Base Language - Feature #6821

java collection performance

10/06/2022 03:09 AM - Constantin Asofiei

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

Description

Related issues:

Related to Base Language - Feature #7045: re-implement "normal" (non-abend) s...	**Test**

History

#1 - 10/06/2022 03:14 AM - Constantin Asofiei

For example, using Java for-each loop for an an ArrayList instead of a for and getting the elements directly is more expensive, as there will be more objects allocated and hasNext() be called each time.

Another example is using Map.entrySet().iterator() when only the value or the key will be accessed.

In a large application, a next() method in an Iterator is being called some 60 million times, for various lists/maps/sets. We need to analyze the iterator usage and improve/reduce it.

#2 - 10/06/2022 04:26 AM - Constantin Asofiei

Some changes are in 3821c/14272. Greg, please review.

#3 - 10/06/2022 04:29 AM - Constantin Asofiei

Eric: you may want to take a look at ScopedList, too.

#4 - 10/26/2022 07:53 AM - Constantin Asofiei

Some notes about the profiling of a large customer application; this includes only JRE or FWD runtime calls:

Related to map access (hashCode calls):

- String.hashCode() - 45 million calls (mostly from Map.get/put/containsKey)
  - Utils.invoke 16 million
  - TableMapper$LegacyFieldInfo.getMutableInfo 1.8 million
  - TemporaryBuffer$ReferenceProxy.bind 1.25 million
  - TableMapper$TempTableMapper.mapClass 1 million
  - HandleChain.delete 880k
  - FieldReference.getGetter/getSetter/getExtent 600k
  - DmoMetadataManager.getDmoInfo(String) 537k
  - SQLQuery.hydrateRecord 440k
  - TriggerTracker.isTriggerEnabled 426k
  - TextOps.matchesList 225k
  - TemporaryBuffer.createPropsMap 180k
  - BaseDataType$Type.of(String) 80k
- HashMap.get - 20 million calls
  - String.hashCode 7 million calls
  - Method.hashCode 8 million (will be solved by Utils.invoke in #6819)
  - TableMapper.getMultableInfo 1.8 million

Related to iterator() calls:

- ArrayList.iterator - 2 million calls - replace with basic 'for' loop
- IdentityHashMap$Values.iterator() - 8 million calls - 7.8 from BufferManager.deregisterBindingBuffer
- ArrayDeque.iterator() - 1.1 million calls
  - DatabaseTriggerManager.peekTrigger 400k
  - DatabaseTriggerManager.getTriggerStatus 160k
  - WidgetPool$WorkArea.scopeStart/scopeFinished 180k
  - ObjectOps.isInitializing 100k
  - ProcedureManager$WorkArea.getStackEntry 60k
  - ProcedureManager$WorkArea.searchInStack 31k

Related to iterator's next() call:

- HashMap$ValueIterator.next() - 20 million calls
  - Session.invalidateRecords - 20 million
- IdentityHashMap$ValueIterator.next() - ~21 million calls
  - BufferManager.deregisterBindingBuffer ~21 million - CA: already fixed in 6129b
- ArrayList$Iter.next() - 7 million calls (~80% is in H2 code)
  - AnnotatedAst.notifyListenerLevelChanged 230k
  - ProcedureManager$WorkArea.scopeFinished 136k
  - BufferImpl.attachDataSource 130k
  - PreselectQuery.getRecordBuffers 113k
  - Collections$UnmodifiableCollection 97k (from RandomAccessQuery.executeImpl, FastCopytHelper.processProperties, and more)
  - TransactionManager.processCommit 96k
  - TransactionManager.processValidate 96k
  - DynamicTablesHelper$CacheKey.hashCode 92k
  - DynamicTablesHelper$CacheKey.equals 184k
  - PreselectQuery.verifyJoins 63k
  - AbstractParameter.processAssignments 56k
- IdentityHashMap$EntryIterator.next() - ~5.8 million calls
  - BufferManager.scopeStart ~5.8 million calls
- ArrayDeque$DeqIterator.next() - 2.2 million calls
  - ObjectOps.isInitializing 1.4 million
  - WidgetPool$WorkArea.scopeFinished/scopeStart 180k
  - ObjectOps.delete 171k
  - ProcedureManager.getStackEntry 114k
  - ObjectOps.instanceAssigned 70k
  - ArrayAssigner$WorkArea.isRegistered 52k
- LinkedHashMap$LinkedEntryIterator.next() - 1.8 million calls
  - AnnotatedAst.duplicateImpl 980k
  - SQLQuery.hydrateRecord 460k
  - BufferImpl.attachDataSource 127k
  - FqlToSqlConverter.expandAlis 80k
  - BufferImpl.fill 80k
  - Persister.update 54k

#5 - 12/13/2022 09:45 AM - Greg Shah

- Assignee set to Stanislav Lomany

#6 - 12/19/2022 04:45 AM - Stanislav Lomany

- Status changed from New to WIP

Constantin,

> For example, using Java for-each loop for an an ArrayList instead of a for and getting the elements directly is more expensive, as there will be more objects allocated and hasNext() be called each time.

