Faster access arrays for users with many role assignments

Source file: doc/eng-us/features/6.0/access-array-prefetch.md Last changed 2026-11-04
https://doc.exponential.earth/learninminutes/exphelp/viewer/features/6.0/access-array-prefetch.md

Read this page if your installation has many users with many role assignments, for example members of dozens or hundreds of teamrooms whose member roles are assigned per teamroom subtree, and the first request of a user after a role change or a cache clear is slow. It describes how the access array of a user is built now, what it gained, and the setting that turns it off.

In short

What changed	The access array of a user is built from the rows of all their roles, loaded in three queries, and the limitations of a policy are turned into their part of the array once, not once per assignment.
Measured	A user with 600 subtree assignments (120 teamrooms × 5 member roles of 20 policies): 3,550 ms and about 48,600 queries before, 244 ms and 4 queries now (14.6×).
Result	The same access array, in the same order: the same modules, functions, policies, limitations and values.
Setting	`site.ini [RoleSettings] AccessArrayPrefetch=enabled` (default); `disabled` builds it role by role as before.
Who must act	Nobody.

When the access array is built

`eZUser::hasAccessTo()` checks every policy against the access array of the user: all policies of all roles assigned to the user and to the user's groups, each assignment with its limitation (Subtree, Section). The array is cached per user (`[RoleSettings] EnableCaching`). It is built again for each user after a role or an assignment changes, after the user cache is cleared, and when a user signs in for the first time. On an installation with 30,000 users, a change of a member role means 30,000 rebuilds, one at the first request of each user.

Why it was slow

A role assigned for 120 subtrees comes back from the database as 120 role objects, one per assignment, each with all its policies. Building the array asked the database

- once for the roles of the user,
- once per assignment for the policies of its role,
- once per policy for its limitations,
- once per limitation for its values, and once more for the limitation the assignment adds,

and turned every row into an object. For the user above that was about 48,600 queries. On a database server reached over the network every query is a round trip of some tenths of a

millisecond, so the queries alone took 10 to 25 seconds there; locally on SQLite the whole build took 3.5 seconds.

On top of that, the parts were merged one by one with `array_merge_recursive()`, which copies the growing array for every role: 74 ms of the 3.5 seconds for this user.

What it does now

1. **Three queries.** `eZRole::accessArrayByUserID()` loads the policies of all the user's roles, their limitations and the values of those limitations in three queries (`eZRole::prefetchAccessRows()`). `eZRole::policyList()`, `eZPolicy::limitationList()` and `eZPolicyLimitation::valueList()` take their rows from there while the array is built, and ask the database as before at any other time.
2. **Once per policy.** The 120 assignments of a role differ only in the limitation the assignment adds (`User_Subtree`, `User_Section`), which `limitationList()` adds as the last limitation. So the part of the array the policy's own limitations make is computed once with `eZPolicyLimitation::limitArray()` and reused for every assignment; the assignment's limitation is added to it the same way (`eZPolicy::prefetchedAccessArray()`). A policy that has a limitation of the assignment's identifier itself goes the way of `limitationList()`, which narrows such a limitation.
3. **One merge.** The parts of a role and the roles of a user are merged in one `array_merge_recursive()` call, which gives the same array as merging them one by one. This also speeds up the build with `AccessArrayPrefetch=disabled`.

The rows are kept only while the array is built and dropped afterwards, also when building it fails, so a persistent worker (Velocity) never answers a later request from them. A build inside a build (an extension that asks for the access array of another user while one is built) gets its own rows and gives the outer build its rows back.

Only the roles, policies and limitations that were loaded ahead are answered from the rows; any other one asks the database as before, also while an array is built. The limitation an assignment adds is built in memory and asks nothing (before, it was one query per policy and assignment, answered with no rows).

Each IN () list carries at most 500 ids (`eZRole::PREFETCH_IN_LIST_SIZE`): Oracle refuses more than 1000, and the lists of a user whose roles have more policies or limitations than that are loaded in several queries. All rows of one role, policy or limitation come from one query. When a query fails, the array is built role by role. On MongoDB, whose persistent layer has no IN () condition, the array is built role by role.

Database	What is asked
SQLite, MySQL/MariaDB, PostgreSQL, Oracle	`ezpolicy WHERE role_id IN (...) AND original_id = 0 ORDER BY id`, `ezpolicy_limitation WHERE policy_id IN (...) ORDER BY id`, `ezpolicy_limitation_value WHERE limitation_id IN (...) ORDER BY value`
MongoDB	role by role, as before

When a read fails

The access array fails closed. When the database answers a read with an error, what could not be read is denied:

Read that fails	What the access array gets
The roles of the user (`eZRole::fetchByUser()`)	no roles
The policies of a role (`policyList()`)	nothing from that role
The limitations of a policy (`eZPolicy::limitationList()`)	nothing from that policy (it is marked `Disabled`); before, it was unlimited (*)
The values of a limitation (`eZPolicyLimitation::valueList()`)	nothing from that policy (the limitation has `ReadFailed`)
Loading the rows ahead	the array is built role by role, which reads, and fails closed, itself

`eZRole::accessArrayByUserID()` logs one error per build with the number of failed reads and the first of them. `eZUser` does not store a user cache built with a failed read (`eZRole::$accessReadFailures` counts them), so the denial lasts only for that request and the next one builds the array again. MongoDB reads are not covered: its persistent layer answers a failed read with no rows.

