The SQL query cache

Source file: doc/eng-us/bc/6.0/sql-query-cache.md Last changed 2026-11-04
https://doc.exponential.earth/exphelp/viewer/bc/6.0/sql-query-cache.md

Read this page if your pages are slow on a full render (the first render after a purge, signed-in pages, the administration interface, scripts) and you want to switch the SQL query cache on, or if you restore databases or write to them from outside Exponential. The query cache answers a `SELECT` again from memory until a write to one of the tables it read makes the result stale. It is **off by default** (`settings/querycache.ini`, `Mode=off`), so an upgrade changes nothing until you turn it on.

In short

What changed	New class `eZDBQueryCache` (`lib/ezdb/classes/ezdbquerycache.php`), hooked into the MySQL/MariaDB (`eZMySQLiDB`), PostgreSQL (`eZPostgreSQLDB`), SQLite (`eZSQLite3DB`) and Oracle (`eZOracleDB` of the ezoracle extension, from its query cache hooks on) drivers. New settings file `querycache.ini`; new buttons in Setup > Caches and a box in Setup > System information. The MongoDB driver does not use it.
Who is affected	Nobody until `Mode` is changed. Once it is on: anyone who writes to the database outside Exponential (a SQL client, a restore, another program) must clear the query cache afterwards. Velocity must be restarted after the update that adds the class.
How to check	`grep -n "^Mode" settings/querycache.ini` and Setup > System information, box **Database queries**
How to fix	Follow Switch it on. After a restore: `php bin/php/ezcache.php --clear-id=querycache --allow-root-user`.

It is the counterpart of the HTTP cache. The HTTP cache removes whole renders for the pages it can store; the query cache speeds up **everything it cannot**: the first render after a purge, pages with query strings, POST responses, the administration interface, signed-in pages with a private permission context, cronjobs and scripts.

On Oracle, besides what is never cached anywhere (see What is never cached), a statement that reads a sequence (`<sequence>.NEXTVAL`, `.CURRVAL`: the driver reads every new row's id that way), the clock (`SYSDATE`, `SYSTIMESTAMP`), the SCN, `SYS_GUID`, `SYS_CONTEXT/USERENV` or `DBMS_RANDOM`, and anything that reads the catalogue (`USER_*`, `ALL_*`, `DBA_*`, `V$*`), is always run. An anonymous PL/SQL block (`DECLARE ...`, `BEGIN ... END;`) counts as a write whose tables cannot be read: it makes every result stale.

Session settings (Oracle `ALTER SESSION`)

ezoracle runs one `ALTER SESSION SET` on every connect (`initializeSession()`): the number format (`NLS_NUMERIC_CHARACTERS`), character length semantics (`NLS_LENGTH_SEMANTICS`), `CURSOR_SHARING`, and with `site.ini [DatabaseSettings] OracleCaseInsensitive=enabled` also `NLS_COMP=LINGUISTIC` and `NLS_SORT`. Such a statement is **no write**. Its parameters become part of the key of every statement of that connection, so a result is answered only to a connection whose session has the same settings, and every connection of an installation, which runs the same statement, shares the results across requests (also with `OracleCaseInsensitive`).

`ALTER SESSION ...`	**What the cache does**
`SET` of only `NLS_*` parameters, `CURSOR_SHARING`, `TIME_ZONE`	no write; the values go into the connection's key (names and unquoted values in upper case, the last value of a parameter wins)
`SET` of anything else, alone or mixed in (`CURRENT_SCHEMA`, `EDITION`, `CONTAINER`, `EVENTS` ...), any other form (`ENABLE PARALLEL DML`), a second statement after `;`	a write whose tables cannot be read: every result stale

The session parameters are kept per driver object (a `WeakMap`), not per request: a persistent worker that keeps its connection keeps its session. `eZDBQueryCache::sessionSettings( $sql )` returns what a statement sets (or null), `eZDBQueryCache::sessionKey( $db )` what the key of a connection carries ('' for MySQL, PostgreSQL, SQLite).

The other drivers set their session through the client library (`mysqli_set_charset()`, the connection string), not through `query()`; a `SET ...` or `PRAGMA ...` that does reach `query()` is no write for the cache and does not change the key.

What it gives (measured on a test installation, 2026-09-27)

Every page runs the same SQL, word for word, on every request: across two requests for the front page, 405 of its 405 distinct statements had the same text, and a quarter (26 %) even repeat inside one request. In `shared` mode a warm page sends **no statement at all** to the database.

Plain Apache (php-fpm), rendered pages

The same pages back to back with `Mode=off` and `Mode=shared`, on a test installation. A query string keeps the HTTP cache out, so every request is a full render. `ab -c 8`, 80 requests anonymous and 40 signed in; "alone" is one request with nothing else running (median of 5). Signed in is an administrator session.

Page	Visitor	Off: pages/s	Shared: pages/s	Gain	Off: alone	Shared: alone	SQL off → shared
/ (front page, layouts)	anonymous	9.4–13.3	41.4–43.0	**3.2–4.4×**	485–608 ms	164–177 ms	545 → 0
`/fitness` (section)	anonymous	30.4–30.7	64.9–76.9	**2.1–2.5×**	294–299 ms	101–104 ms	273 → 0
`/workout` (section)	anonymous	28.5–28.8	60.3–68.9	**2.1–2.4×**	263–303 ms	94–117 ms	269 → 0
/	signed in	2.05	8.17	**4.0×**	582 ms	191 ms	544 → 0
`/fitness`	signed in	4.23	16.98	**4.0×**	263 ms	111 ms	272 → 0
`/workout`	signed in	4.60	18.61	**4.0×**	288 ms	98 ms	268 → 0

In short: a rendered page takes a **third of the time** (front page ~0.6 s → ~0.17 s alone), and the server renders **2–4 times as many pages per second**. The database no longer works for these pages at all, which frees the MariaDB CPU that bounded full renders on this machine (0.2–0.3 s per page).

