In defence of SQL

posted 12 July 2010, updated 13 July 2010

If this title does not interest you, here are some alternative, linkbait titles:

- Why ORM is the Dumbest Idea Ever
- Why NoSQL is a Terrible Idea
- OMADS: the future of data storage
- Why SQL Will Eventually Conquer The World

A little history

SQL was invented in the 1970s at the same time that "large-scale" (read: millions of rows) data stores came into existence. It triumphed over other query languages not because it was particularly great (though it was easier to read), but because it was standard. Everybody building a data store could write to the SQL standard without having to re-train all their clients and customers. It reduced friction all round. It was a huge success.

SQL is awkward

There's no escaping that SQL, as we use it day to day, is not pretty.

Keep in mind that what SQL is really designed to express is relational algebra, a type of logic essentially invented by the ridiculously clever E.F. Codd (along with nearly all the other theoretical underpinnings of relational databases). If you're not familiar with it, I find it helps to think about relational algebra as Venn diagrams: it's about sets intersecting with, unioning with, subtracting from, joining with each other. Find all the fruits in set A, with prices in set B, farmed by the farmer in set C. That kind of thing.

What it's *not* really for is collating, aggregating, and most especially filtering of data sets. The reason count(*) is so awkward is because that's not really what the language was designed to do. GROUP BY and ORDER BY clauses look tacked-on because they are (HAVING is an even more grievous hack, UNIQUE is a disaster, and let's not get started on LIMIT). Of course, in regular use of a data set, you nearly always want to do these things, which is why SQL provides them. SQL, loyal workhorse that it is, is nothing if not willing. But it might not be terribly quick.

So you're right. SQL -- the kind you write every day -- is ugly and awkward. In fact, it looks like hell on legs. And it's often pretty slow. And that's all because you're asking it to do something it, the language, is not really designed to do (whether the *engine* is designed to do it is another question). But it works, and in forty years since its invention we have come up with very little in the way of improvements and nothing close to powerful enough to be a replacement.

What about ORM?

I want to be very, very clear about this: ORM is a stupid idea.

The birth of ORM lies in the fact that SQL is ugly and intimidating (because relational algebra is pretty hard, and very different to most other types of programming). Our programs already have an object-oriented model, and we already know one programming language -- why learn a second language, and

a second model? Let's just throw an abstraction layer on top of this baby and forget there's even an RDBMS down there.

This is obviously silly. You've stored your data in a way that doesn't match your primary use-case, accessible via a language that you are not willing to learn. Your solution is to keep the store and the language and just wrap them in abstraction? Maybe you'd do that if your data were in a legacy system and you needed to write a new front-end, but people slap ORM on *new* projects. Why the hell would you do that?

ORM is slower than just using SQL, because abstraction layers always are. But unlike other abstraction layers, which make up for their performance hit with faster development, ORM layers add almost nothing. In fact, often, if you need to do anything more complicated then a SELECT, you end up writing fragments of SQL or pseudo-SQL languages in order to tell the underlying RDBMS what you're trying to really do.

OMADS: data stores that match the application

ORM is dumb, and people noticed. So clever programmers looked at this ridiculous edifice and realized the real problem: the data store and the use-case were mismatched. So they threw away ORM, SQL, and RDBMS, and wrote lovely new key-value stores, or object stores, or document stores, or searchable indexes, or any of a half-dozen other data structures that more closely matched what they were trying to do. And because these data stores all turned up at a time when nearly all data stores were SQL-interfaced RDBMS, they got the name "NoSQL", even though the actual problem was the Relational model, not SQL itself. And because "Obviously More Appropriate Data Stores", or OMADS, is not catchy enough I guess.*

So I love NoSQL stores. My startup would literally be unable to function without memcache. I think Cassandra is nifty even if Twitter found it not worth the trouble of switching from MySQL. I think Redis is cool if a little buggy. MongoDB is awesome, and I'm probably going to be building a production system based on it quite soon. HDFS I use in production every day, and it still blows my tiny little mind. Really, the only think I dislike about them is the label "NoSQL", which as many people have already pointed out doesn't really say anything about what they **are**, just what they are not. And also because it makes people unfamiliar with the details of the situation think there's something Wrong, Bad or Old Fashioned about SQL. And programmers hate using anything that is any of those things.

What is the relational data model good for anyway?

