



PERCONA  
Performance Consulting Experts

# The MySQL Query Cache

Baron Schwartz

Percona Inc

# The Roadmap

- How it works
- What it isn't
- Myths
- How it uses memory
- Monitoring and status
- Configuration
- Trivia (how it works with InnoDB)

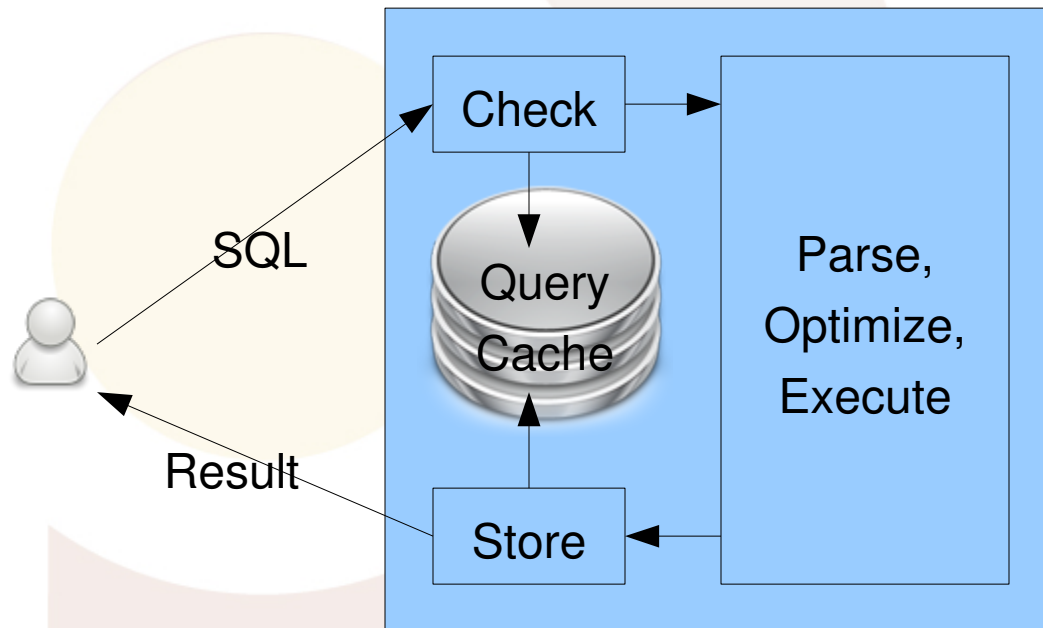
# What is the Query Cache?

---

- Caches the result of a SELECT statement
  - The raw bytes
- When there's a hit, just resends the result
- Does not cache execution plans

