Magic Scaling Sprinkles

Just another WordPress.com weblog

Why Arel?

with 25 comments

The upcoming version 3 of Ruby on Rails will feature a sexy new querying API from ActiveRecord. Here is an example:

```
User.order('users.id DESC').limit(20).includes(:items)
```

You can see that all queries are now chainable. This has two principal benefits: first, you can programmatically generate complex queries (based on user input or other data); and second, you can refactor your queries to eliminate code duplication in a way that you never could before.

The way that such chaining is possible is that each "query" (e.g., `Users.order(...)`) returns not an array of data, but a query object that one can manipulate still further. Only when you try to iterate through the data is a SQL query composed and then executed on the database.

If this sounds to you similar to "named_scope", you're spot on. After I wrote "named_scope" I immediately asked myself "but what if every query was a named_scope? What if named_scope were the rule and not the exception?". I saw, unfortunately, that in order to do this I would need to re-write ActiveRecord.

So I set out to re-write ActiveRecord. I announced my intentions (and then began a quixotic quest that would ultimately fail, then later succeed thanks to other people's help).

I realized, though, that in order to re-write ActiveRecord, it would be best for the purposes of encapsulation to create a whole new self-contained query language; ideally, one that could underlie both DataMapper and ActiveRecord. So I began work on Arel, an Object-Oriented interpretation of the Relational Algebra. The Relational Algebra is a mathematical model for representing "queries" on data. In many ways it resembles SQL (which is modeled on the algebra) but with slightly different jargon ("project" for "select", "select" for "where", etc.).

But the Relational Algebra differs greatly from SQL in that it has the property of "closure" under composition. That is, every operation on a relation (such as adding a "where" clause (aka "selection")) returns a new relation where one can subsequently add more operations. SQL distinguishes queries from result sets: and so queries cannot be composed since queries are just strings (e.g., `SELECT * FROM foo`). To manipulate a SQL query we must manipulate a string. There is a string algebra, but its operations are things like substring, concatenation, substitution, and so forth–not so useful. In the Relational Algebra, there are no queries per se; everything is either a relation or an operation on a relation. Connect the dots and with the algebra we get something like "everything is named_scope" for free.

Ever since I read the book Structure and Interpretation of Computer Programs I have be

fascinated by the concept of closure under composition. (If you haven't read this book, you must; it's like Euclid's Elements). There is a chapter on a Picture Language that represents transformations on pictures; each transformation generates a picture that can be further transformed:

```
(define wave2 (beside wave (flip-vert wave)))
```

The picture is ugly but the code is beautiful

In other words, the picture language has closure under composition. To this day, when I work in completely different languages and on somewhat different problems I'm fascinated by the same concept of closure. This gist which I will someday re-write into a blog-post illustrates how I often use the Decorator pattern with Factories and Dependency Injection to structure programs. This is closure writ-large, and is a different way to think about the architecture of your software than the usual "Domain Driven"/Ontological approach. (Actually, it has much in common with the "Test Driven" approach).

Anyway. Just before I began working on Arel I learned of SqlAlchemy. SqlAlchemy was inspiring, but it was written in Python (and thus useless for RoR) and also had a couple design decisions that I thought were fundamentally (not superficially) wrong. I intended to do something radical, something SQL Alchemy couldn't do: a pure relational algebra with closure under composition even in the presence of aggregations. To explain my motivations though, I want to illustrate two "beautiful" querying problems; one is solved well in SQL, the other not. The first is the technique to do a group-wise maximum of a certain column. In this example, we try to find the most expensive dealer of a given article. for all articles:

```
SELECT s1.article, s1.dealer, s1.price
FROM shop s1
LEFT JOIN shop s2 ON s1.article = s2.article AND s1.price < s2.price
WHERE s2.article IS NULL;
```

This, to me, is one of the great mind-blowing SQL queries. It means something like "find the dealer for which there exists no dealer selling the same product at a lesser price". It is the "double negative" restatement of "most expensive". In other words, "not the lesser expensive". It's this kind of radical restatement of problems that I hope to accomplish with my work. Of course this actually has nothing to do with Arel, although it's expressible trivially in Arel. But it provides some insight into my thinking about why I write software and why I am concerned so much about how software is written.

