
Henrietta Dombrovskaya
Senior Database Architect
Enova
Chicago IL

hdombrovskaya@enova.com
hdombrovskaya.wordpress.com
About the speaker

• Graduated from the University of Saint Petersburg in 1985 with Applied Math major
• At that time there was no CS major, there was no SE department and there was no AIS department, although most of the people were there
• PhD in CS in 1995
• Worked in different industries and Government bodies, including the City of Chicago Mayors Office, New York Department of Education, Pepsi Americas, Chicago Board of Options…

… what is so special about Enova?
One of the oldest trades..
What is Enova doing?..
What is object-relational impedance mismatch and why it is bad?
Why should we care?

It’s all about application performance!
What contributes to database application performance?

Why to use databases?

Because…

*The DBMS is specialized software designed to manage data in the most efficient way.*

Nevertheless, the most common complaint of application developers is …. 
THE DATABASE IS SLOW

WHY???
Database developer: database is always fast, people just don’t know how to write queries!

Application developer: application is perfect, until it hits a database…
May be, we do not have enough hardware?

Our US master PG database runs on
80 thread processors
2.4GHz
512 Gb RAM – almost completely used by disk cache
1066MHz (responses from RAM are 0.9 ns)
I/O 4Gb/sec with avg response time 3ms
I/O utilization: 40%

Even with the best hardware available we can make it only \textbf{twice} faster
Current cost: \textbf{20K} (commodity)
Next – \textbf{100K} – somewhat faster (non-commodity)
Next - \textbf{1,000K} - twice faster (mainframe)
Now - let’s see, *what* is slow...

Where do we *usually* start to look, if we want to see, why the database performance is bad?...
Longest-running queries (slowest queries)

Most frequently running queries

Queries, which take up the most time (top offenders).
Based on these reports - is a database really slow?

```
SELECT * FROM loans WHERE id=?
```

# executions: 8,500,000  
avg time: <10ms  
**total execution time** about 2.5 hours!
May be, we need this?...

May be, that’s how many times this query should be executed?...
Well...

account home controller - 50,000 times during the day

**some** application controllers: over 1,000 database calls *for each* screen refresh.
How could this possibly be happening?!

Let’s take a step back…
First came a program…

Once upon a time there was a program…

Input → Program → Output

Direct access storage: late 60’s

DBMS emerged as specialized programs for centralized data management
Since then we have…

- Imperative programming languages, which tells, *how* to do things

```
for (i:=1, i++, n) do

...

end;
```

*and*

- Declarative data manipulating languages, which define *what* to do:

```
SELECT first_name, last_name FROM people
WHERE id=101
```
So – is anything wrong with that?...

Both imperative programming languages and declarative query languages work perfectly to accomplish the tasks they were designed to accomplish.

The problems start, when we try to make them work together.
What we have at Enova:

Postgres: - RDBMS
Ruby: object-oriented language.
ActiveRecord - Object Relational Mapping (ORM)

ORM:
– Data structures mapping – yes
– Data sets manipulation – no

ORIM – object-relational impedance mismatch
How ActiveRecord works
What this means for application/database interaction

Due to the lack of awareness of the underlying database interaction on the part of the object methods, one controller performs multiple trips to the database.

For example…
A customer comes to the website…
After (s)he logs in...

### Account History

<table>
<thead>
<tr>
<th>Line of Credit #</th>
<th>Opened Date</th>
<th>Line of Credit Limit</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>35618598</td>
<td>08/18/2016</td>
<td>$1000.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>35220581</td>
<td>01/08/2014</td>
<td>$1650.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>35792230</td>
<td>10/30/2013</td>
<td>$1000.00</td>
<td>Declined</td>
</tr>
<tr>
<td>35726218</td>
<td>10/21/2013</td>
<td>$1000.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>35323358</td>
<td>08/16/2013</td>
<td>$800.00</td>
<td>Paid Off</td>
</tr>
</tbody>
</table>

