Java Persistence Performance

A blog on Java, performance, scalability, concurrency, object-relational mapping (ORM), Java Persistence API (JPA), persistence, databases, caching, Oracle, MySQL, NoSQL, XML, JSON, EclipseLink, TopLink, and other fun stuff.

Monday, August 9, 2010

Batch fetching - optimizing object graph loading

Probably the biggest impedance mismatch between *object-oriented* object models and *relational database* data models, is the way that data is accessed.

In a relational model, generally a single **big database query** is constructed to join all of the desired data for a particular use case or service request. The more *complex* the data, the more *complex* the SQL. Optimization is done by avoiding *unnecessary* joins and avoiding fetching *duplicate* data.

In an object model an object or set of objects are obtained and the desired data is collected by *traversing* the object's *relationships*.

With object relation mapping (**ORM**/**JPA**) this typically leads to multiple queries being executed and typically the dreaded **"N queries"** problem, or executing a separate query per object in the original result set.

Consider an example relational model **EMPLOYEE**, **ADDRESS**, **PHONE** tables. **EMPLOYEE** has a foreign key **ADDR_ID** to **ADDRESS ADDRESS_ID**, and **PHONE** has a foreign key **EMP_ID** to **EMPLOYEE EMP_ID**.

To display employee data including address and phone for all part-time employees you would have to following SQL,

Big database query

```
SELECT E.*, A.*, P.* FROM EMPLOYEE E, ADDRESS A, PHONE P WHERE E.ADDR_ID = A.ADDRESS_ID AND E.EMP_ID =
P.OWNER_ID AND E.STATUS = 'Part-time'
```

You would then need to group and format this data to get all the phone numbers for each employee to display.

...

ID	Name	Address	Phone
6578	Bob Jones	17 Mountainview Dr., Ottawa	519-456-1111, 613-798-2222
7890	Jill Betty	606 Hurdman Ave., Ottawa	613-711-4566, 613-223-5678

The corresponding object model defines **Employee**, **Address**, **Phone** classes.
In JPA, Employee has a *OneToOne* to Address, and a *OneToMany* to Phone, Phone has a *ManyToOne* to Employee.

To display employee data including address and phone for all part-time employees you would have the following JPQL,

Simple JPQL

```
Select e from Employee e where e.status = 'Part-time'
```

To display this data you would write the Employee data, get and write the Address from the Employee, and get and write each of the Phones. The displayed result is of coarse the same as using SQL, but the SQL generated is quite different.

...

ID	Name	Address	Phone
6578	Bob Jones	17 Mountainview Dr., Ottawa	519-456-1111, 613-798-2222
7890	Jill Betty	606 Hurdman Ave., Ottawa	613-711-4566, 613-223-5678

N+1 queries problem

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
... followed by N selects to ADDRESS
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 123
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 456
...
... followed by N selects to PHONE
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 789
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 135
...
```

This will of coarse have very pathetic performance (unless all of the objects were already in the cache). There are a few ways to optimize this in JPA. The most common method is to use *join fetching*. A join fetch is where an object, and its related objects are fetched in a single query. This is quite easy to define in JPQL, and is similar to defining a join.

JPQL with join fetch

```
Select e from Employee e join fetch e.address, join fetch e.phones where e.status = 'Part-time'
```

This produces the same SQL as in the SQL case,

SQL for JPQL with join fetch

```
SELECT E.*, A.*, P.* FROM EMPLOYEE E, ADDRESS A, PHONE P WHERE E.ADDR_ID = A.ADDRESS_ID AND E.EMP_ID =
P.OWNER_ID AND E.STATUS = 'Part-time'
```

The code to display the Employee results is the same, the objects are just loaded more efficiently.

JPA only defines join fetches using JPQL, with EclipseLink you can also use the *@JoinFetch* annotation to have a relationship always be join fetched. Some JPA providers will always join fetch any *EAGER* relationship, this may seem like a good idea, but is generally a very bad idea. EAGER defines if the relationship should be loaded, not how it should be accessed from the database. A user may want every relationship loaded, but join fetching every relationship, in particular every *ToMany* relationships will lead

Join the bot revolution with Bot Libre!

Translate

Select Language ▼

Powered by Google Translate

LinkedIn

Resume

on Monster.com

on Indeed.com

DZONE

Total Pageviews

Popular Posts

How to improve JPA performance by 1,825%
The Java Persistence API (JPA) provides a rich persistence architecture. JPA hides much of the low level dull-drum of datab access, free...

Optimizing Java Serialization - Java vs XM vs JSON vs Kryo vs POF
Perhaps I'm naive, but I always thought Java serialization must surely be the fastest a most efficient way to serialize Java obje i...

Batch fetching - optimizing object graph loading to a huge join (outer joins at that), fetching a huge amount of duplicate data. Also for *ManyToOne* relationships such as *parent*, *owner*, *manager* where there is a shared reference (that is probably already in the cache), join fetching this duplicate parent for every child will perform much worse than a separate select (or cache hit).

JPQL does not allow the aliasing of the join fetch, so if you wish to also query on the relationship, you have to join it twice. This is optimized out to a single join for ToOne relationships, and for ToMany relationships you really need the second join to avoid filtering the object's related objects. Some JPA providers do support using an alias for a join fetch, but the JPA spec does not allow it, and EclipseLink does not support this as of yet (but there is a bug logged).

Nesting join fetches are not directly supported by JPQL either (there is also a bug for this). EclipseLink supports nested join fetches through the Query hints "eclipselink.join-fetch" and "`eclipselink.left-join-fetch`".

Nested join fetch query hint