The other query I am fascinated by is the following. Suppose we have a users table and a photos table and we want to select all user data and a *count* of the photos they have created. In SQL, it is expressed like thus:

```
SELECT users.*, photos_aggregation.cnt
FROM users
LEFT OUTER JOIN (SELECT user_id, count(*) as cnt FROM photos GROUP BY user_id) AS
photos_aggregation
ON photos_aggregation.user_id = users.id
```

You'll note the use of the derived table in the subselect. This is terrible, in my opinion. Only advanced SQL programmers know how to write this (I've often asked this question in job interviews I've never once seen anybody get it right). And it shouldn't be hard! It's just what happens when you lack closure under composition. I was determined to be able to write code like the following:

```
photo_counts = photos.
group(photos[:user_id]).
project(photos[:user_id], photos[:id].count)
```

And finally,

```
users.join(photo_counts).on(users[:id].eq(photo_counts[:user_id]))
```

If this seems simple to you, you are as naive as I was when I first started working on Arel. It is anything but simple: Arel needs to recognize when you are joining with relations that are aggregations (i.e., have a "group by") and produce a derived table (a subselect with a unique name) and then join with that. Since we have closure under composition, no matter how many joins and how many aggregations you have, Arel needs to generate unique derived tables and scope all attribute names to them. Since derived tables can be nested infinitely the implementation of this is incredibly tricky; one of the hardest problems I've ever solved, and I'm pretty happy how it all turned out.

But therein lies the rub. All of my open-sourced projects share two attributes. First, the source code is extremely stylized and meant to be /read/ more than /used/. They are literary arguments in code much more than a useful piece of infrastructure. If you read the source-code to Arel you will see a highly affected "combinator" and "interpreter" style of coding.

If you look at Screw.Unit , for example, not only is it an argument about how tests should be written but it is probably the most unusual Javascript code you will ever see; it is written in a "concrete" style inspired by Self's Morphic.

Similarly, Arel is a two-fold argument: this is how you should interact with a database, and this is how software can be written to affect a unique personal style.

The second attribute of all my open-source projects is that I release them and neglect them. This makes me a terrible steward of open-source software. But my aim is to introduce ideas into the world more so than to keep infrastructure alive. (My upcoming open-source project, FlockDB (a distributed graph database) will hopefully be different; we hope to run Twitter off of the open version indefinitely.)

And this explains why I “finished” Arel in May of 2008 but only now has it been integrated into Rails. If it were not for the heroic integration work of Emilio Tagua, the leadership of Bryan Helmkamp, and the support of DHH and Michael Koziarsky, Arel would be but an idea.





Written by nkallen

January 28, 2010 at 11:35 pm

Posted in Uncategorized

« The Meaning of Information Technology
Why I love everything you hate about Java »

25 Responses

Subscribe to comments with RSS.

1. Arel is one of the most beautiful solutions to the problem of query modelling and SQL generation I’ve ever seen.

   I must admit though I didn’t quite understand it until after a bit of prodding by some really smart programmers, telling me I should look at using it within DataMapper. What I found out was that the principles of relational algebra are quite simple conceptually, but they allow you a way to think about and describe complex queries in an even more straight forward manner than SQL does.

   

   Dan Kubb

   January 29, 2010 at 4:51 am

2. Nick,

   You said that before you started work on Arel, you read about SQLAlchemy, but couldn’t use it because it was written in Python. However, Sequel (http://sequel.rubyforge.org) was around back then (unless you started before March 2007), was written in ruby, and did and does pretty much everything Arel currently does and more. Were you aware of Sequel when you chose to start work on Arel?

   Jeremy

   

   Jeremy Evans

January 29, 2010 at 5:23 am

3. This is really mind-blowing, you rock.

Esdras Mayrink

January 29, 2010 at 7:35 am

4. See Linq:
http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

It doesn't have to be just for SQL, either.

Jeff Perrin

January 29, 2010 at 2:03 pm

5. This is a trully fascinating topic.
I have wrote myself about the design pattern in ruby and love to have a grasp on the architectural big picture of an app/library.

It would be wonderfull if you could give us a few more insights on how arel was designed, a sort of quick guide on how to explore/understand your code before we dive in.

charles sistovaris

January 29, 2010 at 3:10 pm

6. "fundamentally wrong design decisions" is a pretty big claim re: SQLAlchemy. I'm going to posit that any relational API Arel allows can be hypothetically arranged in Python via SQLAlchemy as well, since SQLA can emit any SQL possible and certainly considers every relation operation as constructing a new relation (which was my focus as it is yours).

mike bayer

January 29, 2010 at 3:23 pm

7. Linq was certainly an inspiration and Arel to some degree supports non-sql datastores and allows joining across data sources.

