Tune the App, Not the SQL

DBA Sherlock's Adventures in Hibernate/jOOQ Land

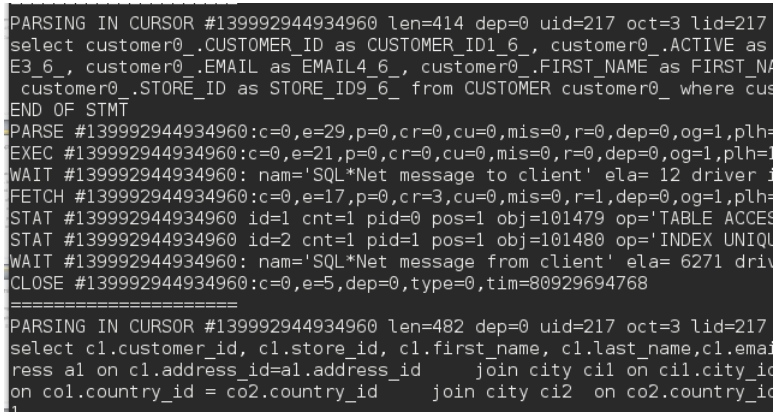
BASEL ▪ BERN ▪ BRUGG ▪ DÜSSELDORF ▪ FRANKFURT A.M. ▪ FREIBURG I.BR. ▪ GENEVA
HAMBURG ▪ COPENHAGEN ▪ LAUSANNE ▪ MUNICH ▪ STUTTGART ▪ VIENNA ▪ ZURICH

trivadis
makes IT easier.

It’s the App, not the SQL

Tuning Application Performance - DBA Sherlock's View

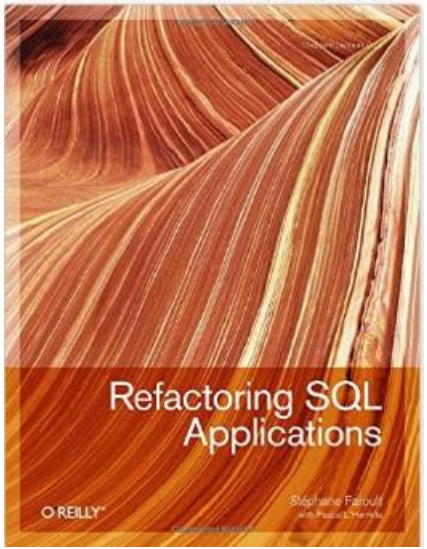
- Perform tracing
- Rank SQL by elapsed time
- Recommend / perform
  - Rewrite of SQL
  - Change of access structures
  - Parameter changes
  - Using hints
- *Hope that overall performance will improve :-)*



trivadis makes IT easier.

OK but … What's Missing?

- We're tuning every statement in isolation …
- What if application logic matters? Retrieving
  - WHAT
  - WHEN, and
  - HOW MUCH of it
- Credits to Stéphane Faroult for emphasizing this aspect



Sherlock: "*What's that application doing*"

- SQL trace shows statements in order of execution
- But … all application processing remains unknown
- In a real-world application, it may be completely impossible to reconstruct the logic

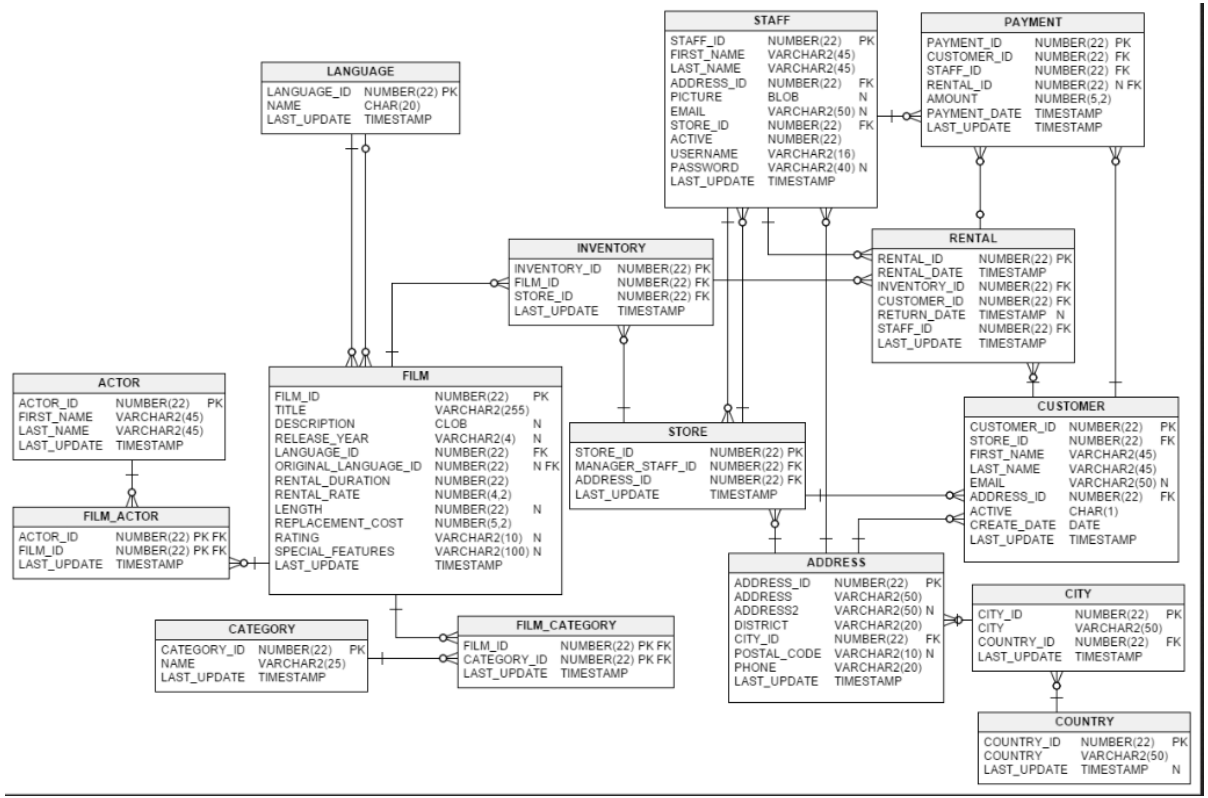
- DBAs and developers need to work ***together*** :-)