Notes on reading the table:

- The ranges are two runs (2026-09-27, 02:3x and 02:4x); the signed-in rows one.

- Signed in, eight concurrent requests share **one** session, and PHP's session lock runs them one after another: p50 is about eight times "alone" (off 1.9 s, shared 0.45 s on `/fitness`). Different users do not wait for each other; "alone" is what one signed-in visitor feels.
- What is left of a render is templates and PHP. For the pages the HTTP cache can store, a hit (380–520 pages/s on Apache) is still far faster; the query cache is what makes the pages it cannot store faster: the first render after a purge, query strings, POSTs, private contexts, the administration interface, scripts.
- A cold cache (the first request after a clear or a write to the tables a page reads) runs every statement as with `Mode=off`, plus the lookups and stores; its overhead has not been measured separately.

Correctness, checked on a test installation

- the rendered HTML is byte-identical with the cache off, cold and warm;
- a write from a CLI script is seen by the web servers at once (the next front-page render re-ran the 125 statements that read the written table, and the one after that ran none);
- a transaction's writes count at `COMMIT` and are forgotten on `ROLLBACK`;
- publishing an object in the administration interface shows the new title in the next read;
- temporary tables (the shop's related-purchase list, search) are never cached.

Settings: `settings/querycache.ini`

Override in `settings/override/querycache.ini.append.php` (or a siteaccess).

INI

```ini
[QueryCacheSettings]
Mode=off          # off | request | shared
Storage=auto      # where shared keeps results: auto | apcu | file
ApcuReserve=25    # percent of APCu kept free for the pool's other users; 0 = off
MaxAge=300        # longest life of a result, in seconds
MaxRows=5000      # results with more rows are not kept
ExcludeTables[]
ExcludeTables[]=ezsession
ExcludeTables[]=ezpending_actions
ExcludeTables[]=ezcollab_notification_rule

[DebugSettings]
Report=enabled    # with the debug output on: accumulators, timing point, report (see "In the deb
ug output")
```

Mode	What is kept	Where
`off`	nothing; the hooks return at once	—
request	a memo for one request: an identical SELECT is not run twice in it	PHP memory
shared	the memo, and results across requests	the memo, then APCu of the server process group or one file per result (`Storage`)

Choosing:

- `request` is safe everywhere and needs nothing: it removes the in-request repeats (about a

quarter of the statements on a layouts page).

- `shared` keeps results in APCu (`apc.enabled=1`; for scripts also `apc.enable_cli=1`) or in files, as `Storage` says. This is the mode that removes the statements between requests.
- `Storage=auto` (the default) takes APCu where the PHP process has it and files otherwise in every process that serves web requests (FPM, and the workers of Exponential Velocity, whose SAPI is `cli`); cronjobs and scripts without APCu keep only the request's memo (their files would only add disk traffic); `Storage=apcu` only APCu (without it `shared` works as `request`, the behaviour before files existed); `Storage=file` always files, even with APCu. Files are `var/<site>/cache/querycache/entries/<2>/<key>`, one serialized result each (read with `allowed_classes` off), written to a temporary file in the same directory and renamed (atomic, on NFS too). A file is named by the key's hex digits only, so no name can reach outside `entries/`. Directories and files are group-writable whatever the umask, and a root process (Velocity, a script run as root) hands them to the owner of the state directory, so the site user's FPM can read and replace them. Every web server that mounts `var/` shares them (a cluster on NFS or a SAN), where APCu is per server. A hit costs a file read and `unserialize()`. Measured on a test installation (PHP 8.4, APCu 5.1.28, files on a local disk), the cache's own cost per statement:

Result	Hit, APCu	Hit, file	Miss and store, APCu	Miss and store, file
1 row	0.02–0.03 ms	0.05–0.08 ms	0.02–0.04 ms	0.17–0.23 ms
26 rows	0.03–0.05 ms	0.07–0.09 ms	0.04–0.05 ms	0.18–0.21 ms
500 rows	0.31–0.42 ms	0.35–0.41 ms	0.39–0.42 ms	0.47–0.63 ms

  A hit from a file takes about twice as long as one from APCu for small results and the same for large ones, where `unserialize()` is most of the cost; storing a file (temporary file, mode, rename) costs 0.15–0.2 ms more for small results and 0.1–0.2 ms more for 500 rows. Both are far below a statement on the database (through the Oracle driver on that installation: 0.2–0.3 ms for a simple statement, about 60 ms for 500 sorted rows, against 0.05–0.1 ms and 0.4 ms for a hit). On NFS or a SAN every file access adds the storage's latency; that was not measured.

  The cronjob part `cache_cleanup` (`cronjobs/querycache_cleanup.php`) removes the files older than `MaxAge` or from before the last clear; clearing the query cache removes them at once. Schedule it with `Storage=file` or `auto` without APCu. Without it the requests sweep the files themselves: a request that stored a file runs the same cleanup at its end, at most every 10 minutes (or `MaxAge`, when that is longer) for the installation, one process at a time (`entries/.swept`, `entries/.sweep.lock`) and for at most half a second, so the files cannot grow without bounds (`eZDBQueryCache::sweepFiles()`).
- `ApcuReserve` (default 25): APCu is one memory for every site of the PHP-FPM pool (the HTTP cache, API keys, a request filter's rate limits …). Full with valid entries, APCu empties itself completely, and everything in it is lost, not only the query cache's results (`apc.ttl` helps only when there are entries nobody read for that long). With this share or less of APCu free, shared mode stops adding results to APCu and keeps them for the request only. The free share comes from `apcu_sma_info()`, read before every store (measured on a test installation with PHP 8.5: 0.17 to 0.22 µs with one segment of 64 MB or 1.2 GB and 20,000 entries, against 0.6 µs for an `apcu_fetch()`), so a long-running process cannot overshoot it. APCu frees an expired entry only when it is deleted by its key or when an allocation fails, and below the reserve the cache allocates nothing: so a miss below the reserve deletes its