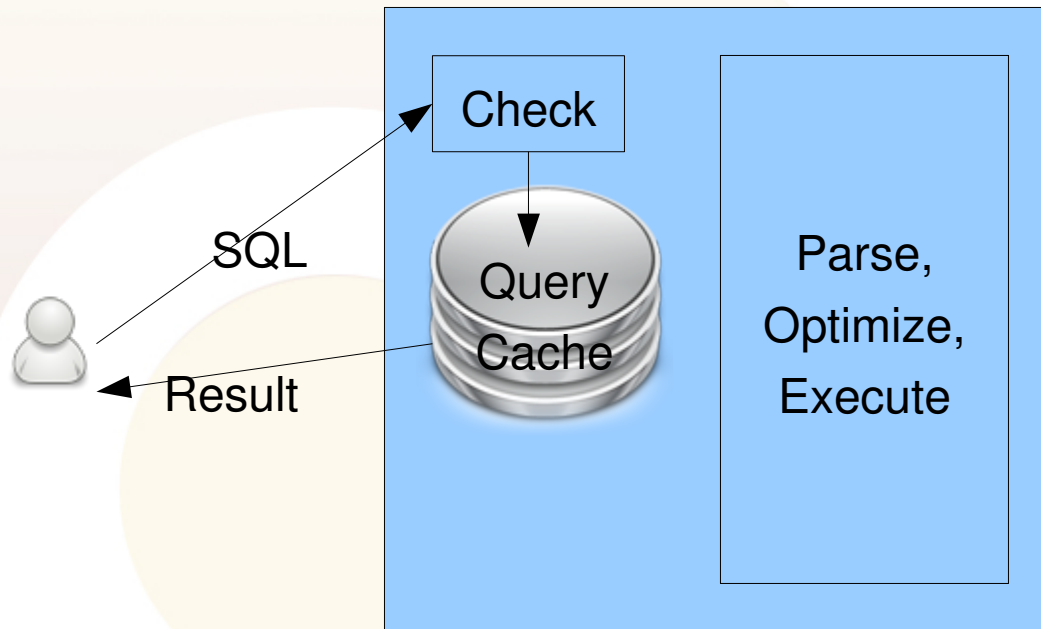
# How it Works

- It's a big hash table
- The hash key is the query text, current database, relevant character sets, etc etc.
  - It's case-sensitive, whitespace-sensitive



# If there's a cache hit

- There's no parsing, optimizing, etc etc



# Not all queries are cacheable

- Temp tables
- User variables
- Non-deterministic functions such as RAND() and NOW()
- Column-level privileges
- LOCK IN SHARE MODE/FOR UPDATE
- User-defined functions

# Query Cache Myths

- “MySQL might serve from the cache if the query contains `SQL_NO_CACHE` and a previous query without it was inserted”
  - False: the query won't match the previous query
- “MySQL doesn't check the query cache if the query contains `CURRENT_DATE`”
  - False: MySQL checks the cache before it parses the query
  - Enabling the query cache adds overhead to all `SELECT` queries

# Query Cache Overhead

- Each SELECT has extra overhead
  - Must check the cache for a hit
  - If it's cacheable and not in the cache, must store the result
- Each UPDATE, INSERT, etc has extra overhead
  - Must check for cached queries to invalidate

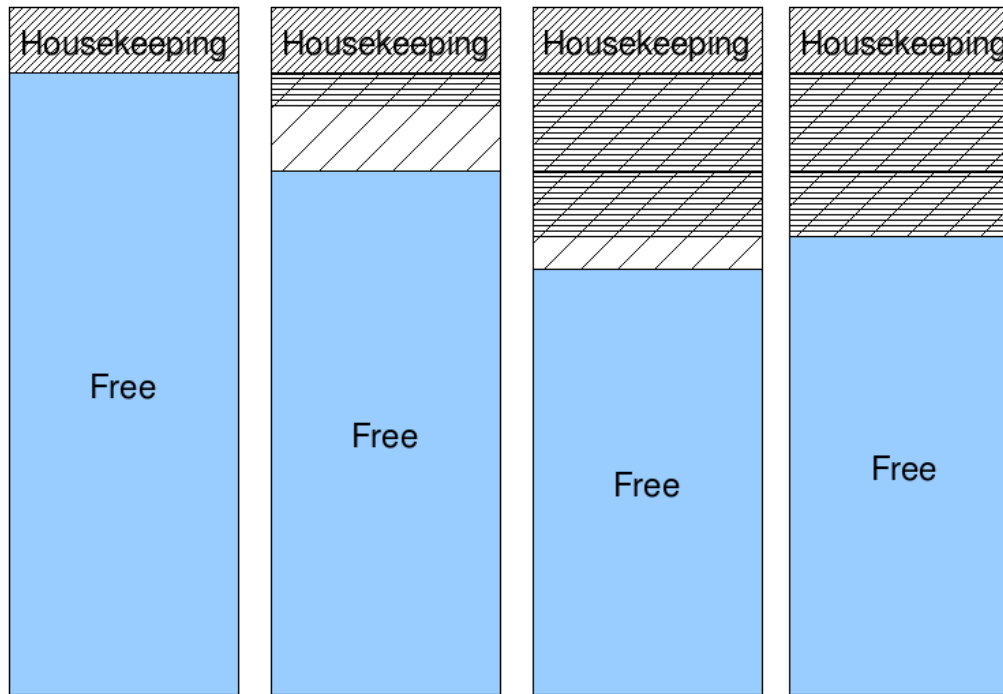
# Memory Usage

- The query cache is completely in-memory
- MySQL allocates one big chunk of memory for it
- MySQL manages its own memory – no malloc()
- Internally, it is structured into “blocks”
  - Hold that thought! These are not traditional blocks
  - Variable-sized blocks, not 4K or 16K or whatever
  - Initially, the entire cache's memory is one big block
  - Blocks can be of several types: free, table list, cache result, etc