### Loan History

<table>
<thead>
<tr>
<th>Loan Type</th>
<th>Loan ID</th>
<th>Funding Date</th>
<th>Loan Amount</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payday Loan</td>
<td>34289957</td>
<td>01/30/2013</td>
<td>$250.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>CAB Loan</td>
<td>32826262</td>
<td>06/04/2012</td>
<td>$700.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>CAB Loan</td>
<td>32792390</td>
<td>05/29/2012</td>
<td>$700.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>CAB Loan</td>
<td>32672252</td>
<td>05/07/2012</td>
<td>$700.00</td>
<td>Paid Off</td>
</tr>
<tr>
<td>Installment Loan</td>
<td>32254565</td>
<td>02/13/2012</td>
<td>$300.00</td>
<td>Paid Off</td>
</tr>
</tbody>
</table>
def initialize(customer)
    @customer = customer
    @customer_extra = customer.customer_extra
    @person = customer.person
    @address = customer.person.try(:address)
    @company = customer.person.try(:company)
    @bank_account = customer.bank_account(true)
    @debit_card = customer.debit_card
    @customer_paydate = customer.customer_paydate(true)
    @paydate_schedule =
    customer.customer_paydate.try(:paydate_schedule)
    @customer_source = customer.customer_source
end
SELECT * FROM customers
    WHERE (customers.id = 12470535)

SELECT * FROM people
    WHERE (people.id = 61657007 AND (type = 'CustomerPerson')) AND ((people.type = 'CustomerPerson'));

SELECT addresses.*, people_addresses.serial_number
FROM addresses
    INNER JOIN people_addresses ON addresses.id = people_addresses.address_id
    WHERE (people_addresses.person_id = 61657007
        AND (eff_end_date is NULL));
But wait, there’s more!

SELECT * FROM approvals WHERE (customer_id = 12470535) ORDER BY processed_on desc
LIMIT 1
SELECT * FROM customers WHERE (customers.id = 12470535)
SELECT * FROM loans WHERE (loans.id = 25563928)
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold', 'on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold', 'on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold', 'on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT count(*) AS count_all FROM loans WHERE (loans.customer_id = 12470535 AND (status_cd in ('applied','approved','on_hold','issued','issued_pmtproc') and loan_type_cd = 'installment'))
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold')) ORDER BY funding_date DESC LIMIT 1
SELECT * FROM loans WHERE (customer_id = 12470535 and status_cd in ('applied','approved','on_hold')) ORDER BY funding_date DESC LIMIT 1

2014 Henrietta Dombrovskaya – Enova
What we should see instead...

```
SELECT * 
FROM customers 
WHERE (customers."id" = 12470535)

SELECT * 
FROM people 
WHERE (people."id" = 61657007 
AND (type = 'CustomerPerson')) 
AND ( (people."type" = 'CustomerPerson' ) );
```

```
SELECT addresses.*, 
people_addresses.serial_number 
FROM addresses 
INNER JOIN people_addresses 
ON addresses.id = 
people_addresses.address_id 
WHERE 
(people_addresses.person_id = 61657007 
AND (eff_end_date is NULL));
```
There are some ways to improve

Eager loading:

SELECT customers."id"
    ,customers."created_by"
    ,..<>,customers."person_id"
    ,..<>,customers_extra."estimated_monthly_living_expense"
    ,..<>,people."type"
    ,<...>,addresses."id"<...>
    ,companies."id" <...>
FROM customers
    LEFT OUTER JOIN customers_extra
    ON customers_extra.customer_id = customers.id
    LEFT OUTER JOIN people
    ON people.id = customers.person_id AND people."type" = 'CustomerPerson'
    LEFT OUTER JOIN people_addresses ON people_addresses.person_id = people.id
    LEFT OUTER JOIN addresses ON addresses.id = <...>
WHERE (customers."id" = 17674188)
But then the next method

```ruby
has_many :bank_accounts, :foreign_key => 'person_id'
  do
    def default_for_customer(customer, reload = false)
      @bank_account_cache = {} if @bank_account_cache.nil? || reload
      ...
    end
  end

executes SELECT again!
```
If we continue with existing frameworks...

