++)
    {
      key_length[x] = strlen(keys[x]);
      value_length[x] = strlen(values[x]);
      rc= memcached_set(memc, keys[x], key_length[x], values[x],
                        value_length[x], (time_t)0, (uint32_t)0);
      if (rc == MEMCACHED_SUCCESS)
        fprintf(stderr,"Key %s stored successfully\n",keys[x]);
      else
        fprintf(stderr,"Couldn't store key: %s\n",memcached_strerror(memc, rc));
    }
  rc= memcached_mget(memc, keys, key_length, 3);
  if (rc == MEMCACHED_SUCCESS)
    {
      while ((return_value= memcached_fetch(memc, return_key, &return_key_length,
                                            &return_value_length, &flags, &rc)) != NULL)
        {
          if (rc == MEMCACHED_SUCCESS)
            {
              fprintf(stderr,"Key %s returned %s\n",return_key, return_value);
            }
        }
    }
  return 0;
}
```

Running the above application:

```
shell> memc_multi_fetch
Added server successfully
Key huey stored successfully
Key dewey stored successfully
Key louie stored successfully
Key huey returned red
Key dewey returned blue
Key louie returned green
```

3.1.5. libmemcached Behaviors

The behavior of libmemcached can be modified by setting one or more behavior flags. These can either be set globally, or they can be applied during the call to individual functions. Some behaviors also accept an additional setting, such as the hashing mechanism used when selecting servers.

To set global behaviors:

```
memcached_return
          memcached_behavior_set (memcached_st *ptr,
                                  memcached_behavior flag,
                                  uint64_t data);
```

To get the current behavior setting:

```
uint64_t
          memcached_behavior_get (memcached_st *ptr,
                                  memcached_behavior flag);
```

Behavior	Description
MEMCACHED_BEHAVIOR_NO_BLOCK	Caused libmemcached to use asynchronous I/O.
MEMCACHED_BEHAVIOR_TCP_NODELAY	Turns on no-delay for network sockets.
MEMCACHED_BEHAVIOR_HASH	Without a value, sets the default hashing algorithm for keys to use MD5. Other valid values include MEMCACHED_HASH_DEFAULT, MEMCACHED_HASH_MD5, MEMCACHED_HASH_CRC, MEMCACHED_HASH_FNV1_64, MEMCACHED_HASH_FNV1A_64, MEMCACHED_HASH_FNV1_32, and MEMCACHED_HASH_FNV1A_32.
MEMCACHED_BEHAVIOR_DISTRIBUTION	Changes the method of selecting the server used to store a given value. The default method is MEMCACHED_DISTRIBUTION_MODULA. You can enable consistent hashing by setting MEMCACHED_DISTRIBUTION_CONSISTENT. MEMCACHED_DISTRIBUTION_CONSISTENT is an alias for the value MEMCACHED_DISTRIBUTION_CONSISTENT_KETAMA.
MEMCACHED_BEHAVIOR_CACHE_LOOKUPS	Cache the lookups made to the DNS service. This can improve the performance if you are using names instead of IP addresses for individual hosts.
MEMCACHED_BEHAVIOR_SUPPORT_CAS	Support CAS operations. By default, this is disabled because it imposes a performance penalty.
MEMCACHED_BEHAVIOR_KETAMA	Sets the default distribution to MEMCACHED_DISTRIBUTION_CONSISTENT_KETAMA and the hash to MEMCACHED_HASH_MD5.
MEMCACHED_BEHAVIOR_POLL_TIMEOUT	Modify the timeout value used by poll(). Supply a signed int pointer for the timeout value.
MEMCACHED_BEHAVIOR_BUFFER_REQUESTS	Buffers IO requests instead of them being sent. A get operation, or closing the connection causes the data to be flushed.
MEMCACHED_BEHAVIOR_VERIFY_KEY	Forces libmemcached to verify that a specified key is valid.
MEMCACHED_BEHAVIOR_SORT_HOSTS	If set, hosts added to the list of configured hosts for a memcached_st structure are placed into the host list in sorted order. This breaks consistent hashing if that behavior has been enabled.
MEMCACHED_BEHAVIOR_CONNECT_TIMEOUT	In nonblocking mode this changes the value of the timeout during socket connection.

3.1.6. libmemcached Command-line Utilities

In addition to the main C library interface, libmemcached also includes a number of command line utilities that can be useful when working with and debugging memcached applications.

All of the command line tools accept a number of arguments, the most critical of which is servers, which specifies the list of servers to connect to when returning information.

The main tools are:

- memcat: Display the value for each ID given on the command line:

```
shell> memcat --servers=localhost hwkey
Hello world
```

- memcp: Copy the contents of a file into the cache, using the file names as the key:

```
shell> echo "Hello World" > hwkey
shell> memcp --servers=localhost hwkey
shell> memcat --servers=localhost hwkey
Hello world
```

- memrm: Remove an item from the cache:

```
shell> memcat --servers=localhost hwkey
Hello world
shell> memrm --servers=localhost hwkey
shell> memcat --servers=localhost hwkey
```

- memslap: Test the load on one or more memcached servers, simulating get/set and multiple client operations. For example, you can simulate the load of 100 clients performing get operations:

```
shell> memslap --servers=localhost --concurrency=100 --flush --test=get
memslap --servers=localhost --concurrency=100 --flush --test=get Threads connecting to servers 100
        Took 13.571 seconds to read data
```

- memflush: Flush (empty) the contents of the memcached cache.

```
shell> memflush --servers=localhost
```

3.2. Using MySQL and memcached with Perl

The Cache::Memcached module provides a native interface to the Memcache protocol, and provides support for the core functions offered by memcached. You should install the module using your hosts native package management system. Alternatively, you can install the module using CPAN:

```
root-shell> perl -MCPAN -e 'install Cache::Memcached'
```

To use memcached from Perl through the Cache::Memcached module, first create a new Cache::Memcached object that defines the list of servers and other parameters for the connection. The only argument is a hash containing the options for the cache interface. For example, to create a new instance that uses three memcached servers:

```
use Cache::Memcached;
my $cache = new Cache::Memcached {
    'servers' => [
        '192.168.0.100:11211',
        '192.168.0.101:11211',
        '192.168.0.102:11211',
        ],
};
```

> **Note**
>
> When using the Cache::Memcached interface with multiple servers, the API automatically performs certain operations across all the servers in the group. For example, getting statistical information through Cache::Memcached returns a hash that contains data on a host by host basis, as well as generalized statistics for all the servers in the group.

You can set additional properties on the cache object instance when it is created by specifying the option as part of the option hash. Alternatively, you can use a corresponding method on the instance:

- servers or method set_servers(): Specifies the list of the servers to be used. The servers list should be a reference to an array of servers, with each element as the address and port number combination (separated by a colon). You can also specify a local connection through a UNIX socket (for example /tmp/sock/memcached). You can also specify the server with a weight (indicating how much more frequently the server should be used during hashing) by specifying an array reference with the mem-

cached server instance and a weight number. Higher numbers give higher priority.

- compress_threshold or method set_compress_threshold(): Specifies the threshold when values are compressed. Values larger than the specified number are automatically compressed (using zlib) during storage and retrieval.
- no_rehash or method set_norehash(): Disables finding a new server if the original choice is unavailable.
- readonly or method set_readonly(): Disables writes to the memcached servers.

Once the Cache::Memcached object instance has been configured you can use the set() and get() methods to store and retrieve information from the memcached servers. Objects stored in the cache are automatically serialized and deserialized using the Storable module.

The Cache::Memcached interface supports the following methods for storing/retrieving data, and relate to the generic methods as shown in the table.

Cache::Memcached Function	Equivalent to
get()	Generic get()
get_multi(keys)	Gets multiple keys from memcache using just one query. Returns a hash reference of key/value pairs.
set()	Generic set()
add()	Generic add()
replace()	Generic replace()
delete()	Generic delete()
incr()	Generic incr()
decr()	Generic decr()

Below is a complete example for using memcached with Perl and the Cache::Memcached module:

```
#!/usr/bin/perl
use Cache::Memcached;
use DBI;
use Data::Dumper;
# Configure the memcached server
my $cache = new Cache::Memcached {
    'servers' => [
                  'localhost:11211',
                  ],
    };
# Get the film name from the command line
# memcached keys must not contain spaces, so create
# a key name by replacing spaces with underscores
my $filmname = shift or die "Must specify the film name\n";
my $filmkey = $filmname;
$filmkey =~ s/ /_/;
# Load the data from the cache
my $filmdata = $cache->get($filmkey);
# If the data wasn't in the cache, then we load it from the database
if (!defined($filmdata))
{
    $filmdata = load_filmdata($filmname);
    if (defined($filmdata))
    {
# Set the data into the cache, using the key
        if ($cache->set($filmkey,$filmdata))
        {
            print STDERR "Film data loaded from database and cached\n";
        }
        else
        {
            print STDERR "Couldn't store to cache\n";
        }
    }
    else
    {
             die "Couldn't find $filmname\n";
    }
}
else
{
    print STDERR "Film data loaded from Memcached\n";
```

```
}
sub load_filmdata
{
    my ($filmname) = @_;
    my $dsn = "DBI:mysql:database=sakila;host=localhost;port=3306";
    $dbh = DBI->connect($dsn, 'sakila','password');
    my ($filmbase) = $dbh->selectrow_hashref(sprintf('select * from film where title = %s',
                                                     $dbh->quote($filmname)));
    if (!defined($filmname))
    {
            return (undef);
    }
    $filmbase->{stars} =
        $dbh->selectall_arrayref(sprintf('select concat(first_name," ",last_name) ' .
                                         'from film_actor left join (actor) ' .
                                         'on (film_actor.actor_id = actor.actor_id) ' .
                                         ' where film_id=%s',
                                         $dbh->quote($filmbase->{film_id})));
    return($filmbase);
}
```

The example uses the Sakila database, obtaining film data from the database and writing a composite record of the film and actors to memcache. When calling it for a film does not exist, you get this result:

```
shell> memcached-sakila.pl "ROCK INSTINCT"
Film data loaded from database and cached
```

When accessing a film that has already been added to the cache:

```
shell> memcached-sakila.pl "ROCK INSTINCT"
Film data loaded from Memcached
```

3.3. Using MySQL and memcached with Python

The Python `memcache` module interfaces to `memcached` servers, and is written in pure python (that is, without using one of the C APIs). You can download and install a copy from Python Memcached.

To install, download the package and then run the Python installer:

```
python setup.py install
running install
running bdist_egg
running egg_info
creating python_memcached.egg-info
...
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing python_memcached-1.43-py2.4.egg
creating /usr/lib64/python2.4/site-packages/python_memcached-1.43-py2.4.egg
Extracting python_memcached-1.43-py2.4.egg to /usr/lib64/python2.4/site-packages
Adding python-memcached 1.43 to easy-install.pth file
Installed /usr/lib64/python2.4/site-packages/python_memcached-1.43-py2.4.egg
Processing dependencies for python-memcached==1.43
Finished processing dependencies for python-memcached==1.43
```

Once installed, the `memcache` module provides a class-based interface to your `memcached` servers. Serialization of Python structures is handled by using the Python `cPickle` or `pickle` modules.

To create a new `memcache` interface, import the `memcache` module and create a new instance of the `memcache.Client` class:

```
import memcache
memc = memcache.Client(['127.0.0.1:11211'])
```

The first argument should be an array of strings containing the server and port number for each `memcached` instance to use. You can enable debugging by setting the optional `debug` parameter to 1.

By default, the hashing mechanism used is `crc32`. This provides a basic module hashing algorithm for selecting among multiple servers. You can change the function used by setting the value of `memcache.serverHashFunction` to the alternate function to use. For example:

```
from zlib import adler32
memcache.serverHashFunction = adler32
```

Once you have defined the servers to use within the memcache instance, the core functions provide the same functionality as in the generic interface specification. A summary of the supported functions is provided in the following table.

Python memcache Function	Equivalent to
get()	Generic get()
get_multi(keys)	Gets multiple values from the supplied array of keys. Returns a hash reference of key/value pairs.
set()	Generic set()
set_multi(dict [, expiry [, key_prefix]])	Sets multiple key/value pairs from the supplied dict.
add()	Generic add()
replace()	Generic replace()
prepend(key, value [, expiry])	Prepends the supplied value to the value of the existing key.
append(key, value [, expiry[)	Appends the supplied value to the value of the existing key.
delete()	Generic delete()
delete_multi(keys [, expiry [, key_prefix]])	Deletes all the keys from the hash matching each string in the array keys.
incr()	Generic incr()
decr()	Generic decr()

> **Note**
>
> Within the Python memcache module, all the *_multi() functions support an optional key_prefix parameter. If supplied, then the string is used as a prefix to all key lookups. For example, if you call:
>
> ```
> memc.get_multi(['a','b'], key_prefix='users:')
> ```
>
> The function retrieves the keys users:a and users:b from the servers.

An example showing the storage and retrieval of information to a memcache instance, loading the raw data from MySQL, is shown below:

```
import sys
import MySQLdb
import memcache
memc = memcache.Client(['127.0.0.1:11211'], debug=1);
try:
    conn = MySQLdb.connect (host = "localhost",
                            user = "sakila",
                            passwd = "password",
                            db = "sakila")
except MySQLdb.Error, e:
     print "Error %d: %s" % (e.args[0], e.args[1])
     sys.exit (1)
popularfilms = memc.get('top5films')
if not popularfilms:
    cursor = conn.cursor()
    cursor.execute('select film_id,title from film order by rental_rate desc limit 5')
    rows = cursor.fetchall()
    memc.set('top5films',rows,60)
    print "Updated memcached with MySQL data"
else:
    print "Loaded data from memcached"
    for row in popularfilms:
        print "%s, %s" % (row[0], row[1])
```

When executed for the first time, the data is loaded from the MySQL database and stored to the memcached server.

```
shell> python memc_python.py
Updated memcached with MySQL data
```

The data is automatically serialized using cPickle/pickle. This means when you load the data back from memcached, you can use the object directly. In the example above, the information stored to memcached is in the form of rows from a Python DB cursor. When accessing the information (within the 60 second expiry time), the data is loaded from memcached and dumped:

```
shell> python memc_python.py
Loaded data from memcached
2, ACE GOLDFINGER
7, AIRPLANE SIERRA
8, AIRPORT POLLOCK
10, ALADDIN CALENDAR
13, ALI FOREVER
```

The serialization and deserialization happens automatically, but be aware that serialization of Python data may be incompatible with other interfaces and languages. You can change the serialization module used during initialization, for example to use JSON, which is more easily exchanged.

3.4. Using MySQL and memcached with PHP

PHP provides support for the Memcache functions through a PECL extension. To enable the PHP memcache extensions, you must build PHP using the --enable-memcache option to configure when building from source.

If you are installing on a Red Hat based server, you can install the php-pecl-memcache RPM:

```
root-shell> yum --install php-pecl-memcache
```

On Debian based distributions, use the php-memcache package.

You can set global runtime configuration options by specifying the values in the following table within your php.ini file.

Configuration option	Default	Description
memcache.allow_failover	1	Specifies whether another server in the list should be queried if the first server selected fails.
memcache.max_failover_attempts	20	Specifies the number of servers to try before returning a failure.
memcache.chunk_size	8192	Defines the size of network chunks used to exchange data with the memcached server.
memcache.default_port	11211	Defines the default port to use when communicating with the memcached servers.
memcache.hash_strategy	standard	Specifies which hash strategy to use. Set to consistent to enable servers to be added or removed from the pool without causing the keys to be remapped to other servers. When set to standard, an older (modula) strategy is used that potentially uses different servers for storage.
memcache.hash_function	crc32	Specifies which function to use when mapping keys to servers. crc32 uses the standard CRC32 hash. fnv uses the FNV-1a hashing algorithm.

To create a connection to a memcached server, create a new Memcache object and then specify the connection options. For example:

```
<?php
$cache = new Memcache;
$cache->connect('localhost',11211);
?>
```

This opens an immediate connection to the specified server.

To use multiple memcached servers, you need to add servers to the memcache object using addServer():

```
bool Memcache::addServer ( string $host [, int $port [, bool $persistent
                [, int $weight [, int $timeout [, int $retry_interval
                [, bool $status [, callback $failure_callback
                ]]]]]] )
```

The server management mechanism within the php-memcache module is a critical part of the interface as it controls the main interface to the memcached instances and how the different instances are selected through the hashing mechanism.

To create a simple connection to two memcached instances:

```
<?php
$cache = new Memcache;
$cache->addServer('192.168.0.100',11211);
$cache->addServer('192.168.0.101',11211);
?>
```

In this scenario the instance connection is not explicitly opened, but only opened when you try to store or retrieve a value. You can enable persistent connections to memcached instances by setting the $persistent argument to true. This is the default setting, and causes the connections to remain open.

To help control the distribution of keys to different instances, use the global memcache.hash_strategy setting. This sets the hashing mechanism used to select. You can also add another weight to each server, which effectively increases the number of times the instance entry appears in the instance list, therefore increasing the likelihood of the instance being chosen over other instances. To set the weight, set the value of the $weight argument to more than one.

The functions for setting and retrieving information are identical to the generic functional interface offered by memcached, as shown in this table.

PECL memcache Function	Equivalent to
get()	Generic get()
set()	Generic set()
add()	Generic add()
replace()	Generic replace()
delete()	Generic delete()
increment()	Generic incr()
decrement()	Generic decr()

A full example of the PECL memcache interface is provided below. The code loads film data from the Sakila database when the user provides a film name. The data stored into the memcached instance is recorded as a mysqli result row, and the API automatically serializes the information for you.