So if your data store should always match your application, what application is it that RDBMS are perfect for? The answer is: all, and none.

We take this for granted these days, but the relational model is pretty magical. Set up a model of your entities, pour data into it, and get answers. How many teachers at the university earn over $100k but teach less than 20 students? How many customers who bought our newest product had never bought anything before? What were sales like on Tuesdays over the last 30 months? You don't have to know in advance what your questions will be; you don't have to write any special code to examine all the rows, or work out the most efficient strategy for combining the results: you just need to know how the data relate each other, and then you can ask ad-hoc questions and *the database knows the answer*. I remember the first time I really grokked that concept, and it filled me with nerdy joy.

If you pick the wrong data structure for your store when you're first writing your application, you can end up -- as happened to a team at my last job -- running crazy, days-long depth-first searches across

distributed document stores in order to perform elementary operations like getting a total count of objects. So if you don't know all the questions you might need to ask about your data, the safest thing to do is put them in an RDBMS. And when you first start a project, you almost never know all the questions you're going to need to ask. So my advice: always use an RDBMS. Just don't *only* use an RDBMS.

Optimize, but be prepared for ad-hoc queries

Is your data really just a giant hash lookup? Then a key-value store is what you want. Do you primarily access your related data via a single key? Then a document store is for you. Do you need full-text searching? Then, dear god, use a text-indexing engine, not an RDBMS. Do you need to answer questions about your data that you can't predict in advance? Then make sure your data also ends up in an RDBMS. Maybe not in real-time, maybe summarized rather than in raw form, but somehow. Then when your co-founder asks "how many Xs happened in Y?" your answer won't be "uh, let me spend half a day writing code to find that out". Just throw down some SQL, and it'll give you an answer -- it'll take 5 minutes to return a single number, but that's a lot faster than half a day.

Because that's what SQL is for.

Post-SQL

If you scroll back to the top you'll see the description of the circumstances that gave birth to SQL: a whole bunch of new data stores came into existence at once, and the lack of a common language created friction and fragmentation. The same thing is happening again with the NoSQL crowd. If you decide to write your app using Cassandra, you better be sure it's what you want, because if you change stores you have to change *all* your code. It's the ultimate lock-in, and it's not the plan of an evil monopolist corporation, it's just an unfortunate side-effect.

Pretty soon, the same sort of clever people who noticed that ORM was a ridiculous hack will notice an opening for an actually useful abstraction layer: a single common API that can access all the NoSQL stores. Maybe it will be Thrift or Avro, but I'm not sure. I'd say the chance is about 50-50 that it will be SQL again.

SQL triumphant

And why not? Awkward it may be, but SQL is a lot more succint and readable than multiple lines of API calls or crazy, math-like relational algebra languages. And there's nothing intrinsically slow about the language itself. If you could run "SELECT * FROM table WHERE ..." on Cassandra, it would be no slower than specifying the same conditions via API calls. In fact, when trying to explain how to use its API, the MongoDB documentation lists the equivalent SQL queries. That's a pretty clear vote for the usability of SQL.

Computer programmers really like new, cool things. So when something like SQL hangs around for nearly 40 years, it either means nobody really cares about it -- I think we're clear that's not the case -- or that there's really nothing else that can do the job quite as well.

So go forth, use your OMADS, keep an RDBMS in your back pocket, and stop being so mean to poor old SQL.

---

* On the off-chance that anybody starts calling these things OMADS, remember: you heard it here first.

Updated 2010-07-13 to fix link to E.F. Codd; thank you Sordina!

tagged with

- cassandra
- data
- databases
- memcache
- mongodb
- nosql
- sql

Comments

- **matt**

posted 12 July 2010
I really dig SQL. It's extremely readable, which is why I was experimenting with JSQL a while back: http://github.com/richtaur/jsql (useless but a neat POC)

These days I'm more of a frontend so I just use what works, is easy and fast, and/or comes packaged with my platform (like Big Table in GAE).

BTW I love that you're such an SQL fanboy :) I always know who to ping with questions!

- **Antonio Ognio**

posted 12 July 2010
Really liked this post. I still feel OK using ORMs since they save you tons of boilerplate code. This kind of abstraction layers could slow things down but really let you be more productive.

