Database - Feature #2275

Feature # 2274 (Closed): improve performance of new database features

cache of runtime-converted, dynamic queries and temp-tables

03/30/2014 03:43 PM - Eric Faulhaber

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Ovidiu Maxiniuc	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	Performance and Scalability Improvements		
billable:	No	**vendor_id:**	GCD

Description

History

#1 - 03/30/2014 03:57 PM - Eric Faulhaber

- Subject changed from create a cache of runtime-converted, dynamic queries to cache of runtime-converted, dynamic queries and temp-tables

As I'm writing this, it's becoming clear that the query cache and temp-table cache probably need to implemented together, so I've changed the name of this task to include both in its scope.

Caching dynamically-converted queries and temp-tables will allow us to avoid the effort of runtime conversion when an identical query is encountered.

Some initial considerations:

- Are the caches shared or private (or both)? Queries can reference either permanent tables, temp-tables, or both. Temp-tables are inherently private to a session, but it is the structure of the table we are interested in, so it may be OK to cache both temp-tables (the DMO interface and implementation classes, ORM mappings) and queries for shared use.
- What are the caches' scope and life cycle? If private, they will be cleaned up with the session, but we don't know how long-lived a session will be. If very long, the cache may grow too large; if very short, the cache may not prove that useful. This suggests we should try to share the caches, if possible, going back to the previous point.
- What type of cache to use? We probably need to use bounded caches to control the amount of memory that can be used. My initial instinct is to use an LRU algorithm. Is LFU more appropriate? Make it configurable? Probably not initially, to keep the complexity down.

#2 - 07/30/2014 12:23 PM - Eric Faulhaber

- Target version changed from Milestone 11 to Milestone 17

#3 - 03/25/2016 01:24 PM - Eric Faulhaber

- Status changed from New to Hold

This set of features is implemented. Only one optional change remains for the dynamic query cache.

Converted, dynamically prepared temp-tables are cached by DynamicTablesHelper. This work is done.

Converted, dynamically prepared queries are cached (see #2488). The existing cache works, but could be smarter about not duplicating entries for essentially the same query against the same buffers, where only the buffer name varies. By doing this, the cache would not fill up as quickly, and we could improve upon the cache hit rate. However, it comes with a slightly more expensive cache look-up, so it is unclear at this time whether it would result in a net performance improvement.

The change would involve parsing the query predicate and determining which buffer(s) it references. If an existing entry exists which has the same predicate, referencing the same backing table(s), but with different names, that query would be used, instead of converting and caching the new query as a unique query. This would increase lookup time slightly, but would save a relatively large amount of time on the query conversion. The improvement depends on how well this savings amortizes across query invocations.

I am leaving this open in M17 for now, but since it is not obvious whether there would actually be an improvement, it is not currently something targeted for work, so I am setting the status to "Hold".

#4 - 04/21/2016 01:38 PM - Eric Faulhaber

- Status changed from Hold to New

- Assignee set to Ovidiu Maxiniuc

#5 - 04/22/2016 03:05 AM - Eric Faulhaber

In addition to the buffers improvement, I bet a lot of predicates differ only in the constant values which are inlined and compared against buffer fields. For example, consider:

```
foo.bar = 1 and foo.bat = "some string"
```

and

```
foo.bar = 2 and foo.bat = "some other string"
```

Assuming foo is backed by the same table for both predicates, these essentially represent the same operation. If there are many such similar predicates and we could factor out the inlined literals, such that both predicates could be serviced by the same cached, converted query, we would have a nice performance win.

#6 - 04/22/2016 08:50 AM - Ovidiu Maxiniuc

This are quite similar to my thought from #2488-34. There are some statistics and an implementation sketch in there, also.

#7 - 05/10/2016 11:53 PM - Eric Faulhaber

- Status changed from New to WIP

#8 - 05/12/2016 12:54 PM - Ovidiu Maxiniuc

We are using now the dynamic conversion mostly for the HQL that are hardcoded at conversion time.
To correctly handle the similar queries we need to extract the literals and replace them with variables. For example,

```
foo.bar = 1 and foo.bat = "some string"
```

should be seen in DynamicQueryHelper.generateJavaTree() as something like:

```
DEFINE VARIABLE __P1__ AS integer.
DEFINE VARIABLE __P2__ AS CHARACTER.
find first foo where foo.bar = __P1__ and foo.bat = __P2__ no-lock.
```

When interpreted by RuntimeJastInterpreter, the two variables need to be extracted initialized with values from original string.

