Runtime Infrastructure - Bug #12161

Live memory allocation optimizations.

11/12/2026 10:35 AM - Stefanel Pezamosca

Status:	Review	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Stefanel Pezamosca	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**reviewer:**	Alexandru Lungu, Constantin Asofiei
vendor_id:	GCD	**production:**	No
case_num:		**env_name:**	
version_reported:		**topics:**	
version_resolved:			

Description

Related issues:

Related to Runtime Infrastructure - Bug #12168: Reduce FWD server heap: read ...	**Test**

History

#1 - 11/13/2026 02:32 AM - Stefanel Pezamosca

Server memory allocation: overview

I found three sources of avoidable server allocation and fixed them in three patches. Serban's load test of the fixes on application B uncovered a fourth, larger one, which is not fixed yet. The next five notes give the details, one topic per note.

#	Issue	Size	Status
1	FastFindCache temp-table maps created at the sizes configured for permanent databases	application A: idle server allocation fell about 70 % with this fix	fixed
2	PreselectQuery FQL text built and copied several times per execution	application A: FQL assembly was 22.7 % of the idle polling session's allocation; the fix cut FQL assembly by 47 %	fixed
3	NOW builds calendars and reads the clock several times	about 1,840 bytes per datetimetz.now() cut to 104; small in both applications	fixed, with a behaviour change
4	TextOps ENTRY / NUM-ENTRIES copy the whole list on every call	application B: about 10.0 GB of 19.3 GB in one load test run (52 %)	not fixed

Where the figures come from

- **Application A** (customer application): I profiled it with no users, where only its background polling session was active (a batch process that polls a task queue about once a second), and under a REST load test. Issues 1-3 come from here. With the whole patch the REST load test allocated about 60 % less (107 GiB against 265 GiB unpatched, in two passes), almost all from issue 1.
- **Application B** (another customer application): Serban tested the patch with a GUI load test of 5 parallel users. The patched code is under 0.5 % of the allocation there, so the test showed no measurable change. Its profile uncovered issue 4.
- **Probes**: afterwards I measured issues 1-4 again in isolation. Java probes call the FWD classes directly. 4GL probes ran on a test server (H2) with the unpatched and the patched jar and, for behaviour, on OE. Both kinds read the JVM's exact per-thread allocation counter.

Patches

The three fixes are three patches against 12161a, one per issue. I will commit them one after the other: 12161a_1_fastfindcache.patch, 12161a_2_fql_assembly.patch (with its unit tests) and 12161a_3_now.patch (with its unit test).

Notes in this series

1. Overview (this note)
2. Issue 1: FastFindCache temp-table maps #12161-2
3. Issue 2: FQL text assembly #12161-3
4. Issue 3: NOW, TODAY, MTIME, TIME, TIMEZONE #12161-4
5. Issue 4: TextOps ENTRY / NUM-ENTRIES (not fixed) #12161-5
6. Other proposed changes (not implemented)

#2 - 11/13/2026 02:36 AM - Stefanel Pezamosca

- File 12161a_1_fastfindcache.patch added

Issue 1: FastFindCache temp-table maps use the sizes configured for permanent databases (fixed)

Where directory.xml sets large L2 / L3 cache-size values for FastFindCache, every temp-table scope that caches a FIND result allocates large hash tables and throws them away: 576 KB per scope with application A's values (L2=10000, L3=100000). With the fix, application A's idle server allocates about 70 % less. Application B sets no such values and is not affected.

Applications A and B, the load tests and the probes are described in the overview note.

Problem

FastFindCache keeps, per table, an L2 map from index to L3 map; each L3 map holds the cached FIND results of one index. putImpl creates these maps with the L2 / L3 cache-size values as their initial capacity (the defaults are 10 and 100). The FastFindCache instances for temp-tables, one per session, use the same values. A temp-table's maps are dropped each time its multiplex scope closes (TemporaryBuffer.doCloseMultiplexScope()), that is, at the end of the life of one temp-table instance. So each temp-table scope that caches one FIND result allocates a 64 KB L2 table and a 512 KB L3 table, then discards them.

