Database - Bug #12020

Replace AdaptiveQuery dynamic mode with PagedResults

10/19/2026 07:10 AM - Alexandru Lungu

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Artur Şcolnic	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**reviewer:**	
vendor_id:	GCD	**production:**	No
case_num:		**env_name:**	
version_reported:		**topics:**	
version_resolved:			

Description

History

#1 - 10/19/2026 07:30 AM - Alexandru Lungu

- Assignee set to Alexandru Lungu

This task is about a proposal to replace the invalidation of AdaptiveQuery.

Today:

- preselect mode of AQ is using a ProgressiveResults to gather rows in increasingly larger brackets
- if a change in the walked index occurs, the AQ will delegate a RAQ to retrieve records one by one (using the FIND NEXT implementation)
- if the value of the column that occurs first in the index is different than the baseline, then the AQ is revalidated and a f1 > <break-value> is used.

The issues of this implementation is that:

- dynamic mode always requires rows one by one.
- revalidation may never occur (e.g. where f1 = 2025 and (f1 = 2025 and f2 > ?)). The sort-band in question (i.e. f1 = 2025) can refer to thousand of records. This is a customer example that iterates all records operated in a certain year. It revalidates only when the year is changed, but this never happens as the WHERE clause enforces a specific where
  - disclaimer, this specific case is optimized through fqlpreprocessor and we can revalidate on the second component of the index if the first component is fixed in WHERE. However, this happens only if the WHERE clause is trivial (conjunction of equalities, so no functions, or operators, etc.).
- ProgressiveResults is a performance bomb. Once the i-th bracket is reached, the query emitted has LIMIT N OFFSET M. That M of the offset is the number of rows skipped **from the beginning of the index/table**. Thus, the query iterates each row a logarithmic number of times.
- ProgressiveResults are hard to invalidate due to cross-session changes. The only guardrail we have is that the last record in a bracket should match the first record of the next bracket. However, it doesn't pro-actively invalidate if a cross-session change occurs while still iterating the bracket.
  - If another session inserts a record right at the beginning, the ProgressiveResults will simply invalidate because the offsets won't match anymore. However, that newly inserted record wasn't going to affect the results that were still to be traversed. This happens because at the offset checking time, we don't have much feedback on how the new/updated/deleted rows affected the index.

My suggestion:

- The PagedResults is a parallel implementation very similar to ProgressiveResults, but it uses a reference row instead of an OFFSET. Thus, instead of emitting LIMIT N OFFSET M to skip the first M rows, it emits WHERE .... AND (<indexed-fields> > <reference-indexed-fields) LIMIT N. The additional clause (fql bundle) is made out of several statements, based on index size. Due to null augmentation, some of the fqls also have f1 > ? or f1 is null.
  - This is only used in NO-LOCK, forward-only or iterating FOR EACH queries in TCP/IP connection mode to emulate prefetching.
  - Current implementation uses BASE 16 and progressivelly increments it.
  - It **resets** the page size back to 16 if another session commits, being proactivelly in invalidating and seeing only the records meant to be seen by the query.
  - Shared-memory database can use PagedResults with **BASE set to 1**.
    - It will work as ProgressiveResults under no cross-session invalidation.
    - It will work as an automatically revalidated query after cross-session invalidation (i.e. reset bracket size to 1).
    - There is no point in distinguishing cross-session commits and intra-session flushes, thus there is no point is pro-actively invalidating

and using a RAQ when the same session commits.

Dirty-share:

- The AQ should still invalidate to RAQ in order to leverage the dirty-share implementation. For the moment, we can distinguish intra-session invalidation due to physical DB update or dirty-share update.
  - in case the change is flushed to the db, it is fine just to reset the PagedResults bracket size back to 1.
  - in case the change was put in dirty-share, then we really need to invalidate.
    - this second technical dept will be mitigated by #8388, because we will be able to pick dirty-changes directly from the DB using PagedResults.