... this problem (ORIM) will never be solved, and we will continue to loose money on timeouts!
Our Solution: Logic Split Unfolded
Our approach allows:

✓ reduce the number of db calls
  (2-10 instead of 500-900 per view rendering)
✓ optimize queries independently from the app.
How we are going to achieve that?

Making the methods data-aware

Contrary to the standard OO approach?
Yes, but…
This is the only way to improve the App/DB interaction.
The sketch of proposed changes
Logic Split methodology

- Disassemble
- Identify data retrieval
- Construct a single query
- Execute
- Use retrieved data in other steps
Example: Amount_Outstanding

$525.00

{"uncollected_principal"=>{:both=>500.0, :debit=>500.0, :credit=>0.0},
"late_fees_ar"=>{:both=>25.0, :debit=>25.0, :credit=>0.0},
"nsf_fee_income"=>{:both=>0.0, :debit=>25.0, :credit=>25.0},
.....}
Definitions

AccountsOutstanding =
    AccountsUncollected +
    FeesOutstanding +
    InterestOutstanding +
    PrincipalAccounts +
    AccountsDue

In turn:
• AccountsUncollected =
    uncollected_principal +
    uncollected_installment_principal
Under the hood: database calls

SELECT
  vl.value AS account,
  SUM(CASE vl.value WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END)
  - SUM(CASE vl.value WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS sum
FROM payment_transactions pt
JOIN valuelists vl ON vl.type_cd = 'transaction_account'
  AND vl.value IN (pt.debit_account_cd, pt.credit_account_cd)
AND loan_id= ?

... and then a value for specific account is selected.
Drawbacks

The method itself would allow retrieving all the information related to one loan “in one shot”.

However, because the application developers are unaware of the underlying layers, there appears to be no difference:

- Whether we obtain the values of all account balances one-by-one by following normal Object-Oriented method logic, in an imperative way
- Or if we obtain them all, simultaneously

Now for some results:
PGBadger log

SELECT
    vl.value AS account,
    SUM(CASE WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) - SUM(CASE
        WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) AS debit,
    SUM(CASE WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS credit
FROM payment_transactions_committed pt
INNER JOIN valuelists vl ON vl.type_cd = "" AND vl.value IN (pt.debit_account_cd, pt
WHERE (loan_id = 0 AND installment_id = 0 AND committed = true) GROUP BY vl.value;

SELECT
    vl.value AS account,
    SUM(CASE WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) - SUM(CASE
        WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) AS debit,
    SUM(CASE WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS credit
FROM payment_transactions_committed pt
INNER JOIN valuelists vl ON vl.type_cd = "" AND vl.value IN (pt.debit_account_cd, pt
WHERE (loan_id = 0 AND committed = true) GROUP BY vl.value;
Modified method

```
{"interest_due":"0.00",
"lender_fees_ar":"0.00",
"uncollected_principal":"500.00",
"late_fees_ar":"25.00",
"uncollected_late_fees":"0.00",
.....
```
Under the hood: database calls

```sql
SELECT
  loan_id,
  sum(CASE WHEN debit_account_cd = 'uncollected_principal'
   THEN pt.amount ELSE 0 END
- CASE WHEN credit_account_cd='uncollected_principal'
   THEN pt.amount ELSE 0 END)
  AS uncollected_principal
,...
  sum(CASE WHEN debit_account_cd = 'unCollected_nsf_fees'
   THEN pt.amount ELSE 0 END
- CASE WHEN credit_account_cd = 'unCollected_nsf_fees'
   THEN pt.amount ELSE 0 END)
  AS uncollected_nsf_fees
  , sum(CASE WHEN debit_account_cd = 'installment_principal'
   THEN pt.amount ELSE 0 END
- CASE WHEN credit_account_cd = 'installment_principal'
   THEN pt.amount ELSE 0 END)
  AS installment_principal
FROM payment_transactions_committed pt
  INNER JOIN loans l ON l.id=pt.loan_id
WHERE loan_id={??}
```
<table>
<thead>
<tr>
<th>Time</th>
<th>Count</th>
<th>Speed</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.895s</td>
<td>3,350</td>
<td>0.001s/0.005s/0.001s</td>
<td>SELECT vl.VALUE AS account, sum (CASE vl.VALUE WHEN pt.debit_account_cd THEN pt.vl.VALUE WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS sum, sum (CASE pt.debit_account_cd THEN pt.amount ELSE 0 END) AS debit, sum (CASE vl.VALUE WHEN pt.amount ELSE 0 END) AS credit FROM payment_transactions pt JOIN valuelists vl IN (pt.debit_account_cd, pt.credit_account_cd) WHERE (loan_id = 0 AND install)</td>
</tr>
<tr>
<td>4.625s</td>
<td>844</td>
<td>0.000s/0.259s/0.005s</td>
<td>UPDATE genesys.channel_messages SET &quot;message_identifier&quot; = 'e', &quot;request_message&quot; = 'channel' = 'e', &quot;owner_id&quot; = 0, &quot;response_message&quot; = 'e', &quot;response_time&quot; = (SHOW EXAMPLES)</td>
</tr>
</tbody>
</table>
... and new