I'm glad I'm "old enough" (at 33) as to have coded a good deal of SQL queries by hand in my life and grokking it :)

In almost 8 years of coding all my queries in SQL mostly by hand (sometimes with hacked-up helpers) in PHP and using and abusing arrays I feel much more natural to use model classes in Python and Ruby frameworks now.

Google AppEngine has done a decent job with GQL for BigTable and the way you use it to retrieve objects not just simpler record containers.

The point of ORMs is not forgetting or avoiding to learn SQL but saving you time and errors writing SQL by hand and hidrating objects on your own. If you're not clueless
you know it's gonna be slower but you're OK paying that price. When the time comes you're better off resorting to plain old SQL to save your butt rather and insisting in doing it in an stupid but ORM-driven way.

I praise OMADS too (liked the term) and think storing the same data in a RDBMS (VoltDB could be a good choice) is definitely the way to go if you want to interrogate your data in time for helping your business not crash.

Please keep posts like this comming :)

Greetings from Lima, Peru.

- **Jeswin Kumar**

posted 12 July 2010
Disclaimer: I do write an ORM toolkit myself, at http://www.agilefx.org

I have to disagree with some of your points regarding ORM.

1. There is hardly any performance hit in 95% of the cases, considering that query generation and compilation takes a fraction of I/O costs.

2. The biggest benefit of ORMs in statically typed systems is, well, Type Safety. You get the compiler to verify large parts of your code.

3. It is common during development (and rarely even in production) to change table schemas. ORM makes it easier, since type checking jumps in and tells you what is broken.

4. And finally, it takes out a ton of boilerplate - I am surprised you haven't noticed this.

There are many, many good applications running well with ORM to dismiss it. Like all apps on Rails, Django. In fact, Microsoft is betting its future data access strategy entirely on ORM (Entity Framework). They even modified C# (with LINQ) to support the object-relational impedance.

- **Sordina**

posted 12 July 2010
> relational algebra, a type of logic essentially invented by the ridiculously clever E.F. Codd

The RA, and Codd links both point to the relational algebra page on Wikipedia.

- **alexp**

posted 13 July 2010
Well, what you say is happening now. First step towards that which I noticed is Moneta (Ruby gem), that is giving you some abstraction over key value stores. I don't think that it's a whole lot of performance overhead and it does let you switch when you realize it's time to.
Overall, you may also want to use orm for sakes of interchangeability of your SQL as well and it certainly gives you advantages.

Im working on a project that involves no orm and uses oracle as a primary data store. Though I don't want to install oracle On my machine but repository idea built-in to project doesn't let me use postgre for development. And yet schema creation involves varchar2 that's simply

inaccessible in postgre. Its quite easy to fix but I'd expect to have some kind of a fallback.... Thanks for the article. I liked it a whole lot.

- **Rob Conery**

posted 13 July 2010
Couple of things: "defence" is spelled "defense". I hate it when people nit my spelling but ... well come on. It's your title :).

NoSQL (Object databases to be specific) have been around since 1974, right when OO languages became prominent. Your argument that SQL has been around for 40 years is unstudied.

This doesn't mean you're wrong - it means you're reactionary. Take some time to know what you're debating; understanding your opposition's points is key to knowing if you're correct.

ORM's aren't actually slower. In some languages it's actually *faster* to precompile a command that is sent in binary format to the driver. Believe it or not: strings take memory and (I know this sounds weird) writing a query in code DOES NOT equal writing it in the DB execution environment. It needs to be handed to a driver, and so on. Some ORMs are really smart this way, and will cache the query calls.

ORM abstraction is much more than "not willing to learn SQL". It's about transitioning an app to other drivers without changing your code. It's about debugging something other than a string, and it's about not destroying your database with unnecessary calls. Between you and me, however, I like SQL alot :).

There's room for both in this world - SQL is for working with data, not working applications.

- **Steve**

posted 13 July 2010
Just a quick FYI Rob, defence is how it's spelt in the UK, and it appear Laurier grew up there (or here, for me :)

- **Rob Grainger**

posted 13 July 2010
Funnily, I'd like to see SQL ditched for the opposite reason - it fails to be a relational language. Codd argues this strongly in his book The Relational Model of Database Management v2. (1990). In the same volume, he gives coherent arguments against various other models of DBMS - many of which still apply to the current crop of NoSQL databases. Saying that, I see a role for these, but probably a different one from the areas where I'd prefer to see an industrial strength RDBMS.