I assume we're taking about replacing

```
for (Buffer buf : buffers)
```

with

```
for (int i = 0; i < buffers.size(); i++)
{
   Buffer buf = buffers.get(i);
```

if buffers is an ArrayList.
I found that there're not so many places where the iterated list is explicity an ArrayList, however there're many cases when it is effectively it - simple analysis shows that an ArrayList is always supplied to it, usually as a function input parameter or return value of other function. Should we replace those cases as well? Performance wise it may be a good idea, but an input parameter / return value has generic List type, and what if another List implementation is passed, like LinkedList which has poor get() perfomance?

> Another example is using Map.entrySet().iterator() when only the value or the key will be accessed.

I don't think there's much difference. HashMap.EntrySet, HashMap.KeySet and HashMap.Values are similar wrappers for the map nodes with no iternal state. And performance-wise they're similair too:

```
final class KeyIterator extends HashIterator implements Iterator<K> {
   public final K next() { return nextNode().key; }
}

final class ValueIterator extends HashIterator implements Iterator<V> {
   public final V next() { return nextNode().value; }
}

final class EntryIterator extends HashIterator implements Iterator<Map.Entry<K,V>> {
   public final Map.Entry<K,V> next() { return nextNode(); }
}
```

#7 - 12/19/2022 04:54 AM - Constantin Asofiei

Stanislav Lomany wrote:

> I assume we're taking about replacing

Correct.

> I found that there're not so many places where the iterated list is explicity an ArrayList, however there're many cases when it is effectively it - simple analysis shows that an ArrayList is always supplied to it, usually as a function input parameter or return value of other function. Should we replace those cases as well? Performance wise it may be a good idea, but an input parameter / return value has generic List type, and what if another List implementation is passed, like LinkedList which has poor get() perfomance?

If there is a Java method with a List argument, iterates that list, but all calls receive a ArrayList, then just replace the parameter type to ArrayList and use a for counter.

> Another example is using Map.entrySet().iterator() when only the value or the key will be accessed.

The point with all the Map and Set iterators is that they allocate lots of short-lived objects.

Something else to consider; please check this test in a profiler:

```
procedure proc0.
   def input param i as int.
end.

function func0 returns int (input i as int).
   return i.
end.

def var i as int.
def var l1 as int64.
def var l2 as int64.

l1 = etime.
do i = 1 to 1000000:
   run proc0(i).
end.
l2 = etime.

message (l2 - l1) "ms".

l1 = etime.
do i = 1 to 1000000:
   dynamic-function("func0", i).
end.
l2 = etime.

message (l2 - l1) "ms".
```

You will see that the Scopeable scopestart/finished do lots of work - please check if it may be possible to condense two or more dictionaries/stacks into a single one, where the element encapsulates what we currently have in separate dictionary/stack.

#8 - 12/19/2022 05:12 AM - Igor Skornyakov

Actually, I doubt that such optimization will result in any significant performance in all situations. See e.g. a discussion here - https://stackoverflow.com/questions/256859/is-there-a-performance-difference-between-a-for-loop-and-a-for-each-loop.

In any case, I believe that generally, it is not a good idea to perform optimization relying just on common sense. It is very easy to overlook some nuances such as an impact of a JIT compiler or the JRE version. I would perform benchmarking/profiling first.

#9 - 12/19/2022 09:18 AM - Greg Shah

> I would perform benchmarking/profiling first.

This is what was done. See #6821-4.

#10 - 12/19/2022 09:25 AM - Stanislav Lomany

> Actually, I doubt that such optimization will result in any significant performance in all situations.

I did some testing, and in this particular ArrayList case the suggested approach is 50% faster and has no memory implications compared to some GC work to garbage collect iterators.

#11 - 12/19/2022 09:31 AM - Stanislav Lomany

> The point with all the Map and Set iterators is that they allocate lots of short-lived objects.

That's true. I mean, are suggesting to replace Map.entrySet().iterator() with Map.keySet()/values().iterator()? If yes, then I'm telling there's no differece between these iterators memory or performance-wise.

#12 - 12/19/2022 09:45 AM - Igor Skornyakov

Greg Shah wrote:

> I would perform benchmarking/profiling first.

This is what was done. See #6821-4.

The number of calls says nothing about potential performance improvement after suggested changes. I've performed multiple profiling sessions and cannot remember that I've ever seen any of the iterator() or next() calls mentioned #6821-4 as CPU hotspots despite a huge number of such calls.