L3 used to be an LRU limit that allocated almost nothing up front. Trunk r16106 (#10337) turned it into a HashMap initial capacity and moved the limit to ReverseLookup.lruCache.

Fix

The temp-table instances of FastFindCache read their sizes from separate cache-size entries, TT-L2 / TT-L3. These default to 10 and 100 and never fall back to L2 / L3. Permanent databases are unchanged: they still use L2 / L3.

Results

Measurement	Unpatched	Patched
Application A, idle server: allocation rate (growth of the young generation over 180 s)	1.55 MB/s	0.46 MB/s
Application A, REST load test: allocation in two passes	265 GiB	107 GiB
Application A, REST load test: share of allocation samples under putImpl	38-46 %	0.3 %
Application A, REST load test: share of CPU samples in FastFindCache.invalidate at temp-table scope close	10-13 %	under 0.5 %
Application A, REST load test: time per pass of the one test where invalidate was 40-50 % of the CPU samples	15.9 s	10.8 s
Probe: 3,000 temp-table scopes with L2=10000 / L3=100000, allocation	1,841 MB	157 MB
Probe: the same, time per run after warm-up	about 600 ms	about 185 ms
Probe: the same with no cache-size configured, allocation	155 MB	154 MB

- The idle server figure is with this fix alone. The REST load test figures are with the whole patch; almost all of the saving is under putImpl.
- The REST load test times are the mean of six passes on one server run per jar; passes of the same jar varied by about 8 %. The other two timed tests of that load test did not change.

#3 - 11/13/2026 02:41 AM - Stefanel Pezamosca

- File 12161a_2_fql_assembly.patch added

Issue 2: PreselectQuery builds and copies the FQL text several times (fixed)

Each query execution built FQL text that was copied again, plus an inverted copy that only backward scrolling reads. The fix removes 20 % to 47 % of what FQL assembly allocates, depending on the workload, and the generated FQL text is unchanged.

Applications A and B, the load tests and the probes are described in the overview note.

Problem

PreselectQuery.assembleFQLImpl does this on every query execution:

- It copies the whole statement into a second StringBuilder with the ORDER BY reversed. Only ProgressiveResults reads that copy, and only when the query scrolls backward.
- SortCriterion.toSortExpression rebuilds its ORDER BY text on every call, although all its fields are final and most instances come from a shared cache.
- FQLExpression.toFinalExpression() builds a String that is then copied into a buffer that starts at 16 characters and grows by copying.

Fix

- PreselectQuery builds the inverted FQL on first use: AssembledFql, a Supplier<String> that ProgressiveResults calls only when scrolling backward.
- SortCriterion caches the text of toSortExpression(false) and toSortExpression(true).
- The new FQLExpression.appendFinalExpression() writes into the caller's buffer, which assembleFQLImpl sizes once using finalLength().

Results

Measurement	Unpatched	Patched	Change
Application A, idle polling session: allocation samples under assembleFQL in 10 minutes	226	119	-47 %
Application A, idle polling session: all allocation samples	997	903	-9 %
Application A, REST load test: allocation under assembleFQL	not recorded	not recorded	about -40 %
Application B: allocation under assembleFQL	58.6 MB	46.0 MB	about -20 %
Probe: 9,000 query executions, allocation (two runs each)	394.0 and 390.8 MB	371.0 and 368.8 MB	-5.7 % (2.6 KB per execution)

- The application A idle figures were measured with the fix for issue 1 already applied. Before this fix, FQL assembly was 22.7 % of that session's allocation. The REST load test figure is with the whole patch.
- In application B the saving is under 0.1 % of the server's allocation.
- In the probe the saving is about a third of what FQL assembly allocates. The run time does not change.

Checks that nothing else changed

- **Same FQL text**: a temporary check rebuilt the text with the old code on every execution and found 0 differences in about 9,000 executions on H2 and 54,000 on PostgreSQL (application A's idle polling session). The inverted text was read, and so compared, only 4 times, all on H2.
- **Same behaviour**: a 4GL test that runs GET LAST / PREV / FIRST / NEXT on 7 scrolling queries gives the same output on OpenEdge and on FWD before and after the fix.