Personally, I do see the use of ORM's, for the type-safety reasons mentioned by some of the correspondents above, and moreover to reduce the mismatch between an OO model and a SQL-based database. The overhead of the mapping layers is typically negligible - most of the overhead of calls is taken up with data access (file access and or network transport).

- **Mike Scott**

posted 13 July 2010
Rob,

Steve's right. Defence is spelled that way in English. Originating in the 14th century from latin defensus through old French defense. The replacement for "s" with "c" in such words is common, e.g. pennies -> pence, Duns -> dunce, fens -> fence.

Consistency is more important in my opinion. If you want to spell defence with an "s", then you should change all words with the same root. So you would have a fense around your garden, not a fence. It doesn't make sense to have a fence as a defense :-p

- **Dominic Pettifer**

posted 13 July 2010
You've completely and totally missed the point of using an ORM. It's simple: you write far less code using an ORM, and less code is best code, it's more maintainable, you're not got multiple violations of the DRY principle, there's fewer chances of bugs etc.

You're saying you'd rather write this:

```
SqlConnection conn = new SqlConnection(mySqlString);

string sql = "SELECT * FROM Products WHERE Id = " + productId;

SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();

if(reader.Read())
{
string productName = reader.GetString(0);
string description = reader.GetString(1);
decimal price = reader.GetDecimal(2);
// ....loads of other properties (snip)...//
}

reader.Close();
conn.Close();
```

...instead of this?

```
Product product = ProductRepository.FromId(productId)
```

Then you are completely mad. Lets count the problems

- 10+ lines of code instead of 1.
- You'll likely have this repeated everywhere = more code, less maintainable.
- Risk of SQL injection problems.

- Using magic strings that won't offer you compile time safety
- You're far more productive with an ORM because the repetative work gets done for you.
- When you need to process the data from an SQL database (display it etc.) you're more than likely to be inside an OO programming language anyway, why not access your data in an OO way.

The birth of ORM DOES NOT lie in the fact that SQL is ugly and intimidating, often the equivalent OO code for running queries is equally intimidating, it lies in the fact that people want to be more productive. People who use ORMs aren't trying to avoid SQL, as you can still write Stored Procedures and have them mapped to your ORM anyway.

Yes, there may be a tiny overhead with it being an abstraction layer, but it's so small compared to the IO overhead of accessing the database in the first place. And often that overhead is usually only the equivelent of buying a slightly faster CPU vs. spending hundreds of thousands $$$$ more on more developers + constracters to maintain a mess of SQL strings and boilerplate code everywhere. Hardware is cheap, developers and time are not.

- **Ron Leisti**

posted 13 July 2010
If you skip to the "Implementations" section in the wikipedia article on Relational algebra, you'll find the following biased (but commonly accepted) bit of wisdom:

"Even the query language of SQL is loosely based on a relational algebra, though the operands in SQL (tables) are not exactly relations and several useful theorems about the relational algebra do not hold in the SQL counterpart (arguably to the detriment of optimisers and/or users)."

Also, apparently Codd himself did not approve of SQL.

I don't think it's fair to put the blame for SQL on the relational model.

- **Jonathan Dickinson**

posted 13 July 2010
I don't agree :(.

You explain the joy of SQL as "and then you can ask ad-hoc questions and the database knows the answer."

This sounds like OLAP - and you should NOT be using a relational model to achieve that (you are going to get really poor performance). OLAP data should be stored as hydrated/computed cubes etc.

OLTP is still relevant in the SQL world. Sure storing my objects in a 3rd normal form is inconvenient; but it's lightning fast and offers great concurrency opportunities. You might be writing a web site; but I am not. Quorum just won't cut it in a real time system; where money, or; God forbid, lives, are on the line.

If the only reason to move to NoSQL is inconvenience; I'll take SQL thanks.

SQL is not slow. Thinking about things in terms of sets and not cursors is pretty much the first step (hash joins/clustered seeks). After that start thinking about how the data gets stored (not using GUIDS as PK, fragmentation). Make sure your relationships are correct (join elimination). Add constraints that tell the server how the data looks (statistics).

If you are getting poor perf out of SQL; have a look on Amazon for some books on the subject.