```
<?php
$memc = new Memcache;
$memc->addServer('localhost','11211');
?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
 <title>Simple Memcache Lookup</title>
</head>
<body>
<form method="post">
  <p><b>Film</b>: <input type="text" size="20" name="film"></p>
<input type="submit">
</form>
<hr/>
<?php
  echo "Loading data...\n";
$value = $memc->get($_REQUEST['film']);
if ($value)
  {
    printf("<p>Film data for %s loaded from memcache</p>",$value['title']);
    foreach (array_keys($value) as $key)
      {
        printf("<p><b>%s</b>: %s</p>",$key, $value[$key]);
      }
  }
 else
   {
     $con = new mysqli('localhost','sakila','password','sakila') or
       die ("<h1>Database problem</h1>" . mysqli_connect_error());
     $result = $con->query(sprintf('select * from film where title ="%s"',$_REQUEST['film']));
     $row = $result->fetch_array(MYSQLI_ASSOC);
     $memc->set($row['title'],$row);
     printf("<p>Loaded %s from MySQL</p>",$row['title']);
   }
?>
```

With PHP, the connections to the memcached instances are kept open as long as the PHP and associated Apache instance remain running. When adding a removing servers from the list in a running instance (for example, when starting another script that mentions additional servers), the connections are shared, but the script only selects among the instances explicitly configured within the script.

To ensure that changes to the server list within a script do not cause problems, make sure to use the consistent hashing mechanism.

3.5. Using MySQL and memcached with Ruby

There are a number of different modules for interfacing to memcached within Ruby. The Ruby-MemCache client library provides a native interface to memcached that does not require any external libraries, such as libmemcached. You can obtain the installer package from http://www.deveiate.org/projects/RMemCache.

To install, extract the package and then run install.rb:

```
shell> install.rb
```

If you have RubyGems, you can install the Ruby-MemCache gem:

```
shell> gem install Ruby-MemCache
Bulk updating Gem source index for: http://gems.rubyforge.org
Install required dependency io-reactor? [Yn]  y
Successfully installed Ruby-MemCache-0.0.1
Successfully installed io-reactor-0.05
Installing ri documentation for io-reactor-0.05...
Installing RDoc documentation for io-reactor-0.05...
```

To use a memcached instance from within Ruby, create a new instance of the MemCache object.

```
require 'memcache'
memc = MemCache::new '192.168.0.100:11211'
```

You can add a weight to each server to increase the likelihood of the server being selected during hashing by appending the weight count to the server host name/port string:

```
require 'memcache'
memc = MemCache::new '192.168.0.100:11211:3'
```

To add servers to an existing list, you can append them directly to the MemCache object:

```
memc += ["192.168.0.101:11211"]
```

To set data into the cache, you can just assign a value to a key within the new cache object, which works just like a standard Ruby hash object:

```
memc["key"] = "value"
```

Or to retrieve the value:

```
print memc["key"]
```

For more explicit actions, you can use the method interface, which mimics the main memcached API functions, as summarized in the following table.

Ruby MemCache Method	Equivalent to
get()	Generic get()
get_hash(keys)	Get the values of multiple keys, returning the information as a hash of the keys and their values.
set()	Generic set()
set_many(pairs)	Set the values of the keys and values in the hash pairs.
add()	Generic add()
replace()	Generic replace()

Ruby MemCache Method	Equivalent to
delete()	Generic delete()
incr()	Generic incr()
decr()	Generic decr()

3.6. Using MySQL and memcached with Java

The com.danga.MemCached class within Java provides a native interface to memcached instances. You can obtain the client from http://whalin.com/memcached/. The Java class uses hashes that are compatible with libmemcached, so you can mix and match Java and libmemcached applications accessing the same memcached instances. The serialization between Java and other interfaces are not compatible. If this is a problem, use JSON or a similar nonbinary serialization format.

On most systems you can download the package and use the jar directly.

To use the com.danga.MemCached interface, you create a MemCachedClient instance and then configure the list of servers by configuring the SockIOPool. Through the pool specification you set up the server list, weighting, and the connection parameters to optimized the connections between your client and the memcached instances that you configure.

Generally you can configure the memcached interface once within a single class and then use this interface throughout the rest of your application.

For example, to create a basic interface, first configure the MemCachedClient and base SockIOPool settings:

```
public class MyClass {
    protected static MemCachedClient mcc = new MemCachedClient();
    static {

        String[] servers =
            {
                "localhost:11211",
            };

        Integer[] weights = { 1 };

        SockIOPool pool = SockIOPool.getInstance();

        pool.setServers( servers );
        pool.setWeights( weights );
```

In the above sample, the list of servers is configured by creating an array of the memcached instances to use. You can then configure individual weights for each server.

The remainder of the properties for the connection are optional, but you can set the connection numbers (initial connections, minimum connections, maximum connections, and the idle timeout) by setting the pool parameters:

```
pool.setInitConn( 5 );
pool.setMinConn( 5 );
pool.setMaxConn( 250 );
pool.setMaxIdle( 1000 * 60 * 60 * 6
```

Once the parameters have been configured, initialize the connection pool:

```
pool.initialize();
```

The pool, and the connection to your memcached instances should now be ready to use.

To set the hashing algorithm used to select the server used when storing a given key you can use pool.setHashingAlg():

```
pool.setHashingAlg( SockIOPool.NEW_COMPAT_HASH );
```

Valid values are NEW_COMPAT_HASH, OLD_COMPAT_HASH and NATIVE_HASH are also basic modula hashing algorithms. For a consistent hashing algorithm, use CONSISTENT_HASH. These constants are equivalent to the corresponding hash settings within libmemcached.

Java com.danga.MemCached Method	Equivalent to
get()	Generic get()
getMulti(keys)	Get the values of multiple keys, returning the information as Hash map using java.lang.String for the keys and java.lang.Object for the corresponding values.
set()	Generic set()
add()	Generic add()
replace()	Generic replace()
delete()	Generic delete()
incr()	Generic incr()
decr()	Generic decr()

3.7. Using the MySQL memcached UDFs

The memcached MySQL User Defined Functions (UDFs) enable you to set and retrieve objects from within MySQL 5.0 or greater.

To install the MySQL memcached UDFs, download the UDF package from http://libmemcached.org/. Unpack the package and run configure to configure the build process. When running configure, use the --with-mysql option and specify the location of the mysql_config command.