key's old entry, and once a minute for the whole pool `eZDBQueryCache::purgeExpired()` deletes every expired query cache entry from APCu's entry list, as long as APCu holds at most 30,000 entries and the list fits below 80 % of `memory_limit` (it costs about 0.8 KB of PHP memory per entry, about 200 ms and 100 MB at 130,000). In a web request it runs at the end, once per request: also in a worker of Exponential Velocity, whose shutdown functions run at the end of every request; a script (cronjob, CLI) runs it at once. It reads APCu's whole entry list (an `APCUIterator` does not list expired entries). Above that only a larger `apc.shm_size` helps. With a small APCu (the default `apc.shm_size` is 32 MB) a reserve of 25 % is 8 MB, and another user holding three quarters of it keeps the query cache out of APCu; the counter shows it. Setup > System information and the Cache tab of the Exp Debug bar count the results held back; the debug report lists them as *Not cached: APCu reserve*. Measured with a 32 MB APCu that another user had filled to 85 %: with `ApcuReserve=10` the cache stored 34 results and held back the rest, the other user's entries stayed; with `0` APCu emptied itself five times and the other user's entries were lost. With the cache's own entries expired and never asked again (one-shot statements, 32 MB APCu at 25 % free), it stored again a minute later, when the purge had freed them. Files (`Storage=file`) have no reserve.

- Several web servers on one `var/` over NFS: whether a result is current is decided by `state.ser`, as for APCu. NFS clients cache file attributes and directory lookups (`acregmin`/`acdirmin` 3 to 30 s by default), so a write or clear on one server can take that long to be seen on another. Mount `var/` with `lookupcache=none` and `actimeo=0` (or `noac`) where that matters; the state file's lock needs NFSv4 or `lockd`. Clock skew between the servers only shifts when the cronjob removes a file, never what is answered.
- `MaxAge` bounds what the cache cannot see: a write made outside Exponential (another program, a SQL client, a restore). Within Exponential every write invalidates at once, whatever `MaxAge` is.
- `ExcludeTables` are tables whose reads are never cached — per-visitor or constantly written tables where a cached answer would never be reused.

After changing the file: clear the INI cache (`php bin/php/ezcache.php --clear-tag=ini --allow-root-user`), reload PHP-FPM and restart Velocity (`./console exp:velocity restart`).

How it works

Class `eZDBQueryCache`, `lib/ezdb/classes/ezdbquerycache.php`. Each SQL driver calls it in three places (a lookup lands on the stored entry first and uses the tables stored with it; the statement's text is parsed only on a miss, since the parsing was about a third of a rendered page's CPU):

- `arrayQuery()` asks `lookup()` before running a statement, and gets the rows or a ticket;
- `arrayQuery()` hands the rows to `store()` after a statement that returned;
- `query()` tells `noteWrite()` of every statement that succeeded.

The key of a statement carries the connection (driver class, database, server, port, socket and user, so two sites on one server and one APCu never share rows), the Oracle session settings, the text and `arrayQuery()`'s parameters. The request's memo keeps at most `eZDBQueryCache::MEMO_MAX` (1,000) results, the least recently used going first: a long script would otherwise keep every result it read, and every write walks the memo.

`eZDBInterface::commit()` and `rollback()` call `afterCommit()` and `afterRollback()`. `ezpKernelWeb` calls `resetRequest()` at the start of every request, so a persistent worker (Velocity, FrankenPHP worker mode) starts each one with an empty memo.

In a persistent worker (checked with Velocity, `ForkPerRequest=disabled`) the reset also drops the worker's copy of the state, so every request reads the state file again at its first lookup or write: the "once a second" below applies within one request only, and another process's write or a replaced state file counts from the worker's next request on (QC-46). The settings, the counters and the "lock failed" note are per request as well. What is kept across requests is only what belongs to a connection: the temporary tables this process created and the Oracle session settings, keyed by the driver object, so a new connection per request starts with none. No connection is kept for the cache's sake. `MEMO_MAX` bounds one request; a rendered page looks up about 550 statements, far fewer of them distinct, so a page never reaches it, and a script that does reads the older results from APCu or the result files again.

Validity

A shared state file, `var/<site>/cache/querycache/state.ser`, holds

```
{ generation, genTime, tables: { table => time of its last write }, id, lockFailed }
```

Every process reads and writes it: Apache/php-fpm, Velocity, scripts and cronjobs each have an APCu of their own, so the state cannot live there. It is written under `flock` and replaced atomically; a process reads it again when its copy is older than one second. The lock is what keeps the state right: `state.lock` must be writable by every user the site runs as, and on NFS it needs NFSv4 or `lockd`. A write that cannot open or take the lock makes everything stale and logs it once per request; that narrows the loss of another process's change between reading and writing the file but does not close it, and on a server where the lock never works every write empties the cache. A file that cannot be written is logged, its temporary file removed; the change then holds in this process only, others see it after `MaxAge`.

A lock failure is also kept in the state file (`lockFailed`, the time), so Setup > System information shows it whichever process failed (a cronjob running as another user, a Velocity worker), and the debug output of the request shows it as well (see below). **Clear the query cache** forgets it: after fixing the permissions, clear the cache and check whether the notice comes back.

The lock is taken on the file at the path. A process that waited for it while the cache directory was deleted would otherwise get the lock on a file that is gone, while the next process creates a new `state.lock` and locks that one, and both would write the state at once. After `flock` the file held is compared with the one at the path (inode and device) and opened again when they differ, up to three times; then the write counts as a lock failure (QC-45).