```
query.setHint("eclipselink.join-fetch", "e.projects.milestones");
```

Join fetching is fine, and the best solution in many use cases, but it is a very relational database centric approach. Another, more creative and object-oriented solution is to use *batch fetching*. Batch fetching is much harder for the traditional relational mindset to comprehend, but once understood is quite powerful.

JPA only defines join fetches, not batch fetches. To enable batch fetching in EclipseLink the Query hint "`eclipselink.batch`" is used, in our example this would be,

Batch fetch query hint

```
query.setHint("eclipselink.batch", "e.address");
query.setHint("eclipselink.batch", "e.phones");
```

In a batch fetch the original query is executed normally, the difference is how the related objects are fetched. Once the employees are retrieved and their first address is accessed, **ALL** of the addresses for **ONLY** the **SELECTED** employees are fetched. There are several different forms of batch fetching, for a *JOIN* batch fetch the SQL will be,

SQL for batch fetch (JOIN)

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
SELECT A.* FROM EMPLOYEE E, ADDRESS A WHERE E.ADDR_ID = A.ADDRESS_ID ANDE.STATUS = 'Part-time'
SELECT P.* FROM EMPLOYEE E, PHONE P WHERE E.EMP_ID = P.OWNER_ID AND E.STATUS = 'Part-time'
```

The first observation is that 3 SQL statements occurred instead of 1 with a join fetch. This may lead one to think that this is less efficient, but it actually is more efficient in most cases. The difference between 1 and 3 selects is pretty minimal, the main issue with the unoptimized case was that N selects were executed, which could be 100s or even 1000s.

The main benefit to batch fetching is that only the desired data is selected. In the join fetch case, the EMPLOYEE and ADDRESS data were duplicated in the result for *every* PHONE. If each employee had 5 phones numbers, 5 times as much data would be selected. This is true for any ToMany relationship and becomes exasperated if you join fetch multiple or nested ToMany relationships. For example if an employee's projects were join fetched, and the project's milestones, for say 5 projects per employee and 10 milestones per project, you get the employee data duplicated 50 times (and project data duplicated 10 times). For a complex object model, this can be a major issue.

Join fetching typically needs to use an *outer* join to handle the case where an employee does not have an address or phone. Outer joins are general much less efficient in the database, and add a row of nulls to the result. With batch fetching if an employee does not have an address or phone, it is simply not in the batch result, so less data is selected. Batch fetching also

Probably the biggest impedance mismatch between object-oriented object models and relational database data models, is the way that data is...

JPQL vs SQL, why not have both
One of the most common questions I see on JPA, is users wanting to know how to write some specific SQL query as a JPQL query. Some of the t...

Batch Writing, and Dynamic vs Parametrized SQL, how well does your database perform?
One of the most effective database optimizations is batch writing. Batch writing is supported by most modern databases and part of the JDBC...

Objects vs Data, and Filtering a JOIN FETCH
JPA deals with objects, and SQL deals with rows. Some developers love JPA because it allows them to use objects in an object-oriented fashi...

What is faster? JVM Performance
Java performance optimization is part analysis, and part superstition and witch craft. In deciding how to optimize their applications a lo...

Got Cache? improve data access by 97,322%
Caching is the most valuable optimization one can make in software development. Making something run faster is nice, but it can never beat ...

EclipseLink 15x faster than other JPA providers
I'm sorry to disappoint those of you looking to see some canned benchmark showing how EclipseLink dominates over all competitors in perf...

But what if I'm not querying by id? (database and cache indexes)
Most data models define a sequence generated numeric primary key. This is the most efficient key to use as it is a single efficient guarant...

Labels

- batch (2)
- batch-fetch (2)
- benchmark (1)
- cache (4)
- cluster (1)
- concurrency (2)

allows for a *distinct* to be used for ManyToOne relationships. For example if the employee's manager was batch fetched, then the distinct would ensure that only the unique managers were selected, avoiding the selecting of any duplicate data.