Sample Use Case

Application Tuning Example – Data

- Sakila schema (https://dev.mysql.com/doc/sakila/en/)
- Ported to Oracle by Lukas Eder https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/Sakila/oracle-sakila-db
- Used for its higher complexity compared to e.g. Oracle Order Entry

Application Tuning Example – Sakila Schema



Application Tuning Example – Use Case

- When a customer browses the film rental database, we want to display films that
  - have been watched by “similar” customers and
  - are in the customer’s preferred category
- Similarity for our purpose being defined as “living in the same country“
- Tailored to structure of Sakila example schema
- Designed to require several multi-table relationships

Sample Frameworks

Frameworks: jOOQ

- Lightweight, Active Record - style JDBC wrapper with advanced code generation capabilities
- Typesafe writing SQL using a fluent API

```
private List<BigDecimal> getFilmIds(Connection conn, List<BigDecimal> similarCustomerIds) {

  List<BigDecimal> filmIds = DSL.using(conn, SQLDialect.ORACLE)
    .select(INVENTORY.FILM_ID)
    .from(INVENTORY)
    .join(RENTAL)
    .on(INVENTORY.INVENTORY_ID.equal(RENTAL.INVENTORY_ID))
    .where(RENTAL.CUSTOMER_ID.in(similarCustomerIds))
    .fetchInto(BigDecimal.class);
  return filmIds;
}
```

Frameworks: Hibernate

- Object-Relational Mapping Framework providing advanced features (e.g., multi-level caching, custom configuration of fetch plans and strategies)
- Keyword: “object-relational impedance mismatch“
- Entity relationships configured in the Java objects

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "customerId")
private Set<Payment> paymentSet;

@JoinColumn(name = "ADDRESS_ID", referencedColumnName = "ADDRESS_ID")
@ManyToOne(optional = false)
private Address address;

@JoinColumn(name = "STORE_ID", referencedColumnName = "STORE_ID")
@ManyToOne(optional = false)
private Store storeId;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "customerId")
private Set<Rental> rentalSet;

public Customer() {
}
```

Tuning jOOQ

jOOQ: Basic Implementation

- Step 1: Load the target customer

```
CustomerRecord customer = getCustomer(conn);
```

```
private CustomerRecord getCustomer(Connection conn) {

  return DSL.using(conn, SQLDialect.ORACLE)
    .selectFrom(CUSTOMER)
    .where(CUSTOMER.CUSTOMER_ID.equal(
      BigDecimal.valueOf(new Random().nextInt(CUSTOMER_MAX_ID) + 1)))
    .fetchOne();

}
```

jOOQ: Basic Implementation

- Step 2: Find similar customers
  - 2a: get target customer's country

```
CountryRecord customerCountry = getCountry(conn, customer.getAddressId());


private CountryRecord getCountry(Connection conn, BigDecimal addressId) {

  return DSL.using(conn, SQLDialect.ORACLE)
    .select(COUNTRY.COUNTRY_ID, COUNTRY.COUNTRY_, COUNTRY.LAST_UPDATE)
    .from(ADDRESS)
    .join(CITY)
    .on(CITY.CITY_ID.equal(ADDRESS.CITY_ID))
    .join(COUNTRY)
    .on(COUNTRY.COUNTRY_ID.equal(CITY.COUNTRY_ID))
    .where(ADDRESS.ADDRESS_ID.equal(addressId))
    .fetchOneInto(CountryRecord.class);

}
```

jOOQ: Basic Implementation

- Step 2: Find similar customers
  - 2b: find customers living in same country

```
List<BigDecimal> similarCustomerIds = getSimilarCustomers(conn, customerCountry.getCountryId());
```

Retrieving IDs only!

```
private List<BigDecimal> getSimilarCustomers(Connection conn, BigDecimal countryId)

  return DSL.using(conn, SQLDialect.ORACLE)
    .select(CUSTOMER.CUSTOMER_ID)
    .from(CUSTOMER)
    .join(ADDRESS)
    .on(CUSTOMER.ADDRESS_ID.equal(ADDRESS.ADDRESS_ID))
    .join(CITY)
    .on(CITY.CITY_ID.equal(ADDRESS.CITY_ID))
    .join(COUNTRY)
    .on(COUNTRY.COUNTRY_ID.equal(CITY.COUNTRY_ID))
    .where(COUNTRY.COUNTRY_ID.equal(countryId))
    .fetchInto(BigDecimal.class);

}
```

jOOQ: Basic Implementation

- Step 3: find customer's preferred category

```
BigDecimal categoryId = getPreferredCategory(conn, customer);
```

Retrieving ID only!