In addition. As far as I understand String.hashCode() can hardly cause any performance issues which can be fixed since the corresponding value is cached. Well, in some situations HashMap can be less efficient than e.g. TreeMap because of e.g. a big number of collisions, but it is a completely different story.

I believe that a big number of the old-style synchronization primitives from the pre-java.util.concurrent era can be a more promising target for optimization.

#13 - 12/19/2022 09:56 AM - Greg Shah

Small improvements over millions of calls add up. The fact that the profiling tools don't show obvious hotspots means that there is not a single cause or a small list of causes, but instead we must implement efficiencies in a wider range of locations. We have already made measurable improvements with this specific technique and as Stanislav notes he sees additional improvements are possible.

If we need to switch data structures to gain efficiencies, we will do so.

We are pursuing many different approaches in parallel (see #4061 for base language and #4011 for persistence). No one single task will be the full answer. I expect this phase of work to require improvements from aproximately 15-20 different tasks.

> I believe that a big number of the old-style synchronization primitives from the pre-java.util.concurrent era can be a more promising target for optimization.

I am open to specific recommendations here. Please create a new task relatred to the old-style synch primitives and document your proposal. We will discuss it further there.

#14 - 12/19/2022 09:57 AM - Igor Skornyakov

Stanislav Lomany wrote:

> Actually, I doubt that such optimization will result in any significant performance in all situations.

> I did some testing, and in this particular ArrayList case the suggested approach is 50% faster and has no memory implications compared to some GC work to garbage collect iterators.

I do believe that one can see some performance boost with the suggested optimization, especially in some standalone artificial tests. What I doubt is that replacing **all** for each loops with for loops can result in an overall performance improvement that will be significant enough to justify the efforts and less code clarity.

#15 - 12/19/2022 10:22 AM - Stanislav Lomany

That may be true. There're about 2k foreach loops, and about 10% of them can be updated. If we want, we can can limit optimization to specific packages, like p2j/persist, p2j/ui, p2j/util.
Greg?

#16 - 12/19/2022 10:26 AM - Greg Shah

The ones that are most important are listed in #6821-4. We aren't suggesting changing things everywhere.

#17 - 12/19/2022 11:28 AM - Constantin Asofiei

- File Back-traces.csv added

For ArrayList.iterator(), the top callers are:

- PreselectQuery.components() - 430k
- AnnotatedAst.notifyListenerLevelChanged() - 170k
- HQLBundle.statements - via Collections's unmodifiable collection - 100k
- ErrorManager.nestedSilent - 120k
- TransactionManager.stopOffEndRegistration - via Collection.addAll - 40k
- ProcedureManager$WorkArea.scopeFinished - 36k
- ScrollableResults.get() - 30k
- DAtaSet.clearImpl - 40k
- AstractParameter$SCope.processAssignments - 26k

See attached for a more comprehensive report after a single run of some tests, in a large customer app. A significant part is in H2, too.

A part of the goal here is to reduce the memory footprint of the iterator usage - for ArrayList, this can be solved by using a for counter instead of a for-each.

For the other cases - some analysis is needed to check if the reason for the large number of iterations is in some other place (or if it can be solved in some other ways).

#18 - 01/02/2023 05:21 PM - Stanislav Lomany

- File 6821.diff added

Please review the diff which replaces "for-each" loops with indexed "for" loops (I'll add headers later), specifically:

1. you may consider changes in some files unnecessary;
2. In EventList, EventList.events is normally an ArrayList but could be a LinkedList in a single case. I replaced LinkedList with ArrayList as a faster implementation for EventList needs.

#19 - 01/03/2023 02:54 PM - Stanislav Lomany

> Related to map access (hashCode calls):
>
> - HandleChain.delete 880k

Maybe we don't need to perform first/lastResource.get(type) because we can determine if the resource is first/last in the handle chain by checking its HandleChain.prev/nextSibling?

```
HandleChain first = wa.firstResource.get(type);
HandleChain last = wa.lastResource.get(type);

if (this == first)
{
   wa.firstResource.put(type, this.nextSibling);
}
if (this == last)
{
   wa.lastResource.put(type, this.prevSibling);
}
```

#20 - 01/04/2023 05:13 AM - Constantin Asofiei

Stanislav Lomany wrote:

> Please review the diff which replaces "for-each" loops with indexed "for" loops (I'll add headers later), specifically:
>
> 1. you may consider changes in some files unnecessary;
> 2. In EventList, EventList.events is normally an ArrayList but could be a LinkedList in a single case. I replaced LinkedList with ArrayList as a faster implementation for EventList needs.

what branch and revision did you use for this patch?

#21 - 01/04/2023 05:17 AM - Constantin Asofiei

Stanislav Lomany wrote:

> Related to map access (hashCode calls):
>
> - HandleChain.delete 880k

> Maybe we don't need to perform first/lastResource.get(type) because we can determine if the resource is first/last in the handle chain by checking its HandleChain.prev/nextSibling?
> [...]