Don't choose NoSQL because it's less inconvenient - that's just lazy. Choose the best tool for the job.

- **MJ**

posted 13 July 2010
SQL is like your wife.

You quickly fall in love with it; you commit your (working) life to it; over time you may take it for granted and flirt with new and younger forms; but you will always go back to it and forget about the others. Plus, every now and again you'll discover something new about it - or it will completely surprise you - and you'll fall in love with it all over again! Like when you get back from work one day and she gets out the hand-cuffs!

- **Eyepoint**

posted 13 July 2010
CQRS is the way to go :)
http://www.udidahan.com/2009/12/09/clarified-cqrs/

- **NNM**

posted 13 July 2010
There's obviously a big rift between SQL and noSQL users. It seems people pick their side, and stay there no matter what.
Posts like Dominic's really show a lack of practise and bad will: seriously,... Do you really think that's how an accomplished programmer does it? SERIOUSLY?
for every query? I know you can't seriously think that.
It's just bad will and predetermination and overcommitment to an answer.
Sotred Procedures. Single connection string. CLASSES.
Overall, I agree with the original post here, exception (mostly): SQL is very pretty and elegant.

- **patrick**

posted 13 July 2010
A couple of people have noted that Codd didn't exactly approve of Chamberlin et al make a complete hash of Codd's original algebra. Seriously, have you ever looked at the grammar for SQL? I was on a project in 1996 that involved implementing a subset of the SQL language. I was given the job of figuring out what the subset should contain and was appalled to discover that the smallest grammar I could find had over 600 productions in it and some parts of the grammar were over 50 levels deep. If you think you understand SQL, believe me, you don't, nobody understands it, the thing is a monster (rather like COBOL actually).

- **jsc42**

posted 13 July 2010
Dominic is not comparing like with like. He has taken a specific programming language implementation of an interface to SQL (looks like ODBC - not a very pretty interface - but that is not the fault of SQL per se) with all of its language dependent trappings and compared it to the one-line final statement from an ORM. For completeness, he should at least include the class definition for the product class and any equivalents to the connection parameters.

A better comparison would be to show the two as they would appear in an interactive session. I don't know any ORMs but I know of several interactive environments where I can just enter just the single line of SQL (without all of the programming language trappings) on the command line / input box and the results are displayed. Are there ORM front-ends that can do that?

- **Parnassus**

posted 13 July 2010
Please stop thinking, you do the world a disservice... You are so confused on so many levels it's even hard to even start arguing with you.

NoSQL is NOT a consequence of ORM sucking.

And it just goes on all the article with so many misunderstandings, like this phrase "even though the actual problem was the Relational model, not SQL itself"....

- **Rob G**

posted 13 July 2010
@Dominic

No one writes that much ADO.NET code to make a simple SQL call. Most of us have written our own SQL helper libraries and Microsoft even distributed a pretty handy one back in the .NET 1.1 days. Those of us writing non-trivial business applications have complex data models better served by parameterized queries or stored procedures, easily called with such code as:

```
SqlParameter[] params = new SqlParameter[ 1 ];
params[ 0 ] = new SqlParameter( "@ProductID", ProductID );
DataSet product = SQLLib.ExecuteDataset( "dbo.ProductGet", params );
```

It's rare that I find myself plucking out one record at a time for much of anything. A 6- or 8-way join is common for robust applications, and an ORM just doesn't save much in this case.

- **Matt**

posted 13 July 2010
Yea, I call pony too. My own ORM has saved me many many lines of code and sure it's slower (in some cases) but my time costs more than a faster server does. It supports a few things like ownership, identifiers and relationships which cover just about every feasible use case. I say slower in some cases because it supports hashing and caching so manages to prevent most database visits from happening.

If I need something complex I've got means of getting at the objects with SQL but I have to say that in a heavily functional 60,000 line project I've needed to use it just once.

- **Karen Lopez**

posted 13 July 2010
One of the other benefits of good relational design is normalization for data integrity reasons. One fact persisted in one place. No update anomalies. Of course, data integrity is a trade off for performance. If performance is more important that data quality, then other design approaches may come out better in the ROI analysis. However, for applications where data integrity is paramount, then performance has to defer to good data quality.

- **Carol**

posted 13 July 2010
I have to agree with a previous post you need to stop before you hurt yourself. I was expecting a thought out article on why SQL is here to stay no matter what the naysayers think and then you go into this thing about the non SQL solutions. So what's your point of the article?

