Why all the Active Record hate? [closed]

Asked 16 years, 4 months ago Modified 13 years, 4 months ago

Viewed 21k times

101

As I learn more and more about OOP, and start to implement various design patterns, I keep coming back to cases where people are hating on Active Record.

Often, people say that it doesn't scale well (citing Twitter as their prime example) -- but nobody actually explains ***why*** it doesn't scale well; and / or how to achieve the pros of AR without the cons (via a similar but different pattern?)

Hopefully this won't turn into a holy war about design patterns -- all I want to know is ****specifically**** what's wrong with Active Record.

If it doesn't scale well, why not?

What other problems does it have?

ruby-on-rails design-patterns oop activerecord

Share

Improve this question

Follow

edited May 21, 2009 at 13:48

asked Aug 11, 2008 at 15:30

Adam Tuttle

19.8k ● 17 ● 83 ● 116

---

10 I guess in general a lot of hate and dislike against design patterns are connected to wrong use. People tend to overuse and use them in wrong context and end up with a more complex solution than the original – terjetyl Nov 27, 2009 at 9:48

1 Ruby's Active Record implementation is more like a ORM. – Jimmy T. Dec 8, 2014 at 17:31

1 There is a social phenomenon which is in order to get appreciation, more recognition, seem smarter and cutting-bleeding-edger, people tend to do repeat mechanically any hype of negation of any current standard, model, widely adopted techs, confusing it with the revolutionary progress to the next wave. – Andre Figueiredo Jan 27, 2017 at 15:48

14 Answers

Sorted by: Highest score (default)

91

There's ActiveRecord the Design Pattern and ActiveRecord the Rails ORM Library, and there's also a ton of knock-offs for .NET, and other languages.

These are all different things. They mostly follow that design pattern, but extend and modify it in many different ways, so before anyone says "ActiveRecord Sucks" it needs to be qualified by saying "which ActiveRecord, there's heaps?"

I'm only familiar with Rails' ActiveRecord, I'll try address all the complaints which have been raised in context of using it.

> @BlaM
>
> The problem that I see with Active Records is, that it's always just about one table

Code:

```
class Person
    belongs_to :company
end
people = Person.find(:all, :include => :company )
```

This generates SQL with `LEFT JOIN companies on companies.id = person.company_id`, and automatically generates associated Company objects so you can do `people.first.company` and it doesn't need to hit the database because the data is already present.

> @pix0r
>
> The inherent problem with Active Record is that database queries are automatically generated and executed to populate objects and modify database records

Code:

```
person = Person.find_by_sql("giant complicated sql que
```

This is discouraged as it's ugly, but for the cases where you just plain and simply need to write raw SQL, it's easily done.

> @Tim Sullivan
>
> ...and you select several instances of the model, you're basically doing a "select * from ..."

Code:

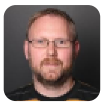
```
people = Person.find(:all, :select=>'name, id')
```

This will only select the name and ID columns from the database, all the other 'attributes' in the mapped objects will just be nil, unless you manually reload that object, and so on.

Share Improve this answer Follow

edited Aug 11, 2008 at 21:11

answered Aug 11, 2008 at 21:02



Orion Edwards
123k ● 66 ● 245 ● 339

Mighty! I didn't know about that specific feature. Yet another pro-AR argument to me to put into my arsenal. – Tim Sullivan Aug 13, 2008 at 17:53

Joining goes beyond the Active Record pattern. – Jimmy T. Dec 8, 2014 at 17:32

"Person.find_by_sql" is not Active Record pattern at all. Its pretty much "Active record" failed me so i need to patch it manually. – magallanes Jun 17, 2016 at 15:12

56

I have always found that ActiveRecord is good for quick CRUD-based applications where the Model is relatively flat (as in, not a lot of class hierarchies). However, for applications with complex OO hierarchies, a DataMapper is probably a better solution. While ActiveRecord assumes a 1:1 ratio between your tables and your data objects, that kind of relationship gets unwieldy with more complex domains. In his book on patterns, Martin Fowler points out that ActiveRecord tends to break down under conditions where your Model is fairly complex, and suggests a DataMapper as the alternative.

I have found this to be true in practice. In cases, where you have a lot inheritance in your domain, it is harder to map inheritance to your RDBMS than it is to map associations or composition.

The way I do it is to have "domain" objects that are accessed by your controllers via these DataMapper (or "service layer") classes. These do not directly mirror the database, but act as your OO representation for some real-world object. Say you have a User class in your domain, and need to have references to, or collections of other objects, already loaded when you retrieve that User object. The data may be coming from many different tables, and an ActiveRecord pattern can make it really hard.