Follow

nkallen

January 29, 2010 at 5:18 pm

8. Sequel did not support (TRANSPARENTLY) joining the same table to itself (needed hints) nor closure when joining with aggretations. I aimed to maintain a perfect illusion where no hints would be necessary; it was all as if you were just working with arrays of data.

nkallen

January 29, 2010 at 5:19 pm

9. SqlAlchemy was amazing and was an inspiration. I don't mean to slight it. I aimed to go further than it could go (at the time, don't know what it's like now). There were places where it was more interested in generating queries than maintaining the illusion you were just working with data. So various hints were necessary, etc. I wanted those hints to disappear, pretend as if you just had sequences of tuples.

nkallen

January 29, 2010 at 5:22 pm

10. that's what I figured. Yes, SQLAlchemy stays close to the semantics of SQL for now. I have always been skeptical of the "just working with data" illusion with regards to simplistic ORMs that hide relational concepts overall, which is why we remain explicit about things like x.join(y) and also the concept of aliasing on a self-referring join (which seems to be something you're making implicit). We introduced some "implicit self-referring join" types of functionality in 0.4 and almost immediately by 0.5 we've walked that back to recommending a more explicit approach again involving alias(). I find that too much magic just gets in the way when it isn't emitting the SQL you're looking for.

mike bayer

January 29, 2010 at 5:30 pm

11. Nick,
I'm extremely interesting in what you mean by "joining across data sources".

I worked in a few Rails projects which had models in different DBs (mainly because of accessing legacy data, and even some times I needed to decorate that and add locally stored info).

Needless to say, SQL queries were almost never optimal, while I could never abstract the query interface correctly.

Thanks,

– nachokb

nachokb

February 3, 2010 at 3:13 am

12. Great work, some time ago I wondered whether Rails was considering switching to SQL generation from a model representing the query ala SQLAlchemy.

    Guess I have my answer

Leon Breedt

February 3, 2010 at 3:41 am

13. Looks like a huge improvement for Rails 3. Looking forward to digging into it.

    One thing I am interested in doing in Rails is to cleanly support prepared statements if the driver supports it (e.g. JDBC under JRuby). I ended up having to hack it in Rails 2.

    Any hints on where to start looking on how to do it in terms of Arel and Rails 3?

    Thanks,
    Gerald

Gerald Boersma

February 4, 2010 at 8:57 pm

14. [...] to a chain of methods performed upon a simpler find. There are lots of good code examples in here! Why Arel? is also a great introduction to this topic and shows the motivation and theory behind the [...]

    Rails 3.0 Beta: 36 Links and Resources To Get You Going

    February 5, 2010 at 11:28 pm

15. Hi Nick,

    Nice post, thank you for writing it! I do have one note on your explanation of the first great mind-blowing SQL query that you present:

```
SELECT s1.article, s1.dealer, s1.price
FROM shop s1
```

Follow

```
LEFT JOIN shop s2 ON s1.article = s2.article AND s1.price < s2.price
WHERE s2.article IS NULL;
```

Since this selects the values from s1, and the join is “s1.price < s2.price WHERE s2.article IS NULL”, I don’t believe the correct formulation is “find the dealer for which there exists no dealer selling the same product at a lesser price”.

I believe the correct formulation is “find the dealer for which there exists no dealer selling the same product at a higher price”.

If we are after the most expensive dealer, then there exists no dealer selling the same article at a higher price. If, instead, there exists no dealer selling the same product at a lesser price, as written in your original post, then the resulting dealer would be the cheapest one.

Vladimir Andrijevik

February 10, 2010 at 10:29 pm

16. Great work on Arel, however there are a couple of points I would like to make:

    The first point is that the SQL you present doesn’t match the spec you give in the original Arel github description here: http://github.com/rails/arel where you say:

    “Now, we’d like to join this with the user table. Naively, you might try to do this:

    SELECT users.*, count(photos.id)
    FROM users
    LEFT OUTER JOIN photos
    ON users.id = photos.user_id
    GROUP BY photos.user_id”

    However the spec you give and the query don’t match, which is why you’re getting unexpected results. Basically the group by should be on users.id rather than photos.user_id If you “correct” this to

    SELECT users.*, count(photos.id)
    FROM users
    LEFT OUTER JOIN photos
    ON users.id = photos.user_id
    GROUP BY users.id

    you get the results you want like this:

    mysql> select users.*, count(photos.id) from users left outer join photos on users.id = photos.user_id group by users.id;
    +—-+——+—————+
    | id | name | count(photos.id) |
    +—-+——+—————+
    | 1 | hai | 3 |

Follow

| 2 | bai | 0 |
| 3 | dumpty | 0 |
+——+———+———+
3 rows in set (0.00 sec)

Of course this does reinforce the point that aggregation queries are awkward!

Secondly, I wondered how Arel deals with the performance implications of nested queries? It's typically harder for an optimiser to work with nested queries and MySQL has particularly poor performance with nested queries. How does Arel deal with this?

GrumpyLittleTed

February 15, 2010 at 12:42 pm

17. My current peeve with Arel is its artificial dependency on ActiveSupport 3, I really wish I could use it in a Rails 2 app, and I am not sure if other ORMs want to depend on ActiveSupport at all anyway.

    Aside from that, love Arel.

Sam

February 15, 2010 at 10:46 pm

18. Interesting work. I'm always watching for ways of allowing developers to deal with relational data without necessarily using SQL or bad SQL generation tools.

    I was looking at the solution to query 2 and wondered why you couldn't do a simple left outer with a group by, avoiding the derived table you don't like? e.g.

