How Modern SQL Databases Come up with Algorithms that You Would Have Never Dreamed Of

Java Devs working with SQL for the first time

Me – @lukaseder

- Founder and CEO at Data Geekery
- Oracle Java Champion
- Oracle ACE

> SQL is a device whose mystery is only exceeded by its power!

Why do I talk about SQL?

SQL is the only ever successful, mainstream, and general-purpose 4GL (Fourth-Generation Programming Language)

And it is awesome!

That’s why the company is called “Oracle”

Why doesn’t anyone else talk about SQL?

Why doesn’t anyone else talk about SQL?

What is SQL?

Who has seen my other talk?

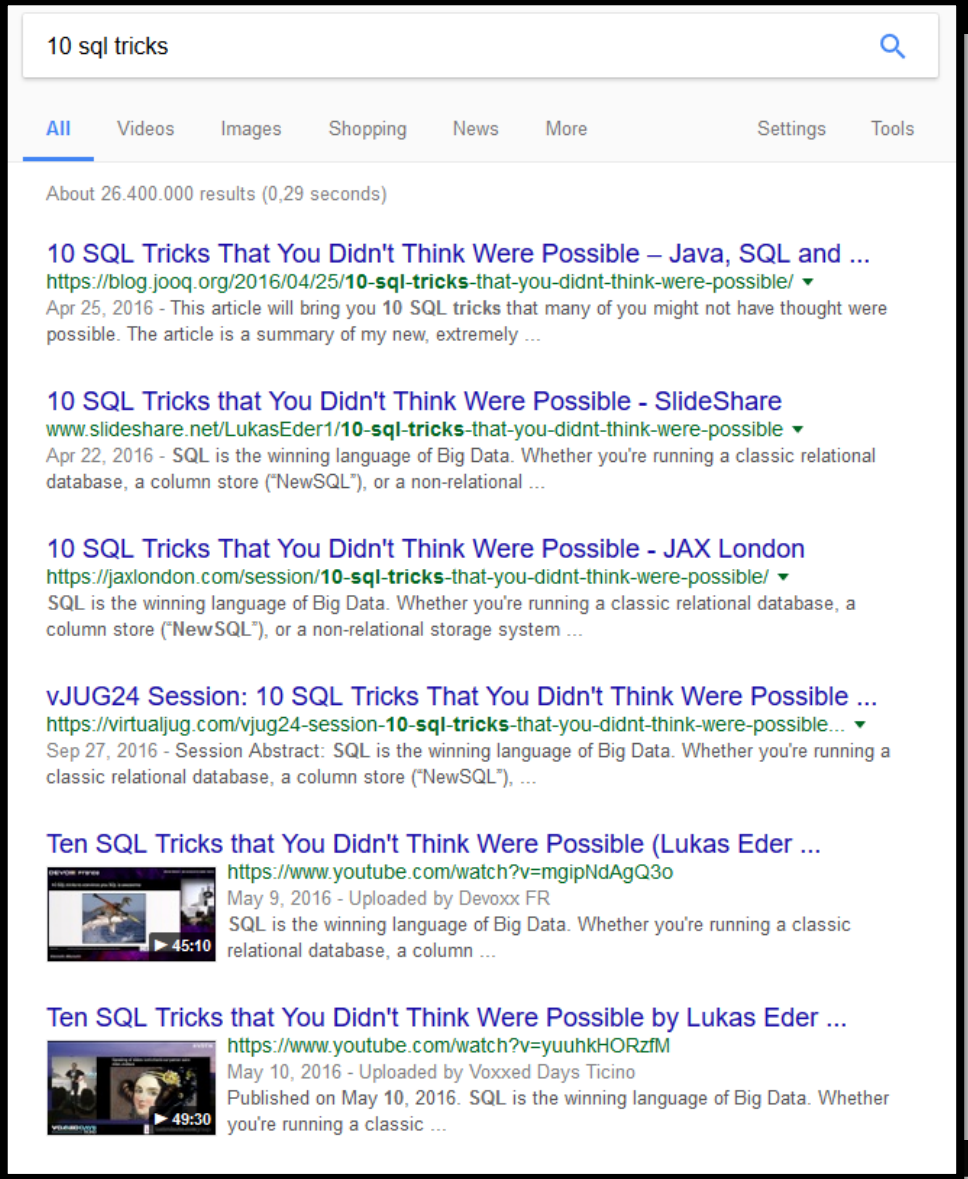
What is SQL?

I mean this one

10 SQL tricks to convince you SQL is awesome

10 SQL tricks to convince you SQL is awesome

Not that hard to find

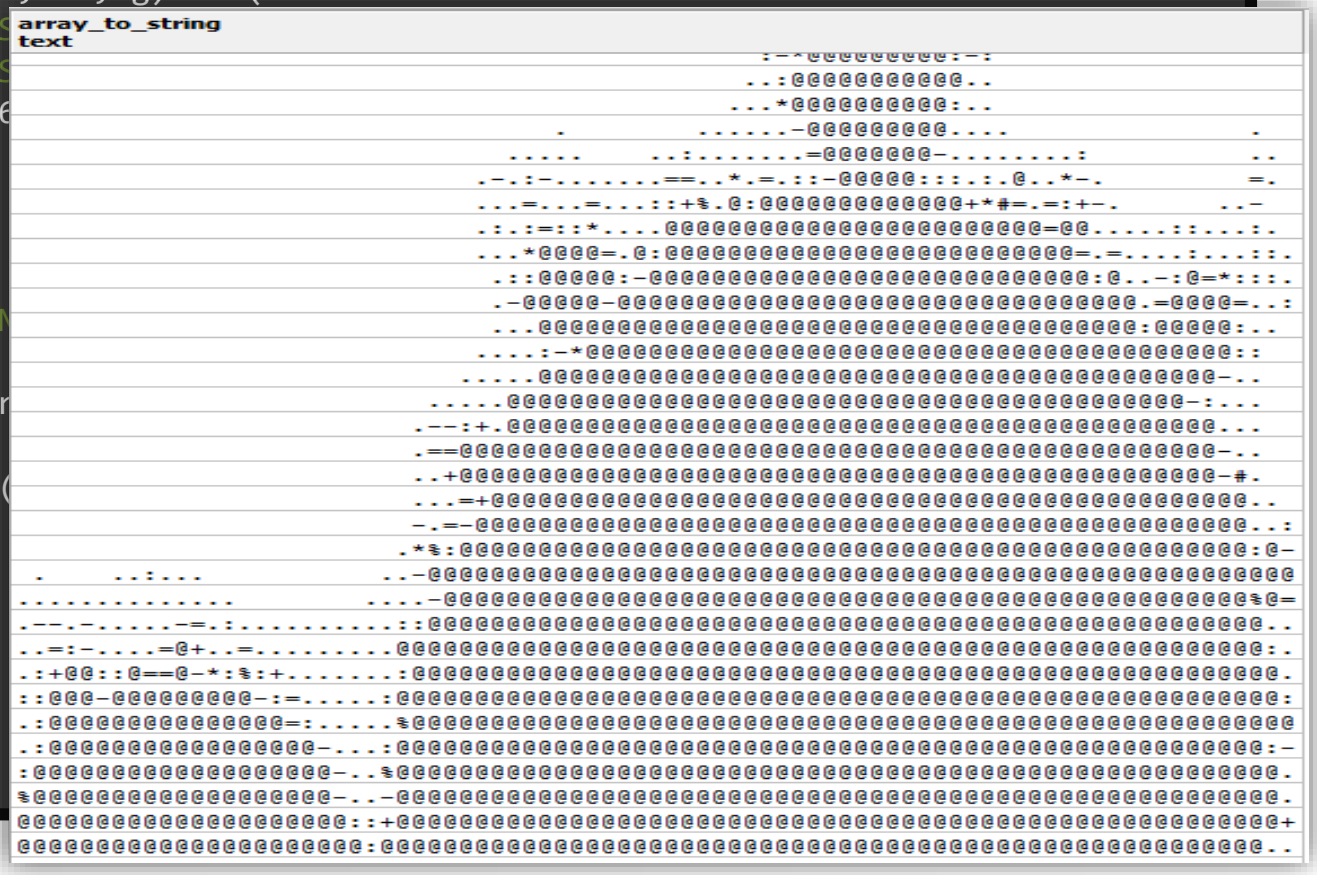


A SQL trick

```
-- Query from http://explainextended.com/2013/12/31/happy-new-year-5/
WITH RECURSIVE q(r, i, rx, ix, g) AS (
  SELECT r::DOUBLE PRECISION * 0.02, i::DOUBLE PRECISION * 0.02,
         .0::DOUBLE PRECISION      , .0::DOUBLE PRECISION, 0
  FROM generate_series(-60, 20) r, generate_series(-50, 50) i
  UNION ALL
  SELECT r, i, CASE WHEN abs(rx * rx + ix * ix) <= 2 THEN rx * rx - ix * ix END + r,
               CASE WHEN abs(rx * rx + ix * ix) <= 2 THEN 2 * rx * ix END + i, g + 1
  FROM q
  WHERE rx IS NOT NULL AND g < 99
)
SELECT array_to_string(array_agg(s ORDER BY r), '')
FROM (
  SELECT i, r, substring(' .:-=+*#%@', max(g) / 10 + 1, 1) s
  FROM q
  GROUP BY i, r
) q
GROUP BY i
ORDER BY i
```

A SQL trick: Generating the Mandelbrot Set

```
-- Query from http://explainextended.com/2013/12/31/happy-new-year-5/
WITH RECURSIVE q(r, i, rx, ix, g) AS (
  SELECT r::DOUBLE PRECIS
         .0::DOUBLE PRECIS
  FROM generate_series(-6
  UNION ALL
  SELECT r, i, CASE WHEN
               CASE WHEN
  FROM q
  WHERE rx IS NOT NULL AN
)
SELECT array_to_string(ar
FROM (
  SELECT i, r, substring(
  FROM q
  GROUP BY i, r
) q
GROUP BY i
ORDER BY i
```



Computation Engine

Your app is sitting on a Ferrari-style computation engine!

What is SQL?

SQL is the original microservice

Just install a single stored procedure in an Oracle XE instance, deploy, done.

What is SQL?

SQL is the original blockchain

What is SQL?

SQL is the original blockchain

```
WITH chain(n, block) AS (
  SELECT 1, standard_hash('Whee', 'MD5')
  FROM dual
  UNION ALL
  SELECT n + 1, standard_hash(block, 'MD5')
  FROM chain WHERE n < 100
)
SELECT block FROM chain
```

BLOCK
A09C8369625100B11BAC2CB3EEC8985A
A1E96C4FC5012D6ACE81118AA70B936E
CC019C8FCFE9A1FB700BE3F29A0F8816
60D1FCFE1EB65E461C5222209B1A9B97
1FA4132BC9C80AB44845C32320BE9166
D6F6393A045730DB96E2A28B67C48B3F
0EEF96F82CEA067BBD98243FDBC80632
DC0E28E54940B64C3A97F70B29C2D576
31626D5A0EB42E7FC4A33A8FEADF1EEA
C954EC9EA210DE59EC0A966A3AF3000D
873A45211E48A9AF5238A0FA4A3E7923
9A5123423B8E53B1C60528067F81317E
B7187EB08A2CE3EBD6468B5C6E323EDE
544D4106F79F87AB3ACA94B2779ED170
BF1908B978ACA29AFEFF8DDE7A0310B0
874C09624A08A15E57186F593F7BD812

Idea credit: @rotnroll666

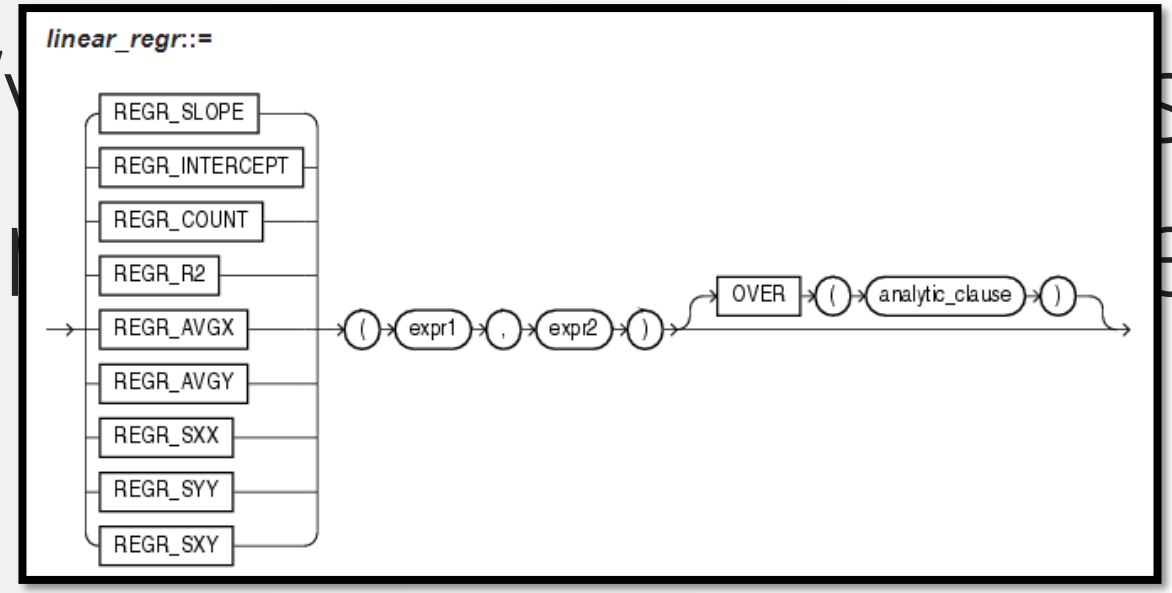
What is SQL?

SQL is the original ML language

What is SQL?

SQL is the original ML language

We'v sion fu es!



But today…

But today, we'll talk about more basic SQL

But today…

But today, we'll talk about more basic SQL (still awesome)

But today…

Is this SQL?

Who can handle this SQL statement here?

```
SELECT *
FROM   person
WHERE  id = 42
```

Easy stuff

Perfect

This talk is for you.

What will I talk about?

- SQL is awesome
- SQL is productive
- SQL is fast

What will I talk about?

- ~~SQL is awesome~~

- SQL is productive
- SQL is fast

What will I talk about?