Instead of loading the User object directly and accessing data using an ActiveRecord style API, your controller code retrieves a User object by calling the API of the UserMapper.getUser() method, for instance. It is that mapper that is responsible for loading any associated objects from their respective tables and returning the completed User "domain" object to the caller.

Essentially, you are just adding another layer of abstraction to make the code more managable. Whether your DataMapper classes contain raw custom SQL, or calls to a data abstraction layer API, or even access an ActiveRecord pattern themselves, doesn't really matter to the controller code that is receiving a nice, populated User object.

Anyway, that's how I do it.

Share Improve this answer
Follow

answered Aug 26, 2008 at 18:10

Sam McAfee
10.1k ● 16 ● 62 ● 65

16 @JoãoBragança - perhaps rather than a sarcastic comment, you could actually explain the difficulties that occur when one's data is sharded - so the rest of us can learn something :) – Taryn East Nov 22, 2011 at 9:30

11

I think there is a likely a very different set of reasons between why people are "hating" on ActiveRecord and what is "wrong" with it.

On the hating issue, there is a lot of venom towards anything Rails related. As far as what is wrong with it, it is likely that it is like all technology and there are situations where it is a good choice and situations where there are better choices. The situation where you don't get to take advantage of most of the features of Rails ActiveRecord, in my experience, is where the database is badly structured. If you are accessing data without primary keys, with things that violate first normal form, where there are lots of stored procedures required to access the data, you are better off using something that is more of just a SQL wrapper. If your database is relatively well structured, ActiveRecord lets you take advantage of that.

To add to the theme of replying to commenters who say things are hard in ActiveRecord with a code snippet rejoinder

> @Sam McAfee Say you have a User class in your domain, and need to have references to, or collections of other objects, already loaded when you retrieve that User object. The data may be coming from many different tables, and an ActiveRecord pattern can make it really hard.

```
user = User.find(id, :include => ["posts", "comments"]
first_post = user.posts.first
first_comment = user.comments.first
```

By using the include option, ActiveRecord lets you override the default lazy-loading behavior.

Share Improve this answer Follow

answered Sep 22, 2008 at 21:35

MattMcKnight
8,260 ● 31 ● 35

9

My long and late answer, not even complete, but a good explanation WHY I hate this pattern, opinions and even some emotions:

1) short version: Active Record creates a "**thin layer**" of "**strong binding**" between the database and the application code. Which solves no logical, no whatever-problems, no problems at all. IMHO it does not provide ANY VALUE, except some **syntactic sugar** for the programmer (which may then use an "object syntax" to access some data, that exists in a relational database). The effort to create some comfort for the programmers

should (IMHO...) better be invested in low level database access tools, e.g. some variations of simple, easy, plain `hash_map get_record( string id_value, string table_name, string id_column_name="id" )` and similar methods (of course, the concepts and elegance greatly varies with the language used).

2) long version: In any database-driven projects where I had the "conceptual control" of things, I avoided AR, and it was good. I usually build a **layered architecture** (you sooner or later do divide your software in layers, at least in medium- to large-sized projects):

A1) the database itself, tables, relations, even some logic if the DBMS allows it (MySQL is also grown-up now)

A2) very often, there is more than a data store: file system (blobs in database are not always a good decision...), legacy systems (imagine yourself "how" they will be accessed, many varieties possible.. but thats not the point...)

B) database access layer (at this level, tool methods, helpers to easily access the data in the database are very welcome, but AR does not provide any value here, except some syntactic sugar)

C) application objects layer: "application objects" sometimes are simple rows of a table in the database, but most times they are **compound** objects anyway, and have some higher logic attached, so investing time in AR objects at this level is just plainly useless, a waste of precious coders time, because the "real value", the "higher logic" of those objects needs to be implemented on top of the AR objects, anyway - with and without AR! And, for example, why would you want to have an abstraction of "Log entry objects"? App logic code writes them, but should that have the ability to update or delete them? sounds silly, and `App::Log("I am a log message")` is some magnitudes easier to use than `le=new LogEntry(); le.time=now(); le.text="I am a log message"; le.Insert();`. And for example: using a "Log entry object" in the log view in your application will work for 100, 1000 or even 10000 log lines, but sooner or later you will have to optimize - and I bet in most cases, you will just use that small beautiful SQL SELECT statement in your app logic (which totally breaks the AR idea..), instead of wrapping that small statement in rigid fixed AR idea frames with lots of code wrapping and hiding it. The time you wasted with writing and/or building AR code could have been invested in a much more clever interface for reading lists of log-entries (many, many ways, the sky is the limit). Coders should **dare to invent new abstractions** to realize their application logic that fit the intended application, and **not stupidly re-implement silly patterns**, that sound good on first sight!

