EOF-Using EOF-EOF Performance Tuning

EOF Performance Tuning

EOF does many nice things for you. But many people learn some things the hard way; like that a relational->object mapping library can have some serious overhead. A common experience with large and complex object model, is that people model their objects, then do a large fetch and find out that bringing in a large set of EO's can be really slow (relative to what they were expecting with JDBC or direct SQL). The overhead of bringing something into an EO can vary, but seeing 100 times speed degradation over raw SQL is certainly plausible; and for many problems, that is just too stiff a penalty. So what to do about it?

Here are some of the approaches you can take to improve things;

Lighten your objects

The largest penalty in an R/O system, is comparing new objects (items) against other cached verions (snapshots) and consolidating and resolving any changes. So adding things as objects, means you will incur that penalty. Still, there are some minor benefits to be had if you lighten the objects.

Your EO's do not have to model everything that is in your table. And you can have many EO's that map to the same table. The valuable part of an EO is that you model in the relationships and can have relationship traversal; but those relationships can have costs. If you thin out objects of relationships you don't need, they will load faster; the same for attributes you don't need. You can even take this a step further; and have multiple flavors of an EO, for what you are doing at a given time. So have an EO that has just the attibutes and relationships that you need for a particular fetch, and another variant for a different type of fetch; and finally the full blown version that you load only in the cases where you need ALL the information for viewing or editing. So most of the your collections of EO's are lighter weight and more targeted variants than ones you need for large edits.

There are other techniques as well; using the little lock attrubute in EOModeler to make many more items opaque (or non mutable) to EOF. EOF compares those attitubutes with locks to figure out if an object in the snapshots is stale or not; so reducing the attributes it needs to compare, can speed up the overhead. Some people take this to the extreme, and set all/most of the attibutes to unlocked, and have one time-stamp that is the only keyed item; then if any attibute changes, that one timestamp will also change, and EOF will still manage the snapshots, but with fewer attributes. But this seems to push more into code, for the sake of saving EOF some runtime maintenance with each objects.

But the point is that the lighter and simpler an EO is, the faster it is. However, the main penalty is on serialization/synchronization at all; so the returns are probably not dramatic enough, in most cases, to justify the measures necessary to create these "optimizations".

Adapt your model

When you are going to be using a relational to object mapping library (like EOF), you should expect that this will change your requirements; enough that you can adapt your model to fit the tool.

If fetching an EO is heavy/slow, then generally the fewer objects you bring in, the faster your system will perform. So if you collapse and denormalize many small tables, into a few bigger ones, you will be doing fewer EO loads, and probably dealing less with all that fault and relationship management of all those little fragments and relationships; which can result in performance savings.

You can do this a little by flattening relationships, or using views in the database to make things appear to be flatter than they are; or you can go right to model and actually flatten. Arguments can be made for each, depending on your requirements.

You can even go further, and start moving complex data structures and relationships into blobs that you manage yourself. This offloads EOF from managing them, and often allows you to speed things up; but the cost is more code maintenance on your part, and of course denormalizing can negatively impact the design so you want to be careful about how zealously you go down this path.

Raw-Rows (late promotion)

One of THE most important techniques for speeding up acces, is using Raw-Rows; or doing late promotion. Basically a raw-row fetch is a way for you to do a database fetch that returns a data structure (array of key-value pairs), instead of fetching and promoting that to a full blown object (EO). This lighter fetch takes much less time, and is much better for large collections.

Then on pages that use those collections can use Raw-Rows, and you promote an object out of that collection to a heavy-weight EO, only on those pages that need to work with an entire object (and all the complexities thereof).

Raw-SQL

Fetches generated by EOF or EOModeler or other tools are generally not as tuned as those generated by a human being; if they were, then machines wouldn't need us, and could program themselves. By creating the Raw-SQL yourself, you can generally tune and optimize fetches yourself, to either fetch in just the parts of table (both columns and/or rows) that you have interest in; again improving the performance and tuning the areas that most need it.