- ~~SQL is awesome~~

- SQL is productive
- SQL is fast

There are two ways of proving this

There are two ways of proving this

1. By example

There are two ways of proving this

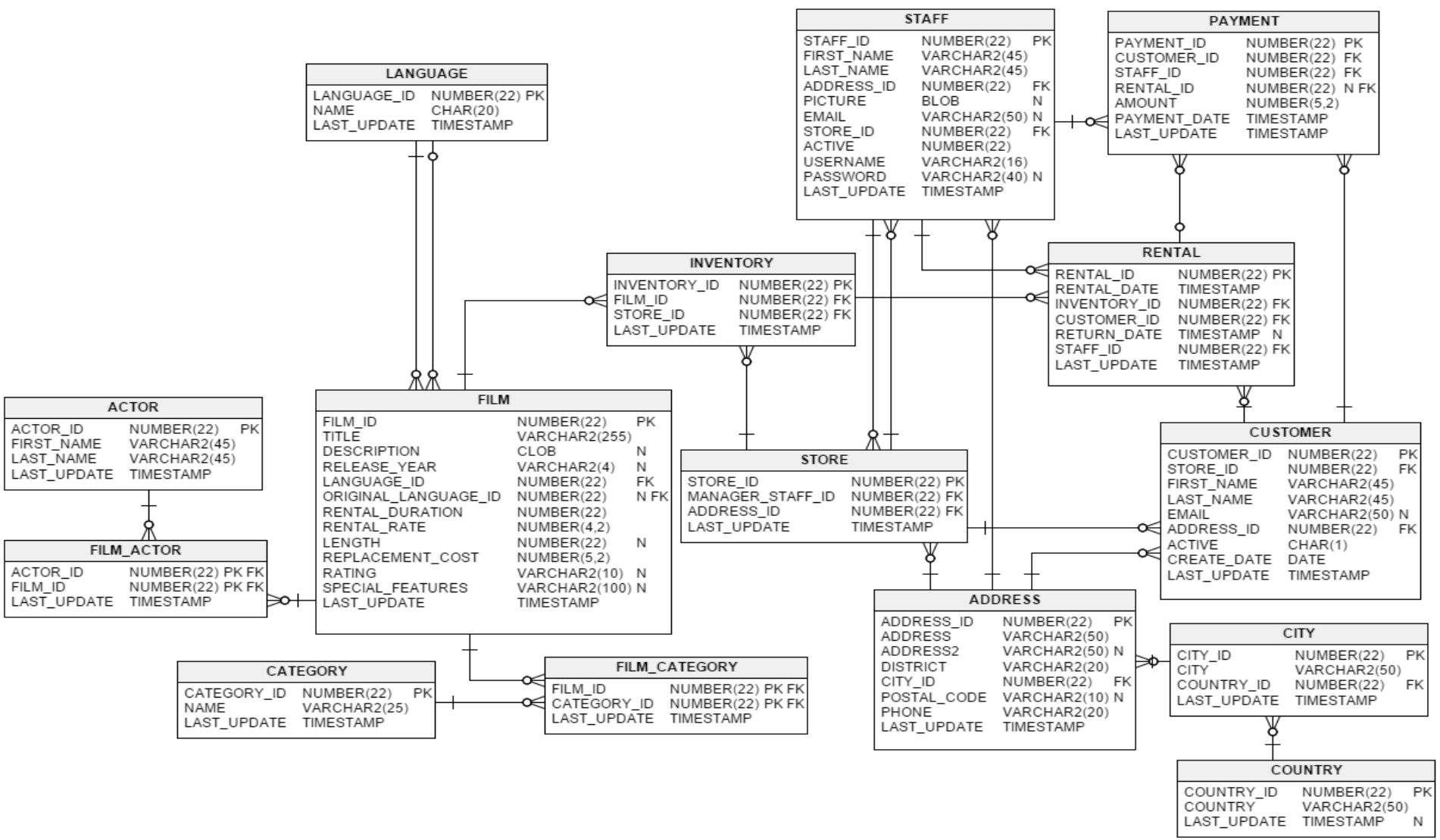
1. By example
2. By alternative (Java)

Disclaimer

This is going to be a high level, conceptual story about SQL

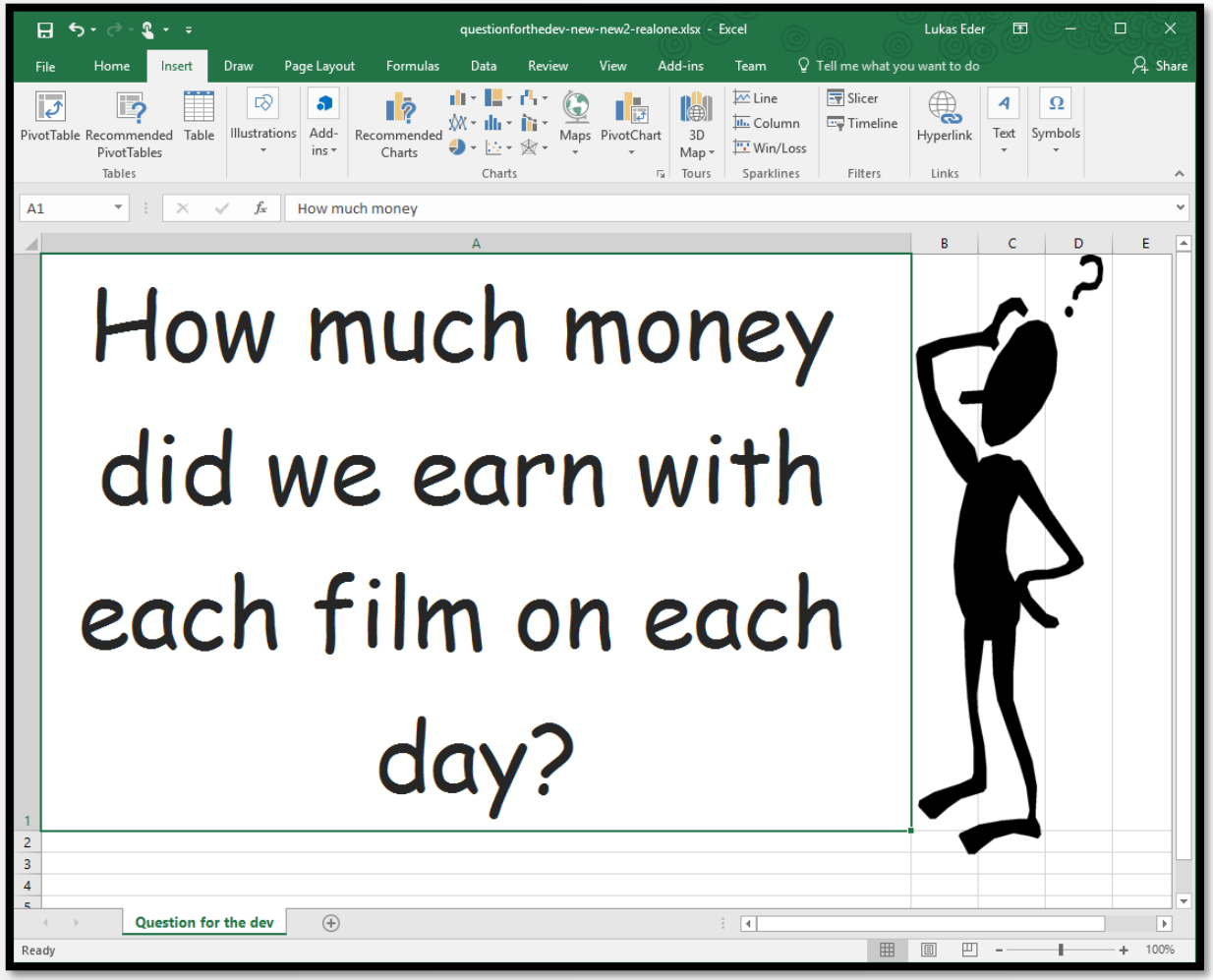
We’re not going into deeper levels.