D) the application logic - implements the logic of interacting objects and creating, deleting and listing(!) of application logic objects (NO, those tasks should rarely be anchored in the application logic objects itself: does the sheet of paper on your desk tell you the names and locations of all other sheets in your office? forget "static" methods for listing objects, thats silly, a bad compromise created to make the human way of thinking fit into [some-not-all-AR-framework-like-]AR thinking)

E) the user interface - well, what I will write in the following lines is very, very, very subjective, but in my experience, projects that built on AR often neglected the UI part of an application - time was wasted on creation obscure abstractions. In the end such applications wasted a lot of coders time and feel like applications from coders for coders, tech-inclined inside and outside. The coders feel good (hard work finally done, everything finished and correct, according to the concept on paper...), and the customers "just have to learn that it needs to be like that", because thats "professional".. ok, sorry, I digress ;-)

Well, admittedly, this all is subjective, but its my experience (Ruby on Rails excluded, it may be different, and I have zero practical experience with that approach).

In paid projects, I often heard the demand to start with creating some "active record" objects as a building block for the higher level application logic. In my experience, this **conspicuously often** was some kind of excuse for that the customer (a software dev company in most cases) did not have a good concept, a big view, an overview of what the product should finally be. Those customers think in rigid frames ("in the project ten years ago it worked well.."), they may flesh out entities, they may define entities relations, they may break down data relations and define basic application logic, but then they stop and hand it over to you, and think thats all you need... they often lack a complete concept of application logic, user interface, usability and so on and so on... they lack the big view and they lack love for the details, and they want you to follow that AR way of things, because.. well, why, it worked in that project years ago, it keeps people busy and silent? I don't know. But the "details" separate the men from the boys, or .. how was the original advertisement slogan ? ;-)

After many years (ten years of active development experience), whenever a customer mentions an "active record pattern", my alarm bell rings. I learned to try to get them **back to that essential conceptional phase**, let them think twice, try them to show their conceptional weaknesses or just avoid them at all if they are undiscerning (in the end, you know, a customer that does not yet know what it wants, maybe even thinks it knows but doesn't, or tries to externalize concept work to ME for free, costs me many precious hours, days, weeks and months of my time, live is too short ...).

So, finally: THIS ALL is why I hate that silly "active record pattern", and I do and will avoid it whenever possible.

EDIT: I would even call this a No-Pattern. It does not solve any problem (patterns are not meant to create syntactic sugar). It creates many problems: the root of all its problems (mentioned in many answers here..) is, that **it just hides** the good old well-developed and powerful

SQL behind an interface that is by the patterns definition extremely limited.

This pattern replaces flexibility with syntactic sugar!

Think about it, which problem does AR solve for you?

Share Improve this answer Follow

edited Nov 27, 2009 at 9:28

answered Nov 27, 2009 at 9:02

Frunsi

7,148 ● 5 ● 38 ● 42

1 It is a data source architectural pattern. Perhaps you should read Fowler's Patterns of Enterprise Application Architecture? I had similar thoughts to yours prior to actually using the pattern/ORM and finding how much it simplified things. – MattMcKnight Dec 16, 2009 at 4:49

1 I share your feelings. I smell something wrong when a framework does not support compound keys.... I avoided any kind of ORM before SQLAlchemy, and we often use it at a lower level, as a SQL generator. It implements Data Mapper and is very flexible. – Marco Mariani Jun 11, 2010 at 10:07

1 Since two days I am involved in a project that uses "state-of-the-art" ORM, maybe the implementations are matured now (in comparison to what i worked with some years ago). Maybe, my mind will change, we'll see in three months :-) – Frunsi Jun 11, 2010 at 12:58

2 The project is done, and you know what? ORM still sucks, I wasted so much time with mapping problems that are easily expressed in a relational way to a bunch of "object-oriented code". Well, of course the ORM provided ways to express

queries in a kind of OOP+SQL-Mix - of course in an OOP-like syntax - but that just took more time than simply writing an SQL query. The abstraction leaked, the "OOPSQLExperiment" on top of OOP - to allow users to write SQL in OOP syntax was the worst idea ever. No, never again. – Frunsi Oct 15, 2010 at 1:25

1 I wrote raw SQL for everything for many years. Rails AR frustrates me sometimes and for passive querying I nearly agree with you but this is what it solves: 1) Makes it suitably hard to save data which fails validation. 2) Tracking what changed in memory since last persist. 3) Using point 2 to write sensible `before_save` callbacks to maintain consistency within the record 4) `after_commit` hooks for external service triggers. 5) A good DSL for organising DDL changes into changesets (migrations). (There's still pain there but having no pattern is worse when > 1 developer.) – Adamantish Mar 16, 2017 at 19:10

6

Some messages are getting me confused. Some answers are going to "ORM" vs "SQL" or something like that.