I don't think this change has a significant impact, so I'll leave it up to you to decide whether to drop it entirely, keep only part of it, or leave it as is.

#4 - 11/13/2026 02:44 AM - Stefanel Pezamosca

- File 12161a_3_now.patch added

Issue 3: NOW, TODAY, MTIME, TIME, TIMEZONE (fixed, with a behaviour change)

datetimetz.now() allocated about 1,840 bytes per call for work it did not need; it now allocates 104. The fix also makes TIME follow SESSION:TIMEZONE, as OpenEdge does. The gain is small in both applications, because they rarely call these built-ins.

The applications, the load tests and the probes are described in the overview note.

Problem

- datetimetz.now() builds up to two GregorianCalendar instances, wraps the time and the offset in decimal (BigDecimal) values only to read them back, and reads the clock several times.
- Separately, TIME (date.secondsSinceMidnight()) ignores SESSION:TIMEZONE. OpenEdge follows it.

Fix

- NOW, TODAY, TIME, MTIME and TIMEZONE each read the clock once.
- The offset for that instant comes from SESSION:TIMEZONE or from TimeZone.getOffset() (new date.currentOffset(long)), and the day, time and offset are set directly.

Behaviour change

- TIME follows SESSION:TIMEZONE, as on OpenEdge.
- When SESSION:TIMEZONE is not set, the patch takes the JVM zone's offset from TimeZone.getOffset() instead of adding the raw offset and the DST savings. The two can differ in some zones. I do not cover that change in this note.

Results

Bytes allocated per call (Java probe; two runs agree to within 1 %):

Built-in	Unpatched	Patched
NOW as DATETIME-TZ (datetimetz.now())	about 1,840	104
NOW as DATETIME (datetime.now())	about 390	104
TIME	488	105
MTIME	about 430	96
TODAY	96	96
TIMEZONE	96	96

- datetimetz.now() takes about 890 ns unpatched and 55-95 ns patched.
- Server probe: 200,000 assignments of NOW to a DATETIME-TZ, then 200,000 each of MTIME, TIME and TODAY, allocate 387.5 MB unpatched and 20.0 MB patched. This is less than 200,000 times the per-call figures above; I did not look into why.
- Behaviour: a 4GL probe of these built-ins, with SESSION:TIMEZONE unset and set to nine offsets, prints the same 144 lines on OpenEdge and on patched FWD. On the unpatched jar the 9 TIME lines differ.
- Neither load test calls these built-ins often enough to show the change.

#5 - 11/13/2026 02:48 AM - Stefanel Pezamosca

Issue 4: TextOps ENTRY / NUM-ENTRIES copy the whole list on every call (not fixed)

This is the largest allocation found: about 10.0 GB of 19.3 GB in one load test run of application B (52 %). ENTRY and NUM-ENTRIES allocate in proportion to the length of the list, not to the entry asked for. A prototype without the copies allocates 64 times less when walking a 100-entry list.

Application B, its load test and the probes are described in the overview note.

Problem

TextOps.entry goes through splitInt(String src, char delim, int element, boolean onlyEntry), which does this on every call:

```
char[] srcChars = src.toCharArray();      // copy of the whole list, 2 bytes per character
int[] subLengths = new int[srcLen + 1];   // 4 bytes per character of the list, not per entry
```

In the figures below, L is the length of the whole list in characters and n is the number of entries in it. "6·L bytes" means 6 bytes for every character of the list. Example: a list of 100 entries that is 1,000 characters long costs about 6,000 bytes per ENTRY call, 2,000 bytes per NUM-ENTRIES call, and 800,000 bytes for a loop over all 100 entries.

- One ENTRY call on a list of L characters allocates about 6·L bytes, whichever entry is asked for.
- numEntriesOf makes the same toCharArray() copy only to count delimiters: about 2·L bytes.
- OpenEdge evaluates the TO expression of a DO loop on every iteration, and FWD does the same. So a loop of ENTRY(i, list) for i = 1 TO NUM-ENTRIES(list) over n entries allocates about 8·n·L bytes.
- The full-split overload splitInt(String, char) and splitIntToChar have the same int[srcLen + 1] pattern.

