Database - Feature #4033

avoid query for FIND unique when the current record in the buffer would match the query result

04/03/2019 06:49 AM - Greg Shah

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Ovidiu Maxiniuc	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Database - Bug #4796: locking regression in RAQ introduced with th...	**WIP**

History

#1 - 04/03/2019 07:08 AM - Greg Shah

In discussions with a customer, it was noted that in the 4GL a FIND (or FIND FIRST etc) is very fast when the same record would be returned from the query. The customer's expectation was that no actual database trip is occurring in this case. For this reason, the code very liberally uses the technique (maybe thousands of times for the same exact record) because they perceive no cost.

The customer also pointed out that his example of FIND FIRST X WHERE field = <unique_condition> was inherently a unique result and didn't need the FIRST to be specified.

This led Constantin to consider the following optimization:

At conversion time, we can calculate when a field (or combination of fields) being referenced in a WHERE is referencing the current buffer AND the field or combination of fields would match a unique result (because of a unique index). In this case, we could check the following against the current buffer:

- The current record in the buffer meets the unique criteria.
- The record is not dirty (unflushed changes in any session, not just my session).
- If these critera are specified on either side of an AND operator.
- Some testing is needed to see if the 4GL handles the case in OR (left side maybe?). The concern here is that if the OR leads to more than 1 match for the query, then an ERROR can be raised under normal conditions. But we don't know if the 4GL still raises this error if the current record matches the condition.

If this happens, we think you can avoid the trip the the database and just immediately return.

To make this easier, we can generate a lambda expression that can be used to test the unique condition at runtime, without any HQL processing or interpretation.

This could save thousands or millions of database round trips. If Progress really does this, then it could explain a lot.

#2 - 04/03/2019 07:17 AM - Greg Shah

One idea to test this:

1. Write a 4GL testcases that does 10000 of the same FIND in a loop using a unique index match AND not in a temp-table. Measure the time in 4GL and FWD.

2. Manually edit the Java version of the testcase to bypass the FIND when the condition is matched by the current buffer. Measure the result in FWD.

#3 - 04/03/2019 08:06 AM - Constantin Asofiei

The table should have more than one record :-)

#4 - 04/03/2019 08:18 AM - Eric Faulhaber

This is an interesting idea, but the details of the execution present some questions.

Greg Shah wrote:

- The current record in the buffer meets the unique criteria.

How do we determine this efficiently, so as not to add overhead to all other types of FINDs which turn out not to meet these criteria?

- The record is not dirty (unflushed changes in any session, not just my session).

How do we determine this efficiently? Within the current session is not too bad, but across sessions potentially is expensive (a dirty database check).

#5 - 04/03/2019 08:23 AM - Eric Faulhaber

Is the assumption that the current session holds a lock on the record between FIND [FIRST] calls?

If NO-LOCK, wouldn't this optimization only apply if -rereadnolock were not in use? Otherwise, any subsequent FIND after the first one could be getting you a stale record.

#6 - 04/03/2019 08:47 AM - Constantin Asofiei

Eric Faulhaber wrote:

- The current record in the buffer meets the unique criteria.

> How do we determine this efficiently, so as not to add overhead to all other types of FINDs which turn out not to meet these criteria?

The idea here is the conversion rules emit this 'lambda expression' (for a tt1.f1 = 10 would be a () -> _isEqual(tt1.getF1(), 10) lambda expression) only if there is an unique index referenced in the WHERE clause (i.e. the FIND can always find only and only one record).

So we don't emit this lambda expression for every FIND, but for only specific cases. And this will help cases where the business logic does lots of FIND which retrieves the same record (already in the buffer). We could further make sure to evaluate the WHERE parameters only once (like tt1.f1 = func0() case, func0() is evaluated only once) , so that if we need to execute the HQL, we don't evaluate these again. Or we could just limit this optimization only when the other operand is a literal, or some expression which doesn't include function calls, dynamic buffer access, etc (which would have side-effects).

- The record is not dirty (unflushed changes in any session, not just my session).

> How do we determine this efficiently? Within the current session is not too bad, but across sessions potentially is expensive (a dirty database check).

Can we make a listener approach and notify the buffer (across sessions) that the table is dirty?

#7 - 04/03/2019 08:50 AM - Constantin Asofiei

Eric Faulhaber wrote:

> Is the assumption that the current session holds a lock on the record between FIND [FIRST] calls?

No, the lock doesn't matter.

> If NO-LOCK, wouldn't this optimization only apply if -rereadnolock were not in use? Otherwise, any subsequent FIND after the first one could be getting you a stale record.

Is not related to -rereadnolock.

#8 - 04/03/2019 08:52 AM - Constantin Asofiei

Regarding Greg's #4033-2 - the idea here is to omit the conversion changes now, and just test a runtime change (which would be faster to implement) by modifying the converted Java code (manually adding the lambda as a parameter to FindQuery), to see the performance impact for a large number of queries retrieving the same record.

#9 - 04/03/2019 09:09 AM - Eric Faulhaber

Constantin Asofiei wrote:

> No, the lock doesn't matter.

My point is that holding a lock on the record is the only way to be sure that the record has not been made stale by an update or deletion in another session, between repeated calls to this unique FIND. This must be true for Progress as well. If the application is written to tolerate receiving potentially stale data and we operate the same way, then this optimization would be ok.

> Is not related to -rereadnolock.

It is complicated by the -rereadnolock parameter, which can force us to go to the database to fetch the latest version of this record, if another buffer in the session already holds the record. Currently, this logic is implemented in the Persistence class, but it would have to be integrated into or accessible from whatever new logic potentially is introduced for this.

#10 - 04/03/2019 09:20 AM - Eric Faulhaber

Constantin Asofiei wrote:

> > How do we determine this efficiently? Within the current session is not too bad, but across sessions potentially is expensive (a dirty database check).
>
> Can we make a listener approach and notify the buffer (across sessions) that the table is dirty?

That itself is likely to introduce an enormous amount of new overhead. I have avoided this approach for other purposes in the past for this reason.

If we are just interested in whether the *table* is dirty, we already can get that information quickly from the dirty share infrastructure, without a query into the dirty database (see DefaultDirtyShareContext.isEntityDirty). However, that only tells us whether the table is dirty in another (or this) session's **uncommitted** transaction. We currently have no mechanism to determine whether a change has been **committed** since the last time we got a record in a FIND, because the assumption has always been that we will go to the database for that information.

#11 - 04/03/2019 09:42 AM - Constantin Asofiei

We are talking about an WHERE clause which is known that can always match on an unique record. If the lambda expression evaluates to true for the buffer's record, the only way that this record is not the intended record is if and only if this record was changed (or delete) and another record was made to match this unique condition.

Can we do a DB refresh for this record? And after that re-evaluate the lambda expression on this new record - if it still matches, then we are on the correct record.

Anyway, all this can be postponed until we have some metrics after testing this change - runtime-only and manually changing the Java code to use the lambda expression.

#12 - 04/03/2019 10:05 AM - Greg Shah

My suggestion was to test without the runtime change. Just write the conditional around the FIND in the converted code.

Also, I'm suggesting trying to get the idea if 4GL does this or not.

#13 - 04/03/2019 12:21 PM - Eric Faulhaber

Constantin Asofiei wrote:

> The table should have more than one record :-)

Yes, it should be a "real" table with a large number of records. Also, Hibernate second level cache and query cache should be enabled for the baseline, for a more realistic comparison.