```
shell> tar zxf memcached_functions_mysql-0.5.tar.gz
shell> cd memcached_functions_mysql-0.5
shell> ./configure --with-mysql-config=/usr/local/mysql/bin/mysql_config
```

Now build and install the functions:

```
shell> make
shell> make install
```

Copy the MySQL memcached UDFs into your MySQL plugins directory:

```
shell> cp /usr/local/lib/libmemcached_functions_mysql* /usr/local/mysql/lib/mysql/plugins/
```

The plugin directory is given by the value of the plugin_dir system variable. For more information, see Compiling and Installing User-Defined Functions.

Once installed, you must initialize the function within MySQL using CREATE and specifying the return value and library. For example, to add the memc_get() function:

```
mysql> CREATE FUNCTION memc_get RETURNS STRING SONAME "libmemcached_functions_mysql.so";
```

Repeat this process for each function to provide access to within MySQL. Once you have created the association, the information is retained, even over restarts of the MySQL server. You can simplify the process by using the SQL script provided in the memcached UDFs package:

```
shell> mysql <sql/install_functions.sql
```

Alternatively, if you have Perl installed, then you can use the supplied Perl script, which checks for the existence of each function and creates the function/library association if it is not already defined:

```
shell> utils/install.pl --silent
```

The --silent option installs everything automatically. Without this option, the script asks whether to install each of the available functions.

The interface remains consistent with the other APIs and interfaces. To set up a list of servers, use the memc_servers_set() function, which accepts a single string containing and comma-separated list of servers:

```
mysql> SELECT memc_servers_set('192.168.0.1:11211,192.168.0.2:11211');
```

> **Note**
>
> The list of servers used by the memcached UDFs is not persistent over restarts of the MySQL server. If the MySQL server fails, then you must re-set the list of memcached servers.

To set a value, use memc_set:

```
mysql> SELECT memc_set('myid', 'myvalue');
```

To retrieve a stored value:

```
mysql> SELECT memc_get('myid');
```

The list of functions supported by the UDFs, in relation to the standard protocol functions, is shown in the following table.

MySQL memcached UDF Function	Equivalent to
memc_get()	Generic get()
memc_get_by_key(master_key, key, value)	Like the generic get(), but uses the supplied master key to select the server to use.
memc_set()	Generic set()
memc_set_by_key(master_key, key, value)	Like the generic set(), but uses the supplied master key to select the server to use.
memc_add()	Generic add()
memc_add_by_key(master_key, key, value)	Like the generic add(), but uses the supplied master key to select the server to use.
memc_replace()	Generic replace()
memc_replace_by_key(master_key, key, value)	Like the generic replace(), but uses the supplied master key to select the server to use.
memc_prepend(key, value)	Prepend the specified value to the current value of the specified key.
memc_prepend_by_key(master_key, key, value)	Prepend the specified value to the current value of the specified key, but uses the supplied master key to select the server to use.
memc_append(key, value)	Append the specified value to the current value of the specified key.
memc_append_by_key(master_key, key, value)	Append the specified value to the current value of the specified key, but uses the supplied master key to select the server to use.
memc_delete()	Generic delete()
memc_delete_by_key(master_key, key, value)	Like the generic delete(), but uses the supplied master key to select the server to use.
memc_increment()	Generic incr()
memc_decrement()	Generic decr()

The respective *_by_key() functions are useful to store a specific value into a specific memcached server, possibly based on a differently calculated or constructed key.

The memcached UDFs include some additional functions:

- memc_server_count()

  Returns a count of the number of servers in the list of registered servers.

- memc_servers_set_behavior(behavior_type, value), memc_set_behavior(behavior_type, value)

Set behaviors for the list of servers. These behaviors are identical to those provided by the libmemcached library. For more information on libmemcached behaviors, see Section 3.1, "Using libmemcached".

You can use the behavior name as the behavior_type:

```
mysql> SELECT memc_servers_behavior_set("MEMCACHED_BEHAVIOR_KETAMA",1);
```

- memc_servers_behavior_get(behavior_type), memc_get_behavior(behavior_type, value)

  Returns the value for a given behavior.

- memc_list_behaviors()

  Returns a list of the known behaviors.

- memc_list_hash_types()

  Returns a list of the supported key-hashing algorithms.

- memc_list_distribution_types()

  Returns a list of the supported distribution types to be used when selecting a server to use when storing a particular key.

- memc_libmemcached_version()

  Returns the version of the libmemcached library.

- memc_stats()

  Returns the general statistics information from the server.

3.8. memcached Protocol

Communicating with a memcached server can be achieved through either the TCP or UDP protocols. When using the TCP protocol you can use a simple text based interface for the exchange of information.

3.8.1. Using the TCP text protocol

When communicating with memcached you can connect to the server using the port configured for the server. You can open a connection with the server without requiring authorization or login. As soon as you have connected, you can start to send commands to the server. When you have finished, you can terminate the connection without sending any specific disconnection command. Clients are encouraged to keep their connections open to decrease latency and improve performance.

Data is sent to the memcached server in two forms:

- Text lines, which are used to send commands to the server, and receive responses from the server.
- Unstructured data, which is used to receive or send the value information for a given key. Data is returned to the client in exactly the format it was provided.

Both text lines (commands and responses) and unstructured data are always terminated with the string \r\n. Because the data being stored may contain this sequence, the length of the data (returned by the client before the unstructured data is transmitted should be used to determine the end of the data.

Commands to the server are structured according to their operation:

- **Storage commands**: set, add, replace, append, prepend, cas

  Storage commands to the server take the form:

```
command key [flags] [exptime] length [noreply]
```

Or when using compare and swap (cas):

```
cas key [flags] [exptime] length [casunique] [noreply]
```

Where:

- command: The command name.
  - set: Store value against key
  - add: Store this value against key if the key does not already exist
  - replace: Store this value against key if the key already exists
  - append: Append the supplied value to the end of the value for the specified key. The flags and exptime arguments should not be used.
  - prepend: Append value currently in the cache to the end of the supplied value for the specified key. The flags and exptime arguments should not be used.
  - cas: Set the specified key to the supplied value, only if the supplied casunique matches. This is effectively the equivalent of change the information if nobody has updated it since I last fetched it.
- key: The key. All data is stored using a the specific key. The key cannot contain control characters or whitespace, and can be up to 250 characters in size.
- flags: The flags for the operation (as an integer). Flags in memcached are transparent. The memcached server ignores the contents of the flags. They can be used by the client to indicate any type of information. In memcached 1.2.0 and lower the value is a 16-bit integer value. In memcached 1.2.1 and higher the value is a 32-bit integer.
- exptime: The expiry time, or zero for no expiry.
- length: The length of the supplied value block in bytes, excluding the terminating \r\n characters.
- casunique: A unique 64-bit value of an existing entry. This is used to compare against the existing value. Use the value returned by the gets command when issuing cas updates.
- noreply: Tells the server not to reply to the command.

For example, to store the value abcdef into the key xyzkey, you would use:

```
set xyzkey 0 0 6\r\nabcdef\r\n
```

The return value from the server is one line, specifying the status or error information. For more information, see Table 3.2, “memcached Protocol Responses”.

- **Retrieval commands**: get, gets

Retrieval commands take the form:

```
get key1 [key2 .... keyn]
gets key1 [key2 ... keyn]
```

You can supply multiple keys to the commands, with each requested key separated by whitespace.

The server responds with an information line of the form:

```
VALUE key flags bytes [casunique]
```

Where:

- key: The key name.
- flags: The value of the flag integer supplied to the memcached server when the value was stored.
- bytes: The size (excluding the terminating \r\n character sequence) of the stored value.
- casunique: The unique 64-bit integer that identifies the item.

The information line is immediately followed by the value data block. For example:

```
get xyzkey\r\n
VALUE xyzkey 0 6\r\n
abcdef\r\n
```

If you have requested multiple keys, an information line and data block is returned for each key found. If a requested key does not exist in the cache, no information is returned.

- **Delete commands**: delete

  Deletion commands take the form:

  ```
  delete key [time] [noreply]
  ```

  Where:

  - key: The key name.
  - time: The time in seconds (or a specific Unix time) for which the client wishes the server to refuse add or replace commands on this key. All add, replace, get, and gets commands fail during this period. set operations succeed. After this period, the key is deleted permanently and all commands are accepted.

    If not supplied, the value is assumed to be zero (delete immediately).
  - noreply: Tells the server not to reply to the command.

  Responses to the command are either DELETED to indicate that the key was successfully removed, or NOT_FOUND to indicate that the specified key could not be found.

- **Increment/Decrement**: incr, decr

  The increment and decrement commands change the value of a key within the server without performing a separate get/set sequence. The operations assume that the currently stored value is a 64-bit integer. If the stored value is not a 64-bit integer, then the value is assumed to be zero before the increment or decrement operation is applied.

  Increment and decrement commands take the form:

  ```
  incr key value [noreply]
  decr key value [noreply]
  ```

  Where:

  - key: The key name.
  - value: An integer to be used as the increment or decrement value.
  - noreply: Tells the server not to reply to the command.

  The response is:

  - NOT_FOUND: The specified key could not be located.
  - value: The new value of the specified key.

  Values are assumed to be unsigned. For decr operations the value is never decremented below 0. For incr operations, the value wraps around the 64-bit maximum.

- **Statistics commands**: `stats`

  The `stats` command provides detailed statistical information about the current status of the `memcached` instance and the data it is storing.

  Statistics commands take the form:

  ```
  STAT [name] [value]
  ```

  Where:

  - `name`: The optional name of the statistics to return. If not specified, the general statistics are returned.
  - `value`: A specific value to be used when performing certain statistics operations.

  The return value is a list of statistics data, formatted as follows:

  ```
  STAT name value
  ```

  The statistics are terminated with a single line, `END`.

  For more information, see Chapter 4, *Getting `memcached` Statistics*.

For reference, a list of the different commands supported and their formats is provided below.

Table 3.1. `memcached` Command Reference

Command	Command Formats
`set`	`set key flags exptime length`, `set key flags exptime length noreply`
`add`	`add key flags exptime length`, `add key flags exptime length noreply`
`replace`	`replace key flags exptime length`, `replace key flags exptime length noreply`
`append`	`append key length`, `append key length noreply`
`prepend`	`prepend key length`, `prepend key length noreply`
`cas`	`cas key flags exptime length casunique`, `cas key flags exptime length casunique noreply`
`get`	`get key1 [key2 ... keyn]`
`gets`	
`delete`	`delete key`, `delete key noreply`, `delete key expiry`, `delete key expiry noreply`
`incr`	`incr key`, `incr key noreply`, `incr key value`, `incr key value noreply`
`decr`	`decr key`, `decr key noreply`, `decr key value`, `decr key value noreply`
`stat`	`stat`, `stat name`, `stat name value`

When sending a command to the server, the response from the server is one of the settings in the following table. All response values from the server are terminated by `\r\n`:

Table 3.2. `memcached` Protocol Responses

String	Description
`STORED`	Value has successfully been stored.
`NOT_STORED`	The value was not stored, but not because of an error. For commands where you are adding a or updating a value if it exists (such as `add` and `replace`), or where the item has already been set to be deleted.
`EXISTS`	When using a `cas` command, the item you are trying to store already exists and has been modified

String	Description
	since you last checked it.
NOT_FOUND	The item you are trying to store, update or delete does not exist or has already been deleted.
ERROR	You submitted a nonexistent command name.
CLIENT_ERROR error-string	There was an error in the input line, the detail is contained in errorstring.
SERVER_ERROR error-string	There was an error in the server that prevents it from returning the information. In extreme conditions, the server may disconnect the client after this error occurs.
VALUE keys flags length	The requested key has been found, and the stored key, flags and data block are returned, of the specified length.
DELETED	The requested key was deleted from the server.
STAT name value	A line of statistics data.
END	The end of the statistics data.

Chapter 4. Getting memcached Statistics

The memcached system has a built in statistics system that collects information about the data being stored into the cache, cache hit ratios, and detailed information on the memory usage and distribution of information through the slab allocation used to store individual items. Statistics are provided at both a basic level that provide the core statistics, and more specific statistics for specific areas of the memcached server.

This information can prove be very useful to ensure that you are getting the correct level of cache and memory usage, and that your slab allocation and configuration properties are set at an optimal level.

The stats interface is available through the standard memcached protocol, so the reports can be accessed by using telnet to connect to the memcached. The supplied memcached-tool includes support for obtaining the Section 4.2, "memcached Slabs Statistics" and Section 4.1, "memcached General Statistics" information. For more information, see Section 4.6, "Using memcached-tool".

Alternatively, most of the language API interfaces provide a function for obtaining the statistics from the server.

For example, to get the basic stats using telnet:

```
shell> telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
stats
STAT pid 23599
STAT uptime 675
STAT time 1211439587
STAT version 1.2.5
STAT pointer_size 32
STAT rusage_user 1.404992
STAT rusage_system 4.694685
STAT curr_items 32
STAT total_items 56361
STAT bytes 2642
STAT curr_connections 53
STAT total_connections 438
STAT connection_structures 55
STAT cmd_get 113482
STAT cmd_set 80519
STAT get_hits 78926
STAT get_misses 34556
STAT evictions 0
STAT bytes_read 6379783
STAT bytes_written 4860179
STAT limit_maxbytes 67108864
STAT threads 1
END
```

When using Perl and the Cache::Memcached module, the stats() function returns information about all the servers currently configured in the connection object, and total statistics for all the memcached servers as a whole.

For example, the following Perl script obtains the stats and dumps the hash reference that is returned:

```
use Cache::Memcached;
use Data::Dumper;
my $memc = new Cache::Memcached;
$memc->set_servers(\@ARGV);
print Dumper($memc->stats());
```

When executed on the same memcached as used in the Telnet example above we get a hash reference with the host by host and total statistics:

```
$VAR1 = {
    'hosts' => {
          'localhost:11211' => {
                    'misc' => {
                          'bytes' => '2421',
                          'curr_connections' => '3',
                          'connection_structures' => '56',
                          'pointer_size' => '32',
                          'time' => '1211440166',
                          'total_items' => '410956',
                          'cmd_set' => '588167',
                          'bytes_written' => '35715151',
                          'evictions' => '0',
                          'curr_items' => '31',
                          'pid' => '23599',
```

```
                          'limit_maxbytes' => '67108864',
                          'uptime' => '1254',
                          'rusage_user' => '9.857805',
                          'cmd_get' => '838451',
                          'rusage_system' => '34.096988',
                          'version' => '1.2.5',
                          'get_hits' => '581511',
                          'bytes_read' => '46665716',
                          'threads' => '1',
                          'total_connections' => '3104',
                          'get_misses' => '256940'
                        },
                    'sizes' => {
                          '128' => '16',
                          '64' => '15'
                        }
                  }
      },
  'self' => {},
  'total' => {
        'cmd_get' => 838451,
        'bytes' => 2421,
        'get_hits' => 581511,
        'connection_structures' => 56,
        'bytes_read' => 46665716,
        'total_items' => 410956,
        'total_connections' => 3104,
        'cmd_set' => 588167,
        'bytes_written' => 35715151,
        'curr_items' => 31,
        'get_misses' => 256940
      }
    };
```

The statistics are divided up into a number of distinct sections, and then can be requested by adding the type to the `stats` command. Each statistics output is covered in more detail in the following sections.

- General statistics, see Section 4.1, "`memcached` General Statistics".
- Slab statistics (`slabs`), see Section 4.2, "`memcached` Slabs Statistics".
- Item statistics (`items`), see Section 4.3, "`memcached` Item Statistics".
- Size statistics (`sizes`), see Section 4.4, "`memcached` Size Statistics".
- Detailed status (`detail`), see Section 4.5, "`memcached` Detail Statistics".

4.1. `memcached` General Statistics

The output of the general statistics provides an overview of the performance and use of the `memcached` instance. The statistics returned by the command and their meaning is shown in the following table.

The following terms are used to define the value type for each statistics value:

- `32u`: 32-bit unsigned integer
- `64u`: 64-bit unsigned integer
- `32u32u`: Two 32-bit unsigned integers separated by a colon
- `String`: Character string

Statistic	Data type	Description	Version
pid	32u	Process ID of the `memcached` instance.	
uptime	32u	Uptime (in seconds) for this `memcached` instance.	
time	32u	Current time (as epoch).	
version	string	Version string of this instance.	

Statistic	Data type	Description	Version
pointer_size	string	Size of pointers for this host specified in bits (32 or 64).	
rusage_user	32u:32u	Total user time for this instance (seconds:microseconds).	
rusage_system	32u:32u	Total system time for this instance (seconds:microseconds).	
curr_items	32u	Current number of items stored by this instance.	
total_items	32u	Total number of items stored during the life of this instance.	
bytes	64u	Current number of bytes used by this server to store items.	
curr_connections	32u	Current number of open connections.	
total_connections	32u	Total number of connections opened since the server started running.	
connection_structures	32u	Number of connection structures allocated by the server.	
cmd_get	64u	Total number of retrieval requests (get operations).	
cmd_set	64u	Total number of storage requests (set operations).	
get_hits	64u	Number of keys that have been requested and found present.	
get_misses	64u	Number of items that have been requested and not found.	
delete_hits	64u	Number of keys that have been deleted and found present.	1.3.x
delete_misses	64u	Number of items that have been delete and not found.	1.3.x
incr_hits	64u	Number of keys that have been incremented and found present.	1.3.x
incr_misses	64u	Number of items that have been incremented and not found.	1.3.x
decr_hits	64u	Number of keys that have been decremented and found present.	1.3.x
decr_misses	64u	Number of items that have been decremented and not found.	1.3.x
cas_hits	64u	Number of keys that have been compared and swapped and found present.	1.3.x
cas_misses	64u	Number of items that have been compared and swapped and not found.	1.3.x
cas_badvalue	64u	Number of keys that have been compared and swapped, but the comparison (original) value did not match the supplied value.	1.3.x
evictions	64u	Number of valid items removed from cache to free memory for new items.	
bytes_read	64u	Total number of bytes read by this server from network.	
bytes_written	64u	Total number of bytes sent by this server to network.	
limit_maxbytes	32u	Number of bytes this server is permitted to use for storage.	
threads	32u	Number of worker threads requested.	
conn_yields	64u	Number of yields for connections (related to the -R option).	1.4.0

The most useful statistics from those given here are the number of cache hits, misses, and evictions.

A large number of get_misses may just be an indication that the cache is still being populated with information. The number should, over time, decrease in comparison to the number of cache get_hits. If, however, you have a large number of cache misses compared to cache hits after an extended period of execution, it may be an indication that the size of the cache is too small and you either need to increase the total memory size, or increase the number of the memcached instances to improve the hit ratio.

A large number of evictions from the cache, particularly in comparison to the number of items stored is a sign that your cache is too small to hold the amount of information that you regularly want to keep cached. Instead of items being retained in the cache, items are being evicted to make way for new items keeping the turnover of items in the cache high, reducing the efficiency of the cache.

4.2. memcached Slabs Statistics

To get the slabs statistics, use the stats slabs command, or the API equivalent.

The slab statistics provide you with information about the slabs that have created and allocated for storing information within the cache. You get information both on each individual slab-class and total statistics for the whole slab.