Business is asking



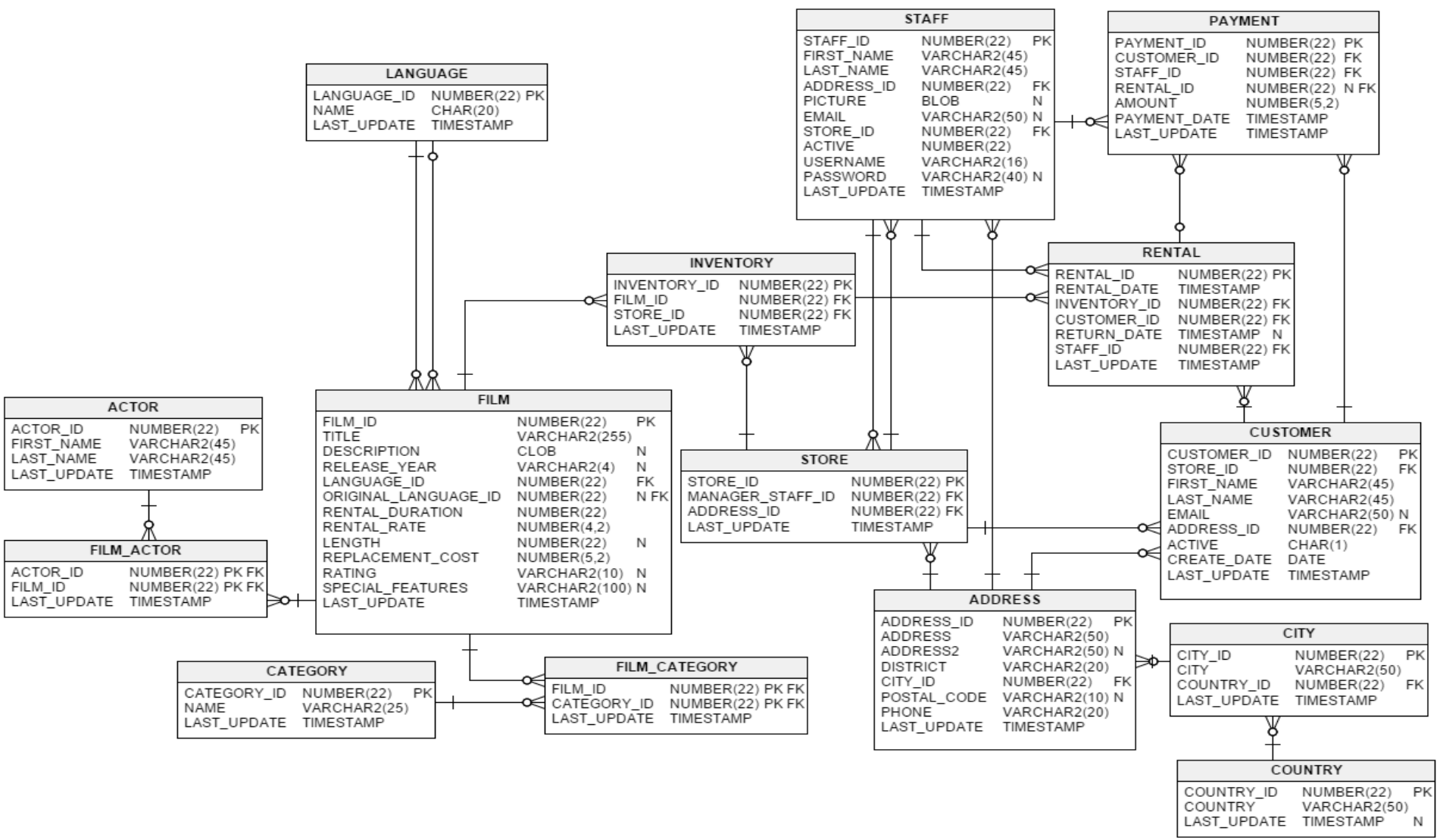
Business is asking

Business is asking

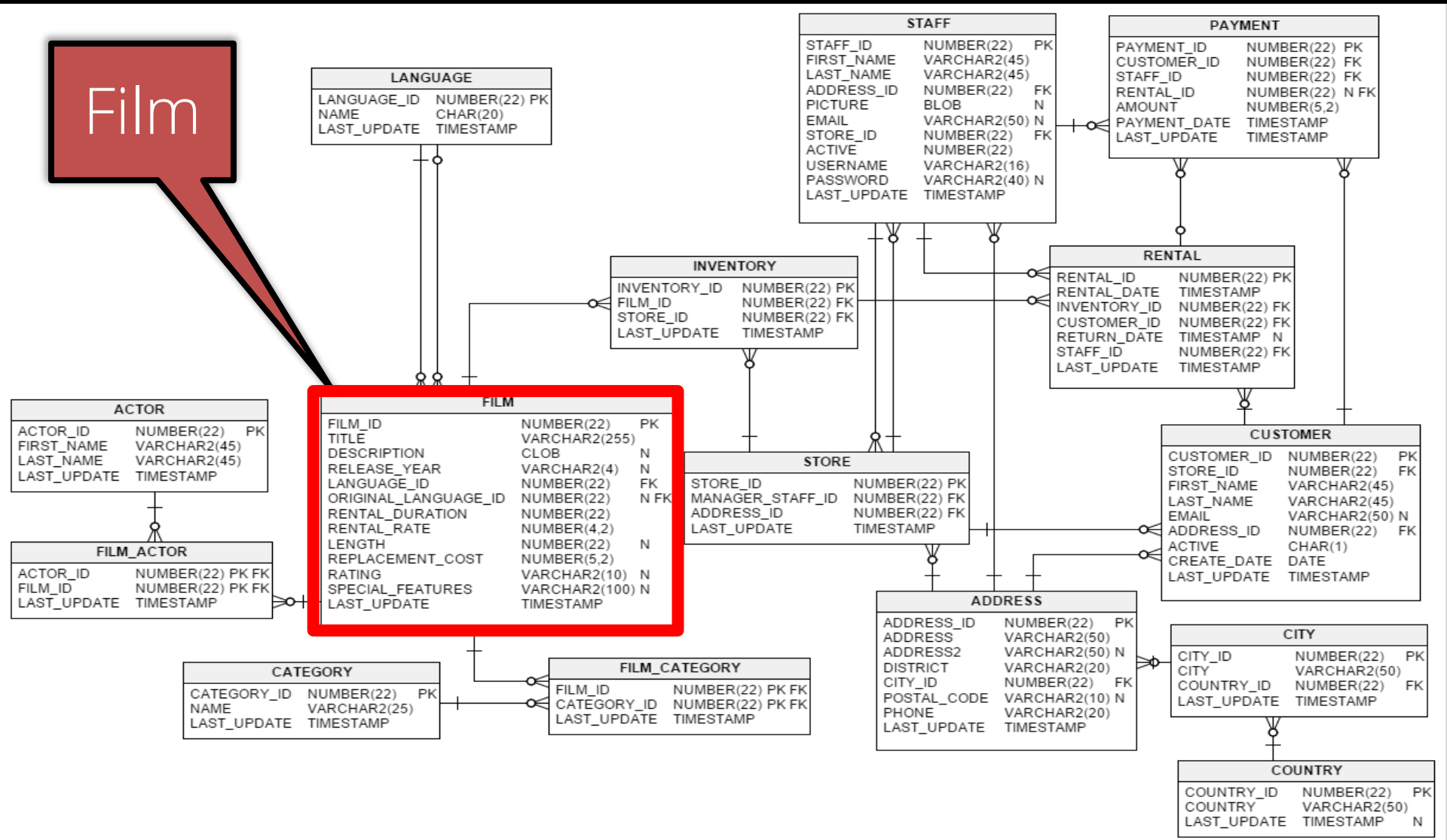


Business is asking

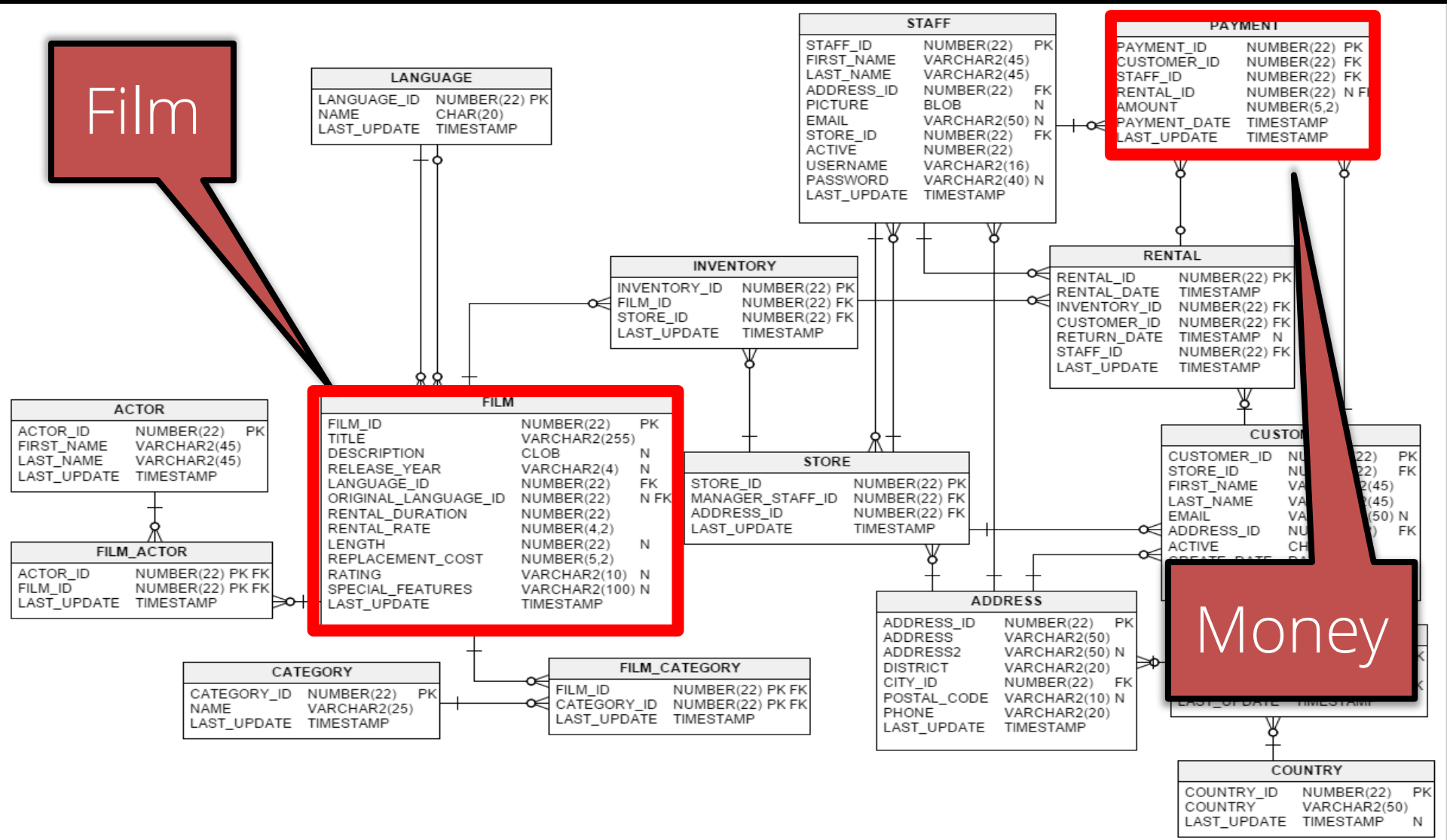
Business is asking



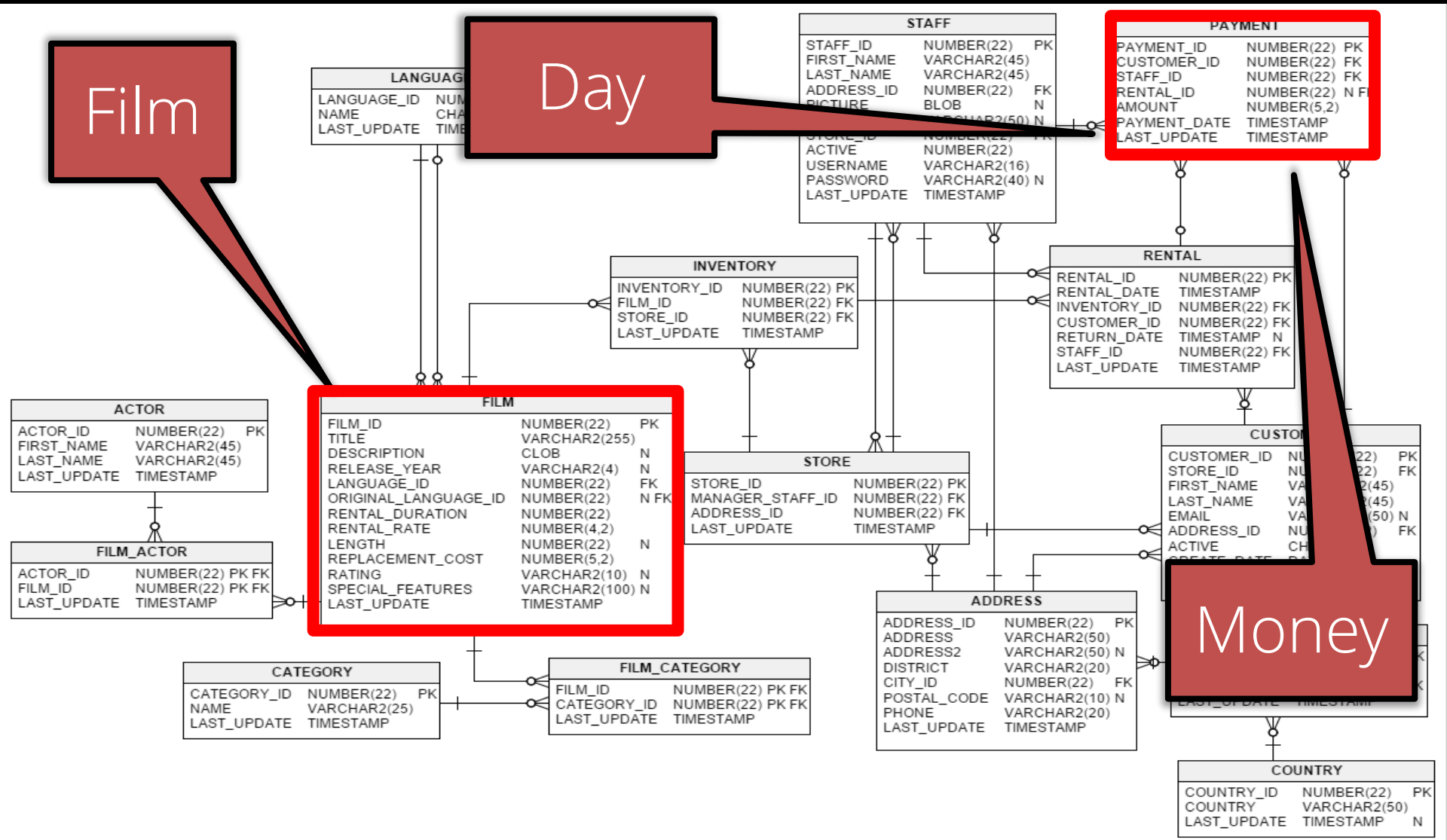
Business is asking



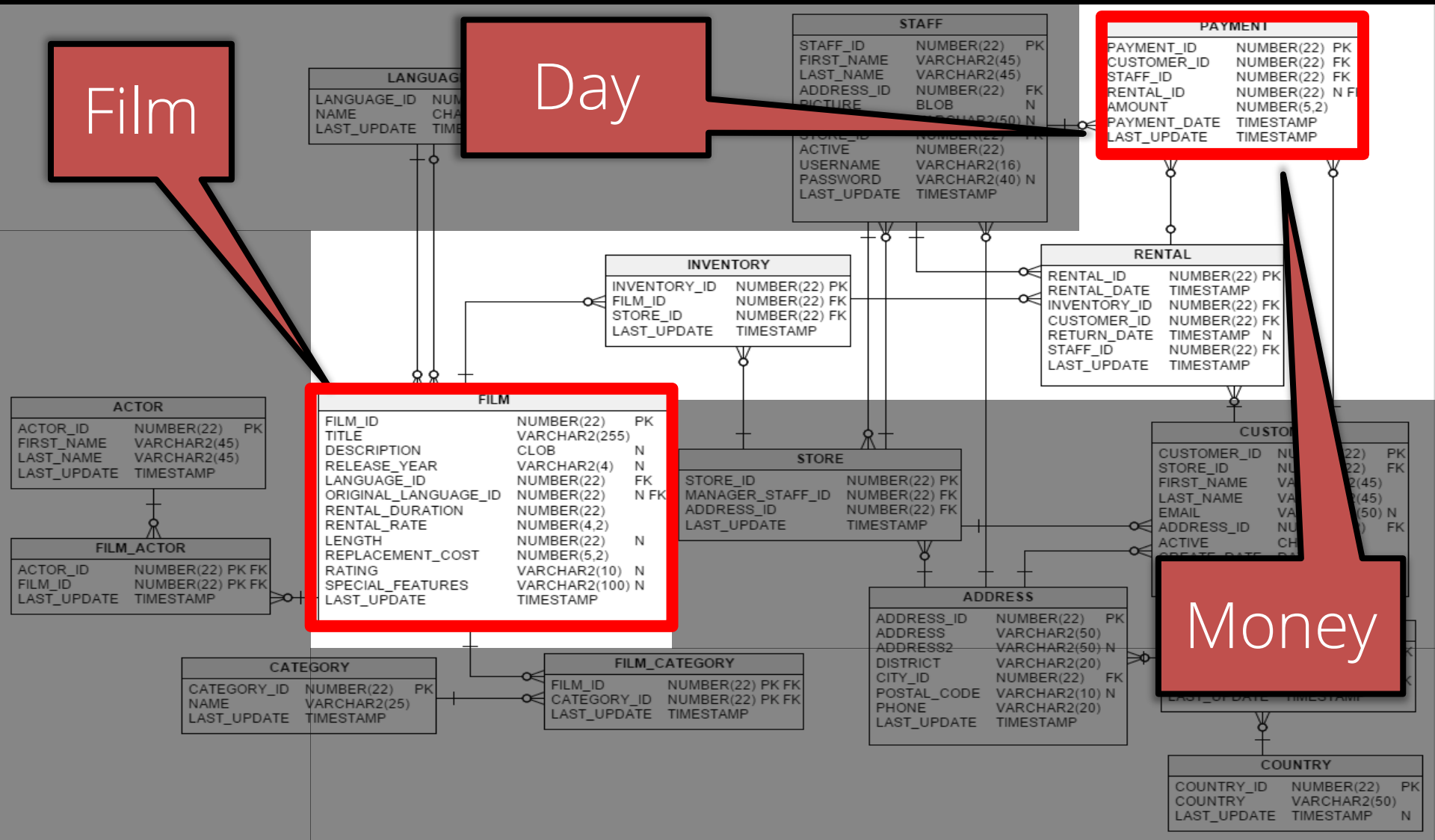
Business is asking



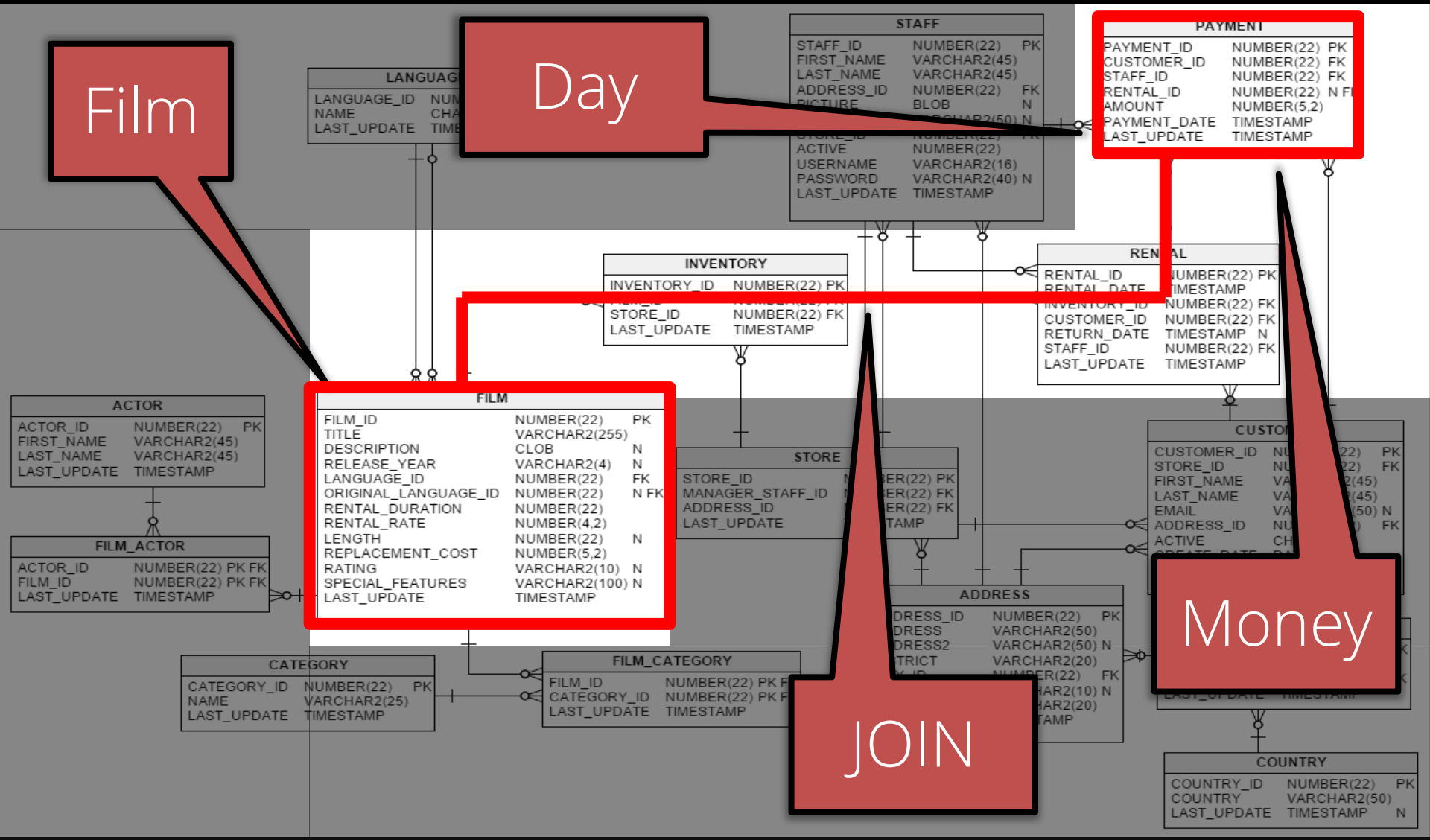
Business is asking



Business is asking



Business is asking



No problemo

Easy, right?

```
SELECT
  title, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, payment_date
ORDER BY title, payment_date
```

Easy, right?

```
SELECT
  title, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment (rental_id)
GROUP BY
ORDER BY
```

Money *per film*

Easy, right?

