Conversion Tools - Feature #11887

rework analytics details report processing to implement a per file intermediate level for performance in large data cases

09/17/2026 07:13 PM - Greg Shah

Status:	Test	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Greg Shah	**% Done:**	100%
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 - 09/17/2026 07:37 PM - Greg Shah

- % Done changed from 0 to 100

- Assignee set to Greg Shah

- Status changed from New to Test

Analytics Detail Report Performance

Introduction

The FWD Analytics web application took roughly ten minutes to open a detail report on a customer's large application. This note records how that time was accounted for, which of the obvious fixes turned out not to work and why, and the design and implementation of the change that was made.

Two independent problems were found. The first was a launch script defect which slowed the entire server by about an order of magnitude and was trivial to fix. The second was the cost of the detail report query itself, which was structural and needed a design change. A third problem, a browser caching defect which hid every web client change, was found while verifying the work.

All measurements below were taken against the report database of a customer's large application: 207,448,414 matches across 431 reports, a 125 GB rptdb.h2.db, on an NVMe SSD with LUKS encryption, 58 GB RAM and 16 cores.

The work was committed to the 11747a branch in two revisions, both referencing #11746. Revision 16766 reworked the report server to load detail reports a file at a time and capped the report server H2 page cache. Revision 16767 made the original all-matches view available alongside the new per-file view, defaulted between them by result count, and added the entity tag fix described below.

Symptom

Opening any detail report for a large match category took about ten minutes. The summary reports were also slow. The server had consumed roughly 27 minutes of CPU in its first hour of life.

Problem analysis

The report server ran with the JIT compiler disabled

deploy/server/report.sh launched the server with these options:

```
dtxt="-Xnoagent -DP2J_HOME=. -Djava.compiler=NONE"
```

On JDK 17, -Djava.compiler=NONE is not the no-op it appears to be. It disables HotSpot's JIT compiler entirely. Measured on the affected machine:

Flags	Elapsed	JIT compilations
(none)	272 ms	103
-Xnoagent only	266 ms	not measured
-Djava.compiler=NONE	**3315 ms**	**1**
-Xint (known interpreted)	3766 ms	not measured

The flag is indistinguishable from -Xint. Every line of Jetty, H2, JSON serialisation and report code was being interpreted.

The origin is a copy-paste defect. Both report.cmd and server.sh carry the legacy JPDA debug incantation -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:.... In server.sh it is correctly guarded:

```
if [ "$debug" == true ]; then
   dtxt="-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,..."
fi
```

report.sh inherited a copy with -Xdebug and -Xrunjdwp stripped out. It therefore kept the part which cripples performance and dropped the part which actually enables debugging.

No maximum heap was set

report.sh set no -Xmx at all, so the JVM defaulted to 25% of RAM, or 15.7 GB. This matters more than usual here because both of the server's large caches are derived from Runtime.maxMemory():

- DatabaseService sets the H2 page cache to maxMemory / 1024 / 2, which was 7.3 GB against a 125 GB database.
- The JanusGraph call graph cache takes about 30% of the heap, which was the 4.6 GB reported in the server log.

That is about 11.6 GB of cache in a 15.7 GB heap, while still caching roughly 6% of the database. By contrast server.sh defaults to -Xmx8192m with explicit GC tuning.

A related inconsistency was found. ReportWorker.calculateCacheSize() caps the same page cache at 8 GB, with this comment:

> on the large heap a big project needs, an uncapped page cache claims enough of the heap to starve the queries which run at the end of the pipeline

That cap had never been applied to DatabaseService, which is the web server's copy of the identical calculation.

Where the time actually went

With the JIT restored, the real detail query for the largest category (cid 151, 4,995,717 matches, 17,228 files, 397,637,450 characters of match text) was timed twice in the same JVM:

Run	Elapsed
cold (first)	340,605 ms
warm (second)	576 ms

A ratio of about 590 to 1. The dominant cost is therefore **cold random I/O**, not CPU and not the query plan. EXPLAIN shows why:

```
FROM "PUBLIC"."MATCH" "M"
    /* PUBLIC.FK_MATCH_CATCID_INDEX_4: CID = ?1 */
...
ORDER BY 4, 5, 6
```

The plan drives from an index on cid alone, then fetches each of the five million matching rows individually from a 125 GB file. At roughly 70 microseconds per random page read that is very close to the 340 seconds observed.