No one has addressed why SQL is really awful, that's because it's not a technological issue it's a personal issue. SQL is simple - deceptively simple. You think just because you know a little something like creating a Select statement, you are empowered with the knowledge of building databases and that is so not true.

The reason people moved away from the SQL Relational models isn't because there's a better mouse trap to be built, or they're trying to fix a technology problem. They're trying to fix their own self created problems and when they did SQL kicked their butt, made them feel stupid because it's so simple, so therefore "if I can't understand it so therefore it's bad."

So far I've observed these NoSQL solutions only go so far and then like that one person who related SQL to being your wife (which is an awesome analogy), you'll come back.

- **TriSys**

posted 13 July 2010
We love SQL and hate Microsoft for dropping DataSet support in ADO.NET and forcing RIA WCF ORM and LINQ on us. Just because the new kids on the block lack real world experience with SQL and RDBMS', the more experienced programme managers should know better and give us tried and tested tools to work with.

- **VampireChicken**

posted 13 July 2010
You've also missed the reason that RDBMSes became popular. Prior to RDBM, the data store model in vogue was to model e.g. a contract as a record containing a header, and N line items.

If your contract had >N line items, you either changed how you did business, or had a huge reconfiguration effort to allow for more line items in a record. Then you changed all of your code to deal with the fact that the data model had changed.

RDBMS made the line item storage independent from the contract header storage. This allowed for the data store to function with minimally invasive changes as the business domain shifted (as they always do).

Flash forward to 2010 (That's about 25 years for those of you keeping score) and you have servers of such speeds and capacities that we can return to the de-normalized data models of old, without needing to maintain fixed-width records, and without runtime penalties.

These new non-relational data stores are no longer "ahead" of the hardware and I think it's time to look at them and seehow they fit into out toolboxes.

- **DaveSF**

posted 13 July 2010
The alternate store movement would be more aptly named NoBtree than NoSql.

Databases which largly fit in memory are punished by read-optimized structures like the b-tree. Full-text indicies several orders of magnitude more unique keys per document addition than a typical RDBMS schema, and their performance on btrees is abysmal. As a result, specialized tools such as Lucene have abandoned btrees in favor of their own write-optimized datastructures. However, these write-optimized benefits can be useful for RDBMS as well. Way back in 1976 D. Severance and G. Lohman showed us why we should be moving to write-optimized datastructures in "Differential files: their application to the maintenance of large databases", and coining of the term Log-Structured-Merge-Tree.

http://portal.acm.org/citation.cfm?
id=320484&dl=GUIDE&coll=GUIDE&CFID=94743458&CFTOKEN=53668684

SQL is not going anywhere, it's the btree whose days are numbered.

- **Kevin Hazzard**

posted 13 July 2010
Ideas that develop organically typically end up with all of the knots, burls & twists that you expect in an any life form. SQL, and our interactions with it, are no different. A shame, really, that the ENIAC developers at the University of Pittsburgh didn't have access to scads of memory in 1948. But they didn't. If they had unlimited memory, E.F. Codd would have gone on to some other greatness I suppose. But the relational database would never have existed. ORMs are just trying to close that gap, albeit in rather poor style. I prefer Ted Neward's characterization of ORMs as "the Vietnam of computer science." Microsoft LINQ (Language Integrated Query) goes in the other direction, moving (some of) the relational algebra into the application, where it would have evolved if the ENIAC folks had enough memory. That's good evolution, IMO.

- **Brian Gladish**

posted 13 July 2010
VampireChicken doesn't quite have it right. Back in the 60s and 70s we had hierarchical and

network databases that allowed any number of detail records for a master. The problem was that they both had to be navigated programmatically to get answers and had links that were physical addresses rather than data (I did a hierarchical project using Burroughs's DMS II, and we put all of the data at the top level, using none of the hierarchical features, to facilitate easy querying). Codd's relational breakthrough meant that the data could be navigated by a processor external to the programming language and was linked by its own content. Unfortunately, he and Chris Date were unable to make SQL truly relational, but we certainly are able to answer questions more quickly now if the data contains the answers.