#14 - 05/01/2019 01:58 PM - Greg Shah

I think that in addition to FIND unique, there are other cases (FIND FIRST/LAST) where we can track enough state change to know that if the query substitution parts are not changed AND the table has not been edited, then the WHERE must return the same record.

Eric and I have discussed this and it seems to be potentially correct.

- The query substitution parts can be cached and checked against newly evaluated versions.
- The table being edited is known by the dirty database manager. We can track this as an invalidation flag (the changes don't need to be tracked in detail to figure this out). The idea is a buffer that is in scope can be registered for tracking invalidation and the dirty database manager can store state for that invalidation using a simple tracking mechanism without storing all the details. We don't need timestamps to do this.

#15 - 07/01/2020 12:45 PM - Eric Faulhaber

- Assignee set to Ovidiu Maxiniuc

#16 - 07/06/2020 05:16 AM - Eric Faulhaber

Ovidiu, please prototype a basic query results cache for use with FIND {FIRST|LAST|unique} queries. The keys are a unique combination of DMO class (or table name), FQL where clause, QueryConstant indicating the "navigation" type of the query, index used, and an array of substitution parameters (if any). The values are DMO instances. The cache is used only by a single user context, so the use of DMO instances is safe and fast. Storing the index will require we look up the index for all such queries in RAQ.initialize, not just when we have a non-null dirty share context.

Before going to the database, any converted FIND which searches for the FIRST, LAST, or a unique record should check the query results cache first. If a record is found, no query is executed. The cache should be checked early, avoiding as much of the normal query set-up work as possible, that would otherwise be done by FindQuery or its parent class. On a cache miss, we go to the database and cache the result, if one is found. We may also want to cache the information that a result is not found, but this is better left for a second-pass prototype if the first yields promising results.

Don't worry about how we would invalidate/remove/expire cache entries just yet. That is too complicated for a quick prototype. We are just trying to get an idea of the performance improvement such a cache can make in optimal conditions, to see whether this is worth pursuing.

#17 - 07/06/2020 06:06 AM - Greg Shah

> Storing the index will require we look up the index for all such queries in RAQ.initialize, not just when we have a non-null dirty share context.

Is this just a map lookup?

#18 - 07/06/2020 06:20 AM - Eric Faulhaber

Greg Shah wrote:

> > Storing the index will require we look up the index for all such queries in RAQ.initialize, not just when we have a non-null dirty share context.
>
> Is this just a map lookup?

The first time a particular sort clause is encountered, there is some analysis to match an index, but it is not an intense effort. That match is cached so that subsequent lookups are simple map lookups. This analysis has not shown up as a bottleneck.

#19 - 07/06/2020 07:12 AM - Greg Shah

I understand. I just want us to keep in mind that it is easily possible to add a parameter to the query init that is a constant which represents the exact index chosen by conversion. The constant would to be something in the DMO class. Since we create such queries very often and (at least in the FIND case) the instances are not kept around, doing anything to eliminate processing for each FIND seems good. The 4GL has no such extra processing, because they resolve everything for FIND at compile time. I'm just proposing that we get every last cycle back.

#20 - 07/06/2020 02:42 PM - Eric Faulhaber

I want to finish the refactoring of the index support from strings to bit sets before we invest time in any conversion change here. In any case, I don't want to complicate the query results cache prototype.

#21 - 07/07/2020 07:34 PM - Ovidiu Maxiniuc

- File RandomAccessQuery.java added

Eric Faulhaber wrote:

> Ovidiu, please prototype a basic query results cache for use with FIND {FIRST|LAST|unique} queries.[...]

I have attached the RandomAccessQuery.java which contains all changes. The FastFind mode can be enable/disable by setting the private ffEnable field.

> Don't worry about how we would invalidate/remove/expire cache entries just yet. That is too complicated for a quick prototype. We are just trying to get an idea of the performance improvement such a cache can make in optimal conditions, to see whether this is worth pursuing.

I did not. At this moment, the naive implementation of initialValue() of ffContextCache will return plain HashMap. This can easily changed to anything smarter than that.

I used the following simple ABL program to test it:

```
DEFINE VARIABLE k AS INTEGER.
ETIME(yes).
   DO k = 1 TO 15:
      FIND FIRST book WHERE book-id EQ k NO-ERROR.
   END.
MESSAGE "to cache:" ETIME.
ETIME(YES).
   DO k = 1 TO 15:
      FIND FIRST book WHERE book-id EQ k NO-ERROR.
   END.
MESSAGE "from cache:" ETIME.
```

#22 - 07/08/2020 07:28 AM - Eric Faulhaber

I've tested this and modified your test case somewhat to stress the caching in different ways, and this definitely seems worth pursuing further.

Note that the optimization can only be done for FIND FIRST, LAST, and unique. NEXT and PREVIOUS won't work, because they are relative to some other record. Also, I don't know at the moment whether we need to differentiate what we are doing by the type of lock being used, but this may be relevant. Requires further thought/analysis...

I would like you to implement a full-featured cache to replace the prototype's hash map, external to RandomAccessQuery. It should be cross-context. I think we will need to make it a two-level lookup.

Cache Structure

The first level is keyed by index, which is a combination of table and index bitset. By "table", I don't necessarily mean table name or DMO interface Class. It is a logical concept of a unique identifier for a combination of database and table. I'm thinking we want this represented by a unique ordinal number, as this (combined with the bitset) would make for a faster hash key than would using a Class<?> or a String. TBD.

The second level is keyed by the components you have in the FastFindKey in the prototype, less the dmo and index (which are implicit to the first level). That is: FQL, navigation type, and substitution parameters.

The value stored at the second level is the RecordIdentifier of the cached record. BTW, we should use RecordIdentfier for the Session cache keys. Session$CacheKey is essentially the same thing. We need to store the identifier as the cache value, rather than the DMO instance itself, because:

- DMO instances cannot be shared across contexts;
- we don't want two sources of cached DMO instances.

TODO: we will need to limit the size of the cache. Best approach?

Cache Misses

The cache begins empty, but sized appropriately for the expected use, so we are not constantly resizing (goes back to fixed size issue mentioned above).

When a cache lookup misses, we find the record, if any, from the database as we normally do. We then update the cache as follows:

- If a DMO was found...
  - if the index used by the FIND is not already mapped in the first level, we add an entry for it;
  - we get the DMO's RecordIdentifier and add it to the second level of the cache.
- If a DMO was not found...
  - if the index used by the FIND is not already mapped in the first level, we add an entry for it;
  - we add a RecordIdentifier to represent the *lack* of result. This can be the DMO entity and a -1L for the primary key. Actually, this should be a singleton instance, e.g., public static final RecordIdentifier NO_RECORD = new RecordIdentifier("", -1L)); the table/entity name is not important. It is just a marker that the database has no match.

Cache Hits

When a cache hit occurs, the value retrieved is used with Session.get(RecordIdentifier) (a new API to which Session.get(String, Long) should delegate) to get the actual DMO. Note that this will result in a database fetch if the query results cache value represents a DMO which has expired from the Session cache.

We may want to compare the RecordIdentifier from the query results cache with the value in the FindQuery's record buffer (if any), *before* calling Session.get. I wonder if we can safely short-circuit calling RB.setRecord if there is a match with the buffer's currentRecord?

For cases where the cached value indicates no record was found, we would clear the buffer (again, if already clear, is it safe to bypass RB.setRecord?).