    Select u.userid, nullif(count(p.*), 0)
    from users u, photos p
    left outer join on u.userid = p.userid
    group by u.userid

    Should be functionally equivalent, although I only ran it in my head so I could be missing something.

mark

February 16, 2010 at 8:20 pm

19. Well I really like arel, however there are a couple of places where SQL is poking through where IMHO it shouldn't.

Follow

User.order(‘users.id DESC’)

I think User.order(asc(‘last_name’,'first_name’)) would have been been a nicer fit, of course you’d need a SortOrder object. The same thing with conditions,

I guess that the main gist of my thoughts is that arel is easier for a database program to parse than SQL, in the first place so it would be really cool to amend the database so it spoke arel instead of sql:->

application -> arel -> sql -> data

to

application -> arel -> data

with

sql -> arel -> data as an option for manual, human interaction with the database

Richard Nicholas

February 21, 2010 at 5:22 pm

20. That “GROUP BY photos.user_id” is a common source of errors. Postgresql would reject the query because there are raw attributes in the SELECT that are not also in the GROUP BY. As a rule, any attribute in the SELECT that is not used in an aggregate should appear in the GROUP BY. When this is not the case the database must choose an arbitrary tuple as discussed here. That ruins the relation.

    select sum(x), y, z GROUP BY y — Bad
    select sum(x), y, z GROUP BY y, z — Good

    It would be nice if Arel could detect these bad queries or even infer the attributes to GROUP BY from SELECT for simple queries.

Sam Danielson

February 24, 2010 at 3:27 am

21. I believe Sam’s comment may be related to the way in with MySQL allows you to shoot yourself in the foot by using GROUP BY in a way that will produce invalid results.

    I’m firmly in the camp that says “when in doubt, always make sure GROUP BY matches the non-aggregate expressions in the SELECT list”. However, it’s not quite as simple as that. In theory, assuming your DB allows it, you can get correct results by referencing the SELECT columns that are the primary key (or a unique key).

See the following explanation, and pay attention to the section titled “Functional dependencies”, and the sections that follow:

http://dev.mysql.com/tech-resources/articles/debunking-group-by-myths.html

Yes, I realize that most of this article is pro-MySQL, but the information about how GROUP BY relates to the concept of “functional dependencies” is helpful no matter which RDBMS you use. It even comes in handy when thinking about normalization.

Dan Kubb

February 25, 2010 at 10:17 pm

22. [...] however, I couldn’t stop looking at Arel. I really enjoyed reading Nick Kallen’s writeup about why he wrote it, and once you sort of get a handle on the general philosophy underlying [...]

metautonomo.us » Blog Archive » Why (fork) Arel?

May 4, 2010 at 3:08 pm

23. I am a quite skilled SQL user, so maybe it is that but I cannot see how you Arel example is easier than the second SQL example.

Using common table expressions it even looks almost identical to me, but it was close enough before that.

```
WITH photos_aggregation AS (SELECT user_id, count(*) as cnt FROM photos GROUP BY user_id)
SELECT users.*, photos_aggregation.cnt
FROM users
LEFT OUTER JOIN photos_aggregation
ON photos_aggregation.user_id = users.id;
```

The first query on the other hand while correct probably gives terrible performance in most (all?) databases. Since it probably will be planned as O(n^2) when it actually just has to be O(n) with another query based on NOT EXISTS. So I would say the first query is the trickier one.

Disclaimer: I have just tested this in PostgreSQL 8.4.

```
SELECT s1.article, s1.dealer, s1.price
FROM shop s1
WHERE NOT EXISTS (SELECT 1 FROM shop s2 WHERE s1.article = s2.article AND s1.price < s2.price);
```

Andreas

Follow

September 18, 2010 at 10:48 pm

24. Sorry, I was wrong about the planning. PostgreSQL optimized it just fine. I just made a lazy typo which fooled the optimizer. Both ways of writing it was optimized into anti-joins just fine.

Andreas

September 18, 2010 at 10:57 pm

25. As you were working on Arel did you consider how other other projects you worked on the extended ActiveRecord, Cache-Money for example, would integrate into the new design?

Ashley

September 25, 2010 at 9:04 pm

Leave a Reply

Enter your comment here...

Fill in your details below or click an icon to log in:

Email (required) (Not published)

Name (required)

Website

Notify me of follow-up comments via email.

Post Comment

Pages

Follow

- About

Search

search site archives

search

Blogroll

- WordPress.com
- WordPress.org

Archives

- July 2010
- February 2010
- January 2010
- November 2009
- December 2008
- November 2008
- October 2008

Categories

- Uncategorized

Meta

-
- Log in
- Site Feed
- Comments Feed
- Back to top

Blog at WordPress.com. Theme: The Journalist v1.9 by Lucian E. Marin.

Follow