<table>
<thead>
<tr>
<th>Time</th>
<th>Count</th>
<th>Duration</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.861s</td>
<td>985</td>
<td>0.001s/0.006s/0.002s</td>
<td><code>SELECT * FROM loans.sum_accounts_by_loan_id_detailed ( NULL, 0, NULL, NULL, 0, NULL, );</code></td>
</tr>
<tr>
<td>1.822s</td>
<td>515</td>
<td>0.001s/0.316s/0.004s</td>
<td><code>SELECT rv.* FROM lexis_nexis.risk_view_reports rv INNER JOIN credit_reports cr ON cr.cr.report_type_cd = '' WHERE ( cr.customer_id = 0 ) ORDER BY cr.inquiry_time DESC LIMIT</code></td>
</tr>
<tr>
<td>1.758s</td>
<td>144</td>
<td>0.003s/0.039s/0.012s</td>
<td><code>SELECT sum ( heap_blks_read ) AS a1, sum ( heap_blks_hit ) AS a2, sum ( idx_blks_read ) AS a4, sum ( toast_blks_read ) AS a5, sum ( toast_blks_hit ) AS a6, sum ( tidx_blks_read ) AS a7, sum ( tidx_blks_hit ) AS a8 FROM pg_statio_user_tables;</code></td>
</tr>
</tbody>
</table>
So… now everybody happy?...
…not really…

What do application developers say at this point?....

Wait! What about the business logic?!
What about the business logic?

✓ We need some business logic to execute joins and selects
✓ Selected results transformations and manipulations do not have to be executed on the database side.
One more time:

- Disassemble method into atomic steps,
- Identify ones which require data retrieval
- Using knowledge about database objects relationships, construct a single query
- Execute
- Use retrieved data in other steps
Let’s review another example – account balance calculation for Lines of credit.
Account_Balance method for LOC

- Obtain account principal balance
- Obtain outstanding fees and interest as of next payment due date
- Calculate the interest credit (unearned interest) for the number of days left before the payment due date
- Obtain existing customer balance
- Calculate the total account balance using the values obtained on steps 1-4.
Traditional object-oriented approach

Account_Balance method calls:

- Principal_Balance,
- Interest_Amount,
- Fees_Amount,
- Customer_Balance
- Interest_Credit

Each of them would interact with a database **independently**.
Drilling down into each of the steps

1. Principal balance as described for account_outstanding requires a single database call.
2. Outstanding interest and fees require one database call each.
3. Interest credit calculation:
   3.1. Obtain the daily interest rate for this customer
   3.2. Obtain base amount, which is used to calculate the total interest
   3.3. Obtain the number of days, for which the interest should be credited:
      3.3.1. Obtain the next payment due date
      3.3.2. Calculate number of days based on obtained date and todays’ date
   3.4. Calculate amount of credit, based on results from the previous three steps
4. Customer balance can be obtained using one database call, same as steps 1-3.
Combining steps with data retrieval

For a given loan, retrieve payment transactions, which show principal balance, current interest, fees and customer balance, also retrieve loan’s daily interest rate and next payment due date.
After disassembling this method..

