

Re: SQL

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-01/msg00276.html>

- *From:* cb@xxxxxxxxxxxxxxxxxxxx (Christian Brunschen)
 - *Date:* Wed, 25 Jan 2006 09:53:41 +0000 (UTC)
-

In article <1138173069.170549.220700@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>, frebe <fredrik_bertilsson@xxxxxxxxxxxx> wrote:

```
>>>* Queries.
>>Those I'd generally consider to be part of the relational model
>Correct.
>
>>>* Transactions.
>> Those are definitely useful, but are not specific to databases
>Correct.
>
>>>* Referential integrity
>>... has to be maintained somehow: dangling references would be a problem
>>in any system
>
>>>* Caching.
>> Caching also isn't specific to RDBMS:es.
>Correct.
>
>My point is that these features are useful non-persistence features
>provided by a DBMS.
>
>You claimed: "If we ignore the persistence aspect, what remains is the
>organization of data according to the relational model. That's
>certainly useful, but it's not 'MANY' features. "
>
>I claim that a (R)DBMS provide MANY useful non-persistence related
>features.
```

Well, the above are 'a few' or even 'several', but not 'many'. One of them – queries – is really a part of the relational model itself (whether you use a query language or some other interface is immaterial). Transactions, as offered by RDBMS:es, are limited to the data stored within the RDBMS, so that if you want to use its transaction capability, you need to store the appropriate data in the RDBMS. Referential integrity, as I said, you still have to maintain yourself: The RDBMS only catches and flags up errors (or handles them in another way, which you must have told the RDBMS to use). Caching is something that you get 'for free' in many other approaches as well.

Re: SQL

>I don't claim that a RDBMS is the only product that may
>provide such features, but currently, for an average enterprise
>application, a RDBMS is the best available product to provide these
>features.

Certainly a RDBMS gives you a useful collection of things to use; but if you wanted to use one or maybe two of them _without_ wanting to use its data storage model, then using the RDBMS won't help you, because it doesn't offer transactions, caching, or referential integrity support for anything other than what is handled within its data model.

So, if you want to use some of those features, without using the data model, an RDBMS would most likely _not_ be the best choice.

>> Do you have any examples of relational databases that have specific features
>> for non-persistent usage? All the relational databases I've looked at
>> (again, a limited number) appear to put a lot of weight on the persistence
>> aspect.

>I just gave you four examples.

None of those features are 'specific features for non-persistent usage': caching, for instance, is probably more appropriate for persistent than for non-persistent usage (if the entire thing is already in memory, what needs to be cached?); both queries, transactions and referential integrity are just as applicable to persistent as to non-persistent data, so also are not 'non-persistent use specific'.

>If you are asking for a RDBMS product
>that is suitable for all-in-RAM use, look at hsqldb.

Cool, I will. [... a quick look later ...] Looks very interesting, and indeed offers primarily memory-based tables (as well as 'cached' ones, for datasets that need to persist, or simply exceed the size of available memory). Thanks for the pointer!

>>>> However, if you look around, I think you will see that the
>>>> _vast_ majority of uses of databases are, in fact, for _persistent_
>>>> storage of data.
>>>Only in the OO world. In the rest of the world there are many examples
>>>of the opposite.
>> I'm open to be educated on the subject – please, could you point me at
>> some examples?
>
>If you look at enterprise applications outside the OO world, you will
>find that they heavily use embedded SQL.

Please, give me some more specific pointers.

>Instead of loading the data
>into memory structures, a select statement is used everytime some data
>is needed. The RDBMS is configured to cache most of the data needed,

Re: SQL

Re: SQL

>into memory. It means that the application asks the RDBMS for data that
>resides in RAM. In this case, the persistence features is not involved
>at all.

So this would be entirely for data that is transient (i.e., the data in that database is not deliberately kept around between executions)?

>Another example is the use of transactions. This feature is not related
>to persistence and enterprise applications uses them a lot.