# Storing Results

- “Allocate” a block for results
  - must be at least `query_cache_min_res_unit` bytes
  - finding an existing free block can take some time
- Store results as they're sent
  - Server does not know in advance how big the entire result will be
  - If bigger than `query_cache_limit`, abort
- When block is full, allocate another
- When done, trim the last block to size

# Storing Results



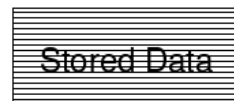
Initial state

Storing results

Results complete

After trimming

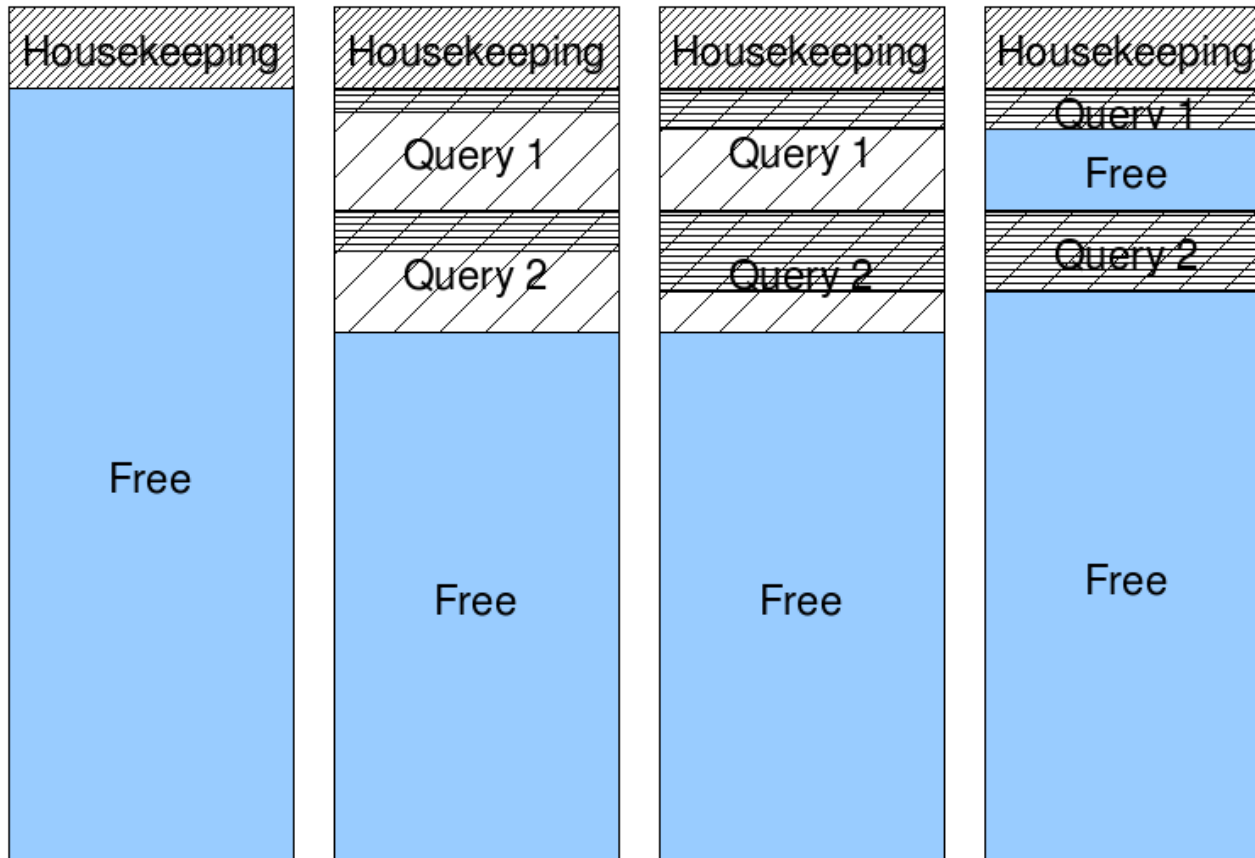
Legend



# Fragmentation

- Happens because of trimming
- Happens because of invalidations
- Is a problem because you get blocks that are too small to use, so memory is wasted
- If `Qcache_lowmem_prunes` is increasing and you have lots of free memory, suspect fragmentation

# Fragmentation



Initial state

Storing two  
results

Results  
complete

After  
trimming

# Monitoring and Status

```
mysql> show global status like 'qcache%';
```

Variable_name	Value
Qcache_free_blocks	11
Qcache_free_memory	16715152
Qcache_hits	49422
Qcache_inserts	8072
Qcache_lowmem_prunes	0
Qcache_not_cached	17404
Qcache_queries_in_cache	32
Qcache_total_blocks	82

# Monitoring and Status

- `Qcache_total_blocks`
  - total number of variable-sized blocks in cache
- `Qcache_free_blocks`