Cache Invalidation

All second level mappings associated with a particular index must be removed when that index is updated. This requirement is the reason the cache is two-level. While this may conservatively invalidate more than is necessary, it is already more granular than Hibernate's second-level query results cache, which invalidates all cached values for a table on any table update. We are at least discriminating by updated index. We need to bias this toward speed. A complex algorithm to verify whether individual cached results are actually affected by a specific index update would be too expensive.

I think the best place from which to do this is the Validation class, at the moment a DMO save occurs. We will need to know exactly which set of indices have been updated, and remove those entries from the cache. We already have some methods in BaseRecord which determine which indices are dirty, but we may need to do some refactoring to get the exact information we need here.

Concurrency

The cache will need to be thread-safe. Implementation TBD, but it must be fast.

ConcurrentHashMap ok?

Let me know if you have any questions or suggestions.

#23 - 07/08/2020 08:07 AM - Greg Shah

Can we expand this usage beyond FIND?

It seems to me that the 4GL uses the same index-based technique for these other cases (making them the equivalent to the FIND cases):

- FIND CURRENT
- FOR {FIRST | LAST | unique_match}
- DO PRESELECT {FIRST | LAST | unique_match}
- REPEAT PRESELECT {FIRST | LAST | unique_match}

Maybe this even could match the GET {FIRST | LAST | CURRENT}?

#24 - 07/08/2020 04:00 PM - Eric Faulhaber

FOR FIRST/LAST already is implemented by RandomAccessQuery, so we should get that for free.

FIND CURRENT should be ok at least if we are re-fetching with a less restrictive lock type than we currently have on the record. In the other direction, I'm not sure (see my comment in the previous post about whether lock type should come into play). However, the more I consider this, the more I think we should be ok, even with different lock types, because we are tracking all changes made by the application to the associated index, and invalidating the cache and forcing a database fetch when anything changes.

Now we get to the higher-hanging, less plentiful fruit...

I haven't looked at DO/REPEAT PRESELECT in a while, so I'm not sure how easily applicable the implementation for these would be. However, they have always been much less commonly used in the code we've seen.

GET operates on an existing result set, so we've already taken a trip to the database by the time GET is called to get the "main" results. GET NEXT is usually the command hit most frequently (PREV less so), as it walks the results of a query. So, I'm not sure there's much advantage to implementing these.

#25 - 07/08/2020 04:05 PM - Eric Faulhaber

Ovidiu wrote (in email):

> I have created the general stubbing for the advanced cache but for the moment I do not have access to BitSet indices, only the name of the current index in RAQ.

I would like to say let's keep the first pass simple and use the index name instead, except that we are working with the bit set representation already in the classes which trigger cache invalidation. So, we will need a bridge from index name to bit set representation in RAQ temporarily. We want to move over to the bit set form in RAQ ultimately anyway, but the dirty share code is not ready for it yet.

#26 - 07/08/2020 04:29 PM - Eric Faulhaber

Please note I have made some updates to #4033-22 to clarify handling of the "no record found" case.

#27 - 07/09/2020 02:44 PM - Ovidiu Maxiniuc

More details from 4011c/11554:

- I added a table/DMO unique ID. This (the Level1key) is the only place used, yet. The whole class/string is reduced to a single int. There is a disadvantage, when it comes to dynamic tables: for the moment these uids cannot be reused;
- I added the missing bridge between indices name and their BitSet representation in RecordMeta. Not sure if this mapping will remain after 4011x, but momentarily it works;
- I am a bit concerned about the fql: it is always obtained as: getHelper().getHQLPreprocessor().getHQL().toFinalExpression(), a bit too complex;
- Instead of RecordIdentifier s values I used simple Long data, with same meaning. The table information from RecordIdentifier is known when the cache is queried, and if the Record is on FWD side, it certainty is in Session's cache so it will be fetched with Session.getCached(DMO, pk) API;
- the computed keys are returned in the event of a cache miss. The very same pair of keys will be used when storing the information in cache. That is a bit of similar code for analysing the cache query result, but I do not believe it could be written more compact (maybe using some lambda expressions).

#28 - 07/10/2020 12:43 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> More details from 4011c/11554:
>
> - I added a table/DMO unique ID. This (the Level1key) is the only place used, yet. The whole class/string is reduced to a single int. There is a disadvantage, when it comes to dynamic tables: for the moment these uids cannot be reused;

Temp-tables are a good point; I did not address these in the initial spec. Clearly, the cache needs to be context-local for temp-tables.

Is there a practical problem caused by the temp-table IDs not being re-usable, or is the problem just theoretical?

> - I added the missing bridge between indices name and their BitSet representation in RecordMeta. Not sure if this mapping will remain after 4011x, but momentarily it works;

Once dirty database has been adapted to use bit sets to represent indices, we probably will get rid of it.

> - I am a bit concerned about the fql: it is always obtained as: getHelper().getHQLPreprocessor().getHQL().toFinalExpression(), a bit too complex;

Please provide some context. What logic needs to obtain it this way?

> - Instead of RecordIdentifier s values I used simple Long data, with same meaning. The table information from RecordIdentifier is known when the cache is queried, and if the Record is on FWD side, it certainty is in Session's cache so it will be fetched with Session.getCached(DMO, pk) API;

The only downside here is that the RecordIdentifier needs to be constructed at every session cache lookup. The idea behind using RecordIdentifier as the cache value was that, once it replaced Session$CacheKey and we have a direct API to get a session-cached record by RecordIdentifier, we will need to construct these key objects (and thus compute a hash code) a lot less often in the query results cache hit case.

> - the computed keys are returned in the event of a cache miss. The very same pair of keys will be used when storing the information in

> cache. That is a bit of similar code for analysing the cache query result, but I do not believe it could be written more compact (maybe using some lambda expressions).

Good idea.

#29 - 07/10/2020 01:38 PM - Eric Faulhaber

I left the topic of synchronization pretty open. Could you please tell me what your thoughts are there? Of course, this does not apply to the temp-table version of the cache.

#30 - 07/10/2020 08:48 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Is there a practical problem caused by the temp-table IDs not being re-usable, or is the problem just theoretical?

The DmoMetadataManager stores them in a simple array for fastest access. The permanent and static temporary tables will keep their uids for the lifetime of the process. Their number is fixed and known at conversion time. The dynamic temp-tables will need to discarded when their life is over. I believe we can use a bitmask for these and store in a different place, like a normal map instead. They will pay some extra price for lookup.

- I am a bit concerned about the fql: it is always obtained as: getHelper().getHQLPreprocessor().getHQL().toFinalExpression(), a bit too complex;

> Please provide some context. What logic needs to obtain it this way?

Sorry, I saw the problem more complicated than it is. The where predicate is enough. We do not need the fully processed fql, just a value to identify the query. Although they are similar from POV of this location, using the where is simpler and faster.

- Instead of RecordIdentifier s values I used simple Long data, with same meaning. The table information from RecordIdentifier is known when the cache is queried, and if the Record is on FWD side, it certainty is in Session's cache so it will be fetched with Session.getCached(DMO, pk) API;

> The only downside here is that the RecordIdentifier needs to be constructed at every session cache lookup. The idea behind using RecordIdentifier as the cache value was that, once it replaced Session$CacheKey and we have a direct API to get a session-cached record by RecordIdentifier, we will need to construct these key objects (and thus compute a hash code) a lot less often in the query results cache hit case.