`id` is set when a file is written where there was none, or none that could be read: the first write of an installation, and the first one after the file was deleted (an `rm -r` of the cache directory, which leaves APCu and perhaps the result files behind). Results carry the id of the state they were stored under and are answered only under that one, so results whose writes the lost file knew are never answered again. The first write of a new installation therefore makes everything stored before it stale, once. A miss reads the state for this.

An entry is kept with the time its query **started**, the tables it read, the state's id and its rows. It is answered only while

- it was stored under the state file in place now (same `id`),

- it was created after `genTime` (the last full clear),
- every table it read was last written before it was created, and
- it is younger than `MaxAge`.

A read that raced a write is therefore always stale, never trusted.

What is never cached

- reads inside a transaction (they may see the transaction's own writes);
- statements whose tables cannot all be read from the text: anything with a parenthesis in a `FROM` list (a derived table, an index hint, `PARTITION (p)`, `TABLE(...)`, `LATERAL (...)`, `ONLY`, a parenthesised join (`JOIN ( b JOIN c ... )`), a comma list after a join (`JOIN b ON ..., c`, `JOIN b USING (i), c`), a quoted name with a space in it, a comment (`--`, `/* */`, and `#` where a backslash escapes), Oracle's `q'[...]'` and PostgreSQL's `E'...'` literals, anything that is not a plain `SELECT`. Each `SELECT` of `UNION`, `MINUS`, `INTERSECT` and `EXCEPT` is read, and `STRAIGHT_JOIN`. Where a backslash in a literal would end the literal elsewhere depending on the database (it escapes in MySQL's literals, and double quotes delimit strings there; not in Oracle's, SQLite's or PostgreSQL's with `standard_conforming_strings` on, the default: `'C:\'`), the driver's `databaseName()` decides (`eZDBQueryCache::backslashEscapes()`); a driver that does not say gets the statement read both ways, and where they differ it is not cached. The same rules apply to writes: a write that cannot be read makes everything stale;
- on every database, a quote that is left unpaired (a literal that does not end) and a quoted name that holds a quote (`"it's"`, `` `it's` ``, SQLite's `[it's]`): read without knowing the name, its apostrophe would pair with the next literal's quote and hide what is in between;
- on PostgreSQL and SQLite (QC-43): a backslash does not escape, so `'C:\'` is a whole literal and the statement is read; `"..."` quotes a name, which is read as a table where it is one (`"public"."ezuser"` is `ezuser`). Not cached: PostgreSQL's `E'...'` strings and dollar quoting (`$$...$$`, `$tag$...$tag$`; `$1` is a parameter and stays readable), `TABLESAMPLE`, a column alias list (`FROM a x (c1, c2)`). `#` is no comment there (an operator in PostgreSQL). A PostgreSQL connection with `standard_conforming_strings` off (the server's setting, or a `SET` in the session) escapes with a backslash as MySQL does: the driver reads the setting from the connection (`eZPostgreSQLDB::standardConformingStrings()`, which costs no statement, the server reports every change), and double quotes then make a statement not cacheable;
- statements whose result is not a function of their tables: `NOW()`, `UNIX_TIMESTAMP()` with no argument, `CURRENT_DATE` and friends, `RAND()`, `UUID()`, `LAST_INSERT_ID()`, `FOUND_ROWS()`, Oracle's `SAMPLE (n)`, locks (`FOR UPDATE`, `LOCK IN SHARE MODE`, `GET_LOCK()`), variables (`@x`, `@@x`), `SQL_NO_CACHE`, `INTO OUTFILE`; PostgreSQL's `clock_timestamp()`, `statement_timestamp()`, `transaction_timestamp()`, `timeofday()`, `txid_current()`, `setval()`, `gen_random_uuid()`, `current_setting()`, every `pg_*()` function (advisory locks, `pg_sleep()` ...), `FOR NO KEY UPDATE`, `FOR KEY SHARE`; SQLite's `changes()`, `total_changes()`, `last_insert_rowid()`, `randomblob()`, `date()`, `time()`, `datetime()`, `julianday()` and `unixepoch()` without an argument; and the literal `'now'` (also `'today'`, `'tomorrow'`, `'yesterday'`, `'localtime'`), which PostgreSQL's casts and SQLite's date functions read as the clock;
- reads of temporary tables: those `CREATE TEMPORARY TABLE` made in this process, and those named like `eZDBInterface::generateUniqueTempTableName()` names them (`ezsearch_tmp_12`, `ezproductcoll_tmp_972`, `eznode_count_3`). A temporary table belongs

to one connection and its name is reused, so another request's rows must never answer for it. Writes to temporary tables invalidate nothing and stay out of the state file;

- results with more than `MaxRows` rows;
- reads of the `ExcludeTables`.

Writes

`INSERT`, `REPLACE`, `UPDATE` (multi-table too), `DELETE` (with `JOIN`s), `TRUNCATE`, `CREATE`/`ALTER`/`DROP TABLE`, `CREATE`/`DROP INDEX` and `RENAME TABLE` mark the tables they name as written now. A write whose tables cannot be read from the text marks **everything** stale (a generation bump), which is always safe. Inside a transaction the tables are collected and marked at the outermost `COMMIT`; on `ROLLBACK` they are dropped.

The forms of PostgreSQL and SQLite (QC-44): `INSERT OR REPLACE`/`OR IGNORE`, `INSERT ... ON CONFLICT`, `UPDATE ONLY a` and `DELETE FROM ONLY a` (the table after `ONLY`), `TRUNCATE a, b` and `DROP TABLE a, b` (every table of the list), `ALTER TABLE ONLY a`, and `ALTER TABLE a RENAME TO b` (both names). `TRUNCATE ... CASCADE` and `DROP ... CASCADE` reach tables and views the text does not name, so they make everything stale; so do a `WITH ... INSERT`, `COPY` and an unnamed `CREATE INDEX ON a`. A name in a write's table list that cannot be read (a quoted name with a space, a subquery) makes the whole write unreadable: before, it was left out, and its table never became stale.