```
private BigDecimal getPreferredCategory(Connection conn, CustomerRecord customer) {

  return DSL.using(conn, SQLDialect.ORACLE)
    .select(CATEGORY.CATEGORY_ID)
    .from(CATEGORY)
    .where(CATEGORY.CATEGORY_ID.equal(BigDecimal.valueOf(
          new Random().nextInt(CATEGORY_MAX_ID) + 1)))
    .fetchOneInto(BigDecimal.class);

}
```

jOOQ: Basic Implementation

- Step 4: Identify films that have been watched by similar customers

```
List<BigDecimal> filmIds = getFilmIds(conn, similarCustomerIds);
```

Retrieving IDs only!

```
private List<BigDecimal> getFilmIds(Connection conn, List<BigDecimal> similarCustomerIds) {

  List<BigDecimal> filmIds = DSL.using(conn, SQLDialect.ORACLE)
    .select(INVENTORY.FILM_ID)
    .from(INVENTORY)
    .join(RENTAL)
    .on(INVENTORY.INVENTORY_ID.equal(RENTAL.INVENTORY_ID))
    .where(RENTAL.CUSTOMER_ID.in(similarCustomerIds))
    .fetchInto(BigDecimal.class);
  return filmIds;
}
```

jOOQ: Basic Implementation

- Step 5: Display films that match both criteria

```
List<FilmRecord> films = getFilmInfo(conn, filmIds, categoryId);

private List<FilmRecord> getFilmInfo(Connection conn, List<BigDecimal> filmIds, BigDecimal categoryId) {

  List<FilmRecord> films = DSL.using(conn, SQLDialect.ORACLE)
    .selectDistinct(FILM.FILM_ID, FILM.DESCRIPTION, FILM.LANGUAGE_ID, FILM.LAST_UPDATE, FILM.LENGTH,
     FILM.ORIGINAL_LANGUAGE_ID, FILM.RATING, FILM.RELEASE_YEAR, FILM.RENTAL_DURATION,
     FILM.RENTAL_RATE, FILM.REPLACEMENT_COST, FILM.SPECIAL_FEATURES, FILM.TITLE)
    .from(FILM)
    .join(FILM_CATEGORY)
    .on(FILM_CATEGORY.FILM_ID.equal(FILM.FILM_ID))
    .join(CATEGORY)
    .on(CATEGORY.CATEGORY_ID.equal(FILM_CATEGORY.CATEGORY_ID))
    .where(FILM.FILM_ID.in(filmIds.stream().limit(1000).collect(Collectors.toList())))
    .and(CATEGORY.CATEGORY_ID.equal(categoryId))
    .fetchInto(FilmRecord.class);
  return films;

}
```

jOOQ: Basic Implementation – Performance (10000 executions)

- Client execution time: 1,009 s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    60000      0.85       1.73          0          0          0           0
Execute  60000      2.55       3.55          0          0          0           0
Fetch   699761     67.39      98.27          0    3166345          0     6552953
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   819761     70.80     103.56          0    3166345          0     6552953

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message from client                699764        0.50        866.76
```

DB : other ~ 1 : 8

jOOQ: Basic Implementation - Performance

- How long do the single steps take?

Step	Elapsed Time (s)	No. Executions
get films from similar customers	69.597	10000
get customers in same country	16.368	10000
get films to display	15.639	10000
get customer country	0.737	10000
get target customer	0.716	10000
get customer preferred category	0.558	10000

jOOQ: Basic Implementation - Performance

- SQL statement execution takes MUCH less time than data transfer over network
- Any attempt at “SQL Tuning” will therefore not have overwhelming effects …
- Sherlock: *“Could you do that with fewer data?”*
- *“Perhaps even with fewer statements?”*
- *“Sure let’s have a look …”*
  - *“We could integrate retrieving the customer’s country into the main get similar customers query …”*
  - *“We should be able to retrieve the data needed to display the films in one step, too”*

jOOQ: Refactoring 1: Combining Steps 2a & 2b

- Step 2 new: Find similar customers in 1 step

```
List<BigDecimal> similarCustomerIds = getSimilarCustomers1Step(conn, customer.getAddressId());

private List<BigDecimal> getSimilarCustomers1Step(Connection conn, BigDecimal addressId) {

  Customer c1 = CUSTOMER.as("c1");
  Address a1 = ADDRESS.as("a1");
  City ci1 = CITY.as("ci1");
  Country co1 = COUNTRY.as("co1");
  Address a2 = ADDRESS.as("a2");
  City ci2 = CITY.as("ci2");
  Country co2 = COUNTRY.as("co2");
  return DSL.using(conn, SQLDialect.ORACLE)
    .select(c1.CUSTOMER_ID).from(c1)
    .join(a1).on(c1.ADDRESS_ID.equal(a1.ADDRESS_ID))
    .join(ci1).on(ci1.CITY_ID.equal(a1.CITY_ID))
    .join(co1).on(co1.COUNTRY_ID.equal(ci1.COUNTRY_ID))
    .join(co2).on(co1.COUNTRY_ID.equal(co2.COUNTRY_ID))
    .join(ci2).on(co2.COUNTRY_ID.equal(ci2.COUNTRY_ID))
    .join(a2).on(ci2.CITY_ID.equal(a2.CITY_ID))
    .where(a2.ADDRESS_ID.equal(addressId))
    .fetchInto(BigDecimal.class);
}
```

jOOQ: Refactoring 1 - Performance

- Client execution time: 970s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    50000      0.77       1.54          0          0          0           0
Execute  50000      2.58       3.39          0          0          0           0
Fetch   673321     64.90      94.96          0    3128232          0     6379218
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   773321     68.26      99.90          0    3128232          0     6379218