The draw backs to *JOIN* batch fetching is that the original query is executed multiple times, so if it is an expensive query, join fetching could be more efficient. Also if only a single result is selected, then batch fetching does not provide any benefit, where as join fetching can still reduce the number of SQL statements executed.

There are a few other forms of batch fetching. EclipseLink 2.1 supports three different batch fetching types, *JOIN*, *EXISTS*, *IN* (defined in the BatchFetchType enum). The batch fetch type is set using the Query hint "eclipselink.batch.type". Batch fetching can also be always enabled for a relationship using the *@BatchFetch* annotation.

Batch fetch query hints and annotations

```
query.setHint("eclipselink.batch.type", "EXISTS");

@BatchFetch(type=BatchFetchType.EXISTS)
```

The *EXISTS* option is similar to the *JOIN* option, but the batch fetch uses an exists and a sub-select instead of a join. The advantage of this is that no distinct is required, which can be an issue with lobs or complex queries.

SQL for batch fetch (EXISTS)

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
SELECT A.* FROM ADDRESS A WHERE EXISTS (SELECT E.EMP_ID FROM EMPLOYEE E WHERE E.ADDR_ID = A.ADDRESS_ID AND
E.STATUS = 'Part-time')
SELECT P.* FROM PHONE P WHERE EXISTS (SELECT E.EMP_ID FROM EMPLOYEE E, WHERE E.EMP_ID = P.OWNER_ID AND
E.STATUS = 'Part-time')
```

The *IN* option is quite different than the *JOIN* and *EXISTS* options. For the IN option the original select is not included, instead an IN clause is used to filter the related objects just for the original object's id. The advantage of the IN option is that the original query does not need to be re-executed, which can be more efficient if the original query is complex. The IN option also supports pagination and usage of cursors, where as the other options do not work effectively as they must select all of the related objects, not just the page. In EclipseLink IN also makes use of the cache, so that if the related object can be retrieved from the cache, it is, and is not included in the IN, so only the minimal required data is selected.

The issues with the IN option is that the set of ids in the IN can be very big, and inefficient for the database to process. If the set of ids is too big, it much be split up into multiple queries. Also composite primary keys can be an issue. EclipseLink supports nested INs for composite ids on databases such as Oracle that support it, but some databases do not support this. IN also requires the IN part of SQL to be dynamically generated, and if the IN batch sizes are not always the same, can lead to dynamic SQL on the database.

SQL for batch fetch (IN)

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID IN (1, 2, 5, ...)
SELECT P.* FROM PHONE P WHERE P.OWNER_ID IN (12, 10, 30, ...)
```

Batch fetching can also be nested, by using the dot notation in the hint.

Nested batch fetch query hint

```
query.setHint("eclipselink.batch", "e.projects.milestones");
```

- eclipselink (15)
- hierarchical (1)
- history (1)
- index (2)
- java (3)
- join-fetch (3)
- jpa (11)
- jpql (2)
- json (3)
- jvm (2)
- kryo (1)
- lists (2)
- maps (3)
- mongo-db (1)
- moxy (1)
- no-sql (2)
- partitioning (1)
- pof (1)
- reflection (2)
- sequence (1)
- serialization (1)
- sql (1)
- synchronized (2)
- tuning (1)
- volatile (2)
- xml (2)

Followers

Followers (64) Next

khairil

Vladimir Blagojevic

Follow

Blog Archive

► 2013 (6)

Something that batch fetching allows that join fetching does not is the optimal loading of a tree. If you set the *@BatchFetch* annotation on a *children* relationship in a tree structure, then a single SQL statement will be used for each level.

So, what does all of this mean? Well, every environment and use case is different, so there is no perfect solution to use in all cases. Different types of query optimization will work better in different situations. The following results are provided as an example of the potential performance improvements from following these approaches.

The results were obtained running in a single thread in JSE accessing an Oracle database over a local network on low end hardware. Each test was run for 60 seconds, and number of operations recorded. Each test run was run 5 times. The high/low results were rejected and the middle 3 results average, the % standard deviation between the runs in included. The numbers themselves are not important, only the % difference between the results.

The source code for this example and comparison run can be found here.

The example performs a simple query for Employees using an unoptimized query, join fetching and each type of batch fetching. After the query each employee's address and phone numbers are accessed. The JPQL NamedQuery queries a small result set of 6 employees by salary from a tiny database of 12 employees.

Simple run (fetch address, phoneNumbers)

Query	Average (queries/minute)	%STD	%DIF (of standard)
standard	5897	0.5%	0
join fetch	14024	1.1%	+137%
batch fetch (JOIN)	11190	4.5%	+89%
batch fetch (EXISTS)	13764	0.4%	+133%
batch fetch (IN)	14341	0.6%	+143%

From the first run's results it seems that join fetching and batch fetching were similar, and about 1.9 to 2.4 times faster than the non optimized query. IN batch fetching seemed to perform the best, and JOIN batch fetching the worst for this small data set.

