

Database - Bug #11953

Consider using the AST transformation pipeline philosophy for FQL preprocessing

10/08/2026 04:23 AM - Alexandru Lungu

Status:	New	Start date:	
Priority:	Normal	Due date:	
Assignee:		% Done:	0%
Category:		Estimated time:	0.00 hour
Target version:		reviewer:	
billable:	No	production:	No
vendor_id:	GCD	env_name:	
case_num:		topics:	
version_reported:			
version_resolved:			
Description			
Related issues:			
Related to Database - Bug #9030: FIND statements should use an internal buffe...		WIP	
Related to Conversion Tools - Feature #11650: Serialize AST or any intermedia...		Review	

History

#1 - 10/08/2026 04:33 AM - Alexandru Lungu

This task is a proposal to dissolve the FQLPreprocessor class and map it onto a TransformDriver **similar** approach.

- Why? There are currently plenty of manipulation steps on the FQL AST that start being interdependent upon each other. There is a complex state that is being analyzed based on the FQL AST and plenty of transformation steps (e.g. simplifications, augmentations, optimizations).
- So? The complexity of FQLPreprocessor makes its extension fragile. For [#9030](#), extending the property matches algorithm to have a lenient version comparing to the strict one used by the CQ\$Optimizer may exploit the fragility easily.
- TRPL? I don't say that TRPL shall be used for FQL ASTs to achieve the annotating and manipulation state, but **I don't exclude its philosophy**. I would prefer to have a cleaner multi-step process that has a this and copy AST.
- Examples:
 - I think the FQL AST is traversed multiple times using custom AST listeners in FQLPreprocessor. The analysis is not streamlined and sometimes we may tend to simply add a new traversal instead of leveraging an existing one just because we are afraid of altering the behavior of existing preprocessor. It would be cleaner to have "FQL annotation", "FQL simplication", "FQL augmentation" so that any FQL related logic will have a structured non-fragile implementation.

I don't have a clean implementation idea, so I want to ask for feedback. My imagination hints that some of the ConversionDriver can be leveraged in the FQLPreprocessor AST work (at least the AST walker and profiles). Maybe we can move the FQL preprocessing to TRPL, but I really fear the performance drawback. If we find an alternative implementation of TRPL with a Java DSL, I think we shall try it out in this scenario, so we can use this as a POC for using Scala (or another JVM language of choice).

Motivation: I dislike having the null augmentation as some bunch of code somewhere in a 100+ lines of code method in a 3000+ lines of code class. Adding new AST manipulation can easily disturb that logic:

- remove redundant where conditions due to analysis -> should be done before/after null augmentation?
- property matches should be identified -> should be done before/after null augmentation? But before or after removing redundant where conditions?
- etc.

#2 - 10/08/2026 07:13 AM - Ovidiu Maxiniuc

Indeed, this is an out-of-the-box idea. But I am a bit reluctant about the performance improvement. Two reasons, at a first thought:

- there are a lot of processing of the tree: nodes are created or deleted, or otherwise transformed. This is the reflection of the class name: preprocessing (inspection of the expression after parsing the where predicate is rather secondary, but also an important goal of the class). Using a TRPL-like transformation pipeline implies creating mirror AST trees (this/copy) for proper manipulation. If the tree is to be traversed multiple times, as FQLPreprocessor does now, the tree will have to be duplicated in the background, increasing the management effort;
- the way I see it, the proposed AST transformation would be nearly 'stateless', the eventual required information will have to be stored in the AST nodes themselves, as annotations. These are actual Maps which imply some computation effort. The FQLPreprocessor uses local variables instead, so it is faster.

I will continue to think of this issue.