Keys

`xxh128` (where PHP has it; `md5` otherwise) of the driver class, database name, server, port, socket, database user, the Oracle session settings, the statement text and `arrayQuery()`'s parameters (offset, limit, column). In APCu the entries are `ezqc:<key>`, with `MaxAge` as their TTL, so APCu can evict them under memory pressure. The counters are `ezqcstat:*` (in APCu only: with the file storage and no APCu there are no counters). As files, the entries are `entries/<first two characters of the key>/<key>`; a lookup checks their age and the state like an APCu entry, the cronjob removes the old ones.

In the administration interface

Setup > Caches

Two rows at the top, with the other most-used buttons:

- **Clear query cache** (`ClearQueryCacheButton`) — a generation bump: every stored result is stale at once, on every server sharing `var/`, without walking APCu. The row shows the mode.
- **Clear HTTP cache** (`ClearHttpCacheButton`) — drops every cached page for every permission context, including Velocity's copies. Greyed out when the HTTP cache is off.

The query cache is also in the cache list below (`querycache`, tags `content` and `ini`), so **Clear all caches**, **Clear content caches**, `ezcache.php --clear-all`, `--clear-tag=content` and `--clear-id=querycache` clear it too.

Setup > System information

The **Database queries** box shows, for the server answering the page:

- the mode, `MaxAge`, `MaxRows`, `ExcludeTables`;
- in `shared` mode, how many results this server holds in APCu and how much memory they take, or with the file storage how many result files there are, their size on disk and their directory;
- the generation, when the cache was last cleared, and how many tables the state file tracks;
- in red, when a write could not take the state lock, the time of the last failure and the lock file to check (from the state file, so any process's failure shows; **Clear the query cache** removes the notice);
- this server's counters since they were last reset: requests, hits, misses, hit rate, statements that could not be cached, writes;
- the tables written most recently, and how long ago;
- **Clear the query cache** and **Reset the counters**.

Counters and APCu entries are **per server**: Apache/php-fpm and Velocity each have their own APCu, so the numbers differ between the admin served by Apache and the admin served by Velocity (port 8080). The state (generation, tables) is shared.

Below it the SQL profile (next section) and its switch.

In the debug output

With the debug output on (`site.ini [DebugSettings] DebugOutput=enabled`) and `querycache.ini [DebugSettings] Report=enabled` (the default), every request shows what the cache did for it. `Report=disabled` keeps all three parts below out of the debug output; the cache itself works the same.

Time accumulators, group *Query cache* (the time is that of the cache's own work, not of the database):

Accumulator	Counts	Time
Hits	statements answered from the cache; they never reach the driver, so they are missing from the driver's accumulator (e.g. *Oracle_queries*, *mysqli_query*)	the lookup that answered
Misses (lookup only)	cacheable statements not in the cache (or stale) that went to the database	only the lookup; the database time stays with the driver's accumulator
Stores	results kept after a miss	copying the rows into the memo (and APCu)
Not cached	statements neither answered nor stored, for one of the reasons the report lists	the lookup

Timing point *Query cache: state read (generation n)*: where the request (or, after `eZDebug::reset()`, the job) first looks at the shared state file; *n* is the generation it found, the number every clear and every write of unknown tables raises. After `eZDebug::reset()` the point comes at once with the next statement when the process already holds the state (the statements of the job may never read it again), otherwise with the first look at it. The report

itself reads no state and sets no point.

Report at the bottom, *Query cache* (text and HTML, every value escaped):

Line	What it means
Mode	`off`, `request` or `shared`; in `shared` mode with its storage, `shared (apcu)` or `shared (file)`, or `shared (no APCu: works as request)` where it has none
Generation	the generation of the state as this request last read it; *not read* when no statement has looked at the state yet
Hits, Misses, Hit rate	as the accumulators; the rate is hits of hits plus misses
Stored	results stored
Not cached	statements not answered and not stored, then one line *Not cached: reason* per reason: *inside a transaction*, *statement (volatile or tables not readable)*, *excluded table* (`ExcludeTables`), *temporary table*, *catalogue table*, *more rows than MaxRows* (that one is also counted as a miss)
Writes	writes the cache saw (each makes the tables it names stale)
Database time saved (estimated)	the sum, over the hits, of the time each statement took the database with fetching its rows when it was stored; an upper estimate. Entries stored before this was recorded add "+ n hits on entries without a duration". Then its share of the page: `186 ms = 186 % of the page so far (100 ms; without the cache about 286 ms, 3 times as long)`; the factor only from 100 % on, where a share alone reads like a part of the page. The chip of the Exp Debug bar gives the same share, `QC 83 % · -186 ms (186 %)`, and the same sentence in its title
Made everything stale	up to 20 writes whose tables cannot be read (a PL/SQL block, `CALL`, an `ALTER SESSION` that is not a session setting ...), then "... and n more"
State lock	only when a write of this request could not take `state.lock`: "could not be taken", with what to check. Every write of the request then made every cached result stale, and the Exp Debug bar's chip turns red

The report counts what happened since it started: from the first statement of the request with the debug output on, or again from the first statement after `eZDebug::reset()`. Its counters are its own, so Setup > System information flushing the request's counters (`eZDBQueryCache::flushStats()`) does not change them. *Not cached* is larger than the *uncacheable* counter of Setup > System information, which leaves out excluded tables and transactions.