The fact is that AR is just a simplification programming pattern where you take advantage of your domain objects to write there database access code.

These objects usually have business attributes (properties of the bean) and some behaviour (methods that usually work on these properties).

The AR just says "add some methods to these domain objects" to database related tasks.

And I have to say, from my opinion and experience, that I do not like the pattern.

At first sight it can sound pretty good. Some modern Java tools like Spring Roo uses this pattern.

For me, the real problem is just with OOP concern. AR pattern forces you in some way to add a dependency from your object to infraestructure objects. These infraestructure objects let the domain object to query the database through the methods suggested by AR.

I have always said that two layers are key to the success of a project. The service layer (where the bussiness logic resides or can be exported through some kind of remoting technology, as Web Services, for example) and the domain layer. In my opinion, if we add some dependencies (not really needed) to the domain layer objects for resolving the AR pattern, our domain objects will be harder to share with other layers or (rare) external applications.

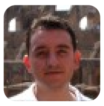
Spring Roo implementation of AR is interesting, because it does not rely on the object itself, but in some AspectJ files. But if later you do not want to work with Roo and have to refactor the project, the AR methods will be implemented directly in your domain objects.

Another point of view. Imagine we do not use a Relational Database to store our objects. Imagine the application stores our domain objects in a NoSQL Database or just in XML files, for example. Would we implement the methods that do these tasks in our domain objects? I do not think so (for example, in the case of XM, we would add XML related dependencies to our domain objects...Truly sad I think). Why then do we have to implement the relational DB methods in the domain objects, as the Ar pattern says?

To sum up, the AR pattern can sound simpler and good for small and simple applications. But, when we have complex and large apps, I think the classical layered architecture is a better approach.

Share Improve this answer Follow

answered Aug 25, 2011 at 9:42



Juanjo
61 ● 1 ● 1

Welcome to SO. Appreciated your comment but this question was closed as not constructive by NullUserException on Dec 17 '11 at 1:17 – Tony Rad Nov 13, 2012 at 17:17

3

> The question is about the Active Record design pattern. Not an orm Tool.

The original question is tagged with rails and refers to Twitter which is built in Ruby on Rails. The ActiveRecord framework within Rails is an implementation of Fowler's Active Record design pattern.

Share Improve this answer

answered Aug 26, 2008 at 19:19

Follow

John Topley
115k ● 47 ● 199 ● 240

2

The main thing that I've seen with regards to complaints about Active Record is that when you create a model around a table, and you select several instances of the model, you're basically doing a "select * from ...". This is fine for editing a record or displaying a record, but if you want to, say, display a list of the cities for all the contacts in your database, you could do "select City from ..." and only get the cities. Doing this with Active Record would require that you're selecting all the columns, but only using City.

Of course, varying implementations will handle this differently. Nevertheless, it's one issue.

Now, you can get around this by creating a new model for the specific thing you're trying to do, but some people would argue that it's more effort than the benefit.

Me, I dig Active Record. :-)

HTH

Share Improve this answer

Follow

answered Aug 11, 2008 at 16:22

Tim Sullivan
16.9k ● 12 ● 75 ● 120

3 "Doing this with Active Record would require that you're selecting all the columns, but only using City." It's actually

extremely easy to specify a select clause. – MattMcKnight Dec 16, 2009 at 4:43

2

Although all the other comments regarding SQL optimization are certainly valid, my main complaint with the active record pattern is that it usually leads to impedance mismatch. I like keeping my domain clean and properly encapsulated, which the active record pattern usually destroys all hope of doing.

Share Improve this answer Follow

answered Aug 26, 2008 at 18:31

Kevin Pang
41.4k ● 38 ● 122 ● 173

ActiveRecord actually *solves* the impedance mismatch problem by letting you code in a OO fashion against a relational schema. – Mauricio Scheffer Feb 4, 2009 at 13:44

Care to elaborate? General consensus is that objects modeled after a relational database are, by definition, not object oriented (since relational databases don't revolve around OO concepts such as inheritance and polymorphism). – Kevin Pang Feb 4, 2009 at 18:36

There are three known ways to map inheritance to a relational schema. Ref: castleproject.org/ActiveRecord/documentation/trunk/usersguide/… – Mauricio Scheffer Feb 4, 2009 at 18:47