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message from client                673324        0.49        819.78
```

jOOQ: Refactoring 1 - Performance

- How long do the single steps take?

Step	Elapsed Time (s)	No. Executions
get films from similar customers	67.894	10000
get similar customers, 1 step	15.447	10000
get films to display	15.322	10000
get target customer	0.713	10000
get customer preferred category	0.578	10000

jOOQ: Refactoring 1 - Performance

- Refactored version is faster by 1,685 ms
- The new version is even faster than part 2 of the original version…!
- Sherlock: *“How is that even possible?”*
- *“Wildly differing execution plans?”* -> No!
- But: In addition to expected measurement errors, there is variance due to random data selection
- Still, we didn’t gain much – no wonder: Room for improvement was small since part 1 of original version already was very fast (737 ms)
- Sherlock: *“Room for improvement should be bigger with the second refactoring, then”*

jOOQ: Refactoring 2: Combining Steps 4 & 5

- Step 4 new: Get films to display (matching all criteria) in 1 step

```
List<FilmRecord> films = getFilmInfo1Step(conn, similarCustomerIds, categoryId);

private List<FilmRecord> getFilmInfo1Step(Connection conn, List<BigDecimal> similarCustomerIds,
  BigDecimal categoryId) {

  List<FilmRecord> films = DSL.using(conn, SQLDialect.ORACLE)
    .selectDistinct(FILM.FILM_ID, FILM.DESCRIPTION, FILM.LANGUAGE_ID, FILM.LAST_UPDATE, FILM.LENGTH,
      FILM.ORIGINAL_LANGUAGE_ID, FILM.RATING, FILM.RELEASE_YEAR, FILM.RENTAL_DURATION,
      FILM.RENTAL_RATE, FILM.REPLACEMENT_COST, FILM.SPECIAL_FEATURES, FILM.TITLE)
    .from(FILM)
    .join(FILM_CATEGORY).on(FILM_CATEGORY.FILM_ID.equal(FILM.FILM_ID))
    .join(CATEGORY).on(CATEGORY.CATEGORY_ID.equal(FILM_CATEGORY.CATEGORY_ID))
    .join(INVENTORY).on(INVENTORY.FILM_ID.equal(FILM.FILM_ID))
    .join(RENTAL).on(RENTAL.INVENTORY_ID.equal(INVENTORY.INVENTORY_ID))
    .where(CATEGORY.CATEGORY_ID.equal(categoryId))
    .and(RENTAL.CUSTOMER_ID.in(similarCustomerIds))
    .fetchInto(FilmRecord.class);
  return films;

}
```

jOOQ: Refactoring 2 - Performance

- Client execution time: 206s
- And the database says...

call	count	cpu	elapsed	disk	query	current	rows
Parse	40000	0.43	1.10	0	0	0	0
Execute	40000	0.94	1.65	0	0	0	0
Fetch	78621	70.99	78.17	0	2607753	0	482374
total	158621	72.36	80.93	0	2607753	0	482374

Event waited on	Times Waited	Max. Wait	Total Waited
SQL*Net message from client	78624	0.48	106.78

jOOQ: Refactoring 2 - Performance

- How long do the single steps take?

Step	Elapsed Time (s)	No. Executions
get similar customers, 1 step	15.620	10000
get films to display, 1 step	63.993	10000
get target customer	0.788	10000
get customer preferred category	0.568	10000

jOOQ: Refactoring 2 - Performance

- This time, combining two application steps results in a speedup of 400%
- Where does the difference come from?

	Refactoring 1	Refactoring 2
SQL elapsed time (s)	100	81
SQL*Net from client (s)	820	107

Tuning jOOQ: Summary

Tuning jOOQ: Summary

- Without even paying attention to details, original performance *was improved by 400% just changing the query logic in the application*
- At the same time, refactoring 1 demonstrated that combining/condensing queries does not necessarily result in speedup
- Instead of relying on intuition, the *actual performance should be checked on application and database sides*

Tuning Hibernate

Hibernate: Basic Implementation

- Step 1: Load the target customer

```
Customer customer = getCustomer(em);
```

```
private Customer getCustomer(EntityManager em) {

  return em.find(Customer.class,
    BigDecimal.valueOf(new Random().nextInt(CUSTOMER_MAX_ID) + 1));

}
```

Hibernate: Basic Implementation

- Step 2: Find similar customers
  - 2a: get target customer's country

```
Country customerCountry = getCountry(em, customer.getAddress().getAddressId());


private Country getCountry(EntityManager em, BigDecimal addressId) {

  return (Country) em.createQuery(
    "select c from Country c join c.citySet ci join ci.addressSet a where a.addressId ="
    + addressId).getSingleResult();

}
```

Hibernate: Basic Implementation

- Step 2: Find similar customers
  - 2b: find customers living in same country

```
List<Customer> similarCustomers = getSimilarCustomers(em, customerCountry);
```

```
private List<Customer> getSimilarCustomers(EntityManager em, Country country) {

  List<Customer> c = em.createQuery(
    "select c from Customer c join c.address a join a.city ci join ci.country co"
    + " where co.countryId =" + country.getCountryId()).getResultList();
  return c;

