Database - Feature #2581

improve performance of non-preselect, multi-table joins

06/05/2015 02:58 PM - Eric Faulhaber

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Eric Faulhaber	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	Cleanup and Stablization for Server Features		
billable:	No	**vendor_id:**	GCD
Description			

History

#1 - 06/05/2015 03:15 PM - Eric Faulhaber

Currently, when we convert (statically or at runtime) constructs like FOR EACH, EACH, ... or FOR EACH, FIRST, ..., etc., we convert to CompoundQuery. Unfortunately, CompoundQuery retrieves all matching records for every table in the join and performs the join at the application server. This implementation can perform very badly in cases where many records are retrieved at any level of the join.

The idea of the performance enhancement is to refactor at runtime the joined components of a CompoundQuery -- to the extent possible -- into server-side joins, to leverage the much more sophisticated query planners of the database server; to minimize the number of individual queries executed; and to minimize the number of records returned to the application server. This will only be possible when components represent tables from the same physical database.

This is not unlike the idea behind AdaptiveQuery/AdaptiveFind, which execute set-oriented queries to replace the record-by-record retrieval idiom used by Progress. Like these implementations, we will need the ability to drop back to the current, "dynamic" mode of CompoundQuery if a record in an index being walked is added, deleted, or changed in such a way as to invalidate the result set being processed, or if a query substitution parameter which needs to be resolved multiple times changes its value after processing of results has begun. This transition will be much more complex to manage with CompoundQuery, since it has to get the multi-table join semantics right, whereas AdaptiveQuery/AdaptiveFind operate on a single table.

As a compromise, we may disallow the transition back to "preselect/server-side join" mode, unlike the single-table implementation, which allows multiple such mode transitions.

#2 - 07/01/2015 11:27 PM - Eric Faulhaber

Ovidiu, can you confirm that something like this SQL works in SQL Server 2012?

```
select *
from person p, pers_addr pa
where p.emp_num = 203
and pa.id = (select id
                 from pers_addr
                 where emp_num = p.emp_num
                 and upper(rtrim(pa.link_type, E' \t\n\r')) = 'FAST'
                 order by link_type asc
                 limit 1);
```

It doesn't have to be this exact syntax (which is supported by PostgreSQL and H2 with some minor changes), but basically I need to know whether the combination of order by and limit (or equivalent syntax) work within a subquery in SQL Server. **The key thing here is that the order by must be applied *before* the limit.**

I understand that in SQL Server 2012 and higher, there is a fetch keyword which acts like limit, something like:

```
fetch next 1 only
```

...but I have no good way to test this at the moment.

I'm trying to optimize P2J runtime support for something like this:

```
for each person no-lock
    where person.emp-num = eno,
    first pers-addr no-lock
    where pers-addr.emp-num = person.emp-num
          and pers-addr.link-type = s
    break by person.emp-num:
   ...
end.
```

This converts to:

```
PresortCompoundQuery query0 = null;

FieldReference byExpr0 = new FieldReference(person, "empNum");

public void init()
{
   RecordBuffer.openScope(persAddr, person);
   query0 = new PresortCompoundQuery();
   query0.enableBreakGroups();
   query0.addComponent(new PreselectQuery(person, "person.empNum = ?", null, "person.empNum asc, person.siteId
 asc", new Object[]
   {
      eno
   }, LockType.NONE));
   RecordBuffer.prepare(persAddr);
   query0.addComponent(new RandomAccessQuery(persAddr, "persAddr.empNum = ? and upper(persAddr.linkType) = ?",
 null, "persAddr.linkType asc", new Object[]
   {
      new FieldReference(person, "empNum"),
      toUpperCase(s)
   }, LockType.NONE), QueryConstants.FIRST);
   query0.addSortCriterion(byExpr0);
}
...
```

If the limit concept is supported by SQL Server, I could manipulate the HQL ASTs at runtime into the HQL equivalent of the SQL query above, which when converted to SQL would perform its join at the database server rather than at the application server. This optimization would be enabled for all 3 of our currently supported databases, and would be disabled for dialects that do not support a subquery result set that is ordered, then limited.