End-game:

- All FOR EACH queries will use PagedResults and will simply reset the bracket size to BASE:
  - BASE is 1 in shared-memory DB
  - BASE is 16 (or computed from -MM) in TCP/IP DB

Performance:

- The PagedResults has slower plans, but faster invalidated modes, both intra- and cross-.
- The ProgressiveResults has faster plans **for small result-sets**, but slower in **larger result-sets** and invalidated modes, both intra- and cross-.
- I think the cross-session invalidation on PagedResults can be improved by not invalidating on any foreign commit, but only on foreign commits that touch the target tables. This is not implemented because of the machinery required (needing to monitor what tables were affected in a foreign commit).
- I think the OR bundled will be eventually optimized by the effort in #11980

Please provide feedback

#3 - 10/29/2026 07:31 AM - Alexandru Lungu

- Assignee changed from Alexandru Lungu to Artur Școlnic

Artur, please explore this idea.

#4 - 10/29/2026 07:42 AM - Artur Școlnic

- Status changed from New to WIP

#5 - 10/30/2026 04:21 AM - Artur Școlnic

I prototyped the proposal (temporary trunk patch, since removed): when AdaptiveQuery goes into dynamic mode on a paging-eligible single-table NO-LOCK walk, it continues with a PagedResults anchored after the current row (the pre-change copy if the walk moved its own row) instead of a RAQ; a later own change drops the page and re-anchors. Compared against trunk and against OpenEdge 12.8 client/server over TCP (-H localhost -S 20991 -Mm 1024, default prefetch).

Walk: FOR EACH dsNonunique WHERE si1 = "PW" NO-LOCK (si1 pinned, as in the customer case). All runs repeated 2-3 times with identical results.

Cost, 20000 rows; "dynamic" = the walk changes an index field of another row on its first row:

Walk	trunk (RAQ dynamic)	paged dynamic, page 16	paged dynamic, frozen page 64
dynamic mode, quiet	69-115 s	58-127 ms	3.0-3.2 s
dynamic mode, another session committing to an unrelated table	71-74 s	5.1-5.3 s	3.1-3.3 s
preselect, quiet	138-208 ms	55-212 ms	3.0-3.3 s
preselect, another session committing	71-103 ms	5.2-5.8 s	3.0-3.5 s

Results, 60 rows, paused on row 5:

- Own changes (create/delete/update/move ahead and behind, move of the current row, inside an outer transaction): paged dynamic mode gives the same rows as the RAQ in all 10 scenarios.
- Foreign commits while paused: OE over TCP froze the whole 60-row remainder (one prefetched message). Trunk and paged page 16 are fresher than OE (they see a change to row 40). A frozen 64-row page matches OE there, but misses rows appended past the end, because it takes a short page (60 < 64) as the end of the walk where OE asks the server once more.

Conclusions:

- Replacing the RAQ with PagedResults is the big win: ~1000x on a long walk, same results on own changes.
- Shrinking the page on any foreign commit should go: it does not match OE over TCP (which ignores commits inside its window) and costs ~5 s per 20000 rows under load, because foreignCommitCount() is per database.
- Candidate design: first page frozen at -prefetchNumRecs (OE parity), geometric growth after it, no shrinking, and one extra query on a short page before ending the walk.
- The "flushed: reset the page, dirty only: invalidate" rule cannot be keyed off AdaptiveQuery.stateChanged: same-session updates and inserts arrive there before they are flushed. Re-anchoring at the next page open, as in the prototype, avoids that.
- Still RAQ: locked walks, joins, client-side WHERE (not paging-eligible). Shared memory not measured, I believe it is not 'in scope' at all here.

Alex, please review the summary of the design below.

Design: PagedResults as AdaptiveQuery's dynamic mode

When a forward NO-LOCK walk is invalidated by a change made in its own session, continue it with reference-row pages instead of a RAQ. Match OpenEdge client/server (TCP) where measured, and do not slow down when other sessions commit.

Why (20000-row walk, FWD trunk):

- Dynamic mode today (RAQ, one FIND NEXT per row): 69-115 s. Preselect: 0.1-0.2 s.
- The prototype (paged dynamic mode): 58-127 ms, same rows as the RAQ in all 10 same-session scenarios.
- OE over TCP (-Mm 1024) delivered the whole remainder of a 60-row walk from its prefetched message: commits by other sessions inside that window were not seen.