The Session works with its 'proprietary' CacheKey key. I switched to it (making it public in Session), so less objects are created. However, there are two issues:

- momentarily CacheKey and RecordIdentifier are a bit incompatible. The former uses the DMO implementation class' name, the latter uses the table name;
- the CacheKey (and RecordIdentifier) has a String component. The Loader used for getting Record s on Session cache miss needs the DMO implementation class.
  I am a bit clicked here. I think we also should switch to the new uids of dmo here.

  > I left the topic of synchronization pretty open. Could you please tell me what your thoughts are there? Of course, this does not apply to the temp-table version of the cache.

Since the temp-tables are contex-local, they do not need synchronization any more. Only the permanent database(s). The problem is, the access not atomic. Especially that it is done in two stages for a cache miss: lookup and update, using the same keys. Between them, a true database access takes place. Normally, I would group all that is a single synchronization block, but this may cause threads to wait too long. As result, the l1 and l2 keys will probably need to be validated when updating the cache.

Note I have committed 40111c/r11555, but it is not compilable because of the Session.get(CacheKey): I cannot extract the dmoImplClass from the key parameter.
And I have another question: who sends the index invalidation message? I think this is related to dirty db checking.

#31 - 07/10/2020 09:51 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The Session works with its 'proprietary' CacheKey key. I switched to it (making it public in Session), so less objects are created. However, there are two issues:

I don't want CacheKey to be public, I want to get rid of it. The session cache will not be staying in the session for long, but that is a different task. It is beyond the scope of this task, but I don't want to expand the access of CacheKey outside the Session class, because that will make that task harder.

- momentarily CacheKey and RecordIdentifier are a bit incompatible. The former uses the DMO implementation class' name, the latter uses the table name;

I don't really consider this an incompatibility. The basic nature of RecordIdentifier is that it is a string and a long. We have long since used the string part for different purposes, so this incompatibility is only an issue if we "leak" the key for other uses, like with the lock manager. Even there, the choice to use the table name instead of the DMO class name was somewhat arbitrary: I chose the string which would in most cases be shorter.

However, I see your point that using this class for different purposes can be confusing, and I prefer your idea of using the faster key (i.e., with the table uids). However, at this point, I want to go with a key that is as fast as is practical, without increasing the scope of this task. I suspect that switching away from a string/long key (like CacheKey or RecordIdentifier) at this point will have further-reaching repercussions, and we can't afford any scope creep for this task. But you have been looking at this more closely, so you tell me. If we stick with a string/long key, I want to use RecordIdentifier and get rid of CacheKey, because they are redundant. If that means we need to change some javadoc to make the flexible nature of the string component more clear, so be it.

In fact, it could be even simpler (and faster). When I first wrote Session, I intentionally avoided using the DMO entity name as part of the key, like Hibernate did. Hibernate has to support a broader range of use cases, where, for example, new primary keys are not necessarily unique across tables. I used a simple Long as the cache key intentionally, because in FWD, primary keys are guaranteed to be unique across all tables of a database, so long as the initial import is done with FWD, and anyone creating new records always uses the p2j_id_generator_sequence for new keys. You later added the DMO entity name and created CacheKey. We had so much change going on at the time that I didn't want to argue this particular point, because technically, your key was safer anyway. Also, I guess we added dependencies on having the class name in Loader.

- the CacheKey (and RecordIdentifier) has a String component. The Loader used for getting Record s on Session cache miss needs the DMO implementation class.
  I am a bit clicked here. I think we also should switch to the new uids of dmo here.

I am not clear on why the cache miss case is any more complicated in terms of needing the DMO implementation class than it is to use Session the way we do now, without any query results cache.

> I left the topic of synchronization pretty open. Could you please tell me what your thoughts are there? Of course, this does not apply to the temp-table version of the cache.

> Since the temp-tables are context-local, they do not need synchronization any more. Only the permanent database(s). The problem is, the access not atomic. Especially that it is done in two stages for a cache miss: lookup and update, using the same keys. Between them, a true database access takes place. Normally, I would group all that is a single synchronization block, but this may cause threads to wait too long. As result, the l1 and l2 keys will probably need to be validated when updating the cache.

The synchronization must be biased toward maximum overall throughput. Remember, this cache is just about an optimization. It is better to lose that optimization in individual cases, if the overall throughput is faster. So, I don't think the entire check/miss/update the cache process needs to be atomic. If another session comes in with a cache check and does not get the benefit of a record having been added to the cache in another user context because of timing, that particular fetch will just be slower (because it now has to go to the database), but it will still be correct. And overall throughput will be less impacted because we are not in a critical section for longer than needed. The only purpose of synchronization is to prevent corruption of the cache data. It is better to get no result from the cache, than a stale one.

> Note I have committed 40111c/r11555, but it is not compilable because of the Session.get(CacheKey): I cannot extract the dmoImplClass from the key parameter.

Thanks, I will have a look.

> And I have another question: who sends the index invalidation message? I think this is related to dirty db checking.

This cache has nothing to do with dirty database checking and should have absolutely no functional dependencies on it. The long term goal is still to remove the dirty share functionality as aggressively as possible, so do not assume it will be active for any given table. Plus, it is never active for temp-tables.

The Validation class must invoke cache invalidation, at the moment just before a DMO is inserted or updated (for real, not the validation-only case). It is at this point that we know exactly which indices are dirty and that they are about to be changed in the database. If the insert/update is rolled back due to a record validation problem, the cache invalidation happens regardless. We can't wait until after the insert/update, because then another session could have gotten stale information from the cache.

#32 - 07/13/2020 03:27 AM - Eric Faulhaber

I reviewed the implementation as of 4011c/11555. In addition to my previous answers in #4033-31, which among other things, discuss the cache key and synchronization, I have some questions/comments.

Why is the FastFindCache$Result class necessary? I am not sure what the L1Key and L2Key instance variables here are for. Shouldn't the result just be the key of the DMO (whatever that ends up being, exactly) in the session cache?

You made changes to the various RAQ.initialize calls. I didn't double-check all the new call paths. Was this change just to flatten the call stack with fewer intermediate calls for performance, or was there a functional change? Either way, please be very sure you did not change any of the default values being passed from less specific to more specific variants of the initialize method.

In RAQ.initialize (the worker variant), please put the result of buffer.getDirtyContext() in a local variable and re-use it.

You can get rid of the RAQ.ffCache variable. That was only for the prototype. The cache will always be enabled. By getting rid of this and

FastFindCache$Result, the cache check code can be simplified. This logic should be in its own method, since it appears to be duplicated.

In the cache check logic, you have this:

```
try
{
   dmo = buffer.getSession().get(ffr.getResult());
}
catch (PersistenceException e)
{
   // TODO: what to do now?
   e.printStackTrace();
}
```

In the exception case, dmo should remain null and we should fall through to the normal find behavior, so I think we're ok. Just instead of e.printStackTrace(), do proper logging (WARNING level, with the stack trace).

There are what appear to be unrelated changes to SequenceManager, TempTableBuilder, TempTableHelper, MetadataManager, Loader, and the Dialect subclasses. Please limit 4011c to cache-related changes.

#33 - 07/13/2020 04:24 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Why is the FastFindCache$Result class necessary? I am not sure what the L1Key and L2Key instance variables here are for. Shouldn't the result just be the key of the DMO (whatever that ends up being, exactly) in the session cache?

The Result class temporarily saves the two keys for the moment of cache insert, avoiding the keys to be created a second time. They are saved only in the event of a cache miss, if we have a cache hit, only the result is returned in the Result object.