Unfortunately, it would require some non-trivial patches to Hibernate (the HQL parser and HQL->SQL generator at minimum), since HQL doesn't support the LIMIT (nor any equivalent) keyword, presumably because they can't make it work consistently across dialects in subqueries, due to varying support for the feature at the database level. One can use Query.setMaxResults, but that doesn't help for a subquery, which is the only way I can conceive to do this at the database server.

#3 - 07/02/2015 05:27 AM - Ovidiu Maxiniuc

Yes, the SQL SERVER has a similar concept for LIMIT, named TOP. It allows to return only the first N rows from a query, in the order the query requested. So yes, ORDER BY is applied before cutting the returned result. IIRC, the only issue here is that it has only one parameter, so paging the results it's a bit tricky. If you only need the 1st row or first N rows, then it's exactly what you need.

The syntax for your testcase in MS SQL looks like this:

```
select *
from person p, pers_addr pa
where p.emp_num = 203
    and pa.id = (select top 1 id
                    from pers_addr
                    where emp_num = p.emp_num
                        and upper(dbo.trimWS_s(pa.link_type)) = 'FAST'
                    order by link_type asc);
```

#4 - 07/14/2015 11:40 PM - Eric Faulhaber

Rebased branch 2581a to trunk rev. 10895.

#5 - 07/21/2015 10:47 PM - Eric Faulhaber

Rebased task branch 2581a from P2J trunk revision 10898.

#6 - 07/31/2015 04:40 PM - Eric Faulhaber

I've implemented this feature for the most part (a lot of code cleanup and documenting left to do). After considerable testing, I've found I've had to roll back the concept significantly, as refactoring many compound queries on the fly had the reverse impact as I had hoped.

The implementation is split between conversion and runtime changes. After a number of attempts, the one conversion change that has survived (so far) is the conversion of the idiom

```
FOR EACH a, { FIRST | LAST } b where b.foo = a.bar [ [ BREAK ] BY sort-phrase ]
```

to:

- PreselectQuery (for cases that used to convert to PresortCompoundQuery with a simple sort phrase); or
- PresortQuery (for cases that used to convert to PresortCompoundQuery with a complex sort phrase or break groups).

See below for what the HQL looks like for a server-side join of this idiom.

The conversion only deals with the entire nested FOR statement. The runtime, on the other hand, will try to optimize individual groupings within the nested FOR statement, on the fly.

Initially, I tried to join as much as is possible from the information we have available at runtime, including all possible combinations of FOR EACH/FIRST/LAST. This broad approach had disasterous performance consequences, causing many compound queries which previously were relatively fast to now take seconds to minutes, or even tens of minutes. There are a variety of reasons for this, but in general, the join significantly complicates the work of the database's query planner. This is further exacerbated by some of the ways we must augment our where clauses. For example, when neither of the fields involved in a join is MANDATORY, we must augment with some fairly verbose NULL checking semantics to mimic Progress' semantic around unknown value comparisons, which differ from SQL (see #2552). This can make for some terribly slow query plans when joining.

Combinations with a top nesting level of FIRST or LAST were particularly problematic, because a FIRST/LAST "loop" is now converted to a subquery. For example:

```
FOR EACH a, FIRST b where b.foo = a.bar
```

now converts to the HQL:

```
from A a, B b where b.id = (select id from B where foo = a.bar order by sortPhraseB asc single) order by sortP
hraseA
```

(Note that this conversion required changes to Hibernate.)

When the outermost nesting loop is FIRST (or LAST), as in FOR FIRST a, EACH b..., putting a subquery on the left side of the join does not make sense, and crafting the HQL otherwise was problematic, for no real performance benefit in the real-world cases I tried. So, I abandoned that aspect of the planned optimization. Fortunately, this is not a normal use case, though I often encountered this situation in real code in situations where I determined that the outermost nesting level in something like a FOR EACH a, FIRST b, FIRST c was not a good join candidate and attempted to start joining below that level.