Relationship film ↔ money

```
SELECT
  title, payme      ate, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, payment_date
ORDER BY title, payment_date
```

Easy, right?

Aggregation per film / date

```
SEL
  t    , payment_date, SUM(amount)
FRO   ilm
JOI   inventory USING (film_id)
JOI   rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, payment_date
ORDER BY title, payment_date
```

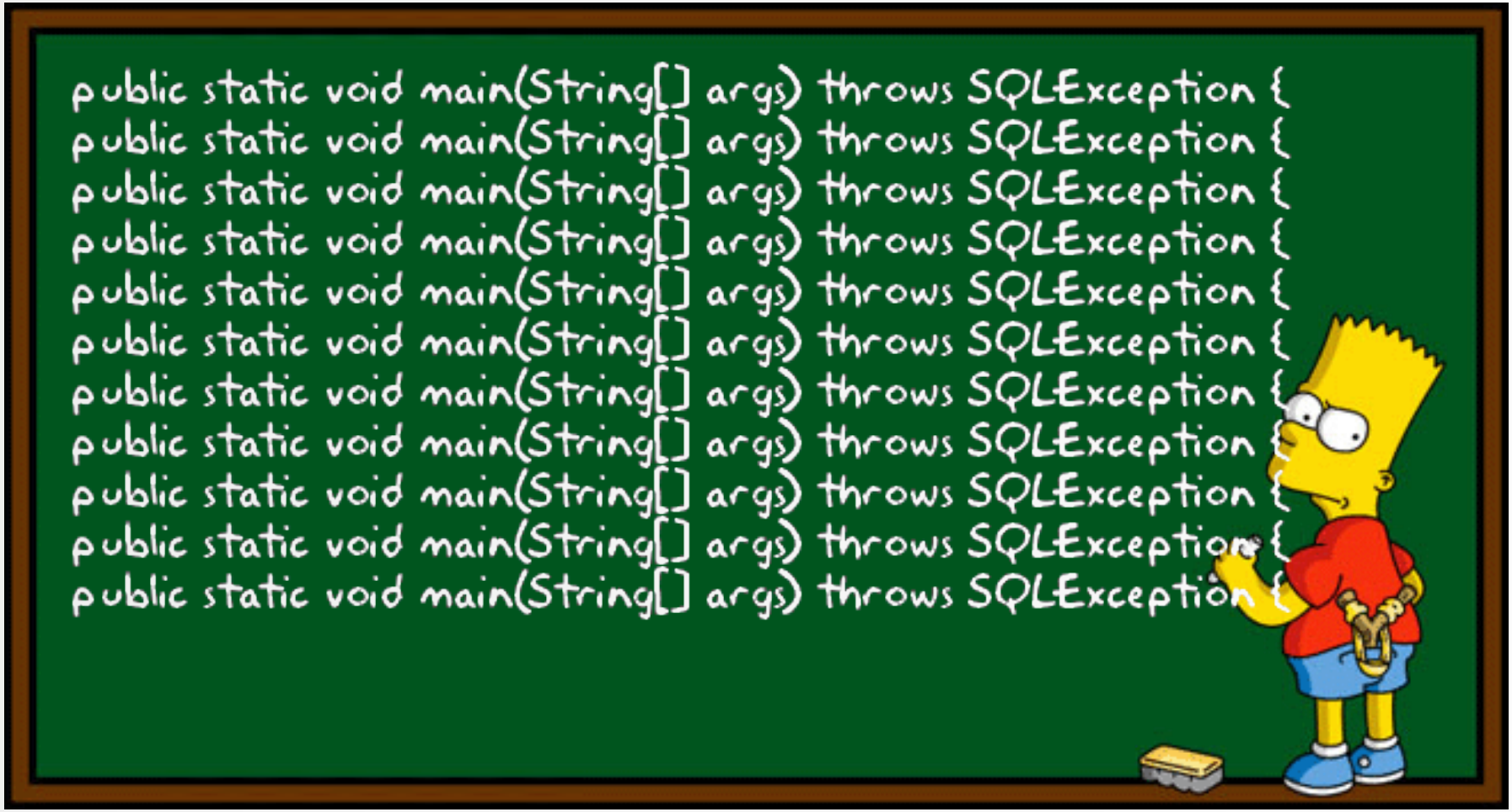
Easy, right?

```
SELECT
  title, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, payment_date
ORDER BY title, payment_date
```

Imperative style

Let’s write this in «classic» Java

“Classic Java”



Let’s get rolling

```
class Film {
}
```

Let’s get rolling

You still with me?

Obviously, Film is a tuple

```
class Film {
   int filmId;
   String title;
}
```

Because, Java

```
class Film {
  int filmId;
  String title;

  int getFilmId() {
    return filmId;
  }

  ...
}
```

Never forget these, of course

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Abstractions, ladies and gentlemen!

```
class FilmFactory {
  Film newFilm() { ... }
}
```

Abstractions, ladies and gentlemen!

```
class FilmFactory {
  Film newFilm() { ... }
}

class FilmFactoryBuilder {
  FilmFactory newFilmFactory() { ... }
}
```

Abstractions, ladies and gentlemen!

```
interface FilmFactory {
  Film newFilm();
}

interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}
```

Just to be sure, super generic!

```
interface FilmFactory {
  Film newFilm();
}

interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}

class FilmFactoryImpl implements FilmFactory {
  @Override
  public Film newFilm() { ... }
}

class FilmFactoryBuilderImpl implements FilmFactoryBuilder {
  @Override
  public FilmFactory newFilmFactory() { ... }
}
```

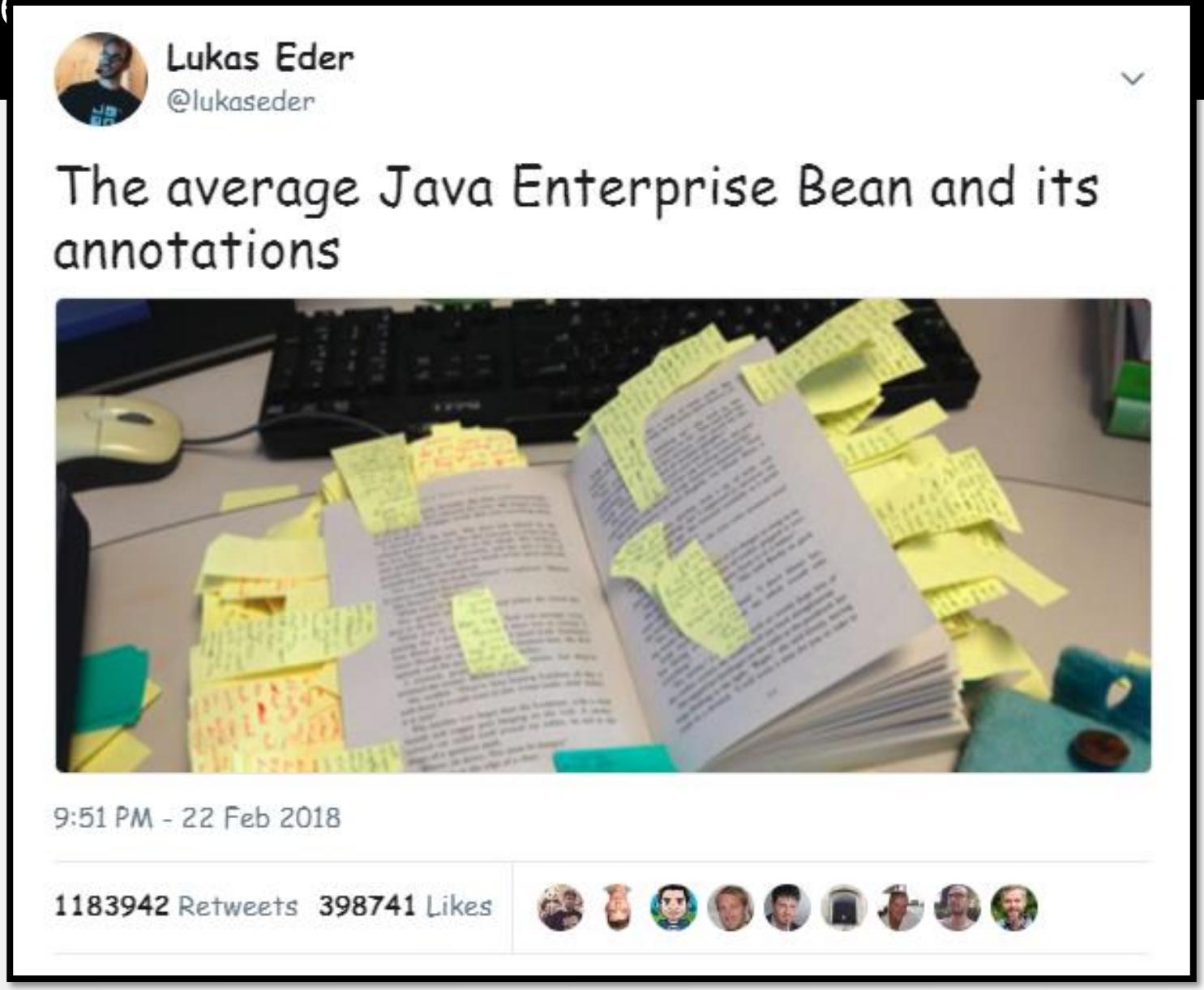
Some Spring

```
interface FilmFactory {
  Film newFilm();
}

interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}

@Bean
class FilmFactoryImpl implements FilmFactory {
  @Override
  public Film newFilm() { ... }
}

@Bean
class FilmFactoryBuilderImpl implements FilmFactoryBuilder {
  @Override
  @EnableCaching
  public FilmFactory newFilmFactory() { ... }
}
```

Some Spring, and a little Lombok

```
@FunctionalInterface
interface FilmFactory {
  Film newFilm();
}

@FunctionalInterface
interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}

@Bean
@EnableAspectJAutoProxy
@EnableAutoConfiguration
class FilmFactoryImpl implements FilmFactory {
  @Override
  public Film newFilm() { ... }
}

@Bean
@Configuration
@NoArgsConstructor
class FilmFactoryBuilderImpl implements FilmFactoryBuilder {
  @Override
  @EnableCaching
  public FilmFactory newFilmFactory() { ... }
}
```

Some Spring and a little Lombok



Oh, what the heck

```
@FunctionalInterface
interface FilmFactory {
  Film newFilm();
}

@FunctionalInterface
interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}

@Bean
@EnableAspectJAutoProxy
@EnableAutoConfiguration
class FilmFactoryImpl implements FilmFactory {
  @Override
  public Film newFilm() { ... }
}

@Bean
@Configuration
@NoArgsConstructor @Cloneable @SneakyThrows
@ResultSetMapping @ConcurrentInitialiserProxyFactory
@SpringPrefetchAdapter @AdapterProxyBeanMethod @AdapterBeanProxyMethod
class FilmFactoryBuilderImpl implements FilmFactoryBuilder {
  @Override
  @EnableCaching
  public FilmFactory newFilmFactory() { ... }
}
```

Oh, what the heck

```
@FunctionalInterface
interface FilmFactory {
  Film newFilm();
}

@FunctionalInterface
interface FilmFactoryBuilder {
  FilmFactory newFilmFactory();
}

@Bean
@EnableAspectJAutoProxy
@EnableAutoConfiguration
class FilmFactoryImpl implements FilmFactory {
  @Override
  @ThisMethodDidntHaveAnnotationsYet @OKWeStillHaveSomeSpaceLeft @LetsSeeIfWeCanReachThePrintMargin
  public Film newFilm() { ... }
}

@Bean
@Configuration
@NoArgsConstructor @Cloneable @SneakyThrows
@ResultSetMapping @ConcurrentInitialiserProxyFactory
@SpringPrefetchAdapter @AdapterProxyBeanMethod @AdapterBeanProxyMethod
@MoreAndMoreAnnotations @CanYouEvenReadThis
@IsThereStillAnyRealLogicLeft
@ThisJokeNeverGetsLame @Annotatiomania @WhoWillBeNextYearsAnnotatiomaniac
class FilmFactoryBuilderImpl implements FilmFactoryBuilder {
  @Override
  @EnableCaching
  public FilmFactory newFilmFactory() { ... }
}
```

Average enterprise bean is visible from space

Average enterprise bean is visible from space

Enough of this. More info here

www.annotatiomania.com

Hmm, relationships?

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  // Hmm... Inventory? Store? Rental? Payment?

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Let’s assume, we copy relational model 1:1

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  List<Inventory> getInventories() { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Let’s assume, we copy relational model 1:1

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  @OneToMany(mappedBy = "film")
  List<Inventory> getInventories() { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Let's assume, we copy relational model 1:1

Anyone seen these lovely things?

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ...
  void setFilmId(int fi...  { ... }
  String getTitle() { ...
  void setTitle(String t...le) { ... }

  @OneToMany(mappedBy = "film")
  List<Inventory> getInventories() { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Let's assume, we copy relational model 1:1

How about this? Care discuss this?