- **Bruce**

posted 13 July 2010
Don't cut down Dominic. My boss has required me to code in this non-ORM way for years, and has finally relented and allowed me to think about putting a row in an object, so that I don't have to repeatedly pass 20 to 50 arguments between a front-end, back-end and stored procs, all replicated for each of create, retrieve, update. And why does he require it that way? Because that's what the only complete working code samples he found told him how to structure an application, and it matched the boiler plate code he wrote to generate all of this mess.

- **curtisjennings**

posted 13 July 2010
Hey Seldo -

The best part of you article is when you drop the opinions and talk about what works for you.

I want to hear more about HDFS, about how you use your data and about when you don't use SQL databases.

Frankly your opinions about ORM and the attempted link-bate distract from your point.

I would also be very interested in what it is that you are doing that is unique to your situation and your startup - as well as your choices.

When you move away from facts and observations that are based on your experience, your article becomes fluffy and you end up appearing to be narrow minded.

Instead of ragging on ORM's which obviously have nothing to do with your work and the kind of development you do.

In conclusion - we are all served by your article and by your attention to SQL and noSQL - but stay focused on the facts and the objective lessons of your current experience please.

We have enough opinions floating around about things that have about as much bearing on reality as, as you mentioned earlier - morality to sexuality.

peace

- **codexena**

posted 13 July 2010
I agree use the right tool for the right application to address the right business need. There are purest who think the world can be solved with a database and others who think such and such data store is the way to go. However, each of these viewpoints have their limitations. I agree with the author that use the right approach for the right problem.

- **Dhananjay Nene**

posted 13 July 2010
The birth of ORM lies in the fact that SQL is ugly and intimidating (because relational algebra is pretty hard, and very different to most other types of programming).

The birth of ORM wasn't because of SQL. It was because people were looking for a way to store objects. And OODBMSs weren't yet measuring up then. The ugliness and intimidating nature as you suggest (assuming thats true) had nothing to do with ORM.

ORM is dumb, and people noticed. So clever programmers looked at this ridiculous edifice and realized the real problem: the data store and the use-case were mismatched.

Not quite. Let us not rewrite history please. People wanted to deal with objects. The relational algebra and databases didn't lend themselves to work at the same level of storage or abstraction as seemed most natural for objects. Clever programmers realised that the data store and the use-case were mismatched. That was the genesis of ORMs.

This is obviously silly. You've stored your data in a way that doesn't match your primary use-case, accessible via a language that you are not willing to learn. Your solution is to keep the store and the language and just wrap them in abstraction? Maybe you'd do that if your data were in a legacy system and you needed to write a new front-end, but people slap ORM on new projects. Why the hell would you do that?

The primary use case leading to the birth of ORMs was wanting to deal with objects as objects - not as tables. The difficulty was that the OODBMSs weren't quite upto the mark. ORM allows objects to be leveraged as objects and the sheer sophistication of relational storage engines to continued to be leveraged even as it takes upon the entire task of attempting to bridge the two and iron out as much quirkiness out of this situation as possible. This driving imperative can still apply to new projects. Thats why ORM are candidates for new projects as well.

My objective here is not to contest the appropriateness, usefulness, efficacy or efficiency of ORMs. Thats beyond the scope of this post. However you paint ORMs in a very different light than the way I saw them through that entire period. A light that doesn't actually do justice to the history of ORMs.

- **Dhananjay Nene**

posted 13 July 2010
Oops .. the blockquotes in my earlier comment got stripped off. Paragraphs 1, 3, and 5 in my earlier comment are actually quotes from this post and were intended to be in blockquotes.

- **pkw**

posted 13 July 2010
I use 3 ORMs: Hibernate, Propel and Doctrine, that are all different and they all access the same Oracle DB. Much of the time I need to write queries that have multiple JOINs or access the same table more than once or use an Oracle specific feature like CONNECT BY. Some of these ORMs can't get that right or take huge amounts of hand holding to do it.

So I usually get the PDO or JDBC connection from the ORM and write my own SQL for these nasty cases and dump the results in a list or map. When the problem is simple and an object is the object, I use the ORM.

Use the right tool for the job, ORM is a power tool, SQL is a hand tool:

I have several power saws including: a table saw, a rip saw and a compound mitre saw for cutting wood. But I also have a couple of dovetail saws and a dozen or so chisels for the times when the power tools just don't cut it (or make a dog's breakfast of it).