I think you're mistaking the Castle Active Record OSS project for Active Record the design pattern. The original question (and my response) are referring to the design pattern. The Castle Active Record project has things baked into it to help with OO development, but the pattern itself does not.
– Kevin Pang Feb 4, 2009 at 19:15

I was just quoting Castle as reference. RoR's ActiveRecord implements Single table inheritance only (martinfowler.com/eaaCatalog/singleTableInheritance.html), but the other strategies are being considered (blog.zerosum.org/2007/2/16/…) – Mauricio Scheffer Feb 4, 2009 at 19:53

1

I love the way SubSonic does the one column only thing. Either

```
DataBaseTable.GetList(DataBaseTable.Columns.ColumnYouW
```

, or:

```
Query q = DataBaseTable.CreateQuery()
               .WHERE(DataBaseTable.Columns.ColumnToFi
q.SelectList = DataBaseTable.Columns.ColumnYouWant;
q.Load();
```

But Linq is still king when it comes to lazy loading.

Share Improve this answer Follow

answered Aug 11, 2008 at 19:50

Lars Mæhlum
6,102 ● 3 ● 29 ● 32

1

@BlaM: Sometimes I justed implemented an active record for a result of a join. Doesn't always have to be the relation Table <--> Active Record. Why not "Result of a Join statement" <--> Active Record ?

Share Improve this answer Follow

answered Aug 11, 2008 at 21:14

Johannes
2,992 ● 4 ● 33 ● 36

1

I'm going to talk about Active Record as a design pattern, I haven't seen ROR.

Some developers hate Active Record, because they read smart books about writing clean and neat code, and these books states that active record violates single resposobility principle, violates DDD rule that domain object should be persistant ignorant, and many other rules from these kind of books.

The second thing domain objects in Active Record tend to be 1-to-1 with database, that may be considered a limitation in some kind of systems (n-tier mostly).

Thats just abstract things, i haven't seen ruby on rails actual implementation of this pattern.

Share Improve this answer Follow

answered Jun 13, 2011 at 7:26

Alex Burtsev
12.6k ● 8 ● 65 ● 89

0

The problem that I see with Active Records is, that it's always just about **one** table. That's okay, as long as you really work with just that one table, but when you work with data in most cases you'll have some kind of join somewhere.

Yes, **join** usually is worse than **no join at all** when it comes to performance, but **join** *usually* is better than **"fake" join** by first reading the whole table A and then using the gained information to read and filter table B.

Share Improve this answer Follow

edited Aug 11, 2008 at 17:21

answered Aug 11, 2008 at 16:46

BlaM
28.8k ● 33 ● 93 ● 106

@BlaM: You're absolutely right. Although I've never used Active Record, I have used other bolted-on ORM systems (particularly NHibernate), and there are two big complaints I have: silly ways to create objects (ie, .hbm.xml files, each of which get compiled into their own assembly) and the performance hit incurred just loading objects (NHibernate can spike a single-core proc for several seconds executing a query that loads nothing at all, when an equivalent SQL query takes almost no processing). Not specific to Active Record of course, but I find most ORM systems (and ORM-like systems) seem to – TheSmurf Aug 11, 2008 at 17:19

There are many alternatives to using hbm.xml files. See for example NHibernate.Mapping.Attributes and fluent-nhibernate. – Mauricio Scheffer Feb 4, 2009 at 13:41

About object creation performance, I've never run into such perf problems, you might wanna check with a profiler. – Mauricio Scheffer Feb 4, 2009 at 13:42

@mausch: Don't need a profiler. It's a fairly well-known issue. Don't know if it applies to the latest version (which I am not using at my job yet). ayende.com/Blog/archive/2007/10/26/… – TheSmurf Feb 4, 2009 at 14:45

4 Using :joins or :includes in the finds IE Customer.find(:all, :include => :contacts, :conditions => "active = 1") will do an SQL join, not a full table scan of either. – Tilendor Jul 10, 2009 at 1:24

0

The problem with ActiveRecord is that the queries it automatically generates for you can cause performance problems.

You end up doing some unintuitive tricks to optimize the queries that leave you wondering if it would have been more time effective to write the query by hand in the first place.

Share Improve this answer Follow

answered Aug 15, 2008 at 14:32

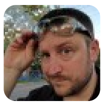
engtech
2,791 ● 3 ● 20 ● 24

0

Try doing a many to many polymorphic relationship. Not so easy. Especially when you aren't using STI.

Share Improve this answer Follow

answered Jul 20, 2009 at 13:39



Alexander Trauzzi
7,387 ● 16 ● 73 ● 123