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  @OneToMany(mappedBy = "film", fetch = FetchType.EAGER)
  List<Inventory> getInventories() { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Let's assume, we copy relational model 1:1

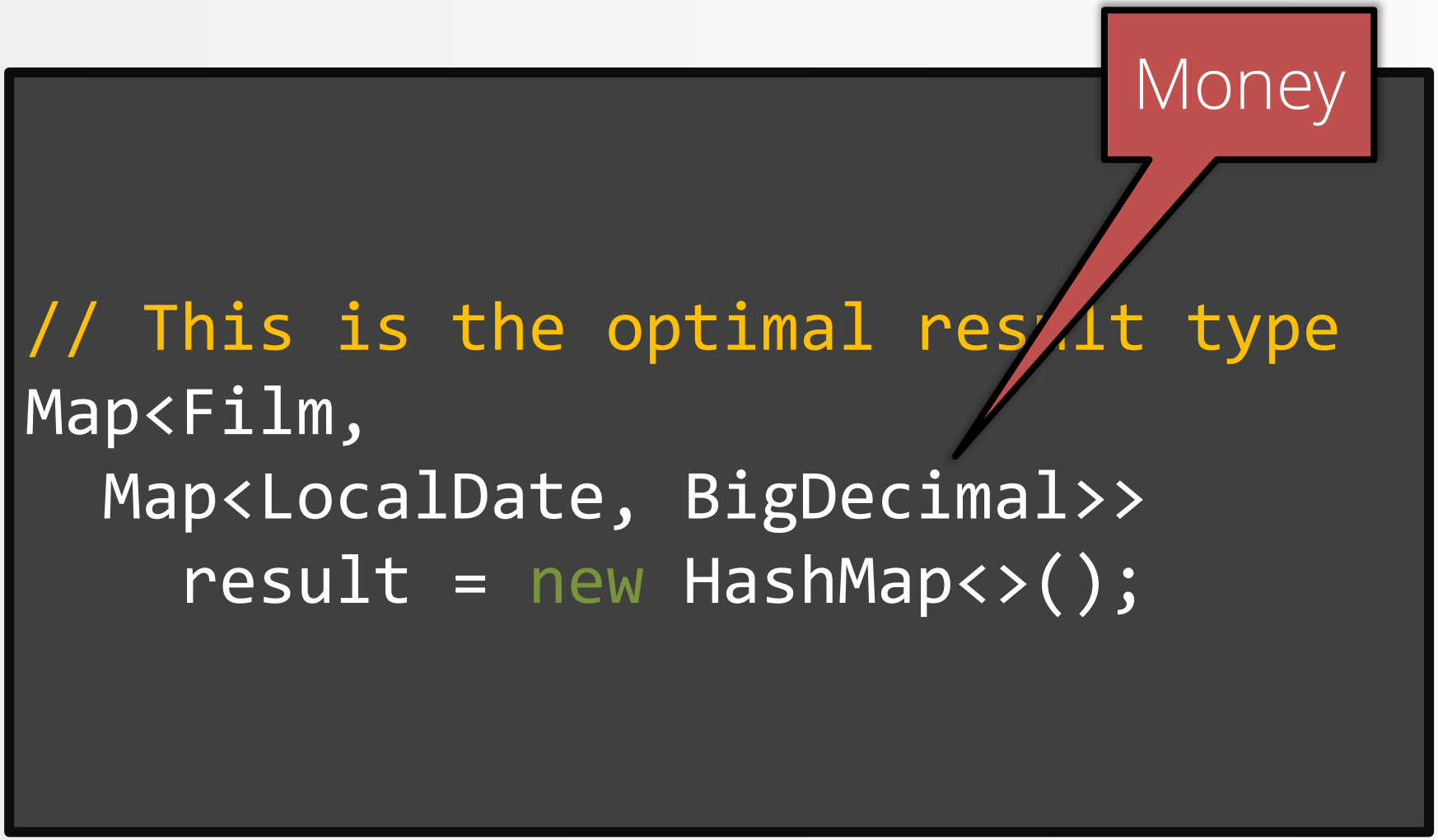
How about this? Care discuss this?

```
class Film {
  int filmId;
  String title;

  int getFilmId() { ... }
  void setFilmId(int filmId) { ... }
  String getTitle() { ... }
  void setTitle(String title) { ... }

  @OneToMany(mappedBy = "film", fetch = FetchType.LAZY)
  List<Inventory> getInventories() { ... }

  int hashCode() { ... }
  boolean equals(Object o) { ... }
  String toString() { ... }

  ...
}
```

Now, calculate. This be the result type

```
// This is the optimal result type
Map<Film,
  Map<LocalDate, BigDecimal>>
    result = new HashMap<>();
```

Now, calculate. This be the result type



Now, calculate. This be the result type

Per film

Money

```
// This is the optimal result type
Map<Film,
   Map<LocalDate, BigDecimal>>
     result = new HashMap<>();
```

Now, calculate. This be the result type

Per film

Per date

Money

```
// This is the optimal result type
Map<Film,
   Map<LocalDate, BigDecimal>>
     result = new HashMap<>();
```

Money, eh?

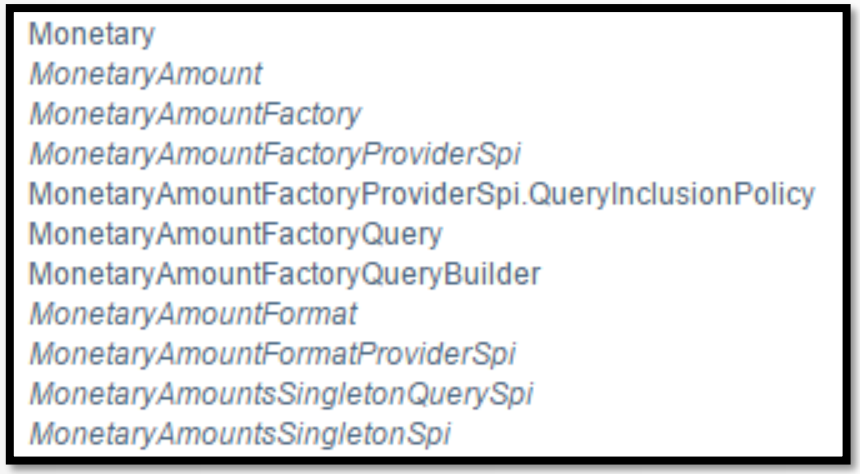
JSR 354 FTW!

(Java Money API)

Money, eh?

JSR 354 FTW!

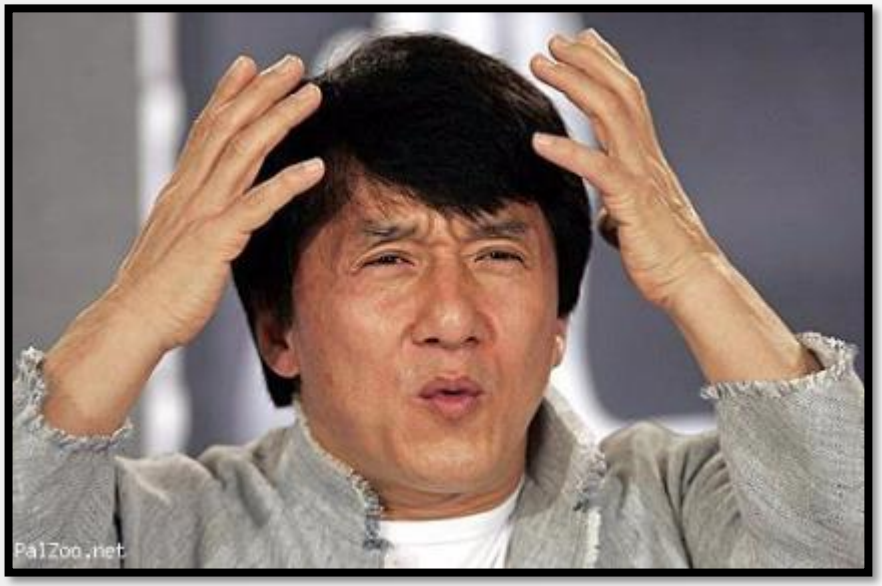
(Java Money API)



Money, eh?

JSR 354 FTW!

(Java Money API)



Monetary
MonetaryAmount
MonetaryAmountFactory
MonetaryAmountFactoryProviderSpi
MonetaryAmountFactoryProviderSpi.QueryInclusionPolicy
MonetaryAmountFactoryQuery
MonetaryAmountFactoryQueryBuilder
MonetaryAmountFormat
MonetaryAmountFormatProviderSpi
MonetaryAmountsSingletonQuerySpi
MonetaryAmountsSingletonSpi

Now, calculate. This be the result type

```
// This is the optimal result type
Map<Film,
  Map<LocalDate, BigDecimal>>
    result = new HashMap<>();
```

Now, calculate. Wait…

```
// Or this?
Map<Film,
   Map<LocalDate, BigDecimal>>
     result = new LinkedHashMap<>();
```

Now, calculate. Or better

```
// Or maybe this?
SortedMap<Film,
   SortedMap<LocalDate, BigDecimal>>
      result = new TreeMap<>();
```

Now, calculate. Aagh

```
// Kewl, wrote some library
MultiKeyMap<
   Film, LocalDate, BigDecimal>
     result = new MultiKeyMap<>();
```

Five hours later

Who in here has felt this way before?

Who in here has felt this way before?

Now let’s calculate, for real

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
```

Easy, so far

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
```

Easy, so far

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
// FIXME 2002-04-01 Will optimize this later
```

Easy, so far

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
// FIXME 2003-04-01 Will optimize this “later”
```

Easy, so far

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
// FIXME 2003-04-01 Will optimize this “later”
// FIXME 2005-04-01 URGENT! This is really slow!
```

Still following?

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films)
  ; // TODO
```

Still following?

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);

  // TODO
}
```

Still following?

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);

  // TODO
}
```

Aaah, lazy initialization!

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);

  if (daily == null) {
    daily = new HashMap<>();
    result.put(film, daily);
  }

  // TODO
}
```

Aaah, lazy initialization!

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.g
  if (daily == null) {
    daily = new HashMap<>();
    result.put(film, daily);
  }
  // TODO
}
```

Aaah, lazy initialization!

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);

  if (daily == null) {
    daily = new HashMap<>();
    result.put(film, daily);
  }

  // TODO
}
```

Let’s have that Map discussion again

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);

  if (daily == null) {
    daily = new LinkedHashMap<>(); // TODO: Better Map!
    result.put(film, daily);
  }

  // TODO
}
```

OK, next

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }

  // TODO
}
```

OK, now looping

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    ; // TODO
}
```

OK, now looping

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      ; // TODO
}
```

OK, now looping

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        ; // TODO
}
```

OK, now looping

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        ; // TODO
}
```

EAGER or LAZY?

OK, now looping

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        ; // TODO
}
```

Or, entity graph?

Entity Graph – How JPA folks write SQL

```
@NamedEntityGraph(
  name = "graph.film.inventories.rentals.payments",
  attributeNodes = @NamedAttributeNode(
    value = "inventories",
    subgraph = "inventories"
  ),
  subgraphs = {
    @NamedSubgraph(
      name = "inventories",
      attributeNodes = @NamedAttributeNode(
        value = "rentals",
        subgraph = "rentals"
      )
    ), ...
  }
)
```

Entity Graph – How JPA folks write SQL

```
@NamedEntityGraph(
  name = "graph.film.inventories.rentals.payments",
  attributeNodes = @NamedAttributeNode(
    value = "inventories",
    subgraph = "inventories"
  ),
  subgraphs = {
    @NamedSubgraph(
      name = "inventories",
      attributeNodes = @NamedAttributeNode(
        value = "rentals",
        subgraph = "rentals"
      )
    ), ...
  }
)
```

Remember this slide?

Your code now visible from Andromeda Galaxy

Your code now visible from Andromeda Galaxy

OK, now looping, done. Right?

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        daily.put(
          payment.getPaymentDate(),
          payment.getAmount());
}
```

OK, now looping, done. Wrong!

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments()) {
        LocalDate date = payment.getPaymentDate();
        BigDecimal amount = payment.getAmount();
        if (daily.containsKey(date))
          daily.put(date, daily.get(date).add(amount));
        else
          daily.put(date, amount);
      }
}
```

OK, now looping, done. Wrong!

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily = result.get(film);
  if (daily == null) { ... }
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments()) {
        LocalDate date = payment.getPaymentDate();
        BigDecimal amount = payment.getAmount();
        if (daily.containsKey(date))
          daily.put(date, daily.get(date).add(amount));
        else
          daily.put(date, amount);
      }
}
```

OK, now looping, done. OK, better with Java 8

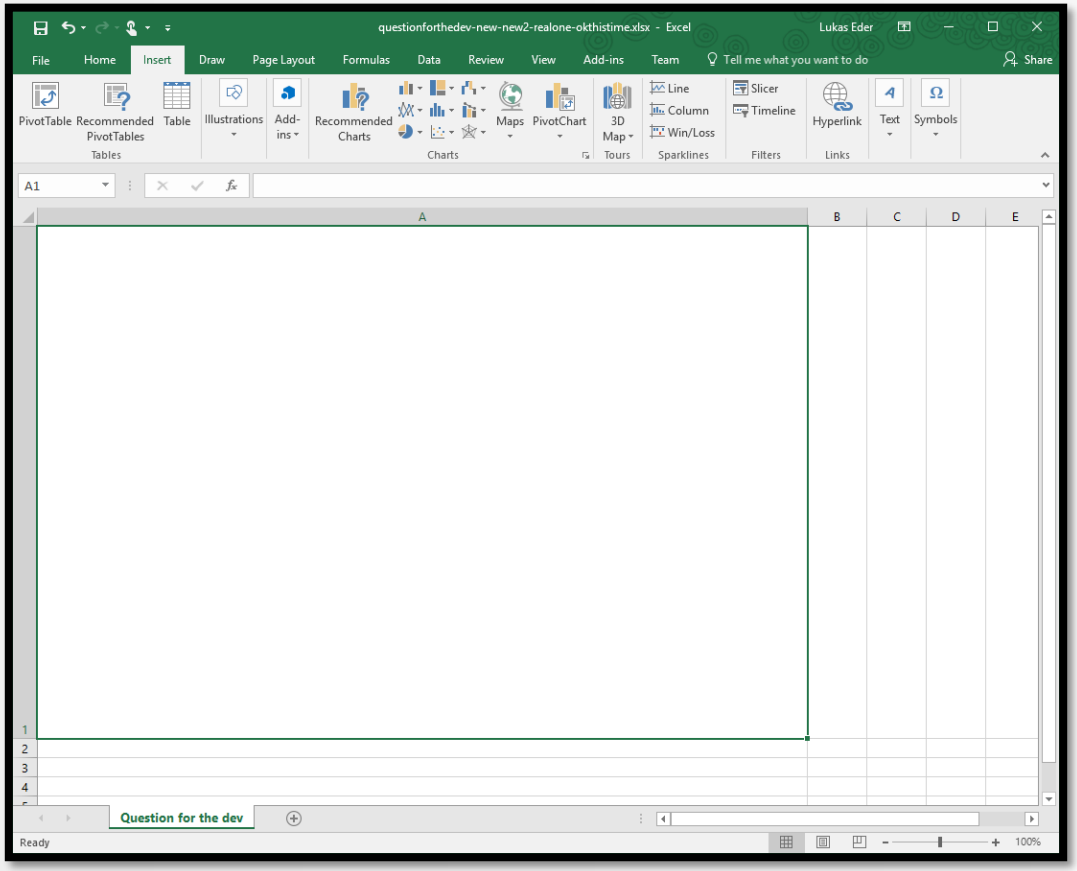
```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        daily.compute(
          payment.getPaymentDate(),
          (k, v) -> v == null
            ? payment.getAmount()
            : payment.getAmount().add(v));
}
```

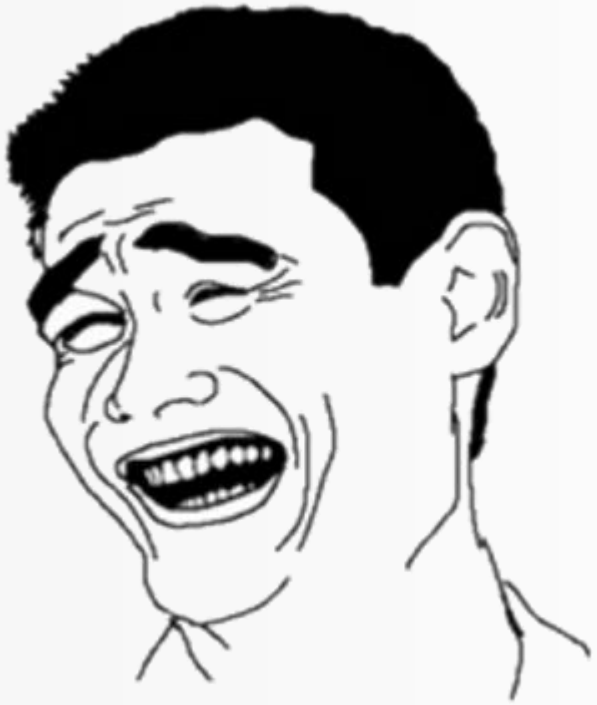
OK, now looping, done. OK, better with Java 8

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment payment : rental.getPayments())
        daily.compute(
          payment.getPaymentDate(),
          (k, v) -> v == null
            ? payment.getAmount()
            : payment.getAmount().add(v));
}
```