> You made changes to the various RAQ.initialize calls. I didn't double-check all the new call paths. Was this change just to flatten the call stack with fewer intermediate calls for performance, or was there a functional change? Either way, please be very sure you did not change any of the default values being passed from less specific to more specific variants of the initialize method.

Yes, they will flatten the call stack, cutting up to three intermediary calls that were just adding the default value of an optional parameter.

> In RAQ.initialize (the worker variant), please put the result of buffer.getDirtyContext() in a local variable and re-use it.

OK. Done.

> You can get rid of the RAQ.ffCache variable. That was only for the prototype. The cache will always be enabled. By getting rid of this and FastFindCache$Result, the cache check code can be simplified. This logic should be in its own method, since it appears to be duplicated.

> In the cache check logic, you have this:
>
> [...]
>
> In the exception case, dmo should remain null and we should fall through to the normal find behavior, so I think we're ok. Just instead of e.printStackTrace(), do proper logging (WARNING level, with the stack trace).

OK, Done.

> There are what appear to be unrelated changes to SequenceManager, TempTableBuilder, TempTableHelper, MetadataManager, Loader, and the Dialect subclasses. Please limit 4011c to cache-related changes.

OK, I reverted them.

Related previous note: indeed CacheKey and RecordIdentifier are, semantically identical. I used the CacheKey to be able to call directly the new Session.get/put(CacheKey) API. I avoided using RecordIdentifier since I noticed CacheKey and RecordIdentifier have different string content. Practically, we can drop CacheKey and replace it with RecordIdentifier.

#34 - 07/13/2020 04:46 AM - Eric Faulhaber

OK, let's use RecordIdentifier. We may need to intern the strings, but this has a performance cost of its own. In the long run, I'd like to use the UID in the key as you suggested, but not now, if it will add significant effort.

The thread safety and invalidation will have to be addressed next.

At this time, BaseRecord does not have an API to report dirty indices (only dirty *unique* indices), but we will need this. The code to gather this type of information is already in that class, but it will need to be refactored a bit to provide exactly what we need for the invalidation.

Let me know your thoughts on the thread safety. It is a bit complicated by the two-level cache. Also, when designing a solution, note that even a get operation on the LRU cache alters the internal state of that object.

#35 - 07/13/2020 06:40 PM - Ovidiu Maxiniuc

I have just committed r11557 of 4011c.
Changes:

- I introduced a numeric identifier for indexes. In fact, the were already used in BaseRecord by getFullyDirtyUniqueIndices() and

getFullyDirtyNonuniqueIndices(). The problem is the BitSet result of these methods were overlapping, both unique and non-unique shared the same id. So I decided to keep the positive id for unique and negative for non-unique. 1-base, so 0 is reserved/invalid. I documented the accessors and usages;
- the Level 1 cache key is now a pair of integers. Fast and simple;
- the FastFindCache$Result was dropped, but the keys were pulled in caller methods. Although K1 is not rather simpler (2 integers) the K2 might be quite complex because of hql String and substitution array. I would prefer to compute it a single time at get and possible reuse the K2 if the cache needs to be updated;
- the Validation.checkUniqueConstraints() will trigger the cache invalidation for indices that are affected by saving a record. However, I feel this is not enough. The same should be done when the records are dropped, too;
- the CacheKey was completely dropped and replaced with RecordIdentifier. I added just a few lines in javadoc to make the programmers aware of overloaded usage;
- the instances of FastFindCache are not members of RAQ. Cannot be disabled;
- I needed access from RecordMeta to DmoMeta so the construction of Impl classes changed a bit. Not big changes, mostly parameters that get saved as final members accessible through getters;
- the only thing that remains is the synchronization for permanent tables. I will handle this tomorrow.

Please review.

#36 - 07/14/2020 05:47 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> I have just committed r11557 of 4011c.
> Changes:
>
> - I introduced a numeric identifier for indexes. In fact, the were already used in BaseRecord by getFullyDirtyUniqueIndices() and getFullyDirtyNonuniqueIndices(). The problem is the BitSet result of these methods were overlapping, both unique and non-unique shared the same id. So I decided to keep the positive id for unique and negative for non-unique. 1-base, so 0 is reserved/invalid. I documented the accessors and usages;

Good.

- the Level 1 cache key is now a pair of integers. Fast and simple;

Good.

- the FastFindCache$Result was dropped, but the keys were pulled in caller methods. Although K1 is not rather simpler (2 integers) the K2 might be quite complex because of hql String and substitution array. I would prefer to compute it a single time at get and possible reuse the K2 if the cache needs to be updated;

So why not hide the complexity of the two level cache implementation entirely?

Since we are storing all found results (including the lack of a record), we know we will always call both get (first) and put (second) each time we use the cache, We also know:

- we need both l1 and l2 keys in the event of a cache hit (for both levels of get); and
- we need both l1 and l2 keys in the event of a cache miss (for at least the first level of get, and for both levels of put).

Since we need both keys in either case, I propose a public, static, inner class called FastFindCache.Key, which stores both l1 and l2 keys. On the get, Key creates both L1Key and L2Key internally. If nothing is found at level 1, the L2Key will still be needed for the put, so the effort is not wasted. Or

you could create it lazily during the put. You could even store a reference to the level 2 map in Key, if one is found, so that a separate lookup in the subsequent put call is not needed. Callers use the public Key instance for both the get and put, and they know nothing of the internals of the cache.

- the Validation.checkUniqueConstraints() will trigger the cache invalidation for indices that are affected by saving a record. However, I feel this is not enough. The same should be done when the records are dropped, too;

Yes, good point. For inserts and deletes, cached results for all indices of that DMO type would be invalidated. Only updates would potentially invalidate a subset of the indices.

- the CacheKey was completely dropped and replaced with RecordIdentifier. I added just a few lines in javadoc to make the programmers aware of overloaded usage;

Good.

- the instances of FastFindCache are not members of RAQ. Cannot be disabled;

Good.

- I needed access from RecordMeta to DmoMeta so the construction of Impl classes changed a bit. Not big changes, mostly parameters that get saved as final members accessible through getters;

OK.

- the only thing that remains is the synchronization for permanent tables. I will handle this tomorrow.

Sounds good.

> Please review.

Code review 4011c/11557:

The logic in findNext doesn't look quite right, because we are caching the result, even if we went down the activateNext path. It should only be cached if we went down the activateFirst path. Same observation for findPrevious.

There are a few other things I think we need to handle.

A RAQ can have a join to another table or embedded CAN-FINDs. Since this cache is based on the indices of a single table, and the cache lookup is based on a single index of that table, we need to skip the cache check if there is a join or an embedded CAN-FIND. The first case should be detectable if join is non-null. I think you can detect the second case if externalBuffers is non-null. However, both cases should be tested to be sure.

The same restriction applies to a client-side where clause expression. In this case whereExpr will be non-null.

Since there is a lot of duplication of the cache logic in RAQ, that logic should be consolidated into one or more methods. In fact, since the cache logic is duplicated in several places, should it be in all these separate places at all? Shouldn't the cache check be centralized in the RAQ.execute method, using the activeBundleKey to differentiate the behavior as needed between FIRST, LAST, and UNIQUE?

There are still some classes in the branch which don't seem to have anything to do with the caching (e.g., TransactionTableUpdater)

The invalidation doesn't look quite right to me.