I think you are right:

- if prevSibling is null, this must be the first resource being deleted, so nextSibling becomes the first resource
- if nextSibling is null, this must be the last resource being deleted, so prevSibling becomes the last resource

You also need checks if this is the single resource in the chain being deleted, in this firstResource/lastResource must remove it completely.

Please check some standalone tests.

#22 - 01/04/2023 08:54 AM - Stanislav Lomany

> what branch and revision did you use for this patch?

3821c rev 14467.

#23 - 01/05/2023 04:25 AM - Constantin Asofiei

- File CheckForEach.java added

Stanislav Lomany wrote:

> what branch and revision did you use for this patch?

3821c rev 14467.

The patch is OK, just a minor formatting issue in EventList.intersects, with the closing braces on lines 2012/2013.

Attached is a space-delimited file with all FOR EACH loops which have a List, ArrayList or Java array as the expression. The format is:

```
classqname:methodname expressionType line
```

Please look through this and fix cases in p2j.persist, goldencode.ast, and p2j.util. Use common sense to prioritize these, you don't have to fix them all (for example, I think PayloadSerializer Java array usage is a good place to fix, and also goldencode.ast cases). I would start with Java arrays first (look for [] in the text). This was ran on 3821c/14468

This file was generated with https://spoon.gforge.inria.fr/ - the file is attached.

#24 - 01/05/2023 04:25 AM - Constantin Asofiei

Keep in mind that this:

```
IdentityHashMap$ValueIterator.next() - ~21 million calls
    BufferManager.deregisterBindingBuffer ~21 million
```

is no longer an issue in 6129b.

#25 - 01/05/2023 04:26 AM - Constantin Asofiei

- File foreach.csv added

Forgot to add the csv file.

#26 - 01/05/2023 06:20 AM - Constantin Asofiei

Constantin Asofiei wrote:

> Attached is a space-delimited file with all FOR EACH loops which have a List, ArrayList or Java array as the expression. The format is:
> [...]
>
> ... This was ran on 3821c/14468

The report was generated with the patch in #6821-18 applied, so these are excluded.

#27 - 01/05/2023 02:37 PM - Stanislav Lomany

> You also need checks if this is the single resource in the chain being deleted, in this firstResource/lastResource must remove it completely. Please check some standalone tests.

Committed as 3821c rev 14472. Please review. I did some standalone testing.

#28 - 01/06/2023 02:14 AM - Constantin Asofiei

Stanislav Lomany wrote:

> > You also need checks if this is the single resource in the chain being deleted, in this firstResource/lastResource must remove it completely. Please check some standalone tests.
>
> Committed as 3821c rev 14472. Please review. I did some standalone testing.

Changes are OK.

Please backport this 3821c/14472 to 6129b and commit it there.

The same for the changes you make in this task, they need to be in 6129b, so please port them to this branch.

#29 - 01/06/2023 11:39 AM - Stanislav Lomany

> TableMapper$LegacyFieldInfo.getMutableInfo 1.8 million

This one looks like caching may be slightly improved, but I don't know how.

> Session.invalidateRecords - 20 million

invalidateRecords has TODO left:

```
* TODO: this is a terribly inefficient algorithm. We scan the entire contents of the cache and compare
* table name strings of each entry to find a subset of records (possibly none). If it is used often, we
* need a better caching strategy to support this use case.
```

What do you think about maintaing a Map of table name -> set of records for this table. Entries are removed on record expiration event.

#30 - 01/06/2023 12:05 PM - Constantin Asofiei

Stanislav Lomany wrote:

> > TableMapper$LegacyFieldInfo.getMutableInfo 1.8 million
>
> This one looks like caching may be slightly improved, but I don't know how.

TableMapper is improved via #6825, don't touch that. I'll have a batch of changes today or tomorrow.

> > Session.invalidateRecords - 20 million
>
> invalidateRecords has TODO left:
> [...]

> What do you think about maintaing a Map of table name -> set of records for this table. Entries are removed on record expiration event.

The downside will be maintaining these separate set of records. Which will require map lookup and set management. But look at the invalidateRecords callers - maybe we can keep just a counter of records per each table, and if there are no cached records for that table, just do nothing?

#31 - 01/06/2023 12:25 PM - Greg Shah

All usage of Session.invalidateRecords() comes from RecordBuffer.pruneSessionCache() which is only called in 2 places, both of this are in BufferImpl.acceptChanges().

Is a given DMO instance always specific to a Session? That DMO is specific to the table already and could be used to store a direct link to the Set of records in the cache. Something would still need to be done to handle the multiplex issue (an iteration of the set and removal of only those matches). Even that would be way faster than the current approach. The invalidation operation where the set could simply be cleared would be very fast for permanent tables.