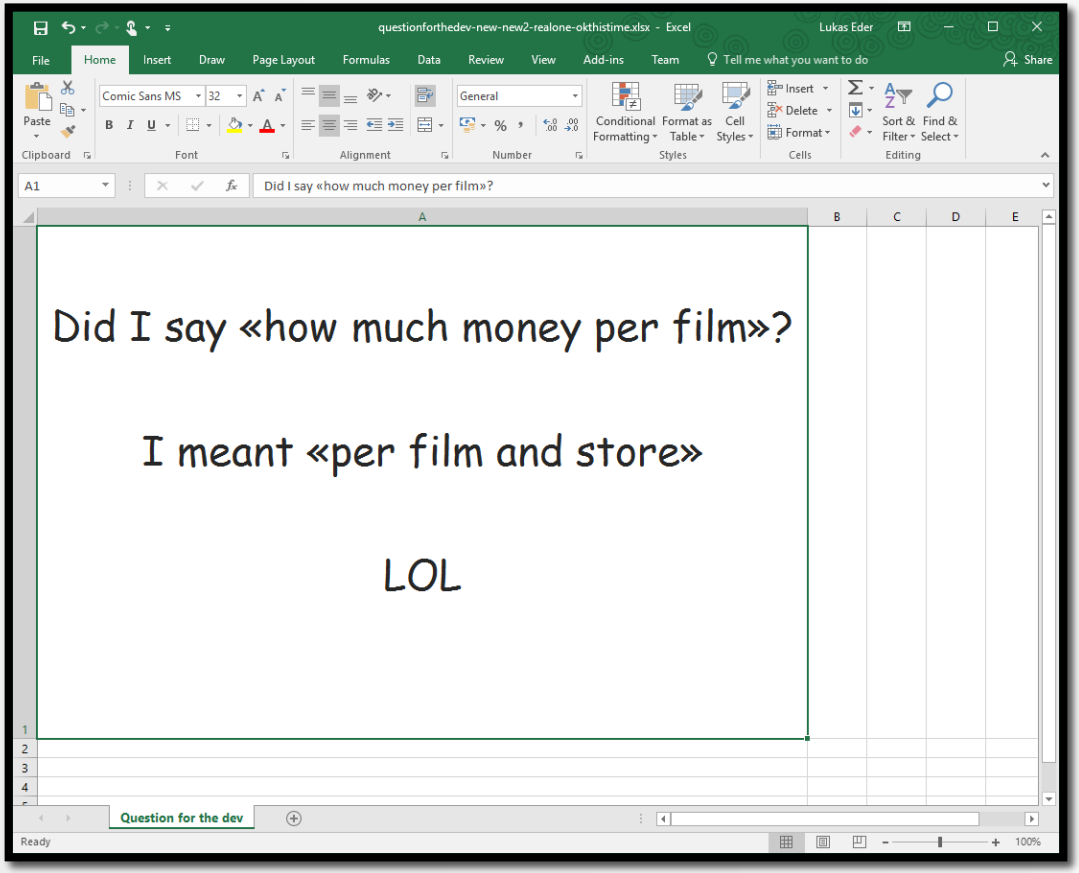
Feel good about yourself?

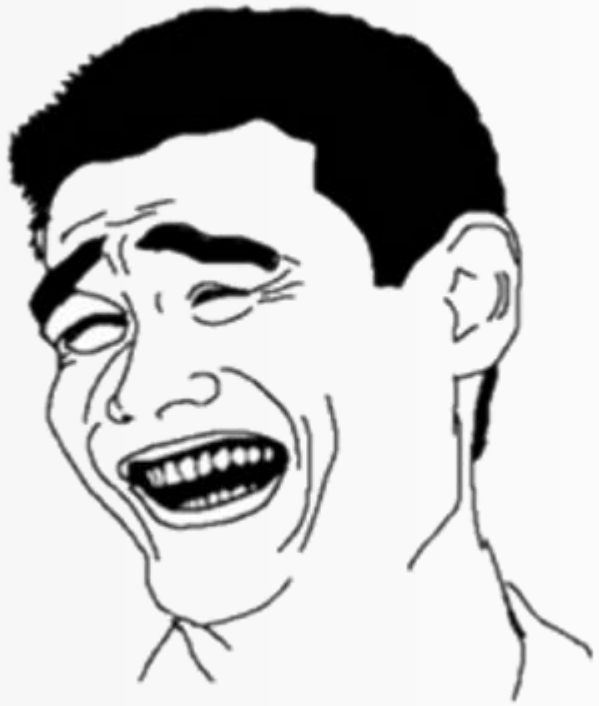
Business is asking again





Business is asking again





Programmer reactions

SQL Developer

Java Developer

Programmer reactions

SQL Developer

Java Developer

Programmer reactions

SQL Developer

Java Developer

SQL dev: I knew this was coming!

```
SELECT
  title, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, payment_date
ORDER BY title, payment_date
```

SQL dev: Swoosh!

```
SELECT
  title,              payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id,                 payment_date
ORDER BY title,                 payment_date
```

SQL dev: Joke's on you!

```
SELECT
  title, store_id, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Java dev: Gaaah, OK

```
Map<Film, Map<LocalDate, BigDecimal>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<LocalDate, BigDecimal> daily =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories())
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
}
```

Java dev: Let's make some room

```
Map<Film,              Map<LocalDate, BigDecimal> > result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
              Map<LocalDate, BigDecimal>  daily =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories())



    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));

}
```

Java dev: Let's see, if I tweak this and then…

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyPerStore =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories()) {
    Map<LocalDate, BigDecimal> daily =
      dailyPerStore.computeIfAbsent(
        inventory.getStoreId(), k -> new HashMap());
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
  }
}
```

Java dev: Let's see, if I tweak this and then…

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyPerStore =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories()) {
    Map<LocalDate, BigDecimal> daily =
      dailyPerStore.computeIfAbsent(
        inventory.getStoreId(), k -> new HashMap());
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
  }
}
```

Java dev: Let's see, if I tweak this and then…

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = r lusOneLoadAllFilms(); // TODO
for (Film film : fi s) {
  Map<Integer, Map  calDate, BigDecimal>> dailyPerStore =
    result.comput  sent(film, k -> new HashMap<>());
  for (Inventory  tory : film.getInventories()) {
    Map<LocalDa  Decimal> daily =
      dailyPer  nputeIfAbsent(
        invent  oreId(), k -> new HashMap());
    for (Rent  : inventory.getRentals())
      for (P  ental.getPayments())
        dai
                te(),
                 null ? p.getAmount() : p.getAmount().add(v));
```

Architect:

Java dev: Let's see, if I tweak this and then

(that's this guy)

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...
List<Film> films = r lusOneLoadAllFilms(); // TODO
for (Film film : fi s) {
  Map<Integer, Map  calDate, BigDecimal>> dailyPe
    result.comput  sent(film, k -> new HashMap<
  for (Inventory  tory : film.getInventories()
    Map<LocalDa  Decimal> daily =
      dailyPer  nputeIfAbsent(
        invent  oreId(), k -> new HashMap(
    for (Rent  : inventory.getRentals())
      for (P  ental.getPayments())
        dai
                te(),
                 null ? p.getAmount() : p.getAmount().add(v));
```

Architect:

Java dev: Let's see, if I tweak this and then…

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = r lusOneLoadAllFilms(); // TODO
for (Film film : fi s) {
  Map<Integer, Map  calDate, BigDecimal>> dailyP
    result.comput  sent(film, k -> new HashMap
  for (Inventory  tory : film.getInventories(
    Map<LocalDa  Decimal> daily =
      dailyPer  nputeIfAbsent(
        invent  oreId(), k -> new HashMap(
    for (Rent  : inventory.getRentals())
      for (P  ental.getPayments())
        dai
                te(),
                 null ? p.getAmount() : p.getAmount().add(v));
```

Architect: We can't have such types.
Factor out in a new class.

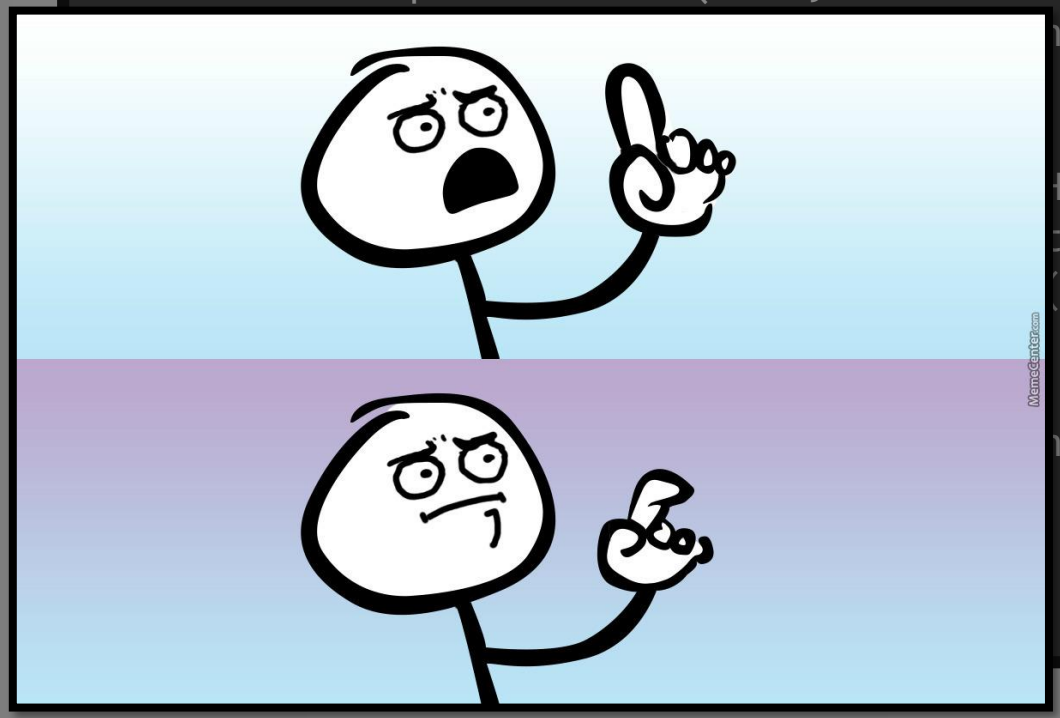
Java dev: Let's see, if I tweak this and then…