The page's time for the share (`eZDBQueryCache::requestMs()`, one time base for the report row and the chip): from eZDebug's script start, which the web kernel sets at the start of every request, also in a persistent worker of Exponential Velocity (so a worker's earlier requests never count), else from PHP's `REQUEST_TIME_FLOAT` (scripts; Velocity sets it for every request too); to where the Exp Debug bar's output starts (`setRequestEnd()`, cleared by `resetRequest()`), else to eZDebug's script stop, else to now. A start that would give a negative time or more than a day is not used (no share is shown): it can only be left from an earlier request. The bar's time chip measures to when its summary is made, a few milliseconds later.

Literals masked. A statement in *Made everything stale* has each string literal replaced by `'?'`, so values from requests and content stay out of the debug output (`eZDBQueryCache::maskLiterals()`). Where a literal ends depends on the database: SQL-standard quoting doubles a quote (`'it''s'`: Oracle, PostgreSQL, SQLite), MySQL also escapes with a backslash (`'it\'s'`, what its `escapeString()` writes), and `'C:\'` is a whole

literal in standard SQL but not in MySQL. Both readings are masked, and an unterminated quote is masked to the end, so no part of a literal shows under either reading; where they disagree, more than the literal is masked (`CALL p( 'it\'s', 1 )` shows as `CALL p( '?'`). Only the first 4 KB of a statement are read, and the line is cut at 200 characters. Double-quoted text (an identifier, or a string in MySQL without `ANSI_QUOTES`) and numbers are shown as they are.

Persistent workers (Exponential Velocity): `resetRequest()`, which the web kernel runs for every request, drops the report, so nothing is carried from one request to the next; after `eZDebug::reset()` the next statement registers the accumulator group and the report again, starts new counters and sets the timing point again.

Cost. With the debug output off the cache only tests, per statement, whether it is on (no call, measured within the noise of a lookup hit of ~2.7 µs); the duration stored with each entry is one `microtime()` per store. With it on, the cost is that of the accumulators.

In the **Exp Debug bar** the same figures are a chip in the pinned bar (hit rate and saved time, the saved time also in percent of the page's time: `QC 83 % · -186 ms (186 %)` on a page of 100 ms means about 286 ms without the cache; amber below 50 %, red below 10 %, in `shared` mode without a storage or when a write could not take the state lock, `debugbar.ini Thresholds[querycache_hit_rate]`) and the report stands at the top of the SQL tab, next to the database's accumulators. The **Cache tab** (setup/managecache) shows what this server holds: entries and memory in APCu (or the result files and their size on disk; files count no hits), the counters since the last reset, the state, the cached results grouped by the tables they read (entries, rows, size, hits as counted by APCu) and the largest entries, with buttons to clear the query cache and to reset the counters.

The SQL profiler

`eZDBInterface::profileSQL()`, called by the MySQL driver after every statement. Off unless `var/tmp/sql_profile.on` exists — one `file_exists()` per request. The switch in Setup > System information creates and removes that file. On, one line per request goes to `var/tmp/sql_profile.log`: statements, `SELECT`s, distinct `SELECT`s, exact repeats, time in the database and in the repeats.

It counts the statements that **reached the database**: with the query cache on, a cached answer is not in it. That makes it the tool to see what the cache leaves: a warm shared-cache page shows 0 statements.

Two further sentinels, for studies: `var/tmp/sql_profile.shapes` adds the most repeated statements (values masked) to the log, and `var/tmp/sql_profile.hashes` writes each request's statement hashes to their own file, to compare two requests. Remove them when done; they write on every request.

Support and maintenance

Switching it on

1. Check APCu: Setup > System information, PHP section, or `php -m | grep apcu` (and the FPM pool's `php_admin_value[apc.enabled]`).
2. Set `Mode=request` in `settings/override/querycache.ini.append.php`; clear the INI cache, reload FPM, restart Velocity. Browse for a day.

3. Set `Mode=shared` the same way. Watch the hit rate in Setup > System information.

Switching it off

`Mode=off`, then the same INI clear, reload and restart. Nothing needs deleting: stale APCu entries expire within `MaxAge` and are never read while the mode is off.

What is normal

- A hit rate of 30–40 % right after a clear or restart, rising as pages are visited. On a quiet site with warm pages, most statements are hits.
- A burst of writes after a cache clear or Velocity restart: the warm-up renders pages, and some renders write (image aliases created on first view update `ezcontentobject_attribute`).
- `state.ser` a few KB; it lists every table ever written, one timestamp each.

What to look at

Symptom	Cause and fix
A page shows old data after a change made with a SQL client, `mysql < dump.sql` or another program	The cache cannot see writes outside Exponential. Clear it (Setup > Caches, or `php bin/php/ezcache.php --clear-id=querycache --allow-root-user`), or wait `MaxAge`. Do this after every restore.
`Class "eZDBQueryCache" not found` in `var/vc/qbix/log/error.log`, admin pages 500 on Velocity only	Velocity loaded the kernel autoload array before the class existed. Restart it: `./console exp:velocity restart --allow-root-user`. Any update that adds kernel classes needs this.
The hit rate stays low	A table the pages read is written on every request. Setup > System information lists the tables written last; find who writes (usually a view counter, a session-like table or a broken image alias regenerated on every view) and fix it, or add the table to `ExcludeTables`.
"APCu is not available to this server" in the box	`Storage=apcu` without APCu: `shared` falls back to `request`. Enable APCu for that PHP (FPM pool, or `apc.enable_cli=1` for Velocity and scripts), or set `Storage=auto` or `file`.
Many files in `var/<site>/cache/querycache/entries/`	The cronjob part `cache_cleanup` does not run. Schedule `php runcronjobs.php -s <siteaccess> cache_cleanup`; a clear of the query cache removes them too.
APCu memory full, other caches evicted	Keep `ApcuReserve` above 0, lower `MaxRows` or `MaxAge`, or raise `apc.shm_size`. Setup > System information shows what the query cache holds.
"results not kept in APCu" in the box or the Cache tab	APCu had no more than `ApcuReserve` percent free. Raise `apc.shm_size`, or find the other users of APCu (Setup > System information, PHP section).
`state.ser` not writable (warnings in the log)	`var/<site>/cache/querycache/` must be writable by every user the site runs as. Files written by root are handed to the directory's owner.