  - number of blocks of type FREE
  - worst-case:  $Qcache\_total\_blocks / 2$
- `Qcache_free_memory`
  - total bytes in FREE blocks

# Monitoring and Status

- Qcache\_hits
  - queries that were returned from the cache
  - hit rate:  $\text{Qcache\_hits} / (\text{Qcache\_hits} + \text{Com\_select})$
- Qcache\_inserts
  - queries that were stored into the cache
- Qcache\_lowmem\_prunes
  - number of cached results discarded to make room for new results
  - fragmentation can cause this to grow

# Monitoring and Status

- `Qcache_not_cached`
  - queries that were uncacheable
  - had non-deterministic function
  - were bigger than `query_cache_limit`
- `Qcache_queries_in_cache`
  - total number of queries in the cache
- `Qcache_invalidations` [doesn't exist]
  - but you can calculate it:  
$$\text{Qcache\_inserts} - \text{Qcache\_queries\_in\_cache}$$

# Avoiding Fragmentation

- You can avoid fragmentation with the block size
  - try setting it close to the average result size
  - $(\text{query\_cache\_size} - \text{Qcache\_free\_memory}) / \text{Qcache\_queries\_in\_cache}$
- You might not be able to pick a good size
  - you have a blend of large and small queries
  - some queries cause a lot of churn
  - you can set the `query_cache_type` to `DEMAND` and use `SQL_CACHE` to select queries that are good to cache