ADM2 code has this loop, for example DO iField = 1 TO NUM-ENTRIES(cColumns) in assignDBRow of adm2/query.p. In application B, ENTRY and NUM-ENTRIES calls from three routines of that file (bufferCopyDBToRO, assignDBRow, bufferCompareDBToRO) account for at least 7.4 GB, 38 % of the server's allocation.

Measured with probes

- ENTRY, per call, for lists of 99 to 9,999 characters: 6·L + 166 bytes, whether the first, middle or last entry is asked for.
- NUM-ENTRIES, per call, same lists: 2·L + 58 to 74 bytes.
- TO expression: a 4GL probe with a counting function as the limit of a 5-iteration DO i = 1 TO loop reports 6 evaluations on OpenEdge and on FWD.
- 4GL probe on the test server: walking a list of 100 entries (999 characters) 2,000 times allocates 1,585 MB, which is 8.1 KB per ENTRY and NUM-ENTRIES pair.

Proposed fix

- ENTRY: scan with String.indexOf(delim, from) and return src.substring(start, end).
- NUM-ENTRIES: count with indexOf.
- Neither needs a copy of the list. The existing special cases must stay: empty list, null-character delimiter, out-of-range entry.

A prototype of this, in the probe only and not in FWD, returns the same value as TextOps for every entry of four lists (10 to 1,000 entries). Walking the 100 entries of the 999-character list (one NUM-ENTRIES and 100 ENTRY calls) allocates 9.4 KB instead of 604 KB.

This will be WIP.

#6 - 11/13/2026 09:50 AM - Șerban Bursuc

- Related to Bug #12168: Reduce FWD server heap: read DMO @Property without reflection, share proxy Method objects added

#7 - 11/16/2026 03:45 AM - Stefanel Pezamosca

- Assignee set to Stefanel Pezamosca

- Status changed from New to WIP

I have created task branch **12161a** from trunk rev. **16850**. I'll proceed with applying the attached patches one by one with a review in between.

#8 - 11/16/2026 08:20 AM - Stefanel Pezamosca

- % Done changed from 0 to 70

Committed to branch 12161a (r16851 to r16854, trunk base r16850):

- r16851 FastFindCache: temp-table instances size their L2/L3 maps from the TT-L2/TT-L3 cache sizes (default 10 and 100) instead of the permanent-database L2/L3 sizes.
- r16852 PreselectQuery, FQLExpression, ProgressiveResults: the inverted FQL is built only when a backward scroll reads it, and the FQL text is assembled in one presized builder without intermediate copies. The generated FQL, forward and inverted, is the same as trunk's.
- r16853 date, datetime, datetimetz: NOW, TODAY, TIME, MTIME and TIMEZONE read the clock once without calendars. The offset is SESSION:TIMEZONE when set, otherwise the JVM zone's real offset at that instant. TIME now follows SESSION:TIMEZONE, as OE does.
- r16854 SortCriterion, PreselectQuery, FQLHelper, QueryComponent: sort criteria append their ORDER BY text directly into the caller's builder, without a temporary string per criterion.

#12161-5 is WIP.

#9 - 11/18/2026 06:24 AM - Stefanel Pezamosca

- % Done changed from 70 to 90

Committed to branch 12161a, r16855 and r16856 (trunk base r16850). r16855 fixes Issue 4 (#12161-5). r16856 holds the fixes from code review; the main one changes the default offset a DATETIME-TZ gets, to match OpenEdge.

- r16855 TextOps, Text: entry(), numEntries() and Text.replaceEntry scan the list for the entry instead of splitting it into a copy, which was the problem described in #12161-5.
- r16856 FastFindCache: the L2/L3 map factories are built once per instance instead of on every put().
- r16856 TextOps: splitInt(String, char, int, boolean) is documented as public API; it has no caller in FWD now.
- r16856 date, datetimetz: with SESSION:TIMEZONE unset, a DATETIME-TZ built from a DATE or DATETIME gets the offset that the zone's current DST rules give at the value's local date and time of day.