But neither are transactions, as offered by RDBMS'es, applicable to things outside the RDBMS:s scope – i.e., outside the data in the RDBMS. So, if all you want is transactions, an RDBMS probably shouldn't be the first place to go.

>> In my experience, even when I was developing procedural systems, in
>> those systems, relational databases where used to work with persistent
>> data. Transient data was generally stored in bespoke data structures in
>> memory. This is without any OO involved.
>This is true for some kind of applications, but normally not for
>enterprise applications.

What precisely is your definition of an 'enterprise application'? I've often thought of them as frequently working with large sets of data, often data that is already in a database. Certainly in such a way that temporary data is created that also needs to be managed, and because the input and output may be coming from databases, using a database for the temporary data would make an eminent amount of sense (keeps all the data handling similar, reusable, regocnisable, easier to maintain); but from your statement above it sounds like you would characterise enterprise applications as using databases not for incoming our outgoing data, but mainly for transient data used only in the process of whatever they are doing?

>Look at an old COBOL program. How advanced are
>the data structures in COBOL? Almost the only thing you can do is to
>traverse an array. All other kind of searched has to be done using a
>select statement. Still COBOL was a very popular language, so the
>concept with letting the DB take care of the collections handling was
>not probably a very bad idea.

So, databases were used to overcome the deficiencies in COBOL's support for data structures? Cool, though I would class that as a workaround. Of course, now we have languages with much better datastructure support, so that workaround is no longer necessary.

>The concept with loading data into advanced structures instead of
>making select calls, was originally caused by performance reason.
>Currently I work with a scheduling application there I have to use this
>concept. The result is bloated and messy code using TreeMap, HashMap,
>ArrayList, etc, and I every minute I wish I could made a select

Re: SQL

Re: SQL

>statement instead. But the time overhead with the interprocess
>communications is simply too high. (One solution would indeed be to use
>hsqldb as an all-in-RAM, in-process DB). But I strongly argue for just
>using this this concept when performance reason force you to do. Using
>select statements will give you much less bloated code.

If you package up your data structures appropriately and offer suitable operations on them, you can end up with a system that becomes similarly easy to use as a database, but still offers you all the performance benefits of using your bespoke data structures.

Also keep in mind that the SQL necessary to operate on a complex database can become quite, um, *_interesting_*, such that even sequences of map lookup, array indexing and pointer traversal can look quite simple and straightforward in comparison. Of course, this very much depends on the level of familiarity of the developer with both the application's language and environment on the one hand, and with SQL and the relation model on the other.

Yes, a RDBMS is something that offers a hugely flexible system for storing data, and a unified interface for accessing and otherwise working with those data. But as you mentioned yourself, performance considerations do come into play as well. It may well be that even an in-memory RDBMS might be too slow for your application.

And there still remains the issue of business logic. Using an OO system, you can keep your data, their interrelationships etc, the primitive data operations, *_and_* their business logic all together. I know that much can be done in RDBMS:es these days by writing stored procedures, adding triggers etc, which allow you to essentially put business logic into the database engine, though I am somewhat wary to use such an approach as that can lock you into a specific database vendor's extension language. Of course, looking at hsqldb, stored procedures etc would be written in Java, just as the rest of the program, and executed potentially within the same virtual machine ... Interesting things to think about.

By the way, thank you for offering useful and civilised discussion and debate :)

>Fredrik Bertilsson
><http://butler.sourceforge.net>

// Christian Brunschen

.

-
- *Follow-Ups:*
 - ◆ *Re: SQL*
 - ◇ *From: frebe*

- **References:**
 - ◆ **SQL**
 - ◇ *From:* priya1
 - ◆ **Re: SQL**
 - ◇ *From:* frebe
 - ◆ **Re: SQL**
 - ◇ *From:* Christian Brunschen
 - ◆ **Re: SQL**
 - ◇ *From:* frebe
- Prev by Date: **Re: SQL**
- Next by Date: **Re: SQL**
- Previous by thread: **Re: SQL**
- Next by thread: **Re: SQL**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**