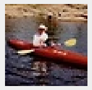
The first run was kind of simple though. It only fetched two relationships. What happens when more relationships are fetched? This next run fetches all 9 of the employee relationships, including OneToOnes, OneToManys, ManyToOnes, ManyToManys, and ElementCollections.

Complex run (fetch address, phoneNumbers, projects, manager, managedEmployees, emailAddresses, responsibilities, jobTitle, degrees)

Query	Average (queries/minute)	%STD	%DIF (of standard)
standard	1438	0.7%	0%
join fetch	1121	0.4%	-22%
batch fetch (JOIN)	3395	3.8%	+136%
batch fetch (EXISTS)	3768	2.6%	+162%
batch fetch (IN)	3893	0.5%	+170%

The second run results show that as more relationships are fetched, join fetching starts to have major issues. Join fetching actually had worse performance that the non optimized query (-22%). This is because the join becomes very big, and a lot of data

Stackoverflow



James
20,696
● 8 ● 54 ● 101

ohloh

ohloh profile for james sutherland

Twitter

Tweets by @j_b_sutherland

Disclaimer

Links

- Wikibook: Java Persistence
- EclipseLink Wiki
- EclipseLink Forums
- ------------
- Paphus Live Chat
- BOT libre!
- LIVE CHAT libre!
- FORUMS libre!

Subscribe To

must be processed. Batch fetching on the other hand did even better than the simple run (2.3 to 2.7x faster). IN batch fetching still performed the best, and JOIN batch fetching the worse.

The amount of data is very small however. What happens when the size of the database and the query are scaled up? I will investigate that next.

Posted by Unknown at 12:03 PM

Labels: batch-fetch , eclipselink , join-fetch , jpa

Posts

Comments

Java Code Geeks

11 comments :

Anonymous November 23, 2010 at 6:06 AM

Please add a link to the second article in the series.

Reply

tunggad December 20, 2010 at 7:08 AM

I really enjoyed this post!

Thank you very much.
Tung Vu

Reply

Abu al-Sous December 27, 2010 at 5:13 PM

amazing thank you so much. I wish there is analysis for Hibernate as well, but this will do

Reply

Rohit Banga February 21, 2011 at 2:46 AM

What is meant by loading a relationship? Refer "...EAGER defines if the relationship should be loaded, not how it should be accessed from the database"

Reply

Rohit Banga February 21, 2011 at 2:54 AM

This comment has been removed by the author.

Reply

João Cavaleiro May 5, 2011 at 3:38 AM

Top post. Thanks!

Reply

Bill Schneider October 27, 2011 at 5:21 PM

Thanks - this is exactly what I was looking for. Batch fetching is starting to look better than join fetching for collections.

Reply

Seb June 27, 2012 at 7:25 AM

James said: In EclipseLink IN also makes use of the cache, so that if the related object can be retrieved from the cache, it is, and is not included in the IN, so only the minimal required data is selected.

That's cool. The native API was NOT doing that. I had to create the code myself.

James said: The issues with the IN option is that the set of ids in the IN can be very big, and inefficient for the database to process. If the set of ids is too big, it much be split up into multiple queries.

I had to split the query myself, since the limit size is DB dependent, that should be handled by EclipseLink. After all, that's the idea of ORM to handle the DB differences.

Reply

Matthew Tallyn June 4, 2013 at 1:30 AM

Great post thank you. What is the most efficient way if you have LOB and BLOB types therefore seemingly SELECT a FROM Enitity a causes the memory to be used up with these memory hungry types. As far as I am aware batch reading won't work with a SELECT a.Field, a.Field2 From Entity a approach, is Join-Fetch the best way in this instance? The entity I am dealing with however has many joins and therefore it may not be as there will be lots of duplication. I've used fetch groups in this instance, but I'm still getting 1 SQL call per join relationship (lazily loaded) which led me to believe there may be a better way.

Reply

James Sutherland June 6, 2013 at 7:16 AM

I would recommend using @Basic(fetch=LAZY) when mapping large LOB fields. This will prevent them from being loaded unless accessed.

I better solution is to put them in their own table, and wrap them in an Entity class, then you have a @OneToOne relationship to them, which gives you more control on when it is loaded and accessed.

Reply

Rodrigo M. Tato Rothamel December 8, 2014 at 10:54 PM

Absolutely superb!

Reply

Enter your comment...

Comment as: beijiang.xu@v ▼ Sign out

Publish Preview

Notify me

Newer Post Home

Subscribe to: Post Comments (Atom)

Powered by Blogger.