#3 - 10/08/2026 08:11 AM - Alexandru Lungu

- Related to Bug #9030: FIND statements should use an internal buffer cursor added

#4 - 10/08/2026 08:11 AM - Teodor Gorghe

- Related to Feature #11650: Serialize AST or any intermediate conversion file into more lightweight form. added

#5 - 10/08/2026 08:46 AM - Teodor Gorghe

I have prototyped an AST AOT approach, which takes these compiled expressions and moves them into a jar, during build time. It proved a faster warm-up time (2-3 seconds), but nothing more.

I think the optimal direction for TRPL is to make a TRPL compiler to Java bytecode (extend our Compiler as Ovidiu said in [#11650](#)), so that JIT inlines it better (right now, the compiled expressions get only into C1). Scala would be as fast as TRPL rules compiler, but there would be a large number of lambda functions which might trigger some slowdowns compared with native Java bytecode.

#6 - 10/08/2026 10:14 AM - Ovidiu Maxiniuc

Ovidiu Maxiniuc wrote:

I will continue to think of this issue.

The idea that emerged during the meet discussion for dealing with the ever increasing complexity of preprocessor.

What if instead of having a single, all-purposes, big class (9kLOC), we would have different preprocessors, with complexity specific to their purpose? A few cases were brought into attention:

- checking whether a query is a simple PK positioning (aka 'reposition by rowid') when all that is needed is to parse and quickly inspect the predicate structure;
- augmenting the predicate with non-boolean elements for specific dialects (I am looking at you, MSSQL);
- specialising the predicate to use FTS, if so configured and the technology is supported and implemented by current dialect (only PSQL for the moment).

As it can be seen, these tasks are pretty independent and the current class crams all, doing multiple traversals of the AST, as needed.

So the normal question is: what if we could have at our disposal of preprocessors specific to a custom task? This way additional nop traversals will be skipped. A set of customised instances could be created but would be difficult to maintain. So the solution would be to construct them dynamically, using a factory, or better: a builder. The method returning the custom preprocessor will have to provide the needed features, as boolean flags or other bitwise combination. Instead of being a monolithic construct, it can be effectively constructed by components which will be chained to be executed in the order imposed by current constraints. These preprocessors do not have to remain static as they were constructed, but (similar to AdaptiveQuery) might reconfigure themselves if the execution flow requires additional features not specified initially, but keeping this at the minimum to avoid returning to original 'heap of features' (although distributed in smaller classes).

Instead of being a monolithic construct, it can be effectively constructed by components which will be chained to be executed in the order imposed by current constraints

Right, but if each unit will traverse the tree, then the performance will be way worse. At least now, some of the analysis is merged within a single tree walk. I would want to keep that. In other words, each such "unit" should be declared as:

- read-only
- transforming

If multiple read-only units are enqueued, they can be resolved using one single traversal. If multiple transforming units are built, they need "copy"/"this" philosophy being implemented. But what happens if we intertwine them (read, read, write, read, write, read)? I suppose each consecutive read steps are merged in a single pass (eventually together with the upcoming write which is not mutating the tree being walked).

PS: each such "unit" we are talking about is like a TRPL rule or rule-set :)

These preprocessors so not have to remain static as they were constructed, but (similar to AdaptiveQuery) might reconfigure themselves if the execution flow requires additional features not specified initially, but keeping this at the minimum to avoid returning to original 'heap of features' (although distributed in smaller classes).

This is interesting. There are cases in AdaptiveQuery where a FQLPreprocessor is used to for AQ PRESELECT mode, but if invalidation occurs, the **same** FQLPreprocessor is used to determine whether the changed DMO matches the where clause (i.e. it is or will be in the result-set of the query).

Right now the implementation builds an all-purpose FQLPreprocessor state that is being checked by the query at arbitrary times and contexts (after open, on invalidation, on reopen / on MSSQL, as a CQ component, with direct H2 access, etc.).

So the apparent goal would be to build an initial "core" FQLPreprocessor (with some preprocessing units), but as events occur, we may need to append new units.

- For example, initially we just plug in the mainWalk logic.
- Afterwards, then a record is changed, the FQL preprocessor should state what properties of the DMO are used in the FQL.
- However, the FQL pipeline we considered already **altered** the AST, so how can we answer the enquiry now? Do we build another pipeline and pay the price of AST traversal?

My point: in trunk we eagerly analyze the AST through **some** walks, but if we do it lazily we may end up with **more** walks as we have to compute some flags that **could have been computed** in the original traversal. Though, we can't know for sure if they were needed back then.

PS: all of the above suggestions are considering caching (FQL to AST, checking if FQL units were no-op for certain FQL and skip next time, etc).

#8 - 10/09/2026 08:30 AM - Octavian Adrian Gavril

Here are some thoughts, from the TRPL side.

What strikes me looking at preprocess() is that the steps do not follow one convention. Around eight of them return a new root, the others either mutate the tree in place or write their results into instance fields. So from the call site you cannot tell whether your tree survives the step, and mainWalk rewrites in place while it is walking. I suspect that is where a good part of the fragility comes from, more than the number of steps.

TRPL solves exactly this, and with one mechanism rather than with its rule language. Per rule set, PatternEngine does copy = source.duplicate(idMap), walks the source tree that no rule is allowed to modify, lets rules edit only the copy, and then the copy becomes the source for the next rule set. A step cannot corrupt its own traversal and cannot see its own half-applied changes. HQLAst is already an Aast, so duplicate(idMap) and annotations are available without new infrastructure.

What I would be careful about is that turning each existing call into a phase object, and nothing more, buys us nothing, it is the same sequence. The three things that would actually make a difference are a single contract for every step (it receives a tree it must not modify and returns the resulting one), the state moving out of instance fields into annotations so that steps stop communicating through the object, and each step declaring which annotations it requires and which it produces so the order can be validated. Only the third one really answers the "before or after null augmentation" question, and with the first two in place the lenient property matches variant needed for #9030 becomes a new step rather than another branch inside collectPropertyMatches. The ordered list of phases is just the scaffolding around those three, and on its own I would not consider it worth the effort.

I would not take the XML and compiled-expression layer with this, since its payoff is changing rules without rebuilding FWD and that does not apply at runtime. If we do want the real engine later, note that we already run TRPL at runtime in persist (ConversionPool, with three profiles under rules/runtime/), so it could be measured rather than guessed. I am not sure that setup fits this case as it stands though, as there is a single engine per profile and popEngine() waits when the stack is empty, so sessions preparing a new where clause would queue on one engine.

#9 - 10/09/2026 08:42 AM - Alexandru Lungu

What I would be careful about is that turning each existing call into a phase object, and nothing more, buys us nothing, it is the same sequence.

It buys nothing from functionality and performance POV. But my concern here is exactly the premises -> the contract. In FQLPreprocessor currently you can do **whatever you want** and nothing stops you. It feels like a big procedural DYI transformation that leverages little to nothing from OOP. The extra engineering that I am dreaming of here will eventually help assessing the Open-Close Principle. For example: where can I add a new analysis step?

- In trunk: this question has a trivial answer: in some AST traversal code hoping that it is not too early or too late. If you want to assess if it is too early or too late, you can by ingesting the **whole** file to understand that. Eventually, you will need to understand whether incorporating in an existing AST traversal is correct ... what if that traversal is also writing. In fact, mainWalk does both property extraction and AST rewriting.
- Suggestion: It would have been easier to extend a AnalyzingPreprocessor and register it in the pipeline.
 - Or at least a PreprocessingUnit with a 1 to 10 weight.
 - Or a well defined phase pipeline: phase 0 is preprocessing, phase 1 is analyzing, phase 2 is optimizing, etc.
 - when you right new units, you only have to care about integrating it in the right bucket.

In conversion, we do this thinking of adding a rule to a certain phase. But current FqlPreprocessor is like having TRPL being within one single big profile (**and file**) doing frame work, DMO work, base language work, mutating the AST in place, etc.