Another major boost in performance here, is that you can traverse your relationships if you select the fields you need from your relationships as part of the raw rows. This also reduces your transaction load (some databases only allow 100 TPM with cheaper licenses), and often increases fetch speed as all data is fetched in the one SQL query with a multiple table SELECT statement (a JOIN). You can also command your EOModel as to wether the relationships perform inner or outer joins which can be a major benefit for checking data integrity. (Check out some SQL sites to learn about INNER, LEFT OUTER, OUTER and RIGHT OUTER joins to see some great data collection methods)

Get closer to the Database (Stored Procedures)

While it seems like it is defeating one of the purposes of a database abstraction layer to get down and dirty with a database (using stored procedures, etc.); it may be a necessary or prudent step.

Stored procedures can do things at the database level that are much faster and lighten the app server load immensely. The cost is a lack of database independence; and you must learn intimate details of the database you are using. But a few well placed StoreProcedures can offer orders of magnitude difference.

Caching

One issue that needs to be discussed on its own is caching. While EO's are slow; they do cache - so subsequent fetches (refetches) are significantly faster than first (primary fetches). This works well on not mutable (static) data. Less well on dynamic data. Still, you can use more than the default cache behaviors. You can fetch raw-rows of non-mutable data into your own arrays and manage your own cache of those. You can also put complex or compound attibutes in a cached attribute and fetch that item, instead of getting that live or constantly recalculating that from all the source parts (prebuild some calculations or collections).

So caching not only means WO+EOF should cache, but you might want to consider what you can cache to lighten various fetch loads as well.

Prefetching / Hinting

Now various prefetches and hints don't really save you time, in that you still have to load all the objects in and pay the same load time for the; but they can move where you pay that time. Sometimes this can dramatically improve the percieved performance, even if it doesn't alter the actual performance. Users might tollerate a 5 or 10 second penalty on one page, if the next 10 things they interact with are all much faster; and vise versa. So it helps to figure out what a user is trying to do, and how they are trying to do it; and tune accordingly.

Reduce your dataset

This may sound obvious, but reduce your dataset whenever possible. For example, no page should really display 5,000 elements - that takes way too long to load and display. Better to use fetch limits or similar type of behaviors to show a user a smaller set of data whenever possible; and then use various techniques to fetch as little data as possible. The smaller the dataset, generally the faster you can work with it.

Conclusion

These are just some of the techniques you can think of to speed up EO fetches.

Some might say that if you have to do all these things that you're doing all the work for EOF for it; or your spending as much time optimizing code as it saved you in the first place. While in a few extreme cases that might be true; generally what happens is that EOF allows you to create a solution very quickly and easily; then you go back and learn where the bottlenecks are (profile), and tune those areas until they perform reasonably for your requirements. This allows the advantages of a RAD (Rapid Application Development) tool, with later the power and speed of a lower level tool; if you'll commit the time to tuning those areas that need to be tuned. And many classes of Apps may require very little tuning.

Vendor-Specific

For vendor-specific performance tuning, refer to the vendor-specific sections.

Profiling EOF

Anjo Krank

Take a look at the ERXAdaptorChannelDelegate in Project Wonder.

Indexes

Chuck Hill

The SQL that EOModeler generates for many to many join tables does not allow full optimization of the join. Many to many join tables are only generated with a single index which is the compound key of both columns.

For example,

```
CREATE TABLE "GROUP_FILE" (
    "FILE_PKEY" NUMERIC NOT NULL,
    "GROUP_PKEY" NUMERIC NOT NULL
  );

  ALTER TABLE "GROUP_FILE"
    ADD PRIMARY KEY
      ("GROUP_PKEY","FILE_PKEY")
    NOT DEFERRABLE INITIALLY IMMEDIATE;
```

This index can only be used to optimize joins on the first column in the index (which is the second column in the table for some reason). The lack of an index for the other column results in a table scan when joining on the other column. This sort of joins happen when following a to many relationship. For example, group.files() will result in SQL similar to this for each element in files():

```
SELECT f.*
  FROM FILE f,
       GROUP g,
       GROUP_FILE gf
  WHERE
        f.PKEY = gf.FILE_PKEY AND
        g.PKEY = gf.GROUP_PKEY AND
        g.PKEY = 10;
```

Only the GROUP_PKEY has an index which is usable for optimizing this query. The join on FILE_PKEY (f.PKEY = gf.FILE_PKEY) is not optimized and results in table scans. You need to manually add an index for this:

```
CREATE INDEX GROUP_FILE_FILE_PKEY
       ON GROUP_FILE (FILE_PKEY);
```

This can make a truly dramatic difference if there are a lot of rows in the join table.