}
```

Hibernate: Basic Implementation

- Step 3: Find customer's preferred category

```
BigDecimal categoryId = getPreferredCategory(conn, customer);
```

```
private Category getPreferredCategory(EntityManager em, Customer customer) {

  return em.find(Category.class,
    BigDecimal.valueOf(new Random().nextInt(CATEGORY_MAX_ID) + 1));

}
```

Hibernate: Basic Implementation

- Step 4: Identify films that have been watched by similar customers

```
List<Inventory> inventories = getInventories(em, similarCustomers);
```

```
private List<Inventory> getInventories(EntityManager em, List<Customer> similarCustomers) {

  Query query = em.createQuery(
    "select i from Inventory i join i.rentalSet r where r.customerId in :ids");
  query.setParameter("ids", similarCustomers);
  List<Inventory> inventories = query.getResultList();
  return inventories;

}
```

Hibernate: Basic Implementation

- Step 5: Display films that match both criteria

```
List<Film> films = getFilmInfo(em, inventories, category);
```

```
private List<Film> getFilmInfo(EntityManager em, List<Inventory> inventories, Category category) {
  Query query = em.createNativeQuery(
    "select distinct f.* from film f join inventory i on (f.film_id=i.film_id) "
    + "join film_category fc on (f.film_id = fc.film_id) join category c on (c.category_id=fc.category_id)"
    + " where i.film_id in :f and c.name= :c",
    Film.class);
  query.setParameter("f", inventories.stream().map(
    i -> i.getFilm().getFilmId()).limit(1000).collect(Collectors.toList()));
  query.setParameter("c", category.getName());
  List<Film> films = query.getResultList();
  return films;

}
```

Hibernate: Basic Implementation - Performance

- Client execution time: 1,616s
- And the database says...

JOOQ:
Client time: 1,009s
DB elapsed: 104s
SQL*Net : 867s

call	count	cpu	elapsed	disk	query	current	rows
Parse	41577	2.54	3.34	0	26	0	0
Execute	41577	5.49	6.29	0	280	0	0
Fetch	667036	88.84	121.36	0	3825253	0	6393616
total	750190	96.88	131.00	0	3825559	0	6393616

DB : other ~ 1 : 11

Elapsed times include waiting on following events:

Event waited on	Times Waited	Max. Wait	Total Waited
SQL*Net message from client	667151	0.76	1432.10

Hibernate: Basic Implementation - Performance

Step	Distinct SQL_IDs	Elapsed Time (s)	No. Executions
get films from similar customers	21	69.837	10000
get films to display	69	43.644	10000
get customers in same country	108	15.661	10000
get customer country	599	2.318	10000
select film0_.FILM_ID as FILM_ID1_7_0_, film0_.DESCRIPTION	1	0.095	958
get target customer	1	0.078	108
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.A	1	0.042	491
get customer preferred category	1	0.007	15
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_	1	0.005	1
select city0_.CITY_ID as CITY_ID1_4_0_, city0_.CITY as CITY2	1	0.001	1


OK Sherlock … What Is Going On?

- SQL trace shows
  - order of execution (raw trace file)
  - execution counts (processed)

- *"What can we deduce?"*

Time for some detective work!

And DBA Sherlock Investigates… [Mystery SQLs, 1/4]

```
select city0_.CITY_ID as CITY_ID1_4_0_, city0_.CITY as CITY2_4_0_,
  city0_.COUNTRY_ID as COUNTRY_ID4_4_0_, city0_.LAST_UPDATE as
  LAST_UPDATE3_4_0_, country1_.COUNTRY_ID as COUNTRY_ID1_5_1_,
  country1_.COUNTRY as COUNTRY2_5_1_, country1_.LAST_UPDATE as
  LAST_UPDATE3_5_1_
from
 CITY city0_ inner join COUNTRY country1_ on city0_.COUNTRY_ID=
  country1_.COUNTRY_ID where city0_.CITY_ID=:1
```

- Occurs: exactly one time, on the first iteration, directly before the *get country* query
- WHY? -> <no solution>
- Problem? -> No: query executed just once

And DBA Sherlock Investigates… [Mystery SQLs, 2/4]

```
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_ID as
  ADDRESS_ID3_16_0_, store0_.LAST_UPDATE as LAST_UPDATE2_16_0_,
  store0_.MANAGER_STAFF_ID as MANAGER_STAFF_ID4_16_0_, address1_.ADDRESS_ID
  as ADDRESS_ID1_2_1_, address1_.ADDRESS as ADDRESS2_2_1_, address1_.ADDRESS2
...
from
 STORE store0_ inner join ADDRESS address1_ on store0_.ADDRESS_ID=
  address1_.ADDRESS_ID inner join CITY city2_ on address1_.CITY_ID=
  city2_.CITY_ID inner join COUNTRY country3_ on city2_.COUNTRY_ID=
  country3_.COUNTRY_ID inner join STAFF staff4_ on store0_.MANAGER_STAFF_ID=
  staff4_.STAFF_ID inner join ADDRESS address5_ on staff4_.ADDRESS_ID=
  address5_.ADDRESS_ID inner join STORE store6_ on staff4_.STORE_ID=
  store6_.STORE_ID where store0_.STORE_ID=:1
```

- occurs exactly one time, on the first iteration, in between several instances of the *get films from similar customers* query
- WHY? -> <no solution>
- Problem? -> No: query executed just once

And DBA Sherlock Investigates… [Mystery SQLs, 3/4]