#32 - 01/06/2023 12:52 PM - Stanislav Lomany

invalidateRecords method is called only for temp-tables. And called only by ACCEPT-CHANGES method (I'm interested what app were you profiling because I couldn't find its usages in what I have). So maintaining a map is an overhead for the most applications. Now I understand why it was left as TODO. I don't think a counter will help because most likely we do have changes to commit for a ACCEPT-CHANGES call.
Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records? Make a directory option to enable this extra mapping for a customer's app?

#33 - 01/06/2023 01:25 PM - Greg Shah

> Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records?

Not exactly. I was suggesting that your idea of the mapping between table name and a Set of the records, could be taken a little further and the reference to the Set could be stored in the DMO such that it would be very fast to access/clear.

My real point here is that it is OK to do something more aggressive/complex if there is a big performance win. More sophisticated caching is certainly worth the effort. I did not analyze all the map maintenance, nor did I look at the various use patterns.

> Make a directory option to enable this extra mapping for a customer's app?

Whatever we do it would be preferred to have the code be standard for all customers.

#34 - 01/06/2023 01:29 PM - Greg Shah

One thing that is not obvious to me is how the cache is used. Since we are using temp-tables with this mechanism, it seems to me that the same RecordIdentifier can be present under different multiplex ids. Do I understand correctly that only the most recently added entry will be kept and any older entry is overwritten? Is that intentional?

#35 - 01/06/2023 01:56 PM - Eric Faulhaber

Greg Shah wrote:

> Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records?

Not exactly. I was suggesting that your idea of the mapping between table name and a Set of the records, could be taken a little further and the reference to the Set could be stored in the DMO such that it would be very fast to access/clear.

What do you mean specifically by "the DMO" in this context? An individual instance? The implementation class? Something else?

#36 - 01/06/2023 02:01 PM - Stanislav Lomany

> Do I understand correctly that only the most recently added entry will be kept and any older entry is overwritten? Is that intentional?

Looks like a good catch: it seems that a record with the same record ID but different multiplex ID it can be returned by the cache.

#37 - 01/06/2023 02:03 PM - Eric Faulhaber

Greg Shah wrote:

> One thing that is not obvious to me is how the cache is used. Since we are using temp-tables with this mechanism, it seems to me that the same RecordIdentifier can be present under different multiplex ids. Do I understand correctly that only the most recently added entry will be kept and any older entry is overwritten? Is that intentional?

You shouldn't have multiple DMO instances with different multiplex IDs but the same RecordIdentifier. Every RecordIdentifier should identify one and only one DMO instance in a user context (for temp-tables) or across the server (for persistent tables). If the multiplex ID is different between two temp-table DMO instances of the same type, the RecordIdentifier must be different as well; there can be only one. If we have drifted from this, it is a bug.

There are special exceptions to this rule for housekeeping DMO instances (like a snapshot copy taken as a reference for query navigation). But these special DMOs must never be stored in the session cache.

#38 - 01/06/2023 02:05 PM - Eric Faulhaber

Stanislav Lomany wrote:

> Do I understand correctly that only the most recently added entry will be kept and any older entry is overwritten? Is that intentional?

Looks like a good catch: it seems that a record with the same record ID but different multiplex ID it can be returned by the cache.

Are you making a theoretical statement, or are you actually seeing this in the wild?

#39 - 01/06/2023 02:07 PM - Stanislav Lomany

I made an assumption that temp-table record IDs are not unique across multiplex IDs, which is wrong, as you're saying. No, I'm not seeing it.

#40 - 01/06/2023 02:18 PM - Greg Shah

Do we assign a new rowid when we copy temp-table records from one temp-table to another, like in table parameter processing?

#41 - 01/06/2023 02:19 PM - Greg Shah

Eric Faulhaber wrote:

> Greg Shah wrote:
>
> > Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records?
>
> Not exactly. I was suggesting that your idea of the mapping between table name and a Set of the records, could be taken a little further and the reference to the Set could be stored in the DMO such that it would be very fast to access/clear.

What do you mean specifically by "the DMO" in this context? An individual instance? The implementation class? Something else?

An individual instance.

#42 - 01/06/2023 03:41 PM - Stanislav Lomany

> Please backport this 3821c/14472 to 6129b and commit it there.

Committed as 6129b rev 14343.

#43 - 01/06/2023 05:53 PM - Eric Faulhaber

Greg Shah wrote:

> Eric Faulhaber wrote:
>
> > Greg Shah wrote:
> >
> > > Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records?
> >
> > Not exactly. I was suggesting that your idea of the mapping between table name and a Set of the records, could be taken a little further and the reference to the Set could be stored in the DMO such that it would be very fast to access/clear.
> >
> > What do you mean specifically by "the DMO" in this context? An individual instance? The implementation class? Something else?
>
> An individual instance.