Generally, code that converts to CompoundQuery is much more sensitive to this optimization approach than code that converts to PresortCompoundQuery. The reason is that the former requires a progressive results fetching approach (bring back a single row first, then a small set, then a larger set, and so on). Processing begins as soon as the first row returns. The latter requires the full result set to be found before processing can commence. If a join performs poorly in the latter case, the cost can be amortized over the full result set (though this still can be bad if the tables are large and the result set is small). Also, these cases tend to be better optimized in the orignal 4GL code, since the same limitation applies. In the former case, the first result is expected immediately, which CompoundQuery generally is pretty good at providing. If the server-side join is slow in this case, it will take a long time before that first result is returned, then the next batch will be slow as well, and so on. In many cases, this can be much worse than the original approach, especially if the loop is abandoned after only one or a small number of results are processed!

I found that on balance, the following CompoundQuery optimized join approaches caused more trouble than they added benefit:

- joins of more than 2 tables;
- joins using more than one pair of fields;
- mixed iteration type joins (e.g., EACH/FIRST, EACH/LAST);
- joins where the join column is not the leading component in an index of the outer table;
- joins nested below the outermost loop level.

So...what does that leave? Not much, unfortunately: EACH-to-EACH joins no more than 2 tables deep, beginning at the outermost, nested loop, using no more than one pair of fields, which in the outer table should be the leading component of an index. Also, it helps if at least one of the fields on either side of the join is MANDATORY (i.e., NOT NULL), because we can provide a simple restriction clause for the join. Even so, this join does not always make things better, but on balance in my testing, it did more good than harm.

At the moment, the implementation is nearly complete. Things left to do:

- massive code cleanup: I refactored a lot of code (including in Hibernate), added quite a bit to our existing classes, did a lot of tinkering that didn't work out;
- documentation: lots of javadoc and file header updates;
- make Hibernate changes platform-neutral: in testing this approach, I was chiefly concerned with making it functional with PostgreSQL; now that the concept is working, I need to make it portable and more robust;
- put Hibernate under version control on devsrv01: so far, we've been working with patches, but we should have a proper git repo.

#7 - 07/31/2015 06:02 PM - Eric Faulhaber

As part of this work, I needed a way to review query performance in a converted application. I added some statistics collection capabilities, which use a combination of statistics Hibernate reports and statistics we collect with some new instrumentation in the P2J persistence layer.

The statistics collected include an accounting of the second level cache use: per-region hit/put/miss counts, number of elements, memory used. They also include global query stats: per-query count; min/max/avg execution time (ms); query cache hit/put/miss counts; and of course the HQL itself. The query stats are further organized by application-level location data. For each combination of location and query, we collect the name of the application source code file, class, method, and line number at which the corresponding P2JQuery object was constructed; the number of times the query was launched from that location; and of course the HQL. Individual substitution parameters are not collected or reported, since the statistics are aggregates; gathering distinct, per-execution data would be prohibitively time- and memory-consuming.

At server shutdown, all the collected data are stored in an H2 database. The class which implements most of this new functionality is DatabaseStatistics. Collection is enabled from the directory using the following node under server/standard/persistence:

```
<node class="container" name="collect-statistics">
  <node class="boolean" name="enabled">
    <node-attribute name="value" value="TRUE"/>
  </node>
  <node class="string" name="db-user">
    <node-attribute name="value" value="perf"/>
  </node>
  <node class="string" name="db-pass">
    <node-attribute name="value" value="perf"/>
  </node>
</node>
```

Currently, the name of the database is hard-coded to perfdb and it is created in a subdirectory of the same name off the current directory, typically <path-to-application-home>/run/server. I'll probably add a configurable path to the directory node above.

The impact of the instrumentation is minimal when statistics collection is disabled. When it is enabled, however, there appears to be a significant performance penalty, due primarly to the use of new Throwable() to collect location statistics for the queries. I've seen it add ~10% overhead in my testing, but it will vary by the density of query instantiation in an application. Also, note that since all the collation of the statistics and their storage into the database happens at server shutdown, this may take a little longer than usual.