```
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.ADDRESS as
  ADDRESS2_2_0_, address0_.ADDRESS2 as ADDRESS3_2_0_, address0_.CITY_ID as
  CITY_ID8_2_0_, address0_.DISTRICT as DISTRICT4_2_0_, address0_.LAST_UPDATE
  as LAST_UPDATE5_2_0_, address0_.PHONE as PHONE6_2_0_, address0_.POSTAL_CODE
  as POSTAL_CODE7_2_0_, city1_.CITY_ID as CITY_ID1_4_1_, city1_.CITY as
  CITY2_4_1_, city1_.COUNTRY_ID as COUNTRY_ID4_4_1_, city1_.LAST_UPDATE as
  LAST_UPDATE3_4_1_, country2_.COUNTRY_ID as COUNTRY_ID1_5_2_,
  country2_.COUNTRY as COUNTRY2_5_2_, country2_.LAST_UPDATE as
  LAST_UPDATE3_5_2_
from
 ADDRESS address0_ inner join CITY city1_ on address0_.CITY_ID=city1_.CITY_ID
  inner join COUNTRY country2_ on city1_.COUNTRY_ID=country2_.COUNTRY_ID
  where address0_.ADDRESS_ID=:1
```

- occurs 491 times, directly before the *get preferred category* query
- WHY? -> <no solution>
- Problem? -> No: takes just 42 ms overall

And DBA Sherlock Investigates… [Mystery SQLs, 4/4]

```
select film0_.FILM_ID as FILM_ID1_7_0_, film0_.DESCRIPTION as
  DESCRIPTION2_7_0_, film0_.LANGUAGE_ID as LANGUAGE_ID12_7_0_,
(...)
  TITLE11_7_0_, language1_.LANGUAGE_ID as LANGUAGE_ID1_0_1_,
  language1_.LAST_UPDATE as LAST_UPDATE2_0_1_, language1_.NAME as NAME3_0_1_,
  language2_.LANGUAGE_ID as LANGUAGE_ID1_0_2_, language2_.LAST_UPDATE as
  LAST_UPDATE2_0_2_, language2_.NAME as NAME3_0_2_
from
 FILM film0_ inner join "LANGUAGE" language1_ on film0_.LANGUAGE_ID=
  language1_.LANGUAGE_ID left outer join "LANGUAGE" language2_ on
  film0_.ORIGINAL_LANGUAGE_ID=language2_.LANGUAGE_ID where film0_.FILM_ID=:1
```

- occurs 958 times, after the *get films from similar customers* query
- WHY? … wait …

And DBA Sherlock Investigates… [Mystery SQLs, 4/4]

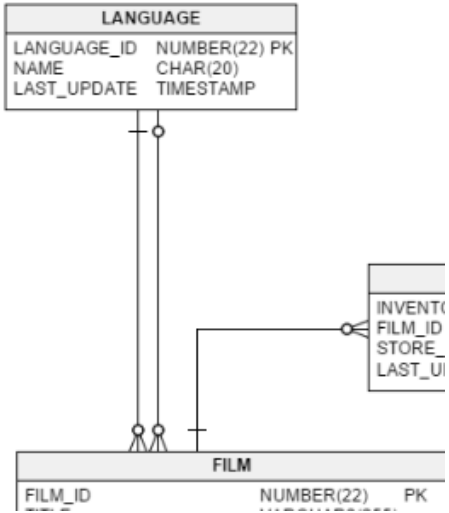
- *"show me that code again"*

```
Query query = em.createNativeQuery(
  "select distinct f.* from film f join inventory i on (f.film_id=i.film_id) "
  + "join film_category fc on (f.film_id = fc.film_id) join category c on (c.category_id=fc.category_id)"
  + " where i.film_id in :f and c.name= :c",
  Film.class);
query.setParameter("f", inventories.stream().map(
  i -> i.getFilm().getFilmId()).limit(1000).collect(Collectors.toList()));
```

- Yaaay …!
- OK but … what exactly is Hibernate doing here?

And DBA Sherlock Investigates… [Mystery SQLs, 4/4]

- Getting films …
- … inner joining LANGUAGE b/c of NOT NULL foreign key constraint from FILM.LANGUAGE_ID
- … outer joining LANGUAGE b/c of NULLABLE foreign key constraint from FILM.ORIGINAL_LANGUAGE_ID
- *"This should disappear automagically with the planned refactorings …"*



And DBA Sherlock Investigates… [Execution Counts]

- Two of the “regular” / expected queries are not executed 10000, but
  - 108 (*get target customer*) resp.
  - 15 times (*get customer preferred category)*
- Which is once per distinct category / customer …
- *“There must be some Hibernate caching going on”*

“Wait … and what is THIS?”

- This is the *get target customer* query …