First, I think we have to invalidate any updated index *before* the change is persisted to the database. Otherwise, another thread could get a stale cache result between the time we perform the save and when we invalidate the cache. In the event of an error persisting the record, we will have invalidated the cache unnecessarily, but I think it's better to lose the optimization than to get incorrect results. The other option here is to figure out a synchronization strategy that lets us invalidate only after a successful save, which does not let another thread in to get a stale cached result. If we go this route, we are still sacrificing performance (overall throughput), just in a different way.

Second, the bit set returned by BaseRecord.getFullyDirty[Non]UniqueIndices is being used. Fully dirty means *every* index component has been

touched. I think we need to be more conservative and invalidate every index which has had *any* of its components touched.

Third, in the case of an insert, we can assume all indices have been touched, and we don't need the DMO to do any work to figure out which were actually dirty at the time of the save. In this case, we can just pass in bit sets of the proper length, with all bits turned on. As you noted, we will need a similar invalidation for a delete (bulk delete as well), for which we can again assume all indices are affected.

The rest of the changes look good to me.

#37 - 07/14/2020 07:32 PM - Ovidiu Maxiniuc

Eric,

I adjusted the code as you suggested. The findNext/findPrevious disappeared by themselves. By extracting the fast-find-cache changes to execute() the code has greatly simplified. The triggers for index invalidation were moved/added before database change. See r11561.

While debugging my testcases I encountered a few issues:

- invalidating the cache for insert/delete (all indices) is a bit costly by iterating all possible L1 keys for a specific table, but probably there is no better solution except for adding a new level. Another alternative could have been to iterate all L1 entries and setValue(null) for those matching the dmoUid. Difficult to estimate the best;
- what do I invalidate on UNDO? The logical answer here is everything, as we really do not know what the jump to savepoint has affected. This mean all instances of find-fast. Well, excluding the no-undo temp-tables;
- I had a thought or two on dirty database and whether those operation. As long as it do not have indices it seems out of the scope of this task.

#38 - 07/14/2020 10:16 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Eric,
>
> I adjusted the code as you suggested. The findNext/findPrevious disappeared by themselves. By extracting the fast-find-cache changes to execute() the code has greatly simplified. The triggers for index invalidation were moved/added before database change. See r11561.

I'll post a review separately.

> While debugging my testcases I encountered a few issues:
>
> - invalidating the cache for insert/delete (all indices) is a bit costly by iterating all possible L1 keys for a specific table, but probably there is no better solution except for adding a new level. Another alternative could have been to iterate all L1 entries and setValue(null) for those matching the dmoUid. Difficult to estimate the best;

I was worried about invalidation actually adding overhead. The fact that you've already noticed a problem in your early testing is confirming that concern. Clearly, we cannot make performance worse by adding noticeable overhead to maintain this cache.

We have to make a decision: is the granularity of tracking by index worth the overhead of invalidation by index? If we reduce the granularity to track results instead by DMO/table, do we lose much of the value of the caching, because we are invalidating results too aggressively?

If you look at the early history entries in this issue, the idea started as just a way to avoid going to the database when we repeat a FIND with the same criteria, multiple times, for the same buffer. The idea evolved into a more general purpose query results cache for single-DMO queries, shared across user contexts. To achieve the original purpose, table-level granularity would be enough.

If we keep the index-level granularity, I would think a third level would be the least expensive way to do this.

I'll think about this as I review the code...

- what do I invalidate on UNDO? The logical answer here is everything, as we really do not know what the jump to savepoint has affected. This mean all instances of find-fast. Well, excluding the no-undo temp-tables;

We cannot invalidate the entire cache every time some user context invokes UNDO. This would result in the cache always being empty and never being useful. We track a lot of information about undoable operations. I will think about this and try to come up with an approach...

- I had a thought or two on dirty database and whether those operation. As long as it do not have indices it seems out of the scope of this task.

I'm not sure what you are asking.

#39 - 07/14/2020 10:49 PM - Eric Faulhaber

Code review 4011c/11561:

In RAQ.initialize, it seems we no longer need to make the lookup of the index information conditional, because FastFindCache.getInstance(buffer.isTemporary()) != null will always evaluate to true.

SequenceManager seems out of place in this branch.

Several classes (TempTableHelper, the Dialect subclasses, Loader, possibly MetadataManager) seem to regress changes added in 4011b which were picked up in the rebase. This probably is because you included them in an earlier revision of 4011c and I asked you to back them out. However, your reversion of the changes apparently resulted in them being ignored during the rebase.

Everything else looks good, though we are still missing synchronization and invalidation on UNDO.

#40 - 07/14/2020 11:02 PM - Eric Faulhaber

Eric Faulhaber wrote:

> Everything else looks good, though we are still missing synchronization and invalidation on UNDO.

Also...maybe I missed it, but did you add invalidation on delete and bulk delete?

#41 - 07/14/2020 11:10 PM - Eric Faulhaber

For the invalidation on UNDO, please see SavepointManager$Block.rollback(Session).

I think what we need to do is, just before the loop which rolls back the elements of the changed set, collect the set of distinct DMO IDs represented by the changed DMOs, then loop through those and invalidate them. This should be done just before the existing loop that executes the rollbacks.

As I look at all the places we need to invalidate the cache for a full table, and how often these code paths are executed (particularly insert and undo operations), I am concerned by how much work is involved in invalidation. To reduce this, I think we need to refactor the cache to have level 1 use just the DMO ID, and to push the index into level 2. I don't think a three level cache is the way to go, unless you can think of a way to make the gets and puts very efficient.

All invalidation will be at level 1, which should mean less work figuring out dirty indices on DMO updates.

Unfortunately, this means we lose the granularity by index, which increases the chances of cache misses in a system where lots of user contexts are active.

Let's give it a try this way and do some profiling to see how it impacts performance.

#42 - 07/14/2020 11:11 PM - Eric Faulhaber

What is your plan regarding thread safety for the shared form of the cache?

#43 - 07/15/2020 06:23 AM - Greg Shah

> I think what we need to do is, just before the loop which rolls back the elements of the changed set, collect the set of distinct DMO IDs represented by the changed DMOs, then loop through those and invalidate them. This should be done just before the existing loop that executes the rollbacks.
>
> As I look at all the places we need to invalidate the cache for a full table, and how often these code paths are executed (particularly insert and undo operations), I am concerned by how much work is involved in invalidation. To reduce this, I think we need to refactor the cache to have level 1 use just the DMO ID, and to push the index into level 2. I don't think a three level cache is the way to go, unless you can think of a way to make the gets and puts very efficient.
>
> All invalidation will be at level 1, which should mean less work figuring out dirty indices on DMO updates.
>
> Unfortunately, this means we lose the granularity by index, which increases the chances of cache misses in a system where lots of user contexts are active.

Now is the time to get this right. The 4GL deals with things at an index level and we should strive to do the same. For example, can you can keep sets of the per-index values such that the entire set of keys can be removed in a single operation without looping? Such sets would only need to be maintained on add/remove but not in normal lookups.

#44 - 07/15/2020 10:31 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> - I had a thought or two on dirty database and whether those operation. As long as it do not have indices it seems out of the scope of this task.
>
> I'm not sure what you are asking.

I was just thinking whether the references to dirty record instance have the chance to be saved in fast-cache. I guess not because two reasons: their tables are not indexed and they are not queries by RAQ.

> In RAQ.initialize, it seems we no longer need to make the lookup of the index information conditional, because FastFindCache.getInstance(buffer.isTemporary()) != null will always evaluate to true.

That's correct. I am dropping the truism.

> Also...maybe I missed it, but did you add invalidation on delete and bulk delete?