Symptom	Cause and fix
"Writes could not take the state lock" in the box, `could not lock ...` `state.lock` in the log, a red QC chip in the Exp Debug bar	`state.lock` cannot be opened or locked: every write makes every cached result stale, so the cache gives almost nothing. Make `var/<site>/cache/querycache/state.lock` writable by every user the site runs as (FPM, Velocity, cron); on NFS, mount with NFSv4 or run `lockd`. Then clear the query cache: the notice goes and stays away while the lock works.
A result looks wrong and you suspect the cache	Set `Mode=off` and compare. If the output is the same, it is not the cache.

Tests

Unit tests, no database: `tests/tests/lib/ezdb/eZDBQueryCacheTest.php` (`@group querycache`, in the `lib` suite). The driver is a stand-in object and the state lives in its own directory under `var/tmp` (`eZDBQueryCache::setStateDir()`), so the tests never touch the site's cache.

	What it proves
QC-01	The tables a `SELECT` reads (joins, aliases, `db.table`, quoted names, table names inside strings ignored); not cacheable: not a plain `SELECT`, volatile functions, locks, `SQL_CALC_FOUND_ROWS`, a subquery in `FROM`, variables
QC-02	The tables a write names, for every verb (`INSERT`, `INSERT IGNORE`, `REPLACE`, `UPDATE` multi-table, `DELETE` with `JOIN`, `TRUNCATE`, `CREATE/ALTER/DROP TABLE`, `CREATE INDEX`, `RENAME TABLE`); an unreadable write is null
QC-03	A miss, a store, then a hit with the same rows, and the counters
QC-38	Statements whose tables cannot all be told are not cached (a derived table, index hint, `PARTITION`, `LATERAL`, `ONLY` or `TABLE()` in `FROM`, a parenthesised join, a comma list after a join, a quoted name with a space, a comment with or without an apostrophe, `q''` and `E''` literals, a backslash that ends a literal differently per database); each `SELECT` of `MINUS`, `INTERSECT`, `EXCEPT`, `UNION` and `STRAIGHT_JOIN` are read; MySQL's `\'` and double-quoted strings stay one literal; a kernel join with `IN ( ... )` lists stays cacheable; `SAMPLE` is volatile
QC-04	A write to a table the result read makes it stale; a write to another table does not
QC-05	An unreadable write and a clear make everything stale; a clear raises the generation
QC-06	In a transaction reads are neither answered nor stored; writes count at `COMMIT` and are forgotten on `ROLLBACK`
QC-07	Temporary tables, by name and by `CREATE TEMPORARY TABLE`: never cached, never in the state
QC-08	`ExcludeTables`, `MaxRows`, `MaxAge`
QC-09	Keys differ by statement, `arrayQuery()` parameters, database and server
QC-10	A stored result is a copy: changing what a caller got changes nothing
QC-11	`Mode=off` stores nothing, writes no state and counts nothing
QC-12	A write by another process, seen through the state file, makes a result stale within a second, even one still in the request's memo; other tables stay current
QC-13	A stored result is answered without parsing the statement again, but not once a table it read has been excluded
QC-14	The database's own catalogue (`sqlite_master`, `sqlite_stat1`, `pg_*`, `information_schema`) is never cached

	What it proves
QC-15	Oracle: sequences, the clock, the SCN, `SYS_GUID`, `SYS_CONTEXT`, `DBMS_RANDOM` and the catalogue are never cached; a PL/SQL block makes everything stale, a bare `BEGIN` does not
QC-16	Oracle: an `ALTER SESSION SET` of `NLS_*`, `CURSOR_SHARING`, `TIME_ZONE` (the statement ezoracle runs on connect, with or without `OracleCaseInsensitive`) writes nothing; any other `ALTER SESSION` (`CURRENT_SCHEMA`, `CONTAINER`, `EVENTS`, `ENABLE ...`, a second statement) makes everything stale
QC-17	With the debug output on: the accumulators of the group *Query cache*, one timing point, the bottom report as text and HTML (escaped) with hits, misses, hit rate, reasons and the writes that made everything stale
QC-18	Without the debug output, or with `Report=disabled`, no accumulator, no report
QC-19	Oracle: a result is answered to the next connections with the same session settings, and connecting raises no generation, also with `NLS_COMP`/`NLS_SORT`
QC-20	Oracle: a result is never answered to a session with other settings (`NLS_SORT=GERMAN` against `BINARY_CI`); a later `ALTER SESSION` gives the connection its new key; with `Mode=off` nothing is noted
QC-21	Literals are masked under both quoting rules (`''`, `\'`, `'C:\'`, unterminated), and no part of one reaches the report, also from a statement longer than 4 KB
QC-22	After `eZDebug::reset()` in the same process the report, its own counters and the timing point start again; `flushStats()` leaves them alone; the next request carries nothing over; with the debug output switched off nothing is registered
QC-23	`debugSnapshot()`: the figures of the report (its own counters) with the debug output on, null without it
QC-24	`inventory()` (with APCu): entries grouped by table, an entry of two tables counted for each, hits from APCu, largest first, no statement text
QC-25	`storage()`: `Storage=file` even with APCu, `auto` APCu (on the command line without APCu none), `apcu` without APCu none, `request` mode none
QC-26	The file storage: one file per result under its two-character directory, a hit in the next request, stale after a write to its table
QC-27	A result file never makes objects (`__PHP_Incomplete_Class`), a broken file is a miss
QC-28	`cleanupFiles()` removes files older than `MaxAge` and temporary files left over, keeps the others; `status()` and `inventory()` count the files (no hits); a clear removes them
QC-29	A result file is only ever named by a key (hex digits): `../`, upper case, an empty or NUL-carrying name write and read nothing
QC-30	The requests sweep the result files without the cronjob: expired ones removed, at most once per interval, one process at a time, within a time budget
QC-31	`Storage=auto` uses the files in every web process without APCu, a Velocity worker (cli SAPI) included, and none in a script
QC-32	`apcuRoom()`: room only with more than `ApcuReserve` percent free, `0` never stops, 25 by default
QC-33	Below the reserve a result stays in the request, is counted (`reserve`) and not put into APCu; with room again it is; files have no reserve
QC-34	`purgeExpired()` deletes the query cache's expired APCu entries and frees their memory, keeps live ones and other users' entries, runs once a minute for the pool, and `apcuRoom()` starts it below the reserve
QC-35	In a web request, a Velocity worker included, the purge runs at the end of the request and is registered once; a script runs it at once
QC-36	One time base for the saved share: the script start (`REQUEST_TIME_FLOAT` without it) to the end the bar sets; a stale or later start gives no share; a new request measures to now again