```
select customer0_.CUSTOMER_ID as CUSTOMER_ID1_6_0_, customer0_.ACTIVE as
  ACTIVE2_6_0_, customer0_.ADDRESS_ID as ADDRESS_ID8_6_0_,
(...)
from
 CUSTOMER customer0_ inner join ADDRESS address1_ on customer0_.ADDRESS_ID=
  address1_.ADDRESS_ID inner join CITY city2_ on address1_.CITY_ID=
  city2_.CITY_ID inner join COUNTRY country3_ on city2_.COUNTRY_ID=
  country3_.COUNTRY_ID inner join STORE store4_ on customer0_.STORE_ID=
  store4_.STORE_ID inner join ADDRESS address5_ on store4_.ADDRESS_ID=
  address5_.ADDRESS_ID inner join STAFF staff6_ on store4_.MANAGER_STAFF_ID=
  staff6_.STAFF_ID inner join ADDRESS address7_ on staff6_.ADDRESS_ID=
  address7_.ADDRESS_ID inner join STORE store8_ on staff6_.STORE_ID=
  store8_.STORE_ID where customer0_.CUSTOMER_ID=:1
```

- *“but we just want (and need!) the customer …?”*

“Wait … and what is THIS?”

- All joins are due to many-to-one relationships …

```
select customer0_.CUSTOMER_ID as CUSTOMER_ID1_6_0_, customer0_.ACTIVE as
  ACTIVE2_6_0_, customer0_.ADDRESS_ID as ADDRESS_ID8_6_0_,
(...)
from CUSTOMER customer0_
inner join ADDRESS address1_ on customer0_.ADDRESS_ID=address1_.ADDRESS_ID    ### from Customer
   inner join CITY city2_ on address1_.CITY_ID=city2_.CITY_ID                 ### from Address
   inner join COUNTRY country3_ on city2_.COUNTRY_ID=country3_.COUNTRY_ID     ### from City
   inner join STORE store4_ on customer0_.STORE_ID=store4_.STORE_ID           ### from Customer
   inner join ADDRESS address5_ on store4_.ADDRESS_ID=address5_.ADDRESS_ID    ### from Store
   inner join STAFF staff6_ on store4_.MANAGER_STAFF_ID=staff6_.STAFF_ID      ### from Store
   inner join ADDRESS address7_ on staff6_.ADDRESS_ID=address7_.ADDRESS_ID    ### from Staff
   inner join STORE store8_ on staff6_.STORE_ID=store8_.STORE_ID              ### from Staff
   where customer0_.CUSTOMER_ID=:1
```

- *“This looks like a quick win”*

Hibernate: Refactoring 1

- Step 1 new: Load the target customer – and just the target customer

```
Customer customer = getCustomer(em);
```

```
private Customer getCustomer(EntityManager em) {

  return (Customer) em.createQuery("select c from Customer c where c.customerId = "
    + BigDecimal.valueOf(new Random().nextInt(CUSTOMER_MAX_ID) + 1))
    .getSingleResult();

}
```

Hibernate: Refactoring 1 - Performance

- Client execution time: 1,755s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    51577      3.08       4.33          0         26          0           0
Execute  51577      5.52       6.35          0        280          0           0
Fetch   684620     87.80     120.18          0    3861110          0     6478751
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   787774     96.42     130.88          0    3861416          0     6478751

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message from client                684628        0.76       1570.22
```

Hibernate: Refactoring 1 - Performance

Step	Distinct SQL_IDs	Elapsed Time (s)	No. Executions
get films from similar customers	21	69.242	10000
get films to display	69	43.510	10000
get customers in same country	108	15.208	10000
get customer country	599	2.296	10000
get target customer NEW	599 (was: 1)	1.450 (was: 78)	10000 (was: 108)
select film0_.FILM_ID as FILM_ID1_7_0_, film0_.DESCRIPTION	1	0.087	958
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.A	1	0.070	599
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_	1	0.025	1
get customer preferred category	1	0.006	15
~~select city0_.CITY_ID as CITY_ID1_4_0_, city0_.CITY as CITY2~~			

Hibernate: Refactoring 1 - Summary

- Reduced response time per execution gets counteracted by much higher execution count – no net wins
- But DBA Sherlock spotted something else now …
- *“Why does the number of distinct SQL_IDs increase?”*
- *“Let’s make sure we’re using bind variables everywhere”*

Hibernate: Refactoring 2

- Steps 1/2 new: Use bind variables instead of literals

```
                                        private Customer getCustomer(EntityManager em) {

                                          return (Customer) em.createQuery("select c from Customer c"
                                            + " where c.customerId = :cstid").setParameter("cstid",
                                              BigDecimal.valueOf(new Random().nextInt(CUSTOMER_MAX_ID) + 1))
                                            .getSingleResult();
                                        }
    private Country getCountry(Entity

      return (Country) em.createQuery("select c from Country c join c.citySet"
        + " ci join ci.addressSet a where a.addressId = :adr_id")
        .setParameter("adr_id", addressId).getSingleResult();

    }

private List<Customer> getSimilarCustomers(EntityManager em, Country country) {

  List<Customer> c = em.createQuery("select c from Customer c join c.address a "
    + "join a.city ci join ci.country co where co.countryId = :ctr_id")
    .setParameter("ctr_id", country.getCountryId())
    .getResultList();
  return c;

}
```

Hibernate: Refactoring 2 - Performance

- Client execution time: 1,770s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    51577      1.15       1.96          0         26          0           0
Execute  51577      5.44       6.48          0          0          0           0
Fetch   680318     90.39     127.22          0    3866777          0     6434677
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   783472     96.98     135.67          0    3866803          0     6434677