You can find the calls in RecordBuffer:7688 (before calling persistence.delete(currentRecord);) and TemporaryBuffer:6054 (before calling removeRecords()). Both of them invoke FastFindCache.invalidate(int dmoUid) variant.

#45 - 07/15/2020 10:51 AM - Ovidiu Maxiniuc

Greg Shah wrote:

> Now is the time to get this right. The 4GL deals with things at an index level and we should strive to do the same. For example, can you can keep sets of the per-index values such that the entire set of keys can be removed in a single operation without looping? Such sets would only need to be maintained on add/remove but not in normal lookups.

Yes, we do this. The L1 key (compound of table uid and index uid) allows access to the set of all records of an index. When an index is invalidated, the all set of records that were cached because the query used the index is dropped.

I am adding at this moment the 3-rd cache level that will allow a quick cache invalidation at table level, at a price of an extra lookup, but the operation should be relatively fast as the keys are plain integers.

#46 - 07/15/2020 03:10 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> For the invalidation on UNDO, please see SavepointManager$Block.rollback(Session).

In the case of a UNDO,LEAVE. statement, things are more complicated. There is another execution path. In this case, the Session.rollback() is invoked. At the moment when conn.rollback() is executed, the savepointManager does not have any block information to extract the list of changed records. Is there a way to save them before?

#47 - 07/15/2020 03:17 PM - Eric Faulhaber

From where is Session.rollback called in this case?

#48 - 07/15/2020 03:21 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> From where is Session.rollback called in this case?

```
      at com.goldencode.p2j.persist.orm.Session.rollback(Session.java:1203)
      at com.goldencode.p2j.persist.Persistence$Context.rollback(Persistence.java:4391)
      at com.goldencode.p2j.persist.BufferManager$TxWrapper.rollback(BufferManager.java:3074)
      at com.goldencode.p2j.util.TransactionManager$WorkArea.notifyMasterCommit(TransactionManager.java:10255)
      at com.goldencode.p2j.util.TransactionManager$WorkArea.access$4900(TransactionManager.java:9750)
      at com.goldencode.p2j.util.TransactionManager.processRollback(TransactionManager.java:6361)
      at com.goldencode.p2j.util.TransactionManager.rollbackWorker(TransactionManager.java:4147)
      at com.goldencode.p2j.util.TransactionManager.rollbackTopLevel(TransactionManager.java:4047)
      at com.goldencode.p2j.util.TransactionManager.access$6500(TransactionManager.java:602)
      at com.goldencode.p2j.util.TransactionManager$TransactionHelper.rollbackTopLevel(TransactionManager.java
:7903)
      at com.goldencode.p2j.util.BlockManager.undoReturnNormalTopLevel(BlockManager.java:7157)
      at com.goldencode.testcases.p4011.Ask$2.body(Ask.java:261)
```

#49 - 07/15/2020 03:34 PM - Eric Faulhaber

SavepointManager registers for the same master transaction rollback hook that is leading to this call, so it should be getting notified in the same loop from here:

```
at com.goldencode.p2j.util.TransactionManager$WorkArea.notifyMasterCommit(TransactionManager.java:10255)
```

I think if you already have the cache invalidation logic in SavepointManager, we should be covered in this case.

#50 - 07/15/2020 03:55 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> I think if you already have the cache invalidation logic in SavepointManager, we should be covered in this case.

I added the cache invalidation code in SavepointManager.rollback(Session session), but it is not called in this case. Could it be a bug that the SavepointManager does not get master transaction rollback notification, or an optimization?

#51 - 07/15/2020 04:04 PM - Eric Faulhaber

It is a bug in SavepointManager.registerBlockHooks. We should be differentiating between full and sub-transaction, calling txHelper.registerTransactionCommit for the former and txHelper.registerCommit for the latter.

We have the full vs. sub-transaction information in BufferManager.beginTx. It should be passed through BufferManager$TxWrapper.registerSavepointHooks to SavepointManager.registerBlockHooks as an additional parameter: fullTx. Please make this change.

#52 - 07/15/2020 05:26 PM - Ovidiu Maxiniuc

- Status changed from New to WIP

Eric,
I committed r11563. It contains the last two issues: 3-level cache and synchronization. I must admit I did not fully tested but it seems stable and I did not notice any penalty from last changes. This is understandable since for synchronization a more elaborate test environment is required.

#53 - 07/15/2020 06:32 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Eric,
> I committed r11563. It contains the last two issues: 3-level cache and synchronization. I must admit I did not fully tested but it seems stable and I did not notice any penalty from last changes. This is understandable since for synchronization a more elaborate test environment is required.

Code review 4011c/11563:

I think it's almost finished.

I think you are right that the 3 level implementation should be ok, since the keys for the first two levels are so simple.

I'm concerned synchronizing on the cache instance might be too coarse for an active system, and may create a bottleneck. We've seen this with H2 synchronizing on the database, for example. However, we can refine this later, if we want. I don't want to want to hold up the first pass for this.

I think the "bulk" invalidate method could be more efficient. The DMOs stored in SavepointManager$Block.changed are not necessarily from distinct tables. There can be many from the same table. In fact, this probably is the common case. By collecting the DMO UIDs in a List in SavepointManager$Block.invalidateFastFind, we potentially are attempting to remove the same UID many times. A Set would be better, I think.

An instance of SavepointManager will never manage a mixture of temp-table records and persistent table records in Block.changed. An instance only works with a single database. So, there is no reason to have conditional logic to determine which cache instance to use in invalidateFastFind. It would be better to pass the knowledge of temp-table vs. persistent to the SavepointManager c'tor from BufferManager.TxWrapper (it knows which database it's working with), and get the right cache instance once. This will save all the calls to get the FFC instance during invalidation, some of which require an expensive context-local lookup.

Sorry if I was not clear earlier about RAQ.initialize. I meant that the first if (dirtyContext != null) conditional needs to go away completely. The index lookup always needs to occur now, because the cache needs it, too. So, we always need the index and indexName. The second if (dirtyContext != null) should be put back, because the dmoSorter is only needed for dirty share processing.

I didn't ask this before, and I'm not asking you to change it, but why did you relocate the registry of DMO classes from DmoClass to DmoMetadataManager?

#54 - 07/15/2020 07:30 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> I'm concerned synchronizing on the cache instance might be too coarse for an active system, and may create a bottleneck. We've seen this with H2 synchronizing on the database, for example. However, we can refine this later, if we want. I don't want to want to hold up the first pass for this.

OK. And maybe we will have some input from test runs to fine-tune some parameters.

> I think the "bulk" invalidate method could be more efficient. The DMOs stored in SavepointManager$Block.changed are not necessarily from distinct tables. There can be many from the same table. In fact, this probably is the common case. By collecting the DMO UIDs in a List in SavepointManager$Block.invalidateFastFind, we potentially are attempting to remove the same UID many times. A Set would be better, I think.

That is correct.

> An instance of SavepointManager will never manage a mixture of temp-table records and persistent table records in Block.changed. An instance only works with a single database. So, there is no reason to have conditional logic to determine which cache instance to use in invalidateFastFind. It would be better to pass the knowledge of temp-table vs. persistent to the SavepointManager c'tor from BufferManager.TxWrapper (it knows which database it's working with), and get the right cache instance once. This will save all the calls to get the FFC instance during invalidation, some of which require an expensive context-local lookup.

That is correct. However, the temp-table vs persistent DMO problem can be decided on-the-fly, analysing the metadata of the first record :).