	What it proves
QC-37	The report row says how many times as long the page would have taken without the cache, from a share of 100 % on, and carries no inline CSS
QC-39	Results kept under a state file that is gone (deleted with the cache directory) are not answered again; what is stored after is
QC-40	A write that cannot take the state lock makes everything stale and logs it
QC-41	The memo keeps at most `MEMO_MAX` results, the least recently used goes first
QC-42	The key carries the database user, port and socket: another user's rows are not answered
QC-43	PostgreSQL and SQLite reads: a backslash does not escape (`standard_conforming_strings` off: it does, read from the connection), quoted names with an apostrophe and unterminated literals are not cached, `E''` and dollar quoting are not, `$1` and `#` stay readable, their volatile functions and `'now'`
QC-44	PostgreSQL and SQLite writes: `ONLY`, `TRUNCATE` and `DROP TABLE` lists, `RENAME TO`, `INSERT OR REPLACE`, `ON CONFLICT`; `CASCADE`, `WITH`, `COPY` and a name that cannot be read make everything stale
QC-45	A lock taken on a `state.lock` that was deleted meanwhile is given up and taken on the new file (a second process, `pcntl_fork()`; skipped without pcntl)
QC-46	A persistent worker starts every request from the state on disk: another process's write counts at once, results kept under a replaced state file are not answered (also those stored in the second the request still held it), the memo is empty, a lock failure is logged once per request
QC-47	A lock failure shows in Setup > System information (`status()` `lock_failed`, from the state file) and in the debug output; a later locked write keeps it, a clear forgets it

Shell

```
php vendor/bin/phpunit --testsuite lib --filter eZDBQueryCacheTest
# OK (47 tests; QC-24, QC-33, QC-34 and QC-35 need apc.enable_cli=1, QC-45 pcntl), about 8 s (QC-08, QC-12, QC-28, QC-34 and QC-45 wait)
```

On a running site (MySQL, the real drivers), each checked on a test installation 2026-09-27:

- the driver hooks, on the live database: temporary tables never cached and their writes kept out of the state, a normal table cached;
- transactions through `eZDB::begin()`/`commit()`/`rollback()`;
- a CLI write invalidating what the web servers hold;
- the rendered page byte-identical with the cache off, cold and warm;
- the timing table above: the same pages, `Mode=off` then `Mode=shared`, anonymous and signed in, with the statements per page from the SQL profiler.

To repeat the timing on another installation: switch the profiler on (Setup > System information), and for each mode set it in `settings/override/querycache.ini.append.php`, clear the INI cache and reload PHP-FPM, request each page three times to warm it, then time it with `ab -n 80 -c 8 'https://<host>/<page>?bench=1'` (the query string keeps the HTTP cache out). The profiler's line for a probe request gives the statements; no line means none reached the database.

Files

File	What it is
`lib/ezdb/classes/ezdbquerycache.php`	the cache

File	What it is
`lib/ezdb/classes/ezmysqlidb.php`, `ezpostgresqldb.php`, `ezsqlite3db.php`	the driver hooks
`lib/ezdb/classes/ezdbinterface.php`	`COMMIT`/`ROLLBACK` hooks, the SQL profiler
`kernel/private/classes/ezpkernelweb.php`	`resetRequest()` per request
`kernel/classes/ezcache.php`	the `querycache` entry in the cache list
`kernel/setup/cache.php`, `design/admin/templates/setup/cache.tpl`	Setup > Caches
`kernel/setup/info.php`, `design/admin/templates/setup/info.tpl`	Setup > System information
`cronjobs/querycache_cleanup.php`, `kernel/private/classes/cronjobs/querycache_cleanup.php`	removes old result files (part `cache_cleanup`)
`settings/querycache.ini`	settings
`var/<site>/cache/querycache/state.ser`	the shared state
`var/<site>/cache/querycache/entries/`	the result files of the file storage

Related pages

- Behaviour changes, 16 to 30 September 2026: SQL query cache (lookup before parsing, `xxh128` keys, catalogue never cached)
- HTTP cache
- Velocity and the Velocity response cache
- Cache clears that move directories aside
- Chronicle, 16 to 30 September 2026
- Changelog 6.0.15

Exponential Copyright © 1998-2026 Exponential Foundation and 7x
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".
This documentation is free documentation, licensed under the GNU Free Documentation License 1.3 or later:
https://www.gnu.org/licenses/fdl-1.3.html.