```
STAT 1:chunk_size 104
STAT 1:chunks_per_page 10082
STAT 1:total_pages 1
STAT 1:total_chunks 10082
STAT 1:used_chunks 10081
STAT 1:free_chunks 1
STAT 1:free_chunks_end 10079
STAT 9:chunk_size 696
STAT 9:chunks_per_page 1506
STAT 9:total_pages 63
STAT 9:total_chunks 94878
STAT 9:used_chunks 94878
STAT 9:free_chunks 0
STAT 9:free_chunks_end 0
STAT active_slabs 2
STAT total_malloced 67083616
END
```

Individual stats for each slab class are prefixed with the slab ID. A unique ID is given to each allocated slab from the smallest size up to the largest. The prefix number indicates the slab class number in relation to the calculated chunk from the specified growth factor. Hence in the example, 1 is the first chunk size and 9 is the 9th chunk allocated size.

The different parameters returned for each chunk size and the totals are shown in the following table.

Statistic	Description	Version
chunk_size	Space allocated to each chunk within this slab class.	
chunks_per_page	Number of chunks within a single page for this slab class.	
total_pages	Number of pages allocated to this slab class.	
total_chunks	Number of chunks allocated to the slab class.	
used_chunks	Number of chunks allocated to an item..	
free_chunks	Number of chunks not yet allocated to items.	
free_chunks_end	Number of free chunks at the end of the last allocated page.	
get_hits	Number of get hits to this chunk	1.3.x
cmd_set	Number of set commands on this chunk	1.3.x
delete_hits	Number of delete hits to this chunk	1.3.x
incr_hits	Number of increment hits to this chunk	1.3.x
decr_hits	Number of decrement hits to this chunk	1.3.x
cas_hits	Number of CAS hits to this chunk	1.3.x
cas_badval	Number of CAS hits on this chunk where the existing value did not match	1.3.x
mem_requested	The true amount of memory of memory requested within this chunk	1.4.1

The following additional statistics cover the information for the entire server, rather than on a chunk by chunk basis:

Statistic	Description	Version
active_slabs	Total number of slab classes allocated.	
total_malloced	Total amount of memory allocated to slab pages.	

The key values in the slab statistics are the `chunk_size`, and the corresponding `total_chunks` and `used_chunks` parameters. These given an indication of the size usage of the chunks within the system. Remember that one key/value pair is placed into a chunk of

a suitable size.

From these stats, you can get an idea of your size and chunk allocation and distribution. If you store many items with a number of largely different sizes, consider adjusting the chunk size growth factor to increase in larger steps to prevent chunk and memory wastage. A good indication of a bad growth factor is a high number of different slab classes, but with relatively few chunks actually in use within each slab. Increasing the growth factor creates fewer slab classes and therefore makes better use of the allocated pages.

4.3. memcached Item Statistics

To get the items statistics, use the stats items command, or the API equivalent.

The items statistics give information about the individual items allocated within a given slab class.

```
STAT items:2:number 1
STAT items:2:age 452
STAT items:2:evicted 0
STAT items:2:evicted_nonzero 0
STAT items:2:evicted_time 2
STAT items:2:outofmemory 0
STAT items:2:tailrepairs 0
...
STAT items:27:number 1
STAT items:27:age 452
STAT items:27:evicted 0
STAT items:27:evicted_nonzero 0
STAT items:27:evicted_time 2
STAT items:27:outofmemory 0
STAT items:27:tailrepairs 0
```

The prefix number against each statistics relates to the corresponding chunk size, as returned by the stats slabs statistics. The result is a display of the number of items stored within each chunk within each slab size, and specific statistics about their age, eviction counts, and out of memory counts. A summary of the statistics is given in the following table.

Statistic	Description	
number	The number of items currently stored in this slab class.	
age	The age of the oldest item within the slab class, in seconds.	
evicted	The number of items evicted to make way for new entries.	
evicted_time	The time of the last evicted entry	
evicted_nonzero	The time of the last evicted non-zero entry	1.4.0
outofmemory	The number of items for this slab class that have triggered an out of memory error (only value when the -M command line option is in effect).	
tailrepairs	Number of times the entries for a particular ID need repairing	

Item level statistics can be used to determine how many items are stored within a given slab and their freshness and recycle rate. You can use this to help identify whether there are certain slab classes that are triggering a much larger number of evictions that others.

4.4. memcached Size Statistics

To get size statistics, use the stats sizes command, or the API equivalent.

The size statistics provide information about the sizes and number of items of each size within the cache. The information is returned as two columns, the first column is the size of the item (rounded up to the nearest 32 byte boundary), and the second column is the count of the number of items of that size within the cache:

```
96 35
128 38
160 807
192 804
224 410
256 222
288 83
320 39
352 53
384 33
416 64
448 51
480 30
```

```
512 54
544 39
576 10065
```

> **Caution**
>
> Running this statistic locks up your cache as each item is read from the cache and its size calculated. On a large cache, this may take some time and prevent any set or get operations until the process completes.

The item size statistics are useful only to determine the sizes of the objects you are storing. Since the actual memory allocation is relevant only in terms of the chunk size and page size, the information is only useful during a careful debugging or diagnostic session.

4.5. memcached Detail Statistics

For memcached 1.3.x and higher, you can enable and obtain detailed statistics about the get, set, and del operations on theindividual keys stored in the cache, and determine whether the attempts hit (found) a particular key. These operations are only recorded while the detailed stats analysis is turned on.

To enable detailed statistics, you must send the stats detail on command to the memcached server:

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to tiger.
Escape character is '^]'.stats detail on
OK
```

Individual statistics are recorded for every get, set and del operation on a key, including keys that are not currently stored in the server. For example, if an attempt is made to obtain the value of key abckey and it does not exist, the get operating on the specified key are recorded while detailed statistics are in effect, even if the key is not currently stored. The hits, that is, the number of get or del operations for a key that exists in the server are also counted.