> Sorry if I was not clear earlier about RAQ.initialize. I meant that the first if (dirtyContext != null) conditional needs to go away completely. The index lookup always needs to occur now, because the cache needs it, too. So, we always need the index and indexName. The second if (dirtyContext != null) should be put back, because the dmoSorter is only needed for dirty share processing.

That was stupid for me. I did not invest much time here so I did the changes blindly. Thanks for you keen eye.

> I didn't ask this before, and I'm not asking you to change it, but why did you relocate the registry of DMO classes from DmoClass to DmoMetadataManager?

I did not relocate. There were two set of maps. I eliminated the one locally used only. The synchronized register method in DmoMetadataManager will make sure no duplicate DmoClass-es are created. At this moment, from the DmoMeta we can directly obtain the Impl class and the RecordMeta, without lookups.

I committed the changes and rebased 4011c to latest 4011b so it can be easily merged. Current revision is 11568.

#55 - 07/15/2020 07:43 PM - Eric Faulhaber

Looks good. Please merge if you're still there :-)

#56 - 07/15/2020 07:49 PM - Ovidiu Maxiniuc

Done. Committed as revision 4011b / 11561.

#57 - 07/21/2020 03:14 PM - Ovidiu Maxiniuc

I found a new issue with FFC.

When we add a new result for the cache, use use the substitution array as part of the key. It seems fine, but this is not always correct.
Consider the inner query component following code:

```
   FOR EACH mtt1 NO-LOCK
        WHERE mtt1.company = company
          AND mtt1.relation = relation,
        FIRST mtt2 NO-LOCK
        WHERE mtt2.company = mtt1.company
          AND mtt2.person = mtt1.person:
     ...
```

It will be converted to:

```
   query3.addComponent(new RandomAccessQuery().initialize(mtt2, "
upper(mtt2.company) = ? and upper(mtt2.person) = ?", null, "mtt2.company asc, mtt2.person asc", new Object[]
   {
      new FieldReference(mtt1, "company", true),
      new FieldReference(mtt1, "person", true)
   }, LockType.NONE), QueryConstants.FIRST);
```

When the outer query component iterates, the inner one will try to access the fast find cache using effectively the same FieldReference (because of the way equals is written), while we are interested in the value of the FieldReference not the object per-se.

Apparently the solution would be to "resolve" the substitution parameters first, but this might trigger other issues because some of these might not like being evaluated multiple times.

#58 - 07/21/2020 05:55 PM - Eric Faulhaber

My first instinct was to pull the argument resolution forward from executeImpl to execute, but that is a lot of refactoring, due to the fact that the argument preparation is shared code with the NEXT/PREVIOUS cases, and is entwined with the use of the FQL bundles.

I think the most expedient solution and the one we should do now is to scan the substitution parameters and disallow the use of the FFC if any are an instance of Resolvable. Fortunately, FOR EACH, FIRST/LAST ... is a less common case compared to FIND FIRST/LAST/unique, so this should not be a big hit, statistically. In addition, some forms of FOR EACH, FIRST/LAST ... constructs will be optimized by the compound query optimizer.

Ultimately, we will want to fix this more thoroughly, which would involve refactoring execute and executeImpl to move the argument preparation for the FIRST/LAST/unique cases from the latter to the former, so that the FFC is working with the resolved parameters.

#59 - 07/22/2020 02:03 PM - Constantin Asofiei

Ovidiu, the fact that you use a dmoUid is not OK for temp-tables. You need to include the multiplex, too.

A define temp-table tt1 field f1 as int may be used from different programs, and in FWD will have the same dmoUid - but is not correct, as each instance is a different 'temp-table' via the _multiplex.

#60 - 07/22/2020 02:14 PM - Ovidiu Maxiniuc

Actually no.
The cache objects for tables-tables are context local. See FastFindCache.getInstance(boolean). Only the permanent tables are shared by different user contexts.

#61 - 07/22/2020 02:16 PM - Constantin Asofiei

Ovidiu Maxiniuc wrote:

> Actually no.
> The temp cache for tables-tables are context local. See FastFindCache.getInstance(boolean). Only the permanent tables are shared by different user contexts.

This is not what I mean. temp-table tt1 in prog1.p is **not** the same as temp-table tt1 in prog2.p.

You are caching via the dmoUid, and a record in tt1 from prog1.p will be 'seen' as a record from prog2.p's tt1. Again, these are **not** the same temp-table in 4GL terms, even if they share the DMO in FWD.

#62 - 07/22/2020 02:20 PM - Ovidiu Maxiniuc

Sorry, you are right. I will prepare a patch for this.

#63 - 07/22/2020 02:47 PM - Eric Faulhaber

- Related to Bug #4796: locking regression in RAQ introduced with the FFC added

#64 - 07/22/2020 02:55 PM - Eric Faulhaber

Code review 4011b/11586:

Please see #4796-8, specifically the points about updateLock and about leaking locks. These were not addressed in 11586.

Is the FFC.INVALID field needed? It does not appear to be used.

The other changes look fine, though Constantin makes a good point on the DMO IDs for temp-tables. If you can address this quickly and you think the lock fixes will take more time, please commit them as separate updates.

#65 - 07/22/2020 03:09 PM - Eric Faulhaber

I think it makes sense to encapsulate the several behaviors of the FFC that are required to be different between temp-tables and persistent tables within the FastFindCache (see RecordLockContext for a similar model). There are enough differences now that I think this is justified; namely:

- no synchronization is needed for temp-tables;
- no record locking is needed for temp-tables;
- the level 1 key implementation needs to be different (i.e., include _multiplex for temp-tables).

Not only will this keep the code that uses the cache simpler (especially if we expand the use cases over time), we will see some performance benefits in the temp-table use case.

#66 - 07/22/2020 04:22 PM - Ovidiu Maxiniuc

The new L1 key implemented in r11588. Please update.

#67 - 07/22/2020 04:35 PM - Eric Faulhaber

Code review 4011b/11588:

Looks good. The buffer.isTemporary() check can be eliminated in the FFC.Key c'tor, since RB.getMultiplexID is polymorphic and will always return null for persistent tables and a non-null value for temp-tables.

#68 - 07/22/2020 05:19 PM - Eric Faulhaber

As I think about the addition of the multiplex ID to the L1 key, it occurs to me that we have a potential memory leak in the FFC now.

The L1 cache is finite for persistent tables, but with the addition of the multiplex ID to the L1 key, it is no longer finite for temp-tables. The multiplex ID is ever increasing, and as old virtual temp-tables go out of use, their associated entries in the L1 cache will remain, unless the L1 cache entry for a UID/multiplex combination is invalidated through normal use. This could be a bulk delete or rollback, or even the delete of a single record. So long as no FIND caches a new result after such an invalidation, we should be ok.

We already are removing all records from a temp-table before it goes out of scope in TemporaryBuffer.doCloseMultiplexScope, so we should be naturally protected by the invalidation that removeRecords does. Nevertheless, we should add an explicit invalidation when a multiplex ID is retired (in the finally block beginning at line 6654 of 4011b rev 11588).

#69 - 07/22/2020 05:20 PM - Eric Faulhaber

Eric Faulhaber wrote:

> As I think about the addition of the multiplex ID to the L1 key, it occurs to me that we have a potential memory leak in the FFC now.

To be fair, I suppose we already had this potential before adding the multiplex ID to the key, considering dynamic temp-tables.

Files

RandomAccessQuery.java	186 KB	07/07/2020	Ovidiu Maxiniuc