From this point forward I will use examples from my test cases. I started with a simple find-first method that will call DynamicQueryHelper.generateJavaTree("FIND FIRST book WHERE isbn = '15bn' NO-LOCK.")

I am working on following algorithm:

- we need to skip the current cache solution and do a quick 4GL parsing.
- when parser ends without error, before invoking prepareQueryTree(progAst), we call a extractParams() method that will iterate all nodes in early AST and replace literals (string/integer/decimal, etc) with manufactured variables of their respective types. Of course their definition must be inserted. This method will return the mapping obtained. Now the AST tree looks like this:

```
block [BLOCK] @0:0
   statement [STATEMENT] @0:0
      DEFINE [DEFINE_VARIABLE] @1:1
         __P1__ [SYMBOL] @1:17
         AS [KW_AS] @1:24
            CHARACTER [KW_CHAR] @1:27
   statement [STATEMENT] @0:0
      FIND [KW_FIND] @1:38
         FIRST [KW_FIRST] @1:43
         record phrase [RECORD_PHRASE] @0:0
            book [TABLE] @1:49
            WHERE [KW_WHERE] @1:54
               expression [EXPRESSION] @0:0
                  = [EQUALS] @1:65
                     isbn [FIELD_CHAR] @1:60
                     __P1__ [VAR_CHAR] @1:67
            NO-LOCK [KW_NO_LOCK] @1:74
```

  The returned map is

```
__P1__ -> '15bn'
```

- only at this moment we should be able to identify if the query can be found in the cache. (I don't have yet an exact cache key format for this but this should be easy to assemble it from a quick traversal of the AST)
- if query not in cache, we continue processing as before, generating the JAST to be interpreted. Eventually we need to:
  - allow additional TRPL code to handle the new define variable statements;
  - remove the generated variables from JAST as they will be passed in as global parameters to RuntimeJastInterpreter;
  - add other changes in TRPL that might be necessary.

- the generated tree will have the HQL a little altered, with substitutions in place for all literals found. This level of generality will allow the JAST to be used for virtual any query with same search pattern;
- save the generated Java tree to new cache
- at the evaluation time, we use the RuntimeJastInterpreter constructor with second parameter initialized with the mapping we obtained at step 2. The implementation should already be in place, although I never tested.

#9 - 05/12/2016 02:24 PM - Eric Faulhaber

This looks good so far.

How do we deal with the buffer(s) in this new cache? It is possible to specify various buffer names which essentially represent the same backing table for the same structural query, but which are named differently from one 4GL query preparation to another. If we identify the buffers' schemaname annotation(s) for cache key purposes, but make the specific buffer instance(s) late-binding to the query, I think this should work.

Is there any drawback you can think of (performance or otherwise) to treating the lock type as a variable as well? I don't know how common it would be to have the same structural query prepared with different lock types, but it can happen, so if there's no penalty for making the lock type variable, we might as well do it.

#10 - 05/13/2016 04:41 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> How do we deal with the buffer(s) in this new cache? It is possible to specify various buffer names which essentially represent the same backing table for the same structural query, but which are named differently from one 4GL query preparation to another. If we identify the buffers' schemaname annotation(s) for cache key purposes, but make the specific buffer instance(s) late-binding to the query, I think this should work.

The list of buffers is part of the QueryCacheKey at this time, too. I am positive we will keep this. I will investigate the solution for using the same generated JAST for multiple buffers belonging to same table, but I don't expect much improvement in this area.

> Is there any drawback you can think of (performance or otherwise) to treating the lock type as a variable as well? I don't know how common it would be to have the same structural query prepared with different lock types, but it can happen, so if there's no penalty for making the lock type variable, we might as well do it.

I will add this too, as a dedicated P4GL variable in the dynamic generated code. There should be no difference from other variables injected in the process, so the impact on performance should be minimum.

You mentioned the performance. Here are my thought/ideas on this area.
I cannot estimate the exact benefit of this new implementation. It will require a few more CPU cycles for processing the input 4GL and extracting the variables, but this should be quick (linear with number of chars in the input query). And then computing the cache key will also add some minimal computation. And, once we have a cache hit, cutting the whole TRPL should give as the performance boost we seek.
There are another things that can influence the performance when interpreted the HQL (now with substitutions /SQL parameters). This is difficult to estimate, but the main factors are the substitutions access (negative impact) and HQL caching (positive impact).