Some day we programmers will learn from the craftsmen what has been common practice for centuries in the more mature crafts.

- **Tim**

posted 14 July 2010
I agree with pkw, mostly, on this and really enjoyed the article. It's important to always use the right tool for the job no matter what you do. Though I always find it useful to have a screwdriver, hammer and duct tape sitting beside me (not to mention a few bandaids).

As far as the article, it's important that some programmers learn there are other options than just ORM's, or straight SQL. Even an ardent Microsoft person I've started dabbling in the OMADS realm (love the acronym by the way). Not because I had to, just because it might be fun to learn something new. Not that I have a use case for it at the moment, but you just never know when a good chance to use will pop up.

Thanks for the good read.

- **Bill Drissel**

posted 14 July 2010
The big deal about relational and SQL is that the designer didn't have to anticipate ALL of the queries that might be mounted against a DBS and build links and chains and paths. Or take months to re-build the DB to answer an ad hoc query.

Regards,
Bill Drissel

- **Richard**

posted 20 July 2010
As a NoSQL fan myself I wouldn't mind one bit if RDMS were indeed marketed as ad hoc

query engines.

- **linlin**

posted 21 July 2010
Links 519 Danni
Love London, Get the Links of London. links of london
London is an amazing city for me, Big Ben, Buckingham House, London Eyes, Red bus, and so on. I had an unforgettable trip in London Last summer. But I met some problem, when I planed to by some souvenir. The souvenir in London is too common which is the same as souvenir in other place, and they also links of london sale were not made in England. Actually, I want some souvenir that is fashion and unique in London, but not is a common cup with a “London” logo, because I did want to buy some extraordinary thing for my girlfriend.
When I wandered in the shopping mall Sloane Square in London, One Classical English style broad attracted me. I was Links of London. I felt loving this brand suddenly. I went into the shop, and found the items links london inside were fantastic and tasty. I thought it was the perfect place to choose a souvenir for my girlfriend. I bought a Flutter & Wow Silver & 18ct Gold Bracelet for my girlfriend. She was very happy. She said it was exactly what she wanted from links of london friendship bracelet London, Fashion, elegant and luxurious. I was also happy with my Links of London choice.
There is a introduction of Links of London, which shows how suitable it can be a souvenir gift to you love.
Links of London design interesting valuable watches, jewelry, tableware and fine, all in sterling silver and 18k gold production, together with assessment of Goldsmiths Hall in London or the Edinburgh Office of the links of london bracelets exclusive mark. Links of London in 1990 was by the creation of Annoushka Ducas and John Ayton, the light that is known for its innovative and affordable gifts and jewelry to establish high-level fame. Brand has developed rapidly over the past decade, with 38 branches and network of senior retail counters,

- **jenny**

posted 28 July 2010
Earning money has online never been this easy and transparent. You would find great tips on how to make that dream amount every month. So go ahead and click here for more details and open floodgates to your online income. All the best.

- **Blessed Geek**

posted 01 August 2010
NoSQL is suitable for those who've had years' experience using SQL. Above that, years' experience is insufficient - one must have had faced the issues of large data quantity, mutating business processes and most of all a distributed user base requiring fast data delivery and synchronisation.

NoSQL is for those who have a clear understanding of the business process and almost every possibility how the dimensionality and primary components of those processes could mutate.

NoSQL is for those with so much SQL experience that they could visualise the candidate data schema and run through their heads all the possible SQL queries and translate those queries into Entity-Relationships and their corresponding objects.

NoSQL is for those who is able to visualise various candidate schemata and realise that using SQL is a hinderance to the progress and mutation of their applications.

SQL is for those unable to visualise schemata and require SQL as an adhoc relationship-discovery tool. Once they sink into SQL they get stuck like stuck in mud or quicksand unable to pull themselves out and unable to express their understanding of data models without the use of SQL.

SQL is a mathematical language and those who understand it do not need to use it - unless, of course, compelled by their employer or contracting client.

- **Dominique De Vito**

posted 17 August 2010
I agree with you: "So go forth, use your OMADS, keep an RDBMS in your back pocket, and stop being so mean to poor old SQL."

That's the reason I wrote: "My own response is: those NoSQL databases could be seen as the database for the middle tiers!"
See http://www.jroller.com/dmdevito/entry/thinking_about_nosql_databases_classification