To turn detailed statistics off, send the stats detail off command to the memcached server:

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to tiger.
Escape character is '^]'.stats detail on
OK
```

To obtain the detailed statistics recorded during the process, send the stats detail dump command to the memcached server:

```
stats detail dump
PREFIX hykkey get 0 hit 0 set 1 del 0
PREFIX xyzkey get 0 hit 0 set 1 del 0
PREFIX yukkey get 1 hit 0 set 0 del 0
PREFIX abckey get 3 hit 3 set 1 del 0
END
```

You can use the detailed statistics information to determine whether your memcached clients are using a large number of keys that do not exist in the server by comparing the hit and get or del counts. Because the information is recorded by key, you can also determine whether the failures or operations are clustered around specific keys.

4.6. Using memcached-tool

The memcached-tool, located within the scripts directory within the memcached source directory. The tool provides convenient access to some reports and statistics from any memcached instance.

The basic format of the command is:

```
shell> ./memcached-tool hostname:port [command]
```

The default output produces a list of the slab allocations and usage. For example:

```
shell> memcached-tool localhost:11211 display
  #  Item_Size  Max_age   Pages   Count   Full?  Evicted Evict_Time OOM
  1      80B        93s       1      20      no        0        0    0
  2     104B        93s       1      16      no        0        0    0
```

```
  3     136B      1335s       1      28      no        0        0    0
  4     176B      1335s       1      24      no        0        0    0
  5     224B      1335s       1      32      no        0        0    0
  6     280B      1335s       1      34      no        0        0    0
  7     352B      1335s       1      36      no        0        0    0
  8     440B      1335s       1      46      no        0        0    0
  9     552B      1335s       1      58      no        0        0    0
 10     696B      1335s       1      66      no        0        0    0
 11     872B      1335s       1      89      no        0        0    0
 12     1.1K      1335s       1     112      no        0        0    0
 13     1.3K      1335s       1     145      no        0        0    0
 14     1.7K      1335s       1     123      no        0        0    0
 15     2.1K      1335s       1     198      no        0        0    0
 16     2.6K      1335s       1     199      no        0        0    0
 17     3.3K      1335s       1     229      no        0        0    0
 18     4.1K      1335s       1     248     yes       36        2    0
 19     5.2K      1335s       2     328      no        0        0    0
 20     6.4K      1335s       2     316     yes      387        1    0
 21     8.1K      1335s       3     381     yes      492        1    0
 22    10.1K      1335s       3     303     yes      598        2    0
 23    12.6K      1335s       5     405     yes      605        1    0
 24    15.8K      1335s       6     384     yes      766        2    0
 25    19.7K      1335s       7     357     yes      908      170    0
 26    24.6K      1336s       7     287     yes     1012        1    0
 27    30.8K      1336s       7     231     yes     1193      169    0
 28    38.5K      1336s       4     104     yes     1323      169    0
 29    48.1K      1336s       1      21     yes     1287        1    0
 30    60.2K      1336s       1      17     yes     1093      169    0
 31    75.2K      1337s       1      13     yes      713      168    0
 32    94.0K      1337s       1      10     yes      278      168    0
 33   117.5K      1336s       1       3      no        0        0    0
```

This output is the same if you specify the `command` as `display`:

```
shell> memcached-tool localhost:11211 display
  #  Item_Size  Max_age   Pages   Count   Full?  Evicted Evict_Time OOM
  1      80B        93s       1      20      no        0        0    0
  2     104B        93s       1      16      no        0        0    0
...
```

The output shows a summarized version of the output from the `slabs` statistics. The columns provided in the output are shown below:

- `#`: The slab number
- `Item_Size`: The size of the slab
- `Max_age`: The age of the oldest item in the slab
- `Pages`: The number of pages allocated to the slab
- `Count`: The number of items in this slab
- `Full?`: Whether the slab is fully populated
- `Evicted`: The number of objects evicted from this slab
- `Evict_Time`: The time (in seconds) since the last eviction
- `OOM`: The number of items that have triggered an out of memory error

You can also obtain a dump of the general statistics for the server using the `stats` command:

```
shell> memcached-tool localhost:11211 stats
#localhost:11211   Field       Value
         accepting_conns           1
                   bytes         162
              bytes_read         485
           bytes_written        6820
              cas_badval           0
                cas_hits           0
              cas_misses           0
               cmd_flush           0
                 cmd_get           4
                 cmd_set           2
             conn_yields           0
   connection_structures          11
```

```
        curr_connections          10
              curr_items           2
               decr_hits           0
             decr_misses           1
             delete_hits           0
          delete_misses           0
              evictions           0
               get_hits           4
             get_misses           0
               incr_hits           0
             incr_misses           2
        limit_maxbytes    67108864
listen_disabled_num           0
                   pid       12981
         pointer_size          32
        rusage_system    0.013911
          rusage_user    0.011876
              threads           4
                 time  1255518565
   total_connections          20
         total_items           2
              uptime         880
             version       1.4.2
```

The memcached-tool provides

Chapter 5. memcached FAQ

Questions

- 5.28: What speed trade offs is there between memcached vs MySQL Query Cache? Where you check memcached, and get data from MySQL and put it in memcached or just make a query and results are put into MySQL Query Cache.
- 5.29: Can we implement different types of memcached as different nodes in the same server - so can there be deterministic and non deterministic in the same server?
- 5.30: What are best practices for testing an implementation, to ensure that it is an improvement over the MySQL query cache, and to measure the impact of memcached configuration changes? And would you recommend keeping the configuration very simple to start?

Questions and Answers

5.1: Can MySQL actually trigger/store the changed data to memcached?

Yes. You can use the MySQL UDFs for memcached and either write statements that directly set the values in the memcached server, or use triggers or stored procedures to do it for you. For more information, see Section 3.7, "Using the MySQL memcached UDFs"

5.2: Can memcached be run on a Windows environment?

No. Currently memcached is available only on the Unix/Linux platform. There is an unofficial port available, see http://www.codeplex.com/memcachedproviders.

5.3: Does the -L flag automatically sense how much memory is being used by other memcached?

No. There is no communication or sharing of information between memcached instances.

5.4: What is the max size of an object you can store in memcache and is that configurable?

The default maximum object size is 1MB. In memcached 1.4.2 and later you can change the maximum size of an object using the -I command line option.

For versions before this, to increase this size, you have to re-compile memcached. You can modify the value of the POWER_BLOCK within the slabs.c file within the source.

In memcached 1.4.2 and higher you can configure the maximum supported object size by using the -I command-line option. For example, to increase the maximum object size to 5MB:

```
$ memcached -I 5m
```

5.5: Is it true memcached will be much more effective with db-read-intensive applications than with db-write-intensive applications?

Yes. memcached plays no role in database writes, it is a method of caching data already read from the database in RAM.

5.6: memcached is fast - is there any overhead in not using persistent connections? If persistent is always recommended, what are the downsides (for example, locking up)?

If you don't use persistent connections when communicating with memcached then there will be a small increase in the latency of opening the connection each time. The effect is comparable to use nonpersistent connections with MySQL.

In general, the chance of locking or other issues with persistent connections is minimal, because there is very little locking within memcached. If there is a problem then eventually your request will timeout and return no result so your application will need to load from MySQL again.

5.7: How does an event such as a crash of one of the memcached servers handled by the memcached client?

There is no automatic handling of this. If your client fails to get a response from a server then it should fall back to loading the data from the MySQL database.

The client APIs all provide the ability to add and remove memcached instances on the fly. If within your application you notice that memcached server is no longer responding, your can remove the server from the list of servers, and keys will automatically be redistributed to another memcached server in the list. If retaining the cache content on all your servers is important, make sure you use an API that supports a consistent hashing algorithm. For more information, see Section 2.4, "memcached Hashing/Distribution Types".

5.8: What's a recommended hardware config for a memcached server? Linux or Windows?

memcached is only available on Unix/Linux, so using a Windows machine is not an option. Outside of this, memcached has a very low processing overhead. All that is required is spare physical RAM capacity. The point is not that you should necessarily deploy a dedicated memcached server. If you have web, application, or database servers that have spare RAM capacity, then use them with memcached.

If you want to build and deploy a dedicated memcached servers, then you use a relatively low-power CPU, lots of RAM and one or more Gigabit Ethernet interfaces.

5.9: Doing a direct telnet to the memcached port, is that just for that one machine, or does it magically apply across all nodes?

Just one. There is no communication between different instances of memcached, even if each instance is running on the same machine.

5.10: Is memcached more effective for video and audio as opposed to textual read/writes

memcached doesn't care what information you are storing. To memcached, any value you store is just a stream of data. Remember, though, that the maximum size of an object you can store in memcached is 1MB, but can be configured to be larger by using the -I option in memcached 1.4.2 and later, or by modifying the source in versions before 1.4.2. If you plan on using memcached with audio and video content you will probably want to increase the maximum object size. Also remember that memcached is a solution for caching information for reading. It shouldn't be used for writes, except when updating the information in the cache.

5.11: If you log a complex class (with methods that do calculation etc) will the get from Memcache re-create the class on the way out?

In general, yes. If the serialization method within the API/language that you are using supports it, then methods and other information will be stored and retrieved.

5.12: If I have an object larger then a MB, do I have to manually split it or can I configure memcached to handle larger objects?

You would have to manually split it. memcached is very simple, you give it a key and some data, it tries to cache it in RAM. If you try to store more than the default maximum size, the value is just truncated for speed reasons.

5.13: Can memcached work with ASPX?

There are ports and interfaces for many languages and environments. ASPX relies on an underlying language such as C# or VisualBasic, and if you are using ASP.NET then there is a C# memcached library. For more information, see .

5.14: Are there any, or are there any plans to introduce, a framework to hide the interaction of memcached from the application; that is, within hibernate?

There are lots of projects working with memcached. There is a Google Code implementation of Hibernate and memcached working together. See http://code.google.com/p/hibernate-memcached/.

5.15: So the responsibility lies with the application to populate and get records from the database as opposed to being a transparent cache layer for the db?

Yes. You load the data from the database and write it into the cache provided by memcached. Using memcached as a simple database row cache, however, is probably inefficient. The best way to use memcached is to load all of the information from the database relating to a particular object, and then cache the entire object. For example, in a blogging environment, you might load the blog, associated comments, categories and so on, and then cache all of the information relating to that blog post. The reading of the data from the database will require multiple SQL statements and probably multiple rows of data to complete, which is time consuming. Loading the entire blog post and the associated information from memcached is just one operation and doesn't involve using the disk or parsing the SQL statement.

5.16: How does memcached compare to nCache?

The main benefit of memcached is that is very easy to deploy and works with a wide range of languages and environments, including .NET, Java, Perl, Python, PHP, even MySQL. memcached is also very lightweight in terms of systems and requirements, and you can easily add as many or as few memcached servers as you need without changing the individual configuration. memcached does require additional modifications to the application to take advantage of functionality such as multiple memcached servers.

5.17: We are caching XML by serialising using saveXML(), because PHP cannot serialize DOM objects; Some of the XML is variable and is modified per-request. Do you recommend caching then using XPath, or is it better to rebuild the DOM from separate node-groups?

You would need to test your application using the different methods to determine this information. You may find that the default serialization within PHP may allow you to store DOM objects directly into the cache.

5.18: How easy is it to introduce memcached to an existing enterprise application instead of inclusion at project design?

In general, it is very easy. In many languages and environments the changes to the application will be just a few lines, first to attempt to read from the cache when loading data and then fall back to the old method, and to update the cache with information once the data has been read.

memcached is designed to be deployed very easily, and you shouldn't require significant architectural changes to your application to use memcached.

5.19: Do the memcache UDFs work under 5.1?

Yes.

5.20: Is the data inside of memcached secure?

No, there is no security required to access or update the information within a memcached instance, which means that anybody with access to the machine has the ability to read, view and potentially update the information. If you want to keep the data secure, you can encrypt and decrypt the information before storing it. If you want to restrict the users capable of connecting to the server, your only choice is to either disable network access, or use IPTables or similar to restrict access to the memcached ports to a select set of hosts.

5.21: Is memcached typically a better solution for improving speed than MySQL Cluster and\or MySQL Proxy?

Both MySQL Cluster and MySQL Proxy still require access to the underlying database to retrieve the information. This implies both a parsing overhead for the statement and, often, disk based access to retrieve the data you have selected.

The advantage of memcached is that you can store entire objects or groups of information that may require multiple SQL statements to obtain. Restoring the result of 20 SQL statements formatted into a structure that your application can use directly without requiring any additional processing is always going to be faster than building that structure by loading the rows from a database.

5.22: File socket support for memcached from the localhost use to the local memcached server?

You can use the -s option to memcached to specify the location of a file socket. This automatically disables network support.

5.23: How expensive is it to establish a memcache connection? Should those connections be pooled?

Opening the connection is relatively inexpensive, because there is no security, authentication or other handshake taking place before you can start sending requests and getting results. Most APIs support a persistent connection to a memcached instance to reduce the latency. Connection pooling would depend on the API you are using, but if you are communicating directly over TCP/IP, then connection pooling would provide some small performance benefit.

5.24: What are the advantages of using UDFs when the get/sets are manageable from within the client code rather than the db?

Sometimes you want to be able to be able to update the information within memcached based on a generic database activity, rather than relying on your client code. For example, you may want to update status or counter information in memcached through the use of a trigger or stored procedure. For some situations and applications the existing use of a stored procedure for some operations means that updating the value in memcached from the database is easier than separately loading and communicating that data to the client just so the client can talk to memcached.

In other situations, when you are using a number of different clients and different APIs, you don't want to have to write (and maintain) the code required to update memcached in all the environments. Instead, you do this from within the database and the client never gets involved.

5.25: How will the data will be handled when the memcached server is down?

The behavior is entirely application dependent. Most applications will fall back to loading the data from the database (just as if they were updating the memcached) information. If you are using multiple memcached servers, you may also want to remove a server from the list to prevent the missing server affecting performance. This is because the client will still attempt to communicate the memcached that corresponds to the key you are trying to load.

5.26: How are auto-increment columns in the MySQL database coordinated across multiple instances of memcached?

They aren't. There is no relationship between MySQL and memcached unless your application (or, if you are using the MySQL UDFs for memcached, your database definition) creates one.

If you are storing information based on an auto-increment key into multiple instances of memcached then the information will only be stored on one of the memcached instances anyway. The client uses the key value to determine which memcached instance to store the information, it doesn't store the same information across all the instances, as that would be a waste of cache memory.

5.27: Is compression available?

Yes. Most of the client APIs support some sort of compression, and some even allow you to specify the threshold at which a value is deemed appropriate for compression during storage.

5.28: What speed trade offs is there between memcached vs MySQL Query Cache? Where you check memcached, and get data from MySQL and put it in memcached or just make a query and results are put into MySQL Query Cache.

In general, the time difference between getting data from the MySQL Query Cache and getting the exact same data from memcached is very small.

However, the benefit of memcached is that you can store any information, including the formatted and processed results of many queries into a single memcached key. Even if all the queries that you executed could be retrieved from the Query Cache without having to go to disk, you would still be running multiple queries (with network and other overhead) compared to just one for the memcached equivalent. If your application uses objects, or does any kind of processing on the information, with memcached you can store the post-processed version, so the data you load is immediately available to be used. With data loaded from the Query Cache, you would still have to do that processing.

In addition to these considerations, keep in mind that keeping data in the MySQL Query Cache is difficult as you have no control over the queries that are stored. This means that a slightly unusual query can temporarily clear a frequently used (and normally cached) query, reducing the effectiveness of your Query Cache. With memcached you can specify which objects are stored, when they are stored, and when they should be deleted giving you much more control over the information stored in the cache.

5.29: Can we implement different types of memcached as different nodes in the same server - so can there be deterministic and non deterministic in the same server?

Yes. You can run multiple instances of memcached on a single server, and in your client configuration you choose the list of servers you want to use.

5.30: What are best practices for testing an implementation, to ensure that it is an improvement over the MySQL query cache, and to measure the impact of memcached configuration changes? And would you recommend keeping the configuration very simple to start?

The best way to test the performance is to start up a memcached instance. First, modify your application so that it stores the data just before the data is about to be used or displayed into memcached.Since the APIs handle the serialization of the data, it should just be a one line modification to your code. Then, modify the start of the process that would normally load that information from MySQL with the code that requests the data from memcached. If the data cannot be loaded from memcached, default to the MySQL process.

All of the changes required will probably amount to just a few lines of code. To get the best benefit, make sure you cache entire objects (for example, all the components of a web page, blog post, discussion thread, etc.), rather than using memcached as a simple cache of individuals rows of MySQL tables. You should see performance benefits almost immediately.

Keeping the configuration very simple at the start, or even over the long term, is very easy with memcached. Once you have the basic structure up and running, the only change you may want to make is to add more servers into the list of servers used by your clients. You don't need to manage the memcached servers, and there is no complex configuration, just add more servers to the list and let the client API and the memcached servers make the decisions.