Measurements

Built 3 times each on a local installation (SQLite, PHP 8.4), for a user to whom 5 member roles of 20 to 22 policies (2 to 4 limitations each, with 2 to 4 values) are assigned for 120 subtrees each, 600 assignments:

Way	Time per build	Queries
Before: role by role, merged one by one	3,550 ms	about 48,600
Role by role, merged once (`AccessArrayPrefetch=disabled`)	3,140 ms	about 48,600
Rows ahead, limitations once per policy, merged once (`enabled`)	244 ms	4

Where the time of the build goes for this user, with the rows loaded ahead but the limitations still computed per assignment: 2 ms for the 600 role objects, 3 ms for the three queries, 651 ms for building the parts of the 600 assignments (the step the reuse per policy removes), 74 ms for merging one by one (1.4 ms in one call).

For users with few roles the gain is smaller: the anonymous user of a demo installation (one role, 12 policies) needs 4 queries instead of 25 and is built 2.6 times as fast.

An alternative was tried and not taken: one SQL query joining roles, policies, limitations and values, with the access array assembled from its rows by code of its own (as an older project patch did). It took 862 ms for the user above, transferred about 67,800 rows, and gave a different access array as soon as a policy carried a limitation of the assignment's identifier: a second implementation of the permission rules drifts from the first.

The same access array

The array is the same as before, in the same order (compared with `===`). The rows loaded ahead are read in the order the methods that ask one by one read them: policies and limitations by id, the values of a limitation by value (the default sort of `eZPolicyLimitationValue`). The objects are built from them as from the database, the methods that turn them into the array are the same, and a policy whose own limitation has the

assignment's identifier is built by `limitationList()` as before.

On the demonstration server (SQLite) every user, every user or group with a role of its own, and throwaway roles with a policy of every limitation kind were built both ways, by the code before the change and by this one, and compared with `===`: 31 subjects, all identical. The throwaway roles carried Class, Section, Owner, Group, Node, Subtree, Language, a state group, ParentOwner, ParentGroup, ParentClass, ParentDepth, SiteAccess, a limitation of an extension (as content limitation handlers add), User_Subtree and User_Section on the policy itself, a limitation without values, values out of order and repeated, a temporary editing copy of a policy, unlimited and `*` policies, and a role of 520 policies; they were assigned plainly, for 40 subtrees and for sections, to a user, to groups, to a nested group and together with the groups of the anonymous user.

Built on the demonstration server (SQLite, PHP 8.5), query cache off, fastest of 5 builds	Before: queries, time	Now: queries, time
Administrator (1 role)	3, 0.2 ms	3, 0.2 ms
Anonymous (1 role)	27, 1.3 ms	4, 1.6 ms
An editor (3 roles)	156, 12.5 ms	4, 3.0 ms
User in groups, 50 assignments of 2 roles	1,843, 130 ms	4, 15 ms
520 policies assigned 41 times	84,802, 7,057 ms	7, 850 ms
All 31 subjects	177,651, 14.3 s	104, 1.6 s

Settings

File	Block	Key	Default	Meaning
`settings/site.ini`	`RoleSettings`	`AccessArrayPrefetch`	`enabled`	`disabled` builds the access array role by role, asking the database for each role, policy and limitation.

Both ways give the same array, so the user caches built by one stay valid under the other.

Tests

- `eZPolicyPrefetchedAccessArrayTest` (no database): the part of a policy computed once and reused gives the same array as `limitationList()` from the same rows, unlimited, with limitations, assigned for a subtree or a section, for a policy with a limitation of the assignment's identifier, and merging in one call gives what merging one by one gave. A policy or limitation that was not loaded ahead asks the database; the limitation an assignment adds asks nothing; a build inside a build gets the outer one its rows back; the IN () lists stay below Oracle's limit.
- `eZRoleAccessArrayPrefetchLiveTest` (on an installation): both ways for every user and user group, identical with `===`, with a role of its own assigned for 40 subtrees and for a section; the loaded rows are dropped afterwards.
- `eZAccessArrayFailClosedTest` (no database): a stub answers each read with an error; roles, role ids, policies, limitations and values that cannot be read deny, with the rows loaded ahead and role by role, and nothing is counted when every read answers.

Related pages

- Content policy limitations of extensions
- Velocity response cache

> **Become an Exponential Partner**
>
> Partners share their information, provide funding, contribute back and develop the products with us, and that is how Exponential grows. A partner website and dashboard are planned for mid-2027: please become a partner today to speed up our growth rate and land new features faster.
>
> Partners: start here | How to apply

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.