… we were able to retrieve all data using one single SELECT statement and two simple Ruby methods.
SELECT l.id AS loan_id,
    ,sum( CASE WHEN debit_account_cd = 'principal'
        AND t.acct_date<= v_current_date
        THEN t.amount ELSE 0 END 
    ) - CASE WHEN credit_account_cd
        AND t.acct_date<= v_current_date
        THEN t.amount ELSE 0 END ) AS amount_payable,
    ,sum (CASE WHEN t.debit_account_cd = 'fees_provisional'
    THEN t.amount ELSE 0 END ) - CASE WHEN t.credit_account_cd = 'fees_provisional'
    THEN t.amount ELSE 0 END ) AS fees_provisional,
    ,sum (CASE WHEN t.debit_account_cd = 'interest_provisional'
    THEN t.amount ELSE 0 END ) - CASE WHEN t.credit_account_cd = 'interest_provisional'
    THEN t.amount ELSE 0 END ) AS interest_provisional,
    st.end_date AS next_closing_date,
    ,l.daily_rate AS interest_rate,
    ,sum (CASE WHEN t.debit_account_cd
        = 'customer_balance'
        THEN -t.amount ELSE 0 END 
    ) - CASE WHEN t.credit_account_cd = 'customer_balance'
    THEN -t.amount ELSE 0 END ) AS customer_balance
FROM loans l
    LEFT OUTER JOIN payment_transactions_committed t ON
        l.id=t.loan_id
    LEFT OUTER JOIN statements st ON
        l.id=st.loan_id
WHERE l.id={?}
GROUP BY l.id ,l.daily_rate ,st.end_date
def account_balance

    <..>

    amt = amt + provisional_fees +
        [ provisional_interest - [unearned_interest,
            0].max, 0 ] .max

    <...

    return [0, amt].max

end
def unearned_interest
    amt = -1 * (principal_amount +
                [(customer_balance - interest_provisional -
                  fees_provisional), 0].max)

    next_closing_date =
    Date.parse(self.next_closing_date)

    <..<>

    interest < 0.0 ? 0 : interest

end
Note, that all data elements can still be retrieved with a single select statement, which won’t be possible within the standard ORM framework.
On a Larger Scale
How satisfied the users are…

Numbers from our weekly report:

<table>
<thead>
<tr>
<th>APP</th>
<th>Page Views (thousands)</th>
<th>Load Time (sec)</th>
<th>% Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old APP 1</td>
<td>21.7</td>
<td>6.83</td>
<td>64.2</td>
</tr>
<tr>
<td>Old APP 2</td>
<td>29.5</td>
<td>5.29</td>
<td>64.4</td>
</tr>
<tr>
<td>New APP</td>
<td>26.5</td>
<td>0.514</td>
<td>99.8</td>
</tr>
</tbody>
</table>
## Execution statistics: Old App vs. New App

<table>
<thead>
<tr>
<th>Controller action</th>
<th>Old Avg # DB calls</th>
<th>New Avg # DB calls</th>
<th>Old Avg Time (sec)</th>
<th>New Avg Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Summary</td>
<td>167</td>
<td>39</td>
<td>1.08</td>
<td>0.19</td>
</tr>
<tr>
<td>Loan Summary</td>
<td>506</td>
<td>50</td>
<td>4.5</td>
<td>0.44</td>
</tr>
<tr>
<td>Loan Payments</td>
<td>36</td>
<td>3</td>
<td>0.11</td>
<td>0.04</td>
</tr>
<tr>
<td>Installments</td>
<td>130</td>
<td>3</td>
<td>0.72</td>
<td>0.018</td>
</tr>
</tbody>
</table>
Avg DB time (sec)

- Customer Summary: Old Avg Time (sec) 1, New Avg Time (sec) 0.5
- Loan Summary: Old Avg Time (sec) 4.5, New Avg Time (sec) 1
- Loan Payments: Old Avg Time (sec) 0.5, New Avg Time (sec) 0.5
- Installments: Old Avg Time (sec) 1, New Avg Time (sec) 0
Installment Presenter page load time

Implementation Date
Now

• We save both time

• *And* money!
Related Work
Were we the first to notice the problem?
Definitely **NOT!**

The problem of object-relational impedance mismatch is a constant discussion topic. In recent years multiple attempts were made to try to resolve this issue with no significant outcome.
Hibernate ORM Service

• Claim: does not hide “the power of SQL” from developers.
• Yes, it allows queries

But...