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = r lusOneLoadAllFilms(); // TODO
for (Film film : fi s) {
  Map<Integer, Map  calDate, BigDecimal>> dailyP
    result.comput  sent(film, k -> new HashMap
  for (Inventory  tory : film.getInventories(
    Map<LocalDa  Decimal> daily =
      dailyPer  nputeIfAbsent(
        invent  oreId(), k -> new HashMap(
    for (Rent  : inventory.getRentals())
      for (P  ental.getPayments())
        dai
                te(),
                 null ? p.getAmount() : p.getAmount().add(v));
```

Architect: We can't have such types. Factor out in a new class. And Factory

And you're like…

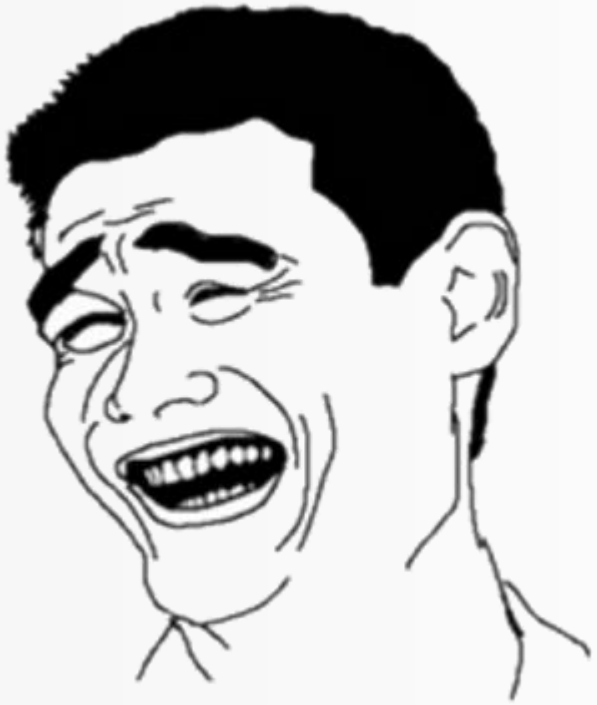
```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyP
    result.computeIfAbsent(film, k -> new HashMap
                                      tories(
                                      HashMap(
                                      tals())
                                      ))
                                      t() : p.getAmount().add(v));
```



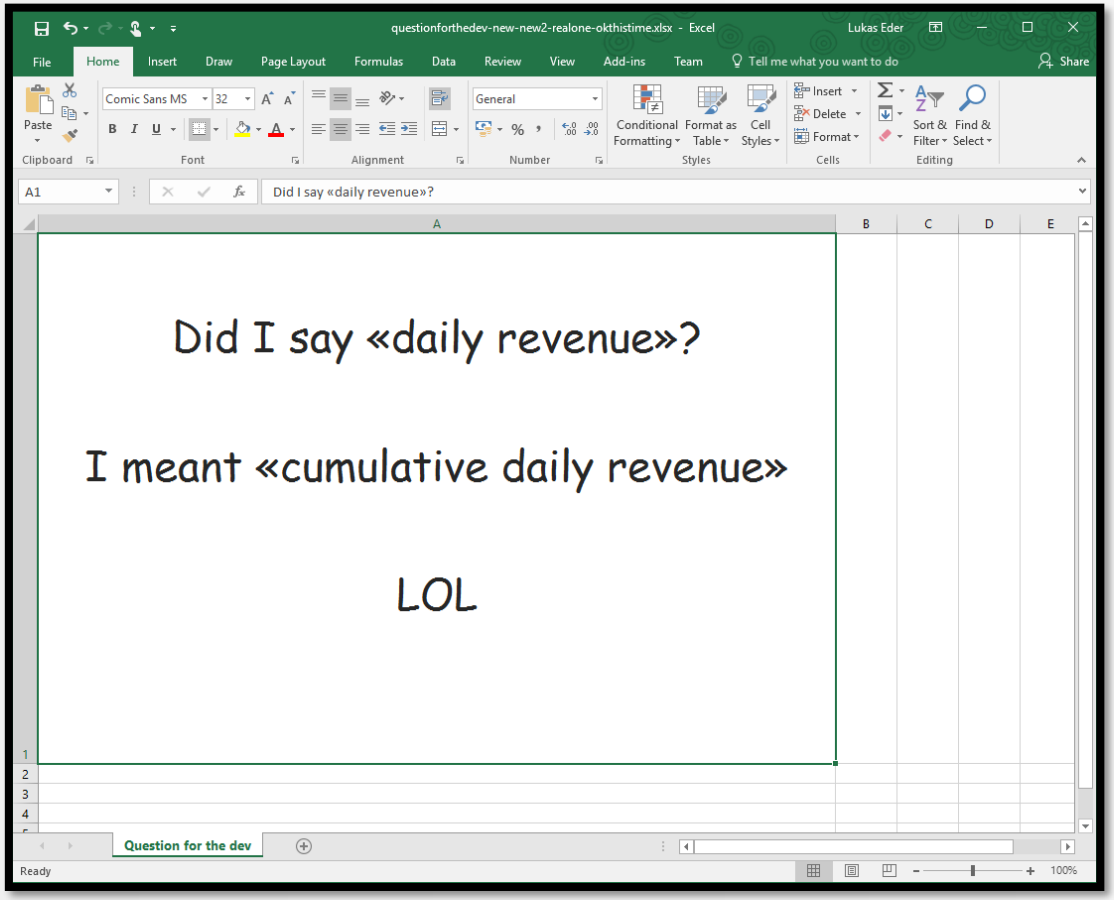
Meanwhile…

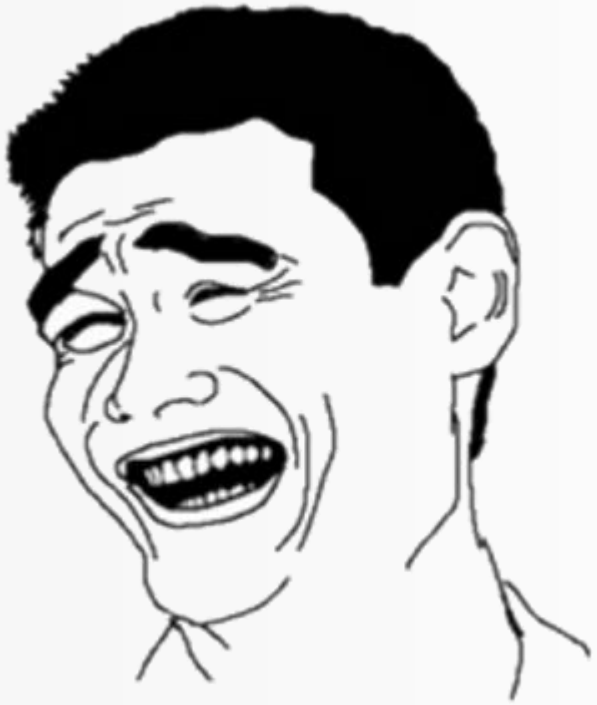
While you and your architect engage in nominal vs structural typing philosophies…

Meanwhile…

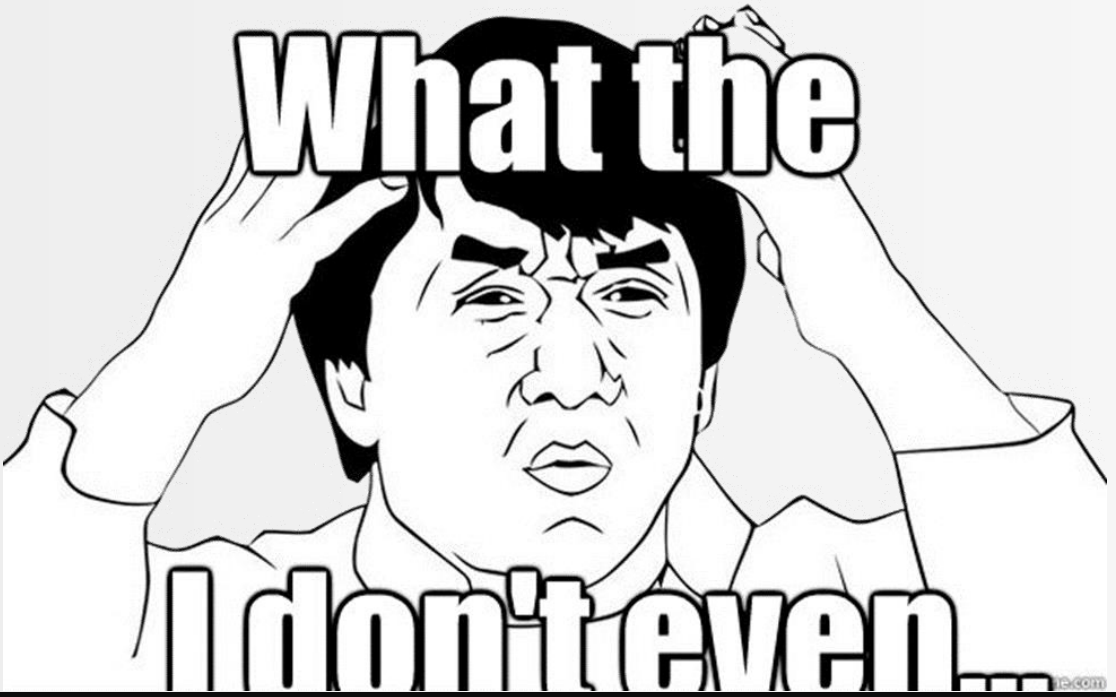


Meanwhile… The inevitable



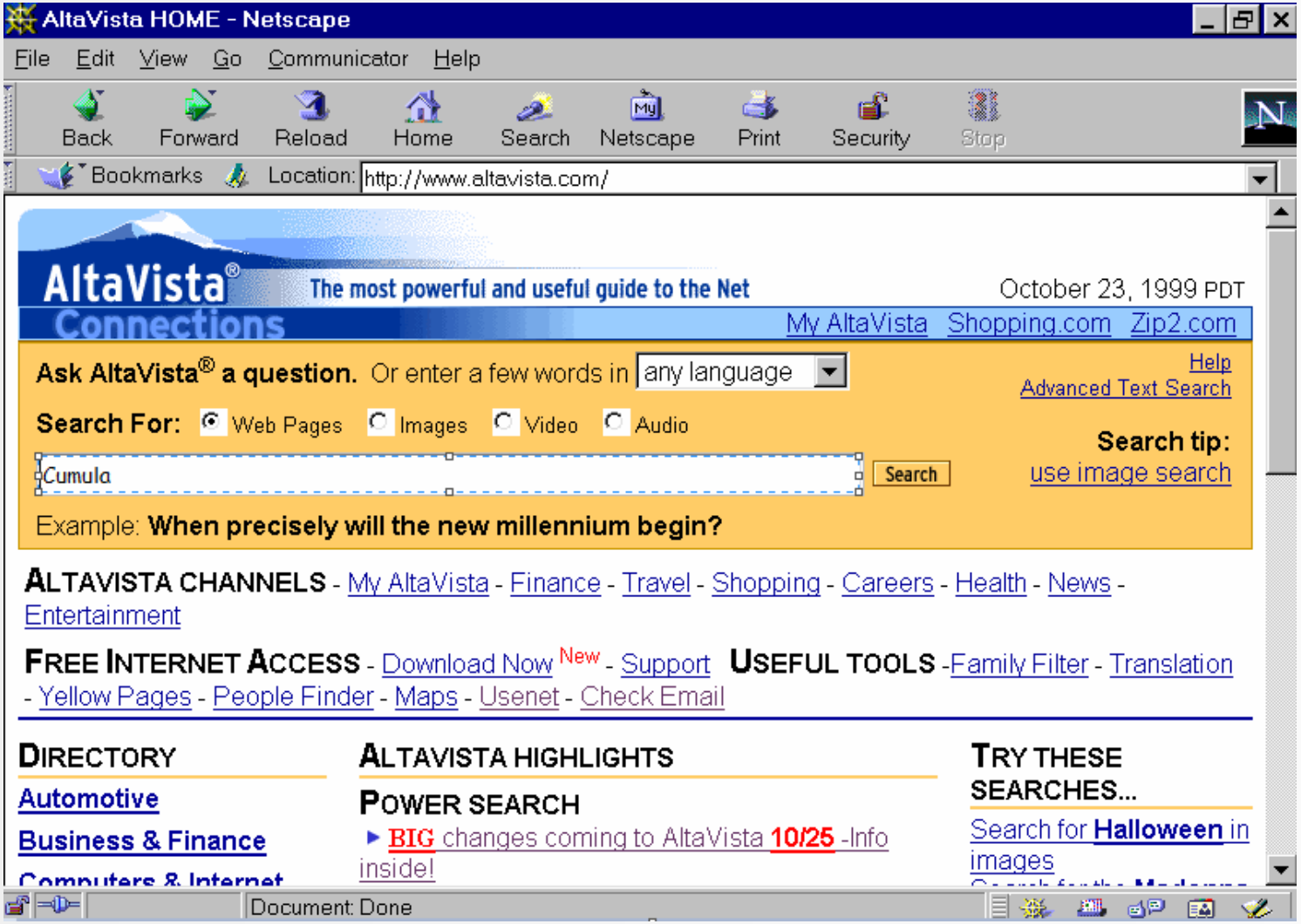


What on earth is a cumulative sum??



What on earth is a cumulative sum??

What on earth is a cumulative sum??



What on earth is a cumulative sum??

$$\frac{d^2\sigma}{d\Omega d\omega'} = r_0^2 \left(\frac{\omega'}{\omega}\right) \sum_f \delta(E_f - E_i - \hbar\Delta\omega) \times \left| \left[\langle f | \sum_j e^{i\mathbf{Q}\cdot\mathbf{r}_j} | i \rangle (\vec{\epsilon}_\alpha \cdot \vec{\epsilon}_{\alpha'}) + \frac{1}{m} \sum_q \left(\frac{\langle f | \vec{\epsilon}_{\alpha'} \cdot \sum_j \mathbf{p}_j e^{-i\mathbf{k}'\cdot\mathbf{r}_j} | q \rangle \langle q | \vec{\epsilon}_\alpha \cdot \sum_j \mathbf{p}_j e^{i\mathbf{k}\cdot\mathbf{r}_j} | i \rangle}{E_i + \hbar\omega - E_q} + \frac{\langle f | \vec{\epsilon}_\alpha \cdot \sum_j \mathbf{p}_j e^{i\mathbf{k}\cdot\mathbf{r}_j} | q \rangle \langle q | \vec{\epsilon}_{\alpha'} \cdot \sum_j \mathbf{p}_j e^{-i\mathbf{k}'\cdot\mathbf{r}_j} | i \rangle}{E_i - \hbar\omega' - E_q} \right) \right] \right|^2,$$

Java dev: 😲

```
// Thanks God it’s Friday!
throw new UnsupportedOperationException();
```

Meanwhile, the SQL developer…

```
SELECT
  title, store_id, payment_date,
  SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Meanwhile, the SQL developer…

```
SELECT
  title, store_id, payment_date,
      SUM(amount)



FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Meanwhile, the SQL developer…

```
SELECT
  title, store_id, payment_date,
  SUM(SUM(amount)) OVER (
    PARTITION BY title, store_id
    ORDER BY payment_date
  )
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Meanwhile, the SQL developer…

```
SELECT
  title, store_id, payment_date,
  SUM(SUM(amount)) OVER (
    PARTITION BY title, store_id
    ORDER BY payment_date
  )
FROM film
JOIN inve    y USING (film_id)
JOIN rer     SING (inventory_id)
JOIN pa      SING (rental_id)
             _date
             date
```

Window function

What does it do?

title character varying(255)	store_id integer	payment_date date	sum numeric	cumulative_sum numeric
ACADEMY DINOSAUR	1	2005-05-30	1.99	1.99
ACADEMY DINOSAUR	1	2005-06-17	0.99	2.98
ACADEMY DINOSAUR	1	2005-07-07	0.99	3.97
ACADEMY DINOSAUR	1	2005-07-08	0.99	4.96
ACADEMY DINOSAUR	1	2005-07-30	1.99	6.95
ACADEMY DINOSAUR	1	2005-07-31	0.99	7.94
ACADEMY DINOSAUR	1	2005-08-02	4.98	12.92
ACADEMY DINOSAUR	1	2005-08-21	4.98	17.90
ACADEMY DINOSAUR	1	2005-08-22	0.99	18.89
ACADEMY DINOSAUR	1	2005-08-23	1.99	20.88
ACADEMY DINOSAUR	2	2005-05-27	0.99	0.99
ACADEMY DINOSAUR	2	2005-06-15	0.99	1.98
ACADEMY DINOSAUR	2	2005-06-21	1.99	3.97
ACADEMY DINOSAUR	2	2005-07-07	0.99	4.96
ACADEMY DINOSAUR	2	2005-07-10	0.99	5.95
ACADEMY DINOSAUR	2	2005-07-27	0.99	6.94
ACADEMY DINOSAUR	2	2005-07-29	1.99	8.93
ACADEMY DINOSAUR	2	2005-07-31	0.99	9.92
ACADEMY DINOSAUR	2	2005-08-18	0.99	10.91

What does it do?