I REALLY dislike the idea of DMO instances having references to sets of other DMO instances. This sounds very messy. DMO instances are meant to be the endpoints of an object graph, not a set of branches with graphs of references to each other.

#44 - 01/07/2023 05:51 PM - Stanislav Lomany

The "for" cycle changes reviewed by Constantin were merged and committed as 6129b rev 14344.

#45 - 01/08/2023 01:36 PM - Stanislav Lomany

> I would start with Java arrays first (look for [] in the text).

Constantin, are you suggesting to replace enhanced "for" loops with indexed loops for *arrays*?

```
int[] arr = new int[10];
for (int i : arr)
```

I'm afraid this can be even counterproductive:
https://stackoverflow.com/questions/70583053/looping-over-array-performance-difference-between-indexed-and-enhanced-for-loop

#46 - 01/09/2023 06:33 AM - Constantin Asofiei

Stanislav Lomany wrote:

> I would start with Java arrays first (look for [] in the text).
>
> Constantin, are you suggesting to replace enhanced "for" loops with indexed loops for *arrays*?
> [...]
> I'm afraid this can be even counterproductive:
> https://stackoverflow.com/questions/70583053/looping-over-array-performance-difference-between-indexed-and-enhanced-for-loop

Stanislav, you are correct, I got carried away with that.

#47 - 01/09/2023 06:58 AM - Greg Shah

Eric Faulhaber wrote:

> Greg Shah wrote:
>
> > Eric Faulhaber wrote:
> >
> > > Greg Shah wrote:
> > >
> > > > Greg, are you suggesting to maintain for temp-tables a map of RecordBuffer.dmoClass -> Set of records?
> > >
> > > Not exactly. I was suggesting that your idea of the mapping between table name and a Set of the records, could be taken a little further and the reference to the Set could be stored in the DMO such that it would be very fast to access/clear.
> > >
> > > What do you mean specifically by "the DMO" in this context? An individual instance? The implementation class? Something else?
> >
> > An individual instance.
>
> I REALLY dislike the idea of DMO instances having references to sets of other DMO instances. This sounds very messy. DMO instances are meant to be the endpoints of an object graph, not a set of branches with graphs of references to each other.

I was thinking of the DMO in its buffer form, not the concept of the specific record that is contained.

#48 - 01/09/2023 07:33 AM - Stanislav Lomany

> PreselectQuery.components() - 430k

Committed 6129b rev 14345: replaced components iterator with ArrayList.get().
Please review.

#49 - 01/09/2023 12:58 PM - Eric Faulhaber

Greg Shah wrote:

> I was thinking of the DMO in its buffer form, not the concept of the specific record that is contained.

It would be fairly straightforward to track all records which are or were at one point held by each buffer, but this wouldn't necessarily be all the records for that buffer's backing table. Other buffers could be defined for that same backing table, which could hold a different or overlapping set of records. If I understand the pruning case correctly, we need to invalidate the union of all these.

For this, we would need some more complex map/set management which would track the superset of all buffers for that table in the local context. It certainly could be done, and the prune operation itself most likely would be more efficient than the current mechanism. What I am concerned with is that we would be adding the overhead of constantly managing these sets of records for what seems like a very specific case of needing to prune them only for ACCEPT-CHANGES calls.

If this is only needed for temp-tables involved in a data set (Ovidiu, please verify this), we could reduce the cases where we are actively tracking these "prunable" records. If this is only needed for data sets, it seems we should manage this information at the data set level. As temp-tables are associated with a data set, they would be notified that they need to update this information when records flow through them.

Ovidiu, can you help me understand the purpose of the pruning? Is it functional, or is it about housekeeping to save memory or to manage the cache? Perhaps we can scope this overhead down even further, for instance, if it can be limited to a certain operation. Should we be temporarily disabling caching while some operation occurs, rather than trying to clean up the cache after that operation? Now that we know there is a bottleneck in the pruning, let's consider whether we can avoid it altogether, while still performing the functionality we need.

#50 - 01/09/2023 08:40 PM - Ovidiu Maxiniuc

The reason pruneSessionCache() is called on afterBuf and beforeBuf after a batch operation executed using SQL directly on database. Since this bypassed the FWD caching mechanism, the latter must be invalidated for the table affected, or else the information in cache(s) might not be correct. The afterBuf data is altered by an UPDATE statement and the beforeBuf by a delete because the ACCEPT-CHANGES basically makes the after table "definitive" by simply deleting the "history" (drops all before images and the now invalid links to them). Since we do not have direct fine control on which records were altered, the whole tables are invalidated. Ultimately, we could iterate the tables individually and process (delete/update then invalidate) the records individually, but this would affect performance.