• It is not an easy task
• Still prompts “natural” solutions
Newer Version of Active Record

- Allows Eager loading and some customer queries, but has the same limitations as Hibernate

- Eager loading may cause an excessive application memory usage
Agile Technology

• acknowledges the existence of ORIM;
• not only the technical impedance mismatch, but also a cultural impedance mismatch:
  “The object-oriented paradigm is based on proven software engineering principles. The relational paradigm, however, is based on proven mathematical principles”.
• raises awareness of the problem

But…
Agile data technology solutions:
• schema changes and/or more careful design – yes
• data refactoring – yes
• dealing with inefficient queries - no
Other related work

• **AppSlueth** tool for application tuning: identifies “delinquent design patterns”; in general does not allow to stay within ORM, or to reuse existing methods

• **SQLAlchemy** tool: allows the object model and database schema to develop in a cleanly decoupled way from the beginning; problem: implies that an application developer is at the same time a database developer, is aware of the best data access paths, and can divert from the OO design standards

• **Dbridge project**: holistic approach to the application optimization.
What’s Next?
Future work

Our goal: to make the Logic Split a company-wide development methodology

Problems:
- Large amount of the legacy code
- No business specifications
- Evolving legacy app
- The human factor
Tasks:

- Separating logic from the Ruby code
- Verifying it with business stakeholders;
- Reusing of our new models
- Continue rewriting
- Clarifying technology
- Constant results measuring
And let me give you just one example…

UK portal current execution times - Loan Controller:

<table>
<thead>
<tr>
<th>Loan type</th>
<th>Avg DB Calls</th>
<th>Max DB Calls</th>
<th>Avg. Page Time</th>
<th>Avg. DB Time</th>
<th>Max DB Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>payday</td>
<td>388</td>
<td>5292</td>
<td>2.83</td>
<td>2.06</td>
<td>40.37</td>
</tr>
<tr>
<td>installment</td>
<td>1112</td>
<td>10863</td>
<td>10.32</td>
<td>6.74</td>
<td>73.53</td>
</tr>
<tr>
<td>oec</td>
<td>274</td>
<td>1594</td>
<td>2.65</td>
<td>1.76</td>
<td>12.50</td>
</tr>
</tbody>
</table>

loan id=6820002, db time: 73.53 sec

SELECT * FROM aperture.loan_search_single_loan (6820002) – 0.26 sec  
SELECT * FROM aperture.loan_installments__api (6820002) – 0.03 sec  
SELECT * FROM aperture.loan_payments__api (6820002) – 0.2 sec
Acknowledgements

Many thanks to:

- Enova CIO/CTO Fred Lee
- Team Raven, and everybody who worked on the Aperture project during its lifetime:
  - Jef Jonjevic
  - Marc Groulx
  - Richard Lee
  - Preeti Dhiman
  - Luda Baklanova
  - Jian Wu
  - Chetana Yogeesh
  - Alan Zoppa
  - Ana Lebon
  - Kevin Glowacz
  - Neha Bhardwaj
  - Ranning Li

- Sheldon Strauch and the entire DB Dev team
- Chad Slaughter and Kurt Stephens
- Ben Heilman, Donny Jekels and Sam Rees
- Dave Trollop and Paul Solomon
Find more at:

- [www.enova.com](http://www.enova.com)
- The World of Data: [hdombrovskaya.wordpress.com](http://hdombrovskaya.wordpress.com)
- [hdombrovskaya@enova.com](mailto:hdombrovskaya@enova.com)