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------    Waited  ----------  ------------
SQL*Net message from client                 680326        0.78       1576.57
```

Hibernate: Refactoring 2 - Performance

Step	Distinct SQL_IDs	Elapsed Time (s)	No. Executions
get films from similar customers	21	70.213	10000
get films to display	69	42.786	10000
get customers in same country bind var	1	16.165 (was: 15.208)	10000
get customer country bind var	1	0.946 (was: 2.296)	10000
get target customer NEW bind var	1	0.771 (was: 1.450)	10000
select film0_.FILM_ID as FILM_ID1_7_0_, film0_.DESCRIPTION	1	0.089	958
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.A	1	0.066	599
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_	1	0.028	2
get customer preferred category	1	0.006	15

Hibernate: Refactoring 2 - Summary

- Reduced parse time noticeable (only) for statements that don't retrieve much data
- *"After this cleanup, let's do the main refactorings"*

Hibernate: Refactoring 3

- Step 2 new: Find similar customers in 1 step

```
List<Customer> similarCustomers = getSimilarCustomers1Step(em, customer.getAddress().getAddressId());

private List<Customer> getSimilarCustomers1Step(EntityManager em, BigDecimal addressId) {

  Query query = em.createNativeQuery("select c1.customer_id, c1.store_id, c1.first_name, c1.last_name,"
    + "c1.email, c1.address_id, c1.active, c1.create_date, c1.last_update "
    + "from customer c1"
    + "     join address a1 on c1.address_id=a1.address_id"
    + "     join city ci1 on ci1.city_id = a1.city_id"
    + "     join country co1 on co1.country_id=ci1.country_id"
    + "     join country co2 on co1.country_id = co2.country_id"
    + "     join city ci2  on co2.country_id=ci2.country_id"
    + "     join address a2 on ci2.city_id=a2.city_id"
    + "     WHERE a2.address_id = :a_id", Customer.class);

  query.setParameter("a_id", addressId);
  List<Customer> c = query.getResultList();
  return c;
}
```

Hibernate: Refactoring 3 - Performance

- Client execution time: 1,749s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    41577      1.06       1.74          0         26          0           0
Execute  41577      5.82       6.55          0          0          0           0
Fetch   685422     85.19     117.90          0    3896141          0     6576043
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   768576     92.08     126.20          0    3896167          0     6576043

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message from client                685430        0.72       1570.53
```

Hibernate: Refactoring 3 - Performance

Step	Distinct SQL_IDs	Elapsed Time (s)	No. Executions
get films from similar customers	21	71.339	10000
get films to display	69	44.003	10000
get similar customers 1 step	1	9.933 ((was: 16.165 + 946)	10000
get target customer NEW bind var	1	0.803	10000
select film0_.FILM_ID as FILM_ID1_7_0_, film0_.DESCRIPTION	1	0.090	958
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.A	1	0.068	599
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_	1	0.026	2
get customer preferred category	1	0.007	15

Hibernate: Refactoring 3 - Summary

- Same as with jOOQ, the refactored query is faster than both original queries taken together
- Again same as with jOOQ, there is no overall win as overall response time is dominated by the display films queries …
- *“Let’s approach the decisive step ...”*

Hibernate: Refactoring 4: Combining Steps 4 & 5

- Step 4 new: Get films to display (matching all criteria) in 1 step

```
List<Film> films = getFilmInfo1Step(em, similarCustomers, category);
```

```
private List<Film> getFilmInfo1Step(EntityManager em, List<Customer> similarCustomers, Category category) {

  Query query = em.createNativeQuery("select distinct f.* from film f "
    + "join film_category fc on (f.film_id = fc.film_id) join category c on (c.category_id=fc.category_id)"
    + " join inventory i on (f.film_id = i.film_id) join rental r on r.inventory_id = i.inventory_id"
    + " where r.customer_id in :s and c.name= :c", Film.class);
  query.setParameter("s", similarCustomers);
  query.setParameter("c", category.getName());
  List<Film> films = query.getResultList();
  return films;
```

Hibernate: Refactoring 4 - Performance

- Client execution time: 487s
- And the database says...

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    30620      0.56       1.12          0         26          0           0
Execute  30620      1.46       2.10          0        135          0           0
Fetch    68511     91.83      99.82          0    2153157          0      466238
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   129751     93.86     103.06          0    2153318          0      466238

Elapsed times include waiting on following events:

Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
SQL*Net message from client                 68519        0.72        365.45
```

Hibernate: Refactoring 4 - Performance

Step	Distinct SQL_IDs	Elapsed Time (s)	No. Executions
get films to display 1 step	21	86.326 (was: 71.339 + 44.003)	10000
get similar customers 1 step	1	15.894	10000
get target customer NEW bind var	1	0.792	10000
select address0_.ADDRESS_ID as ADDRESS_ID1_2_0_, address0_.A	1	0.084	599
select store0_.STORE_ID as STORE_ID1_16_0_, store0_.ADDRESS_	1	0.025	2
get customer preferred category	1	0.006	15

Hibernate: Refactoring 4 - Summary

- With Hibernate, the achieved speedup is at 230%
- With Hibernate, speedup is even more due to reduced network traffic than with jOOQ:

	Original	Refactoring 4
SQL elapsed time (s)	131	103
SQL*Net from client (s)	1432	365

Case over - time for lessons learned …

Lessons learned …

- Don’t just tune SQL, tune the application
- Don’t assume that reduced number of steps / increased complexity of SQL will *always* lead to speedup
- *“Sherlock wait … informant’s calling …”*

Lessons learned …

- “For Hibernate, did you try read-only queries? No need for dirty checking when you don’t do modifications …”
- “Watson, we have work to do …”

Watch out for upcoming season 2 on www.trivadis.com

Thank you!

Sigrid Keydana

Tech Event, February 28 2016

trivadis
makes IT easier.