The schema of the perfdb database is still subject to change; I'm not completely happy with the current one, which looks like this:

database: the database from which stats were collected

id	primary key
run_id	unique id for the server run which produced statistics
name	4GL physical database name
type	0/1/2, corresponding with primary/dirty/meta

cache: Hibernate second-level cache statistics, by cache region

id	primary key
did	foreign key to database primary key
cnt	number of elements in the region (I guess at the time they were collected?)
mem	number of bytes stored in the region
hit	number of cache hits
miss	number of cache misses
put	number of cache puts
region	name of cache region

global_query: Hibernate global, aggregate query statistics

id	primary key
did	foreign key to database primary key
min	minimum execution time in ms
max	maximum execution time in ms
avg	average execution time in ms
hit	number of times results were retrieved from the query cache
miss	query cache misses
put	query cache puts
execnt	number of times query was executed in total

rowcnt	total number of rows found by this query
hql	HQL statement text (includes embedded whitespace)

local_query: localized query statistics collected by P2J instrumentation

id	primary key
gid	foreign key to global_query primary key
file	partially qualified name of application source file (just the portion below the application's pkg-root directory)
class	name of application class in which query was created
method	name of application method in which query was created
line	source file line number at which query was created
cnt	number of times query was instantiated at this location

TODO: store a timestamp of the server run, which would be more user-friendly than the run_id.

To access the database, launch the H2 console in the browser from the run/server directory:

```
java -jar <path-to-h2-jar>/h2.jar
```

Connect using the generic H2 (embedded) profile and the following URL:

```
jdbc:h2:perfdb/perfdb;IFEXISTS=TRUE;ACCESS_MODE_DATA=r
```

Some useful queries:

Join most useful global and local query statistics for the last server run, sorted by (an admittedly rough approximation of) busiest query first:

> select d.name as db, d.type, g.min, g.max, g.hit, l.class, l.line, l.method, g.execnt as gcnt, g.avg, l.cnt as lcnt, g.execnt * g.avg as gtot, g.hql from global_query g join local_query l on l.gid = g.id join database d on g.did = d.id where d.run_id = (select max(run_id) from database) order by gtot desc;

Total second-level cache memory used in last server run:

> select sum(c.mem) from cache c join database d on d.id where d.run_id = (select max(run_id) from database);

All second-level cache stats for the last server run, sorted fullest region first:

> select d.name, c.mem, c.cnt, c.hit, c.miss, c.put, c.region from cache c join database d on d.id = c.did where d.run_id = (select max(run_id) from database) order by cnt desc;

#8 - 08/02/2015 09:55 PM - Eric Faulhaber

Rebased task branch 2581a from P2J trunk revision 10906.

#9 - 08/03/2015 09:04 AM - Paul E

I like this a lot - can you please let me know when this revision is pushed our way?

#10 - 08/11/2015 09:46 PM - Eric Faulhaber

Rebased task branch 2581a from P2J trunk revision 10919. Task branch is now at 10936.

#11 - 08/20/2015 06:36 PM - Eric Faulhaber

Rebased task branch 2581a from P2J trunk revision 10927. Task branch is now at 10945.

#12 - 08/22/2015 07:04 PM - Eric Faulhaber

- % Done changed from 0 to 100

- Status changed from WIP to Closed

Task branch 2581a rev 10947 passed regression testing and was merged into trunk revision 10928. Notification was sent to the team. Branch 2581a was archived.

#13 - 12/22/2015 06:58 AM - Paul E

I've been looking into using the statistic collection DB as a way of understanding slow queries. I think it's really useful.

What I've done is start the P2J server with collection-statistics enabled and ran one test (./entitySEARCHTestCases/RELATIONSHIP/PROPERTY.xml) as this is one of the slowest tests in the full search test suite.

I've then stopped the server so that I can look at perfdb statistics for this test only.

The test took 51.645 secs to execute.

I was expecting a lot of this time to be spent in the database, and expecting that the collection statistics would point me at some high value queries to look at. To my surprise this doesn't seem entirely to be the case.

I ran the following query against the H2 DB:

```
select d.name as db, d.type, g.min, g.max, g.hit, l.class, l.line, l.method, g.execnt as gexecnt, g.avg, l.cnt
 as lcnt, g.execnt * g.avg as gtot, g.hql from global_query g join local_query l on l.gid = g.id join database
 d on g.did = d.id where d.run_id = (select max(run_id) from database) order by gtot desc;
```

(note some minor changes to this query as the schema has changed since note 7 was written. Eric: could you please update note 7 accordingly?)