Note that this call is only valid for BEFORE-TABLEs and only temp-tables are affected. If the method is called on a permanent table or even on a

simple or after-table, it will quickly exit with an error (11887).

This invalidations were added added in r12998. Commit details can be found in #5671-8.

#51 - 01/10/2023 10:27 AM - Greg Shah

> Ultimately, we could iterate the tables individually and process (delete/update then invalidate) the records individually, but this would affect performance.

Don't we already know the exact list of rowids used for all UPDATE statements? We also know the table and multipex ID so we could track these as a list and then use that instead of searching through.

I'm not sure if something similar can be calculated or tracked for deletes but if we can just track the exact list, then a full invalidation would not be needed. Perhaps this could optimize this ACCEPT-CHANGES and also avoid over-aggressive invalidation which would have downstream performance benefits.

#52 - 01/10/2023 11:13 AM - Ovidiu Maxiniuc

Greg Shah wrote:

> Don't we already know the exact list of rowids used for all UPDATE statements? We also know the table and multipex ID so we could track these as a list and then use that instead of searching through.

There is no list of these rowids, but it can be extracted from SQL using a query like:

```
select <PK> from <after-table> where _rowState is null or _rowState = 0 /*ROW_UNMODIFIED*/;
```

before the update.

> I'm not sure if something similar can be calculated or tracked for deletes but if we can just track the exact list, then a full invalidation would not be needed. Perhaps this could optimize this ACCEPT-CHANGES and also avoid over-aggressive invalidation which would have downstream performance benefits.

We are completely dropping the content of this temp table (if the changes were accepted, the before-table is reset) so the invalidation of all records sharing the specific _multiplex is the right solution here.

#53 - 01/10/2023 12:14 PM - Stanislav Lomany

Committed 6129b rev 14351: P2JIndex.components() provides direct access to the components array in order to improve performance.
Please review.

#54 - 01/13/2023 02:21 PM - Stanislav Lomany

Committed 6129b rev 14358: Final round of performance improvements which are replacements of some 'for-each' loops with indexed 'for' loops.
Please review.

#55 - 01/14/2023 04:28 PM - Stanislav Lomany

```
l1 = etime.
do i = 1 to 1000000:
   run proc0(i).
end.
l2 = etime.
message (l2 - l1) "ms".

l1 = etime.
do i = 1 to 1000000:
   func0(i).
end.
l2 = etime.
message (l2 - l1) "ms".
```

It looks interesting to me that functions are executed 10 times slower than procedures due to presence of returnNormal in functions (I didn't look deeper yet).
In 4GL execution of both segments takes ~4s for each. In FWD the first segment takes ~2.5s and the second ~25s.

#56 - 01/14/2023 04:57 PM - Greg Shah

Stanislav Lomany wrote:

> [...]
>
> It looks interesting to me that functions are executed 10 times slower than procedures due to presence of returnNormal in functions (I didn't look deeper yet).
> In 4GL execution of both segments takes ~4s for each. In FWD the first segment takes ~2.5s and the second ~25s.

Hmm, I think it is worth looking deeper. Please go ahead with that.

#57 - 01/16/2023 08:24 AM - Stanislav Lomany

Performance overhead in function vs procedure is cause by throwing ReturnUnwindException. I think that in vast majority of cases RETURN is the last statement in a function, and in this case I suppose that returnNormal can be replaced with setting of return value and letting function to end naturally.

#58 - 01/16/2023 10:40 AM - Greg Shah

- Related to Feature #7045: re-implement "normal" (non-abend) stack unwinding to avoid throwing an exception added

#59 - 01/16/2023 10:41 AM - Greg Shah

Stanislav Lomany wrote:

> Performance overhead in function vs procedure is cause by throwing ReturnUnwindException. I think that in vast majority of cases RETURN is the last statement in a function, and in this case I suppose that returnNormal can be replaced with setting of return value and letting function to end naturally.

Nice find. We always knew it was expensive but now you've demonstrated it is needing serious attention. I'm going to work this in #7045.

#60 - 01/17/2023 06:35 AM - Stanislav Lomany

> You will see that the Scopeable scopestart/finished do lots of work - please check if it may be possible to condense two or more dictionaries/stacks into a single one, where the element encapsulates what we currently have in separate dictionary/stack.

I'm a bit skeptical about this: Scopables doesn't have much uniformity in logic and used data types: they use ScopedDictionaries, PartitionedArrays, Deques.

#61 - 01/17/2023 06:36 AM - Stanislav Lomany

As far as I can see, only pruneSessionCache() issue is left here.