# Defragmenting

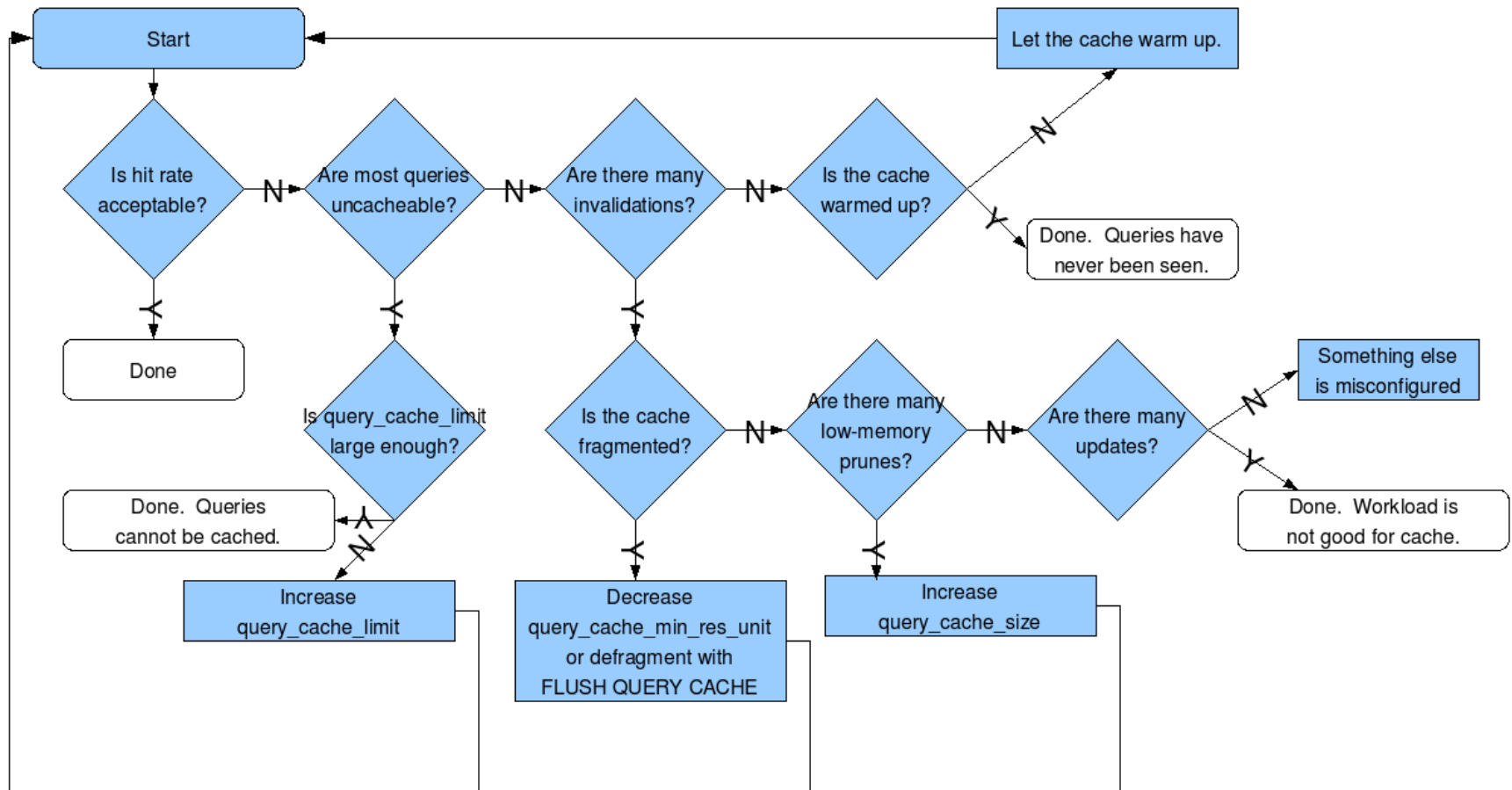
- Use FLUSH QUERY CACHE
- It doesn't flush the cache, it compacts it
- It locks the whole cache
  - effectively locks the whole server

# Tuning

```
mysql> show global variables like 'query_cache%';
```

Variable_name	Value
query_cache_limit	1048576
query_cache_min_res_unit	4096
query_cache_size	16777216
query_cache_type	ON
query_cache_wlock_invalidate	OFF

# Tuning the Query Cache



# InnoDB and the Query Cache

- InnoDB works with the query cache
  - for some value of “works”
- InnoDB tells the server whether a table is cacheable
  - Both for storing results, and for reading results
- Two factors determine this:
  - Your Transaction ID
  - Whether there are locks on the table
  - This is a rough heuristic

# InnoDB and the Query Cache

- If there are locks on a table, it's not cacheable
- If the table's transaction counter is  $>$  yours, you can't access that table in the cache
  - Each table's transaction counter is updated when a txn with locks on the table commits
  - It is updated to the system's transaction ID, not the txn ID of the txn with locks
  - Thus, if you modify a table, you can never read/write it again till you start a new transaction

# Optimizations

- Many small tables instead of one big table
- Batched writes (fewer invalidations)
- Don't make it too big or it stalls—256 MB is plenty
- Consider disabling it entirely