Scope: queries AdaptiveQuery.pagedResultsConfig() already accepts (one table, NO-LOCK, forward-only or iterating, no client-side WHERE, no external buffers, no join). Locked walks, joins and client-side WHERE keep the RAQ. Shared-memory databases are not changed until measured.

Page policy

- First page: W rows, W = prefetch-num-recs (legacy -prefetchNumRecs), default 64 for 12.x (to be confirmed on OE).
- Later pages grow geometrically from W, up to the existing memory cap.
- No shrinking or cutting a page because another session committed: remove capPageOnForeignCommit and the reset in resizePage. Today this costs 5-6 s per 20000 rows under load and does not match OE.
- A short page does not end the walk: run the cascade once more from the last row; only an empty result ends it. Without this, a 64-row page missed rows appended past the end, which OE delivered.

Dynamic mode

1. On a same-session invalidation, create a PagedResults instead of the RAQ (createSimpleQuery).
2. Anchor it after the walk's current row; if the change modified that row, anchor on the event's pre-change snapshot, so the walk continues from the old key.
3. Later same-session changes drop the current page and re-anchor; the next page is read after the flush, on the same connection.
4. No "flushed vs dirty" decision in stateChanged: updates and inserts arrive there before they are flushed.
5. No break-value revalidation for these queries: paged dynamic mode costs the same as preselect.

Code changes

Class	Change
AdaptiveQuery	execute() builds PagedResults instead of DynamicResults/RAQ for eligible queries; stateChanged() keeps the anchor and calls dropPage(anchor) in paged dynamic mode.
PagedResults	anchorAfter(Record), dropPage(Record); first page W thoreign commits; short page re-queries once.
PagedResultsConfig	prefetch-num-recs default 64.

Persistence	foreignCommitCount() no longer used by paging.

Validation: tests/concurrency/paged_walk (10 foreign-commit scenarios) an_change@ (10 own-change scenarios) on FWD and OE over TCP, plus@tests/concurrency/paged_walk_bench@. Confirm W on OE with a 200-row seed and a pause after row 70.

Known differences that remain

- OE over TCP does not show its own inserts or key moves inside the current does today).
- Pages after the first are wider than OE's W-row messages, so deep in a walk FWD can be staler than OE. Not measured on OE; keeping every page at W costs ~3 s per 20000 rows instead of ~0.1 s.
- Dirty share was not exercised by any scenario.

#6 - 10/30/2026 05:09 AM - Alexandru Lungu

> When a forward NO-LOCK walk is invalidated by a change made in its own session, continue it with reference-row pages instead of a RAQ. Match OpenEdge client/server (TCP) where measured, and do not slow down when other sessions commit.

This is the design, indeed. But the tough nut here is the dirty-share. We can't abandon it under no circumstances.

> Candidate design: first page frozen at -prefetchNumRecs (OE parity), geometric growth after it, no shrinking, and one extra query on a short page before ending the walk.

Mind that this is the marginal case of using TCP/IP. The first and foremost line of development here is shared memory used by most customers. Supporting shared memory is more faithful than TCP/IP than vice-versa (a customer using a legacy TCP/IP connection is more lenient with FWD in shared memory than legacy shared memory using TCP/IP in FWD). This is why I want a shared memory first analysis.

> No shrinking or cutting a page because another session committed: remove capPageOnForeignCommit and the reset in resizePage. Today this costs 5-6 s per 20000 rows under load and does not match OE.

I don't understand this. Why would we remove this ... wouldn't it functionally mismatch OE? If the page progressively increased to a 64.000 sized page and the cursor is still in the first half of the page, it will take another 32k records to iterate to see the committed changes of another session. In OE, they are visible after at most 64 iterations. That is why we shrink the page back when another session committed.

> A short page does not end the walk: run the cascade once more from the last row; only an empty result ends it. Without this, a 64-row page missed rows appended past the end, which OE delivered.

Reasonable!

> OE over TCP does not show its own inserts or key moves inside the current does today).

I think this can be easily fixed. Instead of the foreign commit counter, we will drive the same-session changes (stateChanged) so simply invalidate the PagedResults just like foreign commit does.

