

Re: Help! Difficulty understanding DB -> Object mapping

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2005-05/msg00091.html>

- *From:* "H. S. Lahman" <h.lahman@xxxxxxxxxxx>
 - *Date:* Tue, 10 May 2005 17:54:44 GMT
-

Responding to Parker...

Basically what this means is that one should grab as much data as one is /likely/ to need all at once even if sometimes it isn't all used. The bottleneck is getting it into memory as datasets; extracting particular information from the datasets in memory is the fast part.

The problem here is that data loaded into memory has to be managed, it has to be invalidated if data on the server has changed, and this is generally not trivial. There is a strong case to be made for getting the data from the DBMS each time, for simplicity. The performance is usually fine, DBMS's are fast, and they cache too. Also there is the issue of volumes of data, granted, memory is cheap, but with data the only numbers that count are zero, one, and as many as you like.

I agree there is no free lunch. Achieving good RDB access performance requires extra work in a particular application. But the OP expressed a concern over performance, so I took that as a given. [I would add that whoever writes the subsystem needs to understand the real trade-offs, which are not always obvious. Fortunately such subsystems tend to be highly reusable.]

However, the main thing I had in mind in the quoted paragraph is the notion that if one has to access a Customer, say to verify the Customer name provided in the UI, it will usually be more efficient to go ahead and read all the Customer fields that /might/ be of interest later plus any data in relevant related tables at that time rather than doing so with separate accesses on an as-needed basis later. That sort of poor man's caching is almost always worthwhile.

Re: Help! Difficulty understanding DB -> Object mapping

Prefer single complex queries to multiple simple queries.

Yes, but doesn't that contradict your advice to minimize joins? Most DBMS's have been able to give good performance with joins for some time, although in the past you had to have a pretty good knowledge of how the query was executed in order to achieve that performance. When I worked with Sybase many years ago, we fondly referred to the "query optimizer" as a "query pessimizer", and the idea was to work around its shortcomings. But I think most DBMS vendors today have pretty good support for joins.

Yes, there is a contradiction (aka trade-off). But if one needs the data, the join is going to have to be done anyway so the issue is when and how. I also agree that nowadays any problems are more likely to be a side-effect of the schema than any inherent problems in query processing. But there is still a fixed overhead in simply processing a multi-table query, which segues to...

If possible, create specialized indices and store "compiled" joins for queries that are commonly used.

Not necessarily. "Compiled" queries, say in stored procedures, are typically tokenized on the first invocation, and a query plan is computed based on the passed parameters for that invocation. If those parameter values are atypical, the query plan may be off. Besides, processors are so fast these days that the time to compile a query is miniscule compared to the time to retrieve the data.

True. But this just comes back to the notion that optimizing RDB access is a unique subject matter and whoever implements the subsystem had better be familiar with the problem domain. That's why I told the OP that there was no single magic solution.

The performance issue you site here is more of an issue for the RDB server than the application. The RDB may be processing thousands of queries a second so those "minuscule" differences tend to add up and the

Re: Help! Difficulty understanding DB -> Object mapping

DBA will thank the application developers for being well behaved. In addition, depending on how smart the DBMS is, storing the query could result in permanent indices being set up to service it, which could significantly improve performance. But, again, the subsystem developer needs to know about that.

When mapping to the problem solution's needs when performance is a big problem, look for ways to use write caching or anticipatory reads. Cache requests rather than opening long transactions whenever possible, especially if the data is from user keyboard entry.

The problem with delayed updates is that the cached data may become stale. Suppose you read a record, cache it, somebody else reads a record, changes a field and saves it, then you make your save, and overwrite the other users change. A common solution to that is optimistic locking. Basically, when you read your cached data, you also read a last updated timestamp on the record, and if you try to update when that timestamp has changed, your update fails.

Sure. Any time the application does its own caching, read or write, one has to be aware of the risks and implement solutions to mitigate them.

There is nothing wrong with me that could not be cured by a capful of Drano.

H. S. Lahman
hsl@xxxxxxxxxxxxxxxxxxxxxx
Pathfinder Solutions -- Put MDA to Work
<http://www.pathfindermda.com>
blog: <http://pathfinderpeople.blogspot.com/hslahman>
(888)OOA-PATH