Since the query average time is measured in millis, I'm seeing time in the database adding up to about 5 secs for this API call.

I guess I'm partly suffering from accumulated rounding - a temp table query with max execution time of 19ms has an average execution time of 0ms and is executed 21607 times. Perhaps this is a bigger hitter than it appears.

I'll look into timings when running this API at the database level this afternoon and see how things tally up.

#14 - 12/23/2015 05:41 AM - Paul E

Using pg_stat_statements to gather similar data in the DB gives similar results (unsurprisingly).

It's probably useful to do both as it's useful to see the actual SQL for poor performing queries as well as the HQL. I think it also helps to understand the relationship between generated SQL and the corresponding ABL query. Obviously it's also useful to review query plans.

For the example that I'm looking at, the slowest query contains 3 cross joins and is called 3 times.

The pg_stat_statements data confirms that the time spent in the DB, whilst high, is not a particularly large percentage of the overall test running time.

#15 - 12/23/2015 09:32 AM - Eric Faulhaber

To complete the picture, it probably would be useful to perform CPU sampling on the server (and perhaps on the client as well) with visualvm or a similar profiling tool while this test is running, to see where the JVM is spending its time.

#16 - 12/26/2015 08:33 PM - Eric Faulhaber

Paul Eames wrote:

> [...]
> What I've done is start the P2J server with collection-statistics enabled and ran one test (./entitySEARCHTestCases/RELATIONSHIP/PROPERTY.xml) as this is one of the slowest tests in the full search test suite.
> [...]
> The test took 51.645 secs to execute.
>
> I was expecting a lot of this time to be spent in the database, and expecting that the collection statistics would point me at some high value queries to look at. To my surprise this doesn't seem entirely to be the case.

I realize that your focus here is in trying to track down slow database access, but I think I can explain where a lot of the non-database time you are seeing is spent. Consider that when you run tests in single mode, particularly after a fresh P2J server start, there is considerable, initial overhead which will account for a lot of this non-database time.

One component of this is the time spent establishing a connection between BPM and P2J, which can take quite a while. This is not repeated on subsequent runs, since the connection is persistent.

Another component is caching within the P2J server. For instance, dynamic temp-tables are converted once and cached. These will be re-used for subsequent calls into the API. Dynamic queries likewise are converted and cached. There are a number of other operations which benefit from caching. Together, this caching will amortize the costs of expensive operations over the life of a long-running server. Caches are permanent in some cases, least frequently used in others, and least recently used in yet others. So, not everything in a cache will last forever, but the caching is designed to keep the important stuff around.

The combined effect can be very noticeable. For instance, when I run the RELATIONSHIP/PROPERTY.xml test the first time on my system after a server start, it takes ~24 seconds. On subsequent runs, it takes ~8 seconds. I am not sure where most of the time is being saved. Part of it likely is also saved by database-level caching, but I believe a lot of this time difference is due to the BPM/P2J connection and P2J-level caching.

It may make sense for us to focus on a "warm" state rather than a "cold" state when trying to determine where non-database time is being spent, so we are not chasing one-time costs of dynamic conversions or other cached operations, which generally will benefit from caching in a long-running, production system.

#17 - 12/29/2015 04:43 AM - Paul E

Eric, re: note 16:

The last time our overnight pipeline ran the entitySEARCHTestCases/RELATIONSHIP/PROPERTY.xml test it took 45secs. This time is pretty consistent. This time will not include any time to establish connections between BPM and P2J as we run a sanity check test before running this suite, during which the connections are established.

Similarly to you I've seen times as low as 13secs on a warm run.

I generally agree that warm runs are more useful, and much easier, to focus on. I also want a good understanding of what makes the cold run quite so slow though, to ensure I understand how to keep the system warm (and perhaps how to pre-warm it for some APIs).

#19 - 11/16/2016 12:06 PM - Greg Shah

- Target version changed from Milestone 11 to Cleanup and Stablization for Server Features