Why pagination alone does not help

The obvious fix is to stop reading five million rows to display a hundred and fifty. It does not work:

Query	Elapsed	Rows returned
Full detail query	340,605 ms	4,995,717
Same query with LIMIT 150	278,995 ms	150
Restructured to drive from file, LIMIT 150	241,498 ms	150

Fetching one page costs essentially what fetching everything costs. The reason is the ORDER BY f.name, m.line, m.col: the sort key begins with a column of the file table, so the database must materialise and sort every matching row before it can return the first one. LIMIT can discard rows only after that work is done.

Reordering the FROM clause to drive from file in name order, which would in principle allow early termination, does not change the plan. H2's optimiser ignores the hint and still drives from match.

The covering index, and why it was rejected

If the five million random row lookups could be replaced by a sequential index scan, the dominant cost would disappear. This was investigated properly rather than assumed.

Two mechanical preconditions were confirmed by experiment against the FWD H2 fork:

- **H2 does perform true index-only scans.** Two identical tables with identical covering indexes, one carrying a 2 KB filler column absent from both the index and the select list, were queried. The table with the filler is 1.23 GB against 34 MB, yet both queries ran in the same time (13 ms and 20 ms cold, 1 ms each warm). Had the engine been fetching base rows, the larger table would have been far slower.
- **There is no index key size limit to worry about.** Text values up to 200,000 characters were inserted against a wide covering index without error, and the optimiser continued to choose that index.

The proposal nevertheless fails on disk space. The index must duplicate the text column, and that column is what makes the table large:

Measure	Value
Total matches, all reports	207,448,414
MATCH table on disk	95.0 GB
Average match text length	439 characters
SOURCE_LINE table on disk	3.4 GB
Free disk space	**93 GB**

The text alone would need about 91 GB, plus about 6.6 GB of key columns and B-tree overhead, giving an estimated 100 to 110 GB against 93 GB free on a filesystem already 90% full. Building it would also require an external sort of about 91 GB, which needs temporary space that is equally unavailable.

The narrow variant does not rescue the idea. An index on (cid, fid, line, col, id) without text would occupy only about 6.6 GB, but the engine would then have to fetch the base row to obtain the text, reinstating exactly the random reads the index was meant to remove. The text is simultaneously the reason the approach would work and the reason it does not fit.

Finally, a covering index would not have removed the sort in any case, because the sort key begins with a file column which no index on match can supply.

Design

Load a detail report one source file at a time

A detail report is already presented grouped by source file. The list of contributing files is far cheaper to obtain than the matches themselves, because it can be satisfied from the existing index on match (cid, fid) without reading any match row. Measured on the same category:

Query	Elapsed
File list with match counts	6,919 ms
One file's matches, on demand	2 ms to 229 ms

The detail view therefore lists the contributing files with their match counts, and fetches a file's matches when that file is opened.

Why not lazy group expansion

The apparently smaller change, keeping the existing single grid and loading each group's rows as it is expanded, was rejected. The bundled

Tabulator 2.12.0 exposes no group open or close callback, so this would have required patching a third-party library.

The two-level view avoids that entirely and reuses the existing showDetailRows grid unchanged, so per-file match display, sorting, filtering and source drill-down all behave exactly as before.

Whole-category operations move to the server

Two operations genuinely need the whole category and can no longer be served from data held in the browser:

- **Match text search.** No index covers match text, so searching every file costs minutes. It is therefore an explicit action with a warning, never a filter applied while typing. Results are capped at 10,000 rows. Filtering within an already open file remains instant and local.
- **CSV export.** For this category the export is several gigabytes, which no browser will assemble in memory. It is written on the server, under analytics_exports/, and the user is told the path and row count.

Keep the original view for small categories

The original view is better from a user's point of view: one grid, grouped, sorted, filtered and exported entirely in the browser. Its only problem is that its cost grows with the category. On a small project that cost is negligible and the original view should be kept.

The selection is automatic and free. The match count is already known on the client, from the summary row the user clicked, so no probe query is needed. Categories at or below DETAIL_EAGER_LIMIT, set to 50,000 matches, use the original view; larger ones use the file list. At the measured rate of about 56 microseconds per match, that limit corresponds to roughly three seconds.

The threshold is a default, not a rule. Either view can be chosen by hand at any time, as described next.

Switching between the two views

Both views are always reachable. Each one carries a link to the other in its title bar, at the top right, alongside the CSV link:

View currently shown	Link offered	Effect
All matches (the original single grid)	**Show By File**	Replaces it with the file list, loading each file's matches on demand
File list (the per-file view)	**Show All Matches**	Reads every match in the category and shows them in the original single grid

On the file list the **Show All Matches** link sits with the **Search All Matches** and **Export All Matches** links, to the right of the CSV link.