#62 - 02/23/2023 12:28 PM - Constantin Asofiei

7026a/14499 contains some other improvements:

- Avoid the LinkedList overhead, use ArrayDeque instead, for NamedFunction.enclosingScopes.
- In ScopedDictionary and TailedScopedDictionary, avoid the overhead of calculating an empty map when the scope has no dictionary - internally, Node.getDictionary(false) will return null if there is no dictionary; but, getDictionaryAtScope() will still return an empty map, instead of null.
- Avoid listIterator usage, as a plain for loop can be used, in TransactionManager.nearestTopLevel.

Tijs: you've last touched ScopedDictionary and TailedScopedDictionary, please review these changes.

#63 - 02/24/2023 05:03 AM - Tijs Wickardt

- Status changed from WIP to Review

Reviewing.

#64 - 02/24/2023 07:00 AM - Tijs Wickardt

Reviewed p2j 7026a -r14497..14499 .
The changes are good.
Remark on Avoid the overhead of calculating an empty map :
Previously the empty map originated from Collections.emptyMap().
This returns a singleton, so we are only removing the overhead of assigning that singleton to a member.
Am I missing something here (probably)?
The old behavior, using the immutable empty map, guards against NPE's and makes the intention clearer than null .

#65 - 02/24/2023 07:01 AM - Tijs Wickardt

- Status changed from Review to WIP

#66 - 02/24/2023 07:04 AM - Constantin Asofiei

Tijs Wickardt wrote:

> Reviewed p2j 7026a -r14497..14499 .
> The changes are good.
> Remark on Avoid the overhead of calculating an empty map :
> Previously the empty map originated from Collections.emptyMap().
> This returns a singleton, so we are only removing the overhead of assigning that singleton to a member.
> Am I missing something here (probably)?
> The old behavior, using the immutable empty map, guards against NPE's and makes the intention clearer than null .

The overhead is seen when this code is executed millions of times - I agree that using an immutable empty map the code is cleaner, but every member access of the map (clean, contains, get, etc) is expensive when executing lots of times. Plus the actual invocation to get the empty map via the supplyImmutableEmptyMap is costly.

#67 - 02/24/2023 07:09 AM - Tijs Wickardt

Constantin Asofiei wrote:

> The overhead is seen when this code is executed millions of times - I agree that using an immutable empty map the code is cleaner, but every member access of the map (clean, contains, get, etc) is expensive when executing lots of times. Plus the actual invocation to get the empty map via the supplyImmutableEmptyMap is costly.

Agreed, all small things count on a hot path. If the impact is noticeable, I fully agree.

Did you profile this? (I see #6821-55 but it seems high level).

#68 - 02/24/2023 07:10 AM - Constantin Asofiei

Tijs Wickardt wrote:

> Did you profile this? (I see #6821-55 but it seems high level).

Yes, this was from profiling a customer's application.

#69 - 02/24/2023 07:11 AM - Tijs Wickardt

Constantin Asofiei wrote:

> Yes, this was from profiling a customer's application.

Great. Review complete, approved.

#70 - 03/03/2023 07:00 AM - Alexandru Lungu

Greg Shah wrote:

> > Ultimately, we could iterate the tables individually and process (delete/update then invalidate) the records individually, but this would affect performance.
>
> Don't we already know the exact list of rowids used for all UPDATE statements? We also know the table and multipex ID so we could track these as a list and then use that instead of searching through.
>
> I'm not sure if something similar can be calculated or tracked for deletes but if we can just track the exact list, then a full invalidation would not be needed. Perhaps this could optimize this ACCEPT-CHANGES and also avoid over-aggressive invalidation which would have downstream performance benefits.

I read your discussion regarding Session.invalidateRecords, because I stumbled upon a hot-spot on HashMap.ValueIterator.next() caused by this method. I see that you link this method very closely to ACCEPT-CHANGES, but in my tests, 75% of the back traces lead to TemporaryBuffer.doCloseMultiplexScope (when multiplex is no longer referenced), 22% to TemporaryBuffer.removeRecords (when doing full remove) and 3% to TemporaryBuffer.loopDelete (when doing full remove). All of these 3 are widely used in large customer applications. None of the samples were related to ACCEPT-CHANGES.

In a server start-up, I've got: 2.887 invalidateRecords calls, 1.774.419 table mismatches, 7.374 multiplex mismatches and 117ms wasted in the for loop
In a POC run, I've got: 44.768 invalidateRecords calls, 26.603.934 table mismatches, 75.784 multiplex mismatches and 1.004ms wasted in the for loop

Therefore, should we consider that this method has a wider usage and should be optimized? I don't have an idea for this yet - thinking.

Files

Back-traces.csv	22.3 KB	12/19/2022	Constantin Asofiei
6821.diff	90.8 KB	01/02/2023	Stanislav Lomany
CheckForEach.java	2.92 KB	01/05/2023	Constantin Asofiei
foreach.csv	147 KB	01/05/2023	Constantin Asofiei