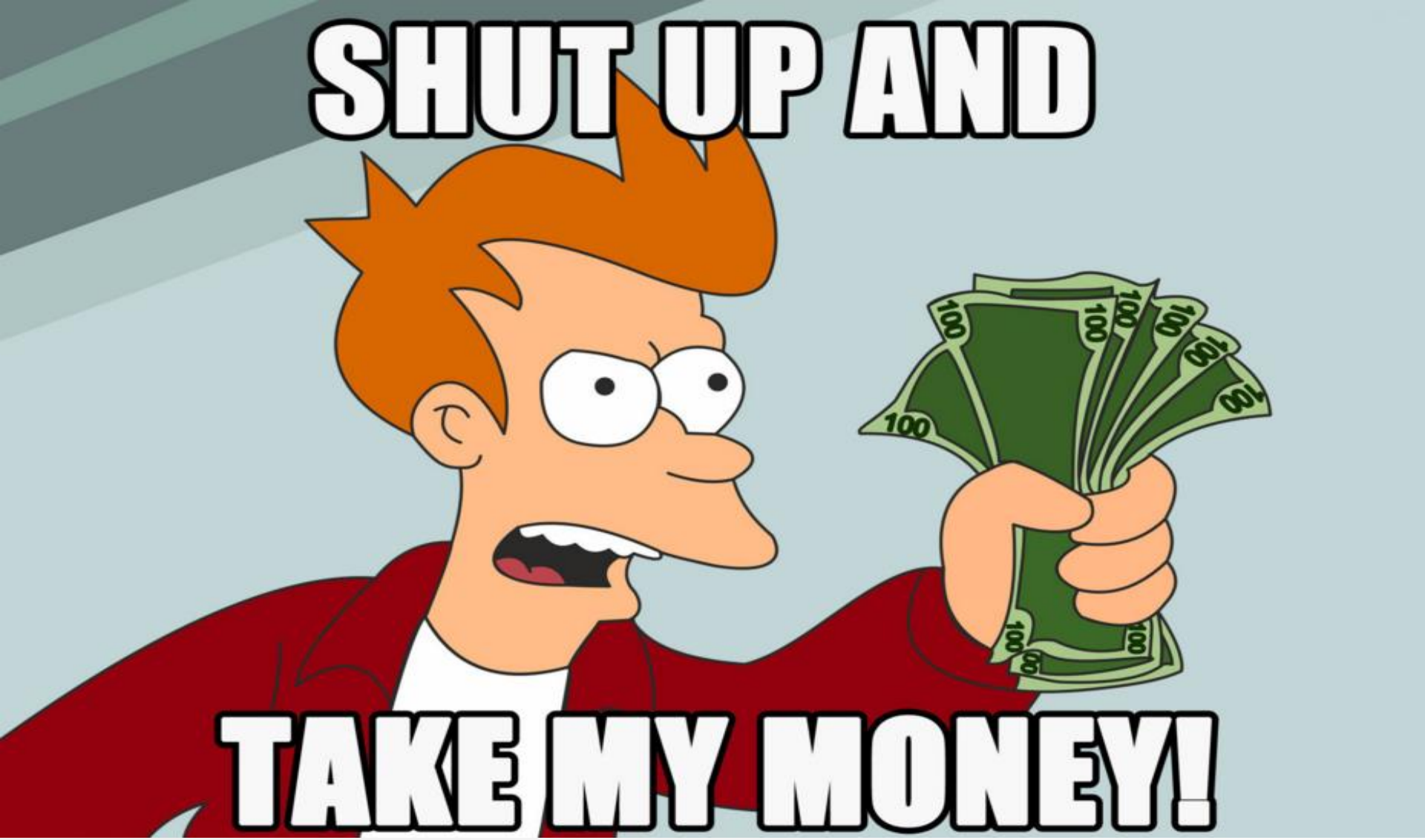
title character varying(255)	store_id integer	payment_date date	sum numeric	cumulative_sum numeric
ACADEMY DINOSAUR	1	2005-05-30	1.99	1.99
ACADEMY DINOSAUR	1	2005-06-17	0.99	2.98
ACADEMY DINOSAUR	1	2005-07-07	0.99	3.97
ACADEMY DINOSAUR	1	2005-07-08	0.99	4.96
ACADEMY DINOSAUR	1	2005-07-30	1.99	6.95
ACADEMY DINOSAUR	1	2005-07-31	0.99	7.94
ACADEMY DINOSAUR	1	2005-08-02	4.98	12.92
ACADEMY DINOSAUR	1	2005-08-21	4.98	17.90
ACADEMY DINOSAUR	1	2005-08-22	0.99	18.89
ACADEMY DINOSAUR	1	2005-08-23	1.99	20.88
ACADEMY DINOSAUR	2	2005-05-27	0.99	0.99
ACADEMY DINOSAUR	2	2005-06-15	0.99	1.98
ACADEMY DINOSAUR	2	2005-06-21	1.99	3.97
ACADEMY DINOSAUR	2	2005-07-07	0.99	4.96
ACADEMY DINOSAUR	2	2005-07-10	0.99	5.95
ACADEMY DINOSAUR	2	2005-07-27	0.99	6.94
ACADEMY DINOSAUR	2	2005-07-29	1.99	8.93
ACADEMY DINOSAUR	2	2005-07-31	0.99	9.92
ACADEMY DINOSAUR	2	2005-08-18	0.99	10.91

What does it do?

title character varying(255)	store_id integer	payment_date date	sum numeric	cumulative_sum numeric
ACADEMY DINOSAUR	1	2005-05-30	1.99	1.99
ACADEMY DINOSAUR	1	2005-06-17	0.99	2.98
ACADEMY DINOSAUR	1	2005-07-07	0.99	3.97
ACADEMY DINOSAUR	1	2005-07-08	0.99	4.96
ACADEMY DINOSAUR	1	2005-07-30	1.99	6.95
ACADEMY DINOSAUR	1	2005-07-31	0.99	7.94
ACADEMY DINOSAUR	1	2005-08-02	4.98	12.92
ACADEMY DINOSAUR	1	2005-08-21	4.98	17.90
ACADEMY DINOSAUR	1	2005-08-22	0.99	18.89
ACADEMY DINOSAUR	1	2005-08-23	1.99	20.88
ACADEMY DINOSAUR	2	2005-05-27	0.99	0.99
ACADEMY DINOSAUR	2	2005-06-15	0.99	1.98
ACADEMY DINOSAUR	2	2005-06-21	1.99	3.97
ACADEMY DINOSAUR	2	2005-07-07	0.99	4.96
ACADEMY DINOSAUR	2	2005-07-10	0.99	5.95
ACADEMY DINOSAUR	2	2005-07-27	0.99	6.94
ACADEMY DINOSAUR	2	2005-07-29	1.99	8.93
ACADEMY DINOSAUR	2	2005-07-31	0.99	9.92
ACADEMY DINOSAUR	2	2005-08-18	0.99	10.91

3GL vs. 4GL

Do you get the message?



https://www.jooq.org/training/

3GL vs. 4GL

SQL is a 4GL

3GL vs. 4GL

In SQL, we only *declare* the result.

3GL vs. 4GL

The database will figure out the algorithm for us.

3GL vs. 4GL

Algorithms are *boring* (mostly)

3GL vs. 4GL

We don’t care about

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation
- Data type details (HashMap<A, B> or List<Entry<A, B>>?)

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation
- Data type details (HashMap<A, B> or List<Entry<A, B>>?)
- Local variables

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation
- Data type details (HashMap<A, B> or List<Entry<A, B>>?)
- Local variables
- Caches

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation
- Data type details (HashMap<A, B> or List<Entry<A, B>>?)
- Local variables
- Caches
- Loops

3GL vs. 4GL

We don’t care about

- Writing algorithms (using them is a different story)
- Premature Optimisation
- Data type details (HashMap<A, B> or List<Entry<A, B>>?)
- Local variables
- Caches
- Loops
- Initialisation (or NullPointerException)

3GL vs. 4GL

Let's have a discussion about Optional!

We do[illegible]are about

- Writing algorith[illegible]them is a different story)
- Premature Optim[illegible]
- Data type details (H[illegible]ap<A, B> or List<Entry<A, B>>?)
- Local variables
- Caches
- Loops
- Initialisation (or NullPointerException)

3GL vs. 4GL

We do care about

3GL vs. 4GL

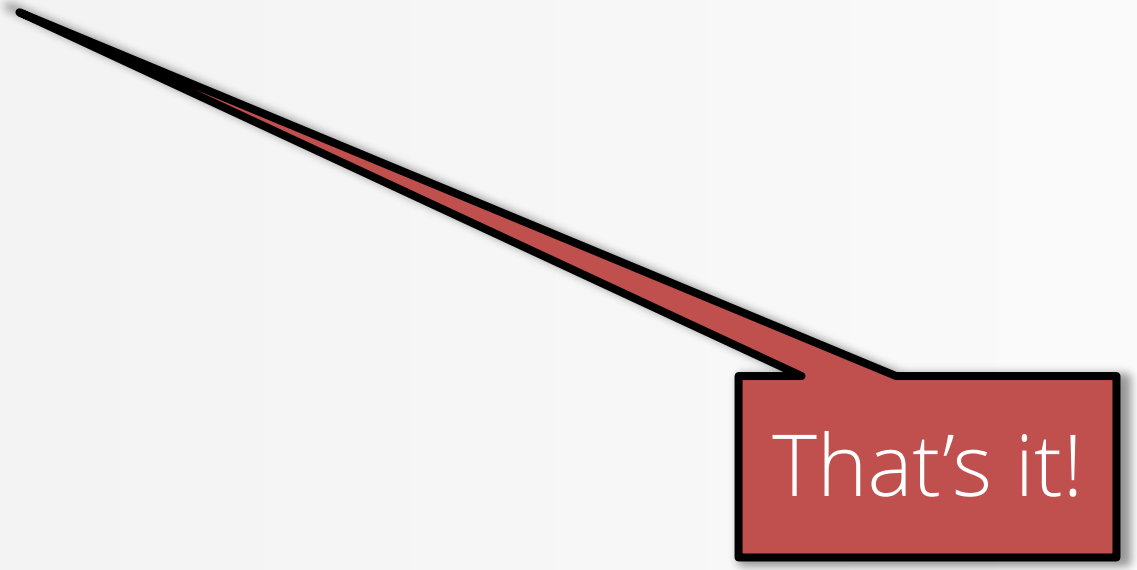
We do care about

- Business logic
- Just a bit of indexing

3GL vs. 4GL

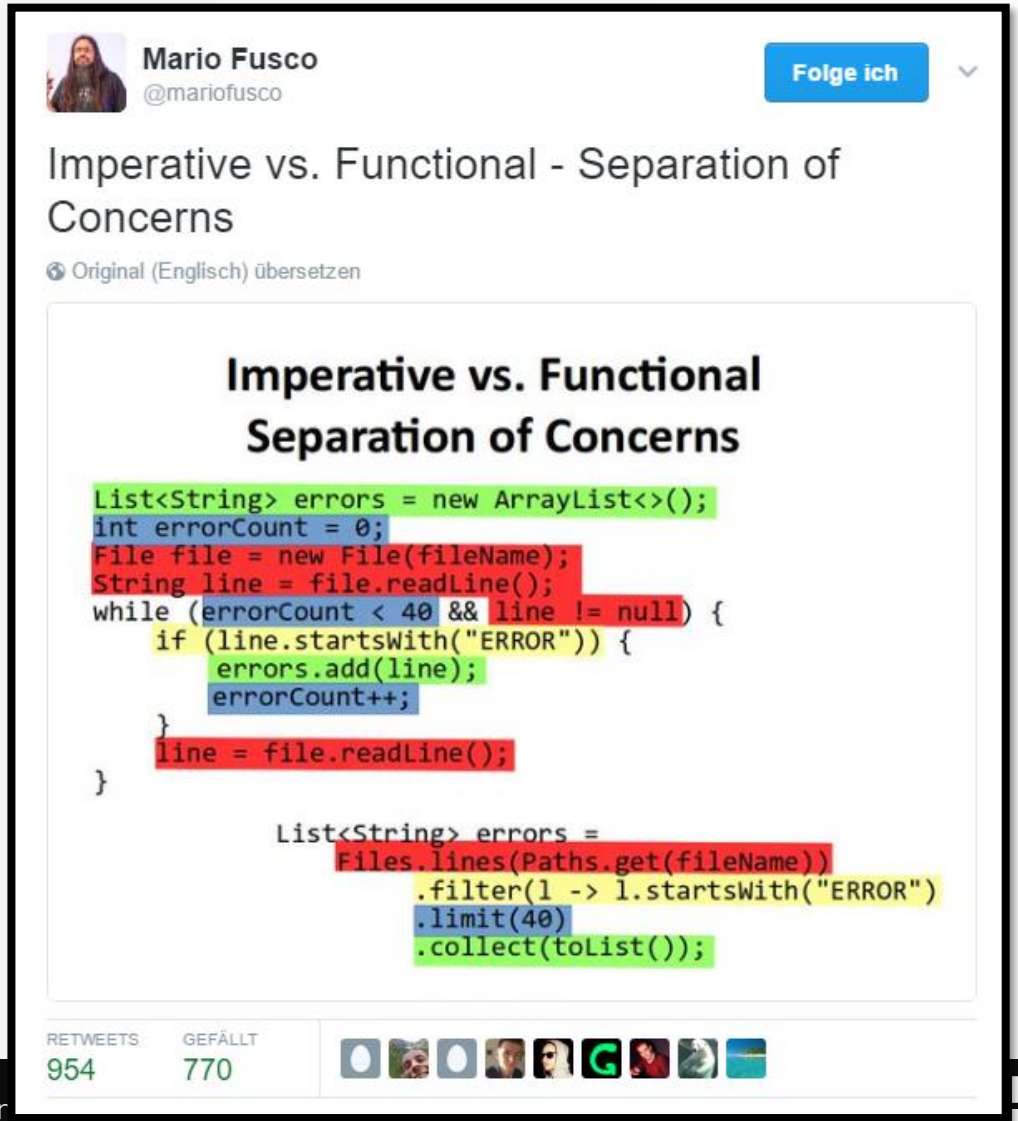
We do care about

- Business logic
- Just a bit of indexing



Is Java 8 Streams “declarative” ?