Which view appears first, before any link is used, depends only on the category's match count:

- At or below 50,000 matches, the all-matches view opens. This is every category on a small project.
- Above 50,000 matches, the file list opens.

Switching is immediate and takes effect for the report being viewed. Clicking a link discards the current view and rebuilds the report in the other one, so the data is refetched in the form that view needs; there is no partial or mixed state.

Choosing **Show All Matches** for a category above the threshold first asks for confirmation, naming the match count and warning that reading them all can take several minutes. Cancelling leaves the file list untouched. The other direction, **Show By File**, is always fast and never prompts.

The choice is not remembered. Using the back arrow and reopening the same category applies the automatic choice again. This is deliberate: it means an accidental switch into a multi-minute load cannot become a persistent setting which makes every later report slow.

The 50,000 threshold is the client constant DETAIL_EAGER_LIMIT in report.js. Lowering it makes the file list take over sooner, raising it keeps the original view for larger categories. It is documented in place with the measured cost per match, so it can be retuned against a particular project rather than guessed at.

Implementation

Changes

The FWD changes were committed to the 11747a branch in two revisions:

File	Entry	Revision	Change
DatabaseService.java	005	16766	Capped the H2 page cache at 8 GB, mirroring ReportWorker
ReportApi.java	012	16766	Four new APIs: detail file list, per-file matches, server side search, server side CSV export
DetailFileRow.java	001	16766	New bean for a file group and its match count

report.js	007 sub-entry	16766, 16767	File list view and per-file loading, server side search and export, then the view mode switch
ReportWebServer.java	006	16767	Serve entity tags for static content

The launch script fix is separate. deploy/server/report.sh is not shipped by FWD; each converted project carries its own copy. The change there was to remove -Djava.compiler=NONE and the vestigial -Xnoagent, and to add -Xmx32768m, the GC tuning server.sh already uses, and optional GC logging. Because the script is project-local, **any other project holding a copy of this script has the same defect and needs the same fix**.

The count(*) trap

The first implementation of the file list query used count(m.id) and took **218,742 ms**, barely better than the original. m.id is not part of the index on (cid, fid), so counting it obliged the engine to fetch every match row and reinstated the problem the query existed to avoid.

Changing it to count(*), which requires no column at all, keeps the scan index-only and brought the query to **6,919 ms**, a factor of 32. The constant carries a comment recording this, because the regression is silent and a plausible-looking edit reintroduces it.

Per-file rows omit constant fields

The per-file query returns only id, line, col and text. The file name and AST ID are constant across the result and the caller already holds them in the file group it requested, so repeating them on every row would have dominated the response. The client reattaches them.

Stale cached web client

While verifying the work it became clear that web client changes were not reaching the browser at all. The static content is served from inside p2j.jar, and the build normalises jar entry timestamps to the ZIP epoch:

```
Last-Modified: Fri, 01 Feb 1980 05:00:00 GMT
```

That date never changes between builds. No ETag and no Cache-Control header were sent. A browser which had once cached a file therefore revalidated against a date which never moved, was told 304 Not Modified, and kept its stale copy indefinitely. With no Cache-Control on the response, heuristic freshness based on a 1980 timestamp would also let many browsers skip revalidation for years.

The fix is two lines on the static resource handler, with a comment recording the cause so it is not removed as redundant:

```
rootHandler.setEtags(true);
rootHandler.setCacheControl("no-cache");
```

Verified against the running server: a request carrying the current entity tag receives 304, and one carrying a stale tag receives 200 with fresh content.

One limitation should be recorded. Because Last-Modified is frozen at the ZIP epoch, the weak entity tag Jetty derives here is effectively a function of content length. A rebuild which happened to produce a file of identical length would collide and go stale again. Setting preserveFileTimestamps on the jar task would close that gap, at the cost of reproducible builds; it has not been done.

Results

Operation	Before	After
Report server startup	2410 ms	905 ms
Open detail report, 4,995,717 matches	about 279,000 ms	6,919 ms
View one file's matches	included in the above	2 ms to 229 ms
Open a detail report below 50,000 matches	unchanged	unchanged, original view retained

Not done, and open points

- **The covering index** remains the only change which would speed up every report rather than the detail view alone. It is blocked on disk space, not on correctness, and would become available if the filesystem gained roughly 120 GB of headroom.
- **Match text storage.** 207 million matches averaging 439 characters of stored text are why the database is 125 GB, of which the match table is 95 GB. Truncating stored match text would shrink the database dramatically, but it would change report contents and has not been pursued.
- **Browser testing.** The implementation is verified at query, build and HTTP level. The user interface flow itself was not exercised by automated means during this work.