DATETIME-TZ default offset
Fix: the new date.currentRulesOffset(LocalDateTime) builds the date's year's transitions from the zone's current rules (ZoneRules.getTransitionRules()). A zone with no recurring rules (a fixed offset, or explicit transitions only, such as Africa/Casablanca) uses its offset at the same month, day and time of the current year. date.getDefaultOffset(Date) calls it at local midnight (a DATE assigned to a DATETIME-TZ, a date-only string). datetimetz.getDefaultTimezoneOffset(date) calls it at the value's time of day (DATETIME-TZ(d, mtime), strings with a time, DATETIME operands). SESSION:TIMEZONE, when set, still wins, and NOW, TODAY, TIME, MTIME and TIMEZONE are unchanged.

#10 - 11/18/2026 10:25 AM - Stefanel Pezamosca

- % Done changed from 90 to 100

- Status changed from WIP to Review

Please review 12161a, r16851 to r16859 (trunk base r16850). It now holds only the allocation optimizations: r16857 to r16859 return the offset rules of NOW, TODAY, TIME, MTIME, TIMEZONE and the DATETIME-TZ defaults to trunk's, so the branch changes no date or time result. The OpenEdge compatibility fixes for those offsets (#12161-4, #12161-9) are in a separate patch, outside this branch.

Branch content

- #12161-2 FastFindCache: temp-table instances size their L2/L3 maps from TT-L2/TT-L3 (default 10 and 100); the map factories are built once per instance.
- #12161-3 PreselectQuery, FQLExpression, ProgressiveResults, SortCriterion, FQLHelper, QueryComponent: FQL text assembled in one presized builder, inverted FQL built only on backward scroll, sort criteria appended directly into the builder.
- #12161-4 date, datetime, datetimetz: NOW, TODAY, TIME, MTIME and TIMEZONE read the clock once, without calendars, with trunk's offset rules.
- #12161-5 TextOps, Text: ENTRY, NUM-ENTRIES and Text.replaceEntry scan the list instead of copying it.

Some results from the load tests:

Run	Server allocation	TextOps allocation	Young GCs	GC pauses
Unpatched, cold	28,675 MB	12,677 MB	13	828 ms
Patched, cold	17,774 MB	1,717 MB	10	568 ms
Unpatched, warm	20,933 MB	12,444 MB	6	198 ms
Patched, warm	10,316 MB	1,649 MB	3	163 ms

- Server allocation falls 51 % warm and 38 % cold, almost all from ENTRY / NUM-ENTRIES: TextOps.splitInt (9.0 GB warm) and numEntriesOf (1.5 GB) are gone; the new scan allocates 67 MB.
- The heap after GC is unchanged (lowest about 2.5 GB cold, 3.0 GB warm): the branch removes garbage, not retained data.
- The other changes are under about 100 MB per run in these flows, within sampling noise; their effect is measured in #12161-2 to #12161-4 (the NOW figures with r16853's offset rules). r16856 to r16859 were not measured in these flows.
- Largest remaining allocators, not changed here: H2 temp-table rows (Table.getTemplateRow, 10.8 %) and TextOps.concat (10.1 %, mostly ADM2 assignDBRow).

#11 - 11/18/2026 10:26 AM - Stefanel Pezamosca

- reviewer Alexandru Lungu, Constantin Asofiei added

#12 - 11/19/2026 03:32 AM - Stefanel Pezamosca

Branch 12161a was rebased to trunk revision 16857. The new branch revision is 16866.

#13 - 11/23/2026 08:37 AM - Constantin Asofiei

I'm OK with the changes.

Files

12161a_1_fastfindcache.patch	4.72 KB	11/13/2026	Stefanel Pezamosca
12161a_2_fql_assembly.patch	38.9 KB	11/13/2026	Stefanel Pezamosca
12161a_3_now.patch	30.1 KB	11/13/2026	Stefanel Pezamosca