> Pages after the first are wider than OE's W-row messages, so deep in a walk FWD can be staler than OE. Not measured on OE; keeping every page at W costs ~3 s per 20000 rows instead of ~0.1 s.

Are wider to improve performance, but **do not affect functionality**. Any foreign commit will bounce back the page to the BASE size (currently defaults to 16). I agree that OE doesn't get results in incrementally larger page, but FWD will trim the large page on the first invalidation, so it will "emulate" single sized pages. As long as the batch size is not fully deterministic (at least in FWD we don't compute the actual page size dynamically), there is an epsilon to be considered (i.e. there might be a couple of rows more or less in the window of FWD than OE; this invariant allows us to use incremental pages and bounce back to 16).

> Dirty share was not exercised by any scenario.

#8388 is a prerequisite of this task. PagedResults shall be emitted with UNION ALL just like RAQ does in #8388. This way, we never invalidate, but rather send complex queries once an invalidation occurs. As we progress further, the active bundle will get less and less complex - re-validating implicitly.

I think there are multiple phases for this task that will be adopted in trunk one-by-one:

Implementation of a per-table foreign commit counter

We need a **powerful** and **safe** foreign commit counter in the FWD server. We need to create this mechanism and test the hell out of it.
The effort from 10806a and #10222 will be wrapped-up in one single powerful ForeginCommitCounter class that will handle this mechanism (or something similar to this.

To make use of it, we will:

- replace the PagedResults foreign commit counter with the per-table one **and merge it like that**.
- leverage it to fix #10222 and invalidate FFC

To test:

- we need to assess cluster mode very carefully to check if it is scalable.
- we need to test multi-tenancy and ensure foreign commit are broadcast correctly intra-tenant in private vs shared mode
- we need to do memory leak tests to ensure that every one commit will accumulate some state in the FWD server.

Goal: we implement the per-table foreign commit counter mechanism, test functionality/performance/ memory and merge to trunk.

PS: Mind that we already have the global events that are broadcast cross-session. This was used in dirty-share, but afaik it is completely disabled in most/all environments. Lets see if we either leverage that or purge that. I don't want both such events being broadcasted **and** a separate foreign commit mechanism in place. If we need to keep the global events **only** for cross-session dirty-share, so be it. Lets document it that way and keep it as such.

Make AdaptiveQuery sensitive to other session commits

- preferably #10806 will be done by this time

This is basically integrating the foreign commit logic in AQ invalidation. At this time, RAQ is used and not PagedResults.
We need to test the hell out of this before moving on with PagedResults optimization for forward-only. We need to ensure that functionally the AQ invalidation will work as in OE in terms of multiple session committing at the same time.

Switch RAQ for PagedResults

- #8388 should be ready
- #11980 should be ready

This is the core logic of this task as you described previously. I am trying to think if we can split this as well to separate PagedResults logic from dirty-share ... I guess not. When we will reach this point, we will do some brainstorm to identify the right scope.

Make PagedResults available to non-forward only

PagedResults implementation does not depend on moving only forward. As long as we have a reference row, we can always emit the query with reversed order by and gather the results in the other way around. In fact, this will make reverse iterations faster than trunk which **doesn't invalidate**. Trunk with PREV is ruling out the re-validation bets completely.

#7 - 11/02/2026 03:41 AM - Artur Școlnic

Agreed on the phases, shared memory first, #8388 as prerequisite and the page epsilon.

- **Shrinking on foreign commits:** you are right, we keep it. My point came from a 60-row walk over TCP, where the whole set fits in one message. The 5-6 s cost I measured came from commits to an unrelated table, so the per-table counter is what removes it.
- **Own inserts/moves:** the quoted line was truncated; it should read "OE over TCP does not show its own inserts or key moves inside the current window; FWD does (as the RAQ does today)". So FWD is fresher than OE TCP there, and invalidating PagedResults from stateChanged keeps that behaviour, which is most likely the shared-memory one. I will confirm with the shared-memory run.
- **Global events:** I will cover leverage vs purge in the phase 1 inventory.