#11 - 05/13/2016 09:33 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Eric Faulhaber wrote:
>
> > How do we deal with the buffer(s) in this new cache? It is possible to specify various buffer names which essentially represent the same backing table for the same structural query, but which are named differently from one 4GL query preparation to another. If we identify the buffers' schemaname annotation(s) for cache key purposes, but make the specific buffer instance(s) late-binding to the query, I think this should work.
>
> The list of buffers is part of the QueryCacheKey at this time, too. I am positive we will keep this. I will investigate the solution for using the same generated JAST for multiple buffers belonging to same table, but I don't expect much improvement in this area.

I seem to recall an idiom used in the current project where the same query appears over and over, and the only difference is that a different (ascending) number is appended to the buffer name (I *think* this name represents essentially the same buffer, insofar as the query is concerned). We treat these as different queries currently. This is the case I am hoping to address. I believe you previously analyzed the dynamic queries running through the current cache implementation for that project. Please revisit that analysis at the appropriate time, and confirm whether this is the case, and if so, that we will address this with the new implementation.

> You mentioned the performance. Here are my thought/ideas on this area.
> I cannot estimate the exact benefit of this new implementation. It will require a few more CPU cycles for processing the input 4GL and extracting the variables, but this should be quick (linear with number of chars in the input query). And then computing the cache key will also add some minimal computation. And, once we have a cache hit, cutting the whole TRPL should give as the performance boost we seek.
> There are another things that can influence the performance when interpreted the HQL (now with substitutions /SQL parameters). This is difficult to estimate, but the main factors are the substitutions access (negative impact) and HQL caching (positive impact).

This all sounds fine. Considering that when we have a cache miss with the current implementation, we are doing the parsing anyway as the first step in the runtime conversion, it sounds like the primary penalty we will be paying with the new approach is doing the parse for the cases where the old cache implementation would have resulted in a hit already (plus the work of extracting the variables, which seems minimal). I expect that on balance, we will come out ahead with the new implementation, as I expect our hit ratio will be better for most applications.

#12 - 05/13/2016 10:05 AM - Ovidiu Maxiniuc

The summary/statistics can be found at #2488-34 and 35. I don't recall the exact target. I can reactivate the log and eventually the timers when implementation works as I expect.

#13 - 05/16/2016 02:08 PM - Ovidiu Maxiniuc

Created task branch 2275a from trunk 11030. Last revision is 11031. The update is based on the algorithm described in note 8 above.

It was not thoroughly tested, I will do that tomorrow, along with some timings o see the improvements.
I tried to add to this revision support for locking, too, but unfortunately, the conversion does not support the workaround (this would be similar to adding a variable reference in P4GL instead of locking clause - it works for dynamic queries but not in static one). The code remained in place but disabled by commenting the 'entry point'. I am glad that, after cleanup, not many files were affected (I was expected at least the RuntimeJastInterpreter to be altered).

I am not happy with the 'manufactured' variable names. Maybe we should discuss about this to find a better solution. I commented the code (Java and TRPL) where this is handled.

Another thing to discuss is the cache key. I don't know if my solution is optimum. It requires a second traversal of the intermediary progress ast.

I am not aware how the implementation handles similar queries on different buffer of same table. I need to test this. I am not aware if such constructs are present in our projects.

I am going to testing phase, and expect minimum changes. Please review.

#14 - 05/17/2016 01:03 PM - Eric Faulhaber

Ovidiu, can you please help me understand how the variable_definitions.rules changes (both annotations and conversion) fit into the picture?

#15 - 05/17/2016 02:26 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Ovidiu, can you please help me understand how the variable_definitions.rules changes (both annotations and conversion) fit into the picture?

The variable references are not generated unless javaname and classname are added by annotations/variable_definitions.rules. I guess I could add these manually in the injectVariable(), and drop this rule from the chain of TRPL. The reason to keep it would be to have all processing (static & dynamic) in a single place.
I just double checked and the other variable_definitions seems now not mandatory. I will remove it.

#16 - 05/17/2016 03:29 PM - Ovidiu Maxiniuc

Encountered an issue with new processing. The queries like

```
FOR EACH t1 WHERE  t1.f1 = 1 AND    t1.f1 = '1'  AND    t1.f2 <> ? AND t1.f3 <> ?
```

are not handled well. The field f1 is integer.
The new generated hql looks like this:

```
from T1Impl as t1 where (t1.f1 = ?0 and t1.f1 = ?1 and t1.f2 is not null and t1.f3 is not null) order by t1.d2
 asc
```

with two parameters: [1: int64, "1":char].

Static and older dynamic implementation handles this by forcing a cast from char to int for second parameter. The new solution will fail with following PostgreSQL error:

```
[05/17/2016 19:06:47 GMT] (org.hibernate.engine.jdbc.spi.SqlExceptionHelper:WARN) SQL Error: 0, SQLState: 4288
3
[05/17/2016 19:06:47 GMT] (org.hibernate.engine.jdbc.spi.SqlExceptionHelper:ERROR) ERROR: operator does not ex
ist: integer = character varying
  Hint: No operator matches the given name and argument type(s). You might need to add explicit type casts.
  Position: 919

Caused by: org.postgresql.util.PSQLException: ERROR: operator does not exist: integer = character varying
  Hint: No operator matches the given name and argument type(s). You might need to add explicit type casts.
  Position: 919
    at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2182)
    at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:1911)
    at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:173)
    [...]
```

#17 - 05/18/2016 02:39 PM - Ovidiu Maxiniuc

After more research on testcase from note 16 that causes some failures:

- I tried to isolate the testcase with TEMP-TABLES. Hibernate will generate valid SQL code with H2 dialect so queries end successfully.
- only the persistent database reports the errors reported (PostgreSQL)

I used the isolated testacase and run it against 4GL. When the query if executed statically:

```
(1) FIND FIRST ttt WHERE  ttt.f1 = '0010' AND ttt.f2 = 1 AND ttt.f2  = "1" AND ttt.f3 NO-LOCK NO-ERROR.
(2) FIND FIRST ttt WHERE  ttt.f1 = '0010' AND ttt.f2 = q__P1__ AND ttt.f2  = q__P2__ AND ttt.f3 NO-LOCK NO-ERR
OR.
```

both cases fail to compile (error is: ** Incompatible data types in expression or assignment. (223))
OTOH, in dynamic case:

```
bh:FIND-FIRST("WHERE  ttt.f1 = '0010' AND ttt.f2 = 1 AND ttt.f2  = ~"1~" AND ttt.f3", NO-LOCK).
QUERY qh:QUERY-PREPARE("FOR EACH ttt WHERE  ttt.f1 = '0010' AND ttt.f2 = 1 AND ttt.f2  = ~"1~" AND ttt.f3").
```

4GL will execute without complaints.

In this case the PostgreSQL dialect is strange (well, Hibernate in fact). When using the inlined hql as in (1) the query is successful. This is the older way dynamic queries were executed. However, the new implementation, will replace the literals with SUBSTS of the type the replaced literal. In this case Hibernate is not able to construct the SQL and fails with the above mentioned error.

I added the testcase as uast/dynamic-queries/p2275.p.

#18 - 05/19/2016 10:38 AM - Ovidiu Maxiniuc

Eric,
I found the errors logged in /var/log/postgresql/postgresql-9.2-p2j_enUS.log. I don't think hibernate is able to do anything for us.
This is very strange that if I copy/paste the statements logged there and replace the %1, %2 parameters with values to be passed in, they work, even the type is not matching, probably the planner will insert automatically the casting or maybe evaluate the expression when it parses them. However, in the case of parametrized queries sent from program/application, PostgreSQL expect the correct type of the parameters.

To fix this we could add casting or automatically convert the injected variable to required type. But operation this needs to know the datatype for each position/parameter, information we do not have just in the DynamicQueryHelper.

Do you have any other ideas in this matter?

#19 - 05/19/2016 10:54 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> To fix this we could add casting or automatically convert the injected variable to required type. But operation this needs to know the datatype for each position/parameter, information we do not have just in the DynamicQueryHelper.

Not sure I understand why we don't have this information. We have the Progress AST, which has a token type for the literal value it parsed (STRING, NUM_LITERAL, DEC_LITERAL, etc.) -- this is what becomes the variable, right? We also have the token type of the opposite operand (FIELD_INT,

etc.), so we would know the target of the cast/conversion. What am I missing?

Separate question: where do q__P1__ and q__P2__ come from in your static example above?

#20 - 05/19/2016 11:01 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Ovidiu Maxiniuc wrote:
>
> > To fix this we could add casting or automatically convert the injected variable to required type. But operation this needs to know the datatype for each position/parameter, information we do not have just in the DynamicQueryHelper.
>
> Not sure I understand why we don't have this information. We have the Progress AST, which has a token type for the literal value it parsed (STRING, NUM_LITERAL, DEC_LITERAL, etc.) -- this is what becomes the variable, right? We also have the token type of the opposite operand (FIELD_INT, etc.), so we would know the target of the cast/conversion. What am I missing?

This is correct. I need to check the operator and the opposite operand and convert the variable type/value. It might add some additional time penalty, but I will try the fastest solution. Thanks for clearing this up.

> Separate question: where do q__P1__ and q__P2__ come from in your static example above?

Those are the variable I injected in lieu of literals. They are of the same types as the removed literals, at this moment.

#21 - 05/19/2016 11:34 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> I need to check the operator and the opposite operand and convert the variable type/value. It might add some additional time penalty, but I will try the fastest solution. Thanks for clearing this up.

The following is a bit of a tangent, but I wanted to document a really good idea Greg had so I don't lose track of it. Let's get the core implementation of the smart cache working first, then we can come back around to this...

To avoid as much performance overhead as possible. why not keep the simple cache implementation we currently have as a first level cache? This

would catch any exact matches quickly, as we do today. Then, make the new, smart cache implementation a second level cache, which only gets invoked on a miss of the first level cache. If we get a hit on the second cache, we put the found JAST into the first level cache under the simple key, then return the result. If we miss on the second cache, we convert the query as usual and add the result to both caches with the appropriate keys.

The cache lookup on the second level cache is more expensive than that of the first, in that we are parsing the expression -- but we only do that if the exact match didn't work, AND we have to parse the expression anyway as the first step of the conversion, if there's no hit on the second level cache. So, the only new penalty is the incremental cost of preparing the second level cache key (beyond the normal parsing we would do for conversion), and the extra put operations on cache misses, due to having a second cache.

There is of course the consideration of the extra memory required to have two caches instead of one, but in my opinion, this is a very good trade-off. The JASTs are not duplicated, just the references to them are. So, the additional memory is only for the keys, and the overhead of the cache structures themselves. If the smart cache implementation is doing its job and the dynamic queries are distributed as we expect (i.e., similar in structure, but with varying literals), the size of the second-level cache should naturally be smaller than that of the first.

#22 - 05/20/2016 01:42 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> I am not happy with the 'manufactured' variable names. Maybe we should discuss about this to find a better solution. I commented the code (Java and TRPL) where this is handled.

These are only for intermediate use, right? Do they ever have to be legal names for any parsing purposes? If not, why not use illegal characters, so they cannot be mis-processed in TRPL code, in the (extremely rare) case that the current naming convention causes a conflict with real variables. Or is there some other aspect you don't like about them?

#23 - 05/20/2016 02:03 PM - Ovidiu Maxiniuc

[ECF: moved this entry to #3081-3]

#24 - 05/20/2016 02:16 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Ovidiu Maxiniuc wrote:
>
> > I am not happy with the 'manufactured' variable names. Maybe we should discuss about this to find a better solution. I commented the code (Java and TRPL) where this is handled.
>
> These are only for intermediate use, right? Do they ever have to be legal names for any parsing purposes? If not, why not use illegal characters, so they cannot be mis-processed in TRPL code, in the (extremely rare) case that the current naming convention causes a conflict with real variables. Or is there some other aspect you don't like about them?

Well, the code evolved a little since. Now the implementation does not relay on TRPL to add some variable processing, they are set directly when the

variable is injected. The node are used downstream when generating the JAST, but their definition is stripped and their value is stored in param map fed to interpreter.

In my example I used some __Pn__ pattern for testcases. This is invalid in P4GL so I switched to q__Pn__. The implementation uses varName = "hqlParam" + (i + 1) + "dq" pattern (see extractParams() before calling injectVariable()).However I am thinking of going back to first one: this is valid in Java and the way we generate identifiers makes collisions impossible and even that the identifier is invalid in P4GL, P2J conversion routines don't 'know' so there should not be any conflict.

#25 - 05/23/2016 08:15 AM - Eric Faulhaber

Ovidiu, please note that one of the changes in task branch 3109a (now P2J trunk 11032) was to increase the DQH cache size. Please carry this change along into your implementation of the 2-level cache, and make it configurable in the directory.

#26 - 05/23/2016 08:49 AM - Ovidiu Maxiniuc

Eric,
I am working on double cache solution. I encountered the following issue which I sensed but was not able to describe precisely:
In order to generify a predicate, it is preprocessed and its literals are replaced with variable. The generated JAST contains references to those 'placeholders'. To use the 1st level cache, the query must not have any parameter (no literal at all or they must not be converted) because otherwise the variables will not be initialized and the query will fail.
So, only non-literal queries can be stored in level 1 cache (the fast access, which does not require 4GL parsing). Also it does not make any sense to store these queries in level2 cache. In conclusion the two caches will store different partitions of the query set.

To take full-advantage of the double-cache solution we need to identify a-priori somehow the 'literal'-queries that HIT most the cache and fully convert them without parameterizing them fist so they could be stored in level#1 cache. Even if we fully-convert them a few times, in the long time, the faster access based on level#1 cache will compensate the time needed by 4GL parsing needed to create the access key for level#2 cache.

#27 - 05/23/2016 09:34 AM - Eric Faulhaber

I've been thinking about the same problem with the 2-level cache, but I came to a different conclusion. The first level cache is faster at generating a key (no parsing required). The second level cache takes longer to generate a key, but its parameterization allows better re-use of similar queries and avoids full conversion in a number of cases that fully convert today. Can't we combine these strengths?

What I'm envisioning is that the first time we encounter a query:

1. first level cache creates a fast key
2. first level cache lookup fails
3. second level cache creates a slow key + possible parameter artifacts
4. second level cache lookup fails
5. full conversion occurs, resulting in a parameterized AST
6. we put parameterized AST into second level cache, indexed by slow key
7. we put parameters (if any) and parameterized AST into first level cache, indexed by fast key
8. we execute the query

The next time we encounter the same query (exact match, including any literals), the flow would be:

1. first level cache creates a fast key
2. first level cache lookup succeeds (value contains parameters (if any) and parameterized AST)
3. we execute the query

No new puts are needed in this scenario. If we encounter the same base query (but with different literals), the flow would be:

1. first level cache creates a fast key
2. first level cache lookup fails
3. second level cache creates a slow key + possible parameter artifacts
4. second level cache lookup succeeds
5. we put parameters (if any) and parameterized AST into first level cache, indexed by fast key

6. we execute the query

Do you see a problem with that approach?

#28 - 05/23/2016 09:56 AM - Eric Faulhaber

The key to my suggested approach working is that the second scenario above cannot be appreciably slower than the current approach of executing a non-parameterized query after a cache hit. Or, if it is slower, the extra cost must be outweighed by the cost savings from avoiding multiple conversions of the same base query (i.e., differentiated only by literals). So, the question is: can the parameters be extracted, stored, and prepared in such a way that doesn't add appreciable expense to the execution process, compared to the current use of a non-parameterized JAST?

With this approach, we don't need to identify which queries would best be parameterized and which would not. I think trying to get that right would be error-prone.

#29 - 05/23/2016 10:02 AM - Ovidiu Maxiniuc

Thanks, Eric.
The golden idea was to save the (default) parameters, too, in with 1st level cache, so we could 'rehydrate' a parametrized query with missing values. If the slow key matched, the parameters should match also, otherwise the key is bad.

The parameters set (a map of usually no more than 5 elements) is really small compared to its JAST tree.

I am adjusting it right now and start ETF testing.

#30 - 05/24/2016 10:36 AM - Eric Faulhaber

Code review 2275a/11038:

The changes look fine. None of this code is used in the normal regression test environment, and I think we have tested with the ETF sufficiently. Please merge to trunk.

#31 - 05/24/2016 02:08 PM - Ovidiu Maxiniuc

The task branch 2275a was committed to trunk as revision 11034 and then archived. Notification was sent to team.
The server project was also updated to reflect the default values for cache sizes (revision 66).

#32 - 07/12/2016 12:02 PM - Eric Faulhaber

- Status changed from WIP to Closed

- % Done changed from 0 to 100

#33 - 11/16/2016 12:31 PM - Greg Shah

- Target version changed from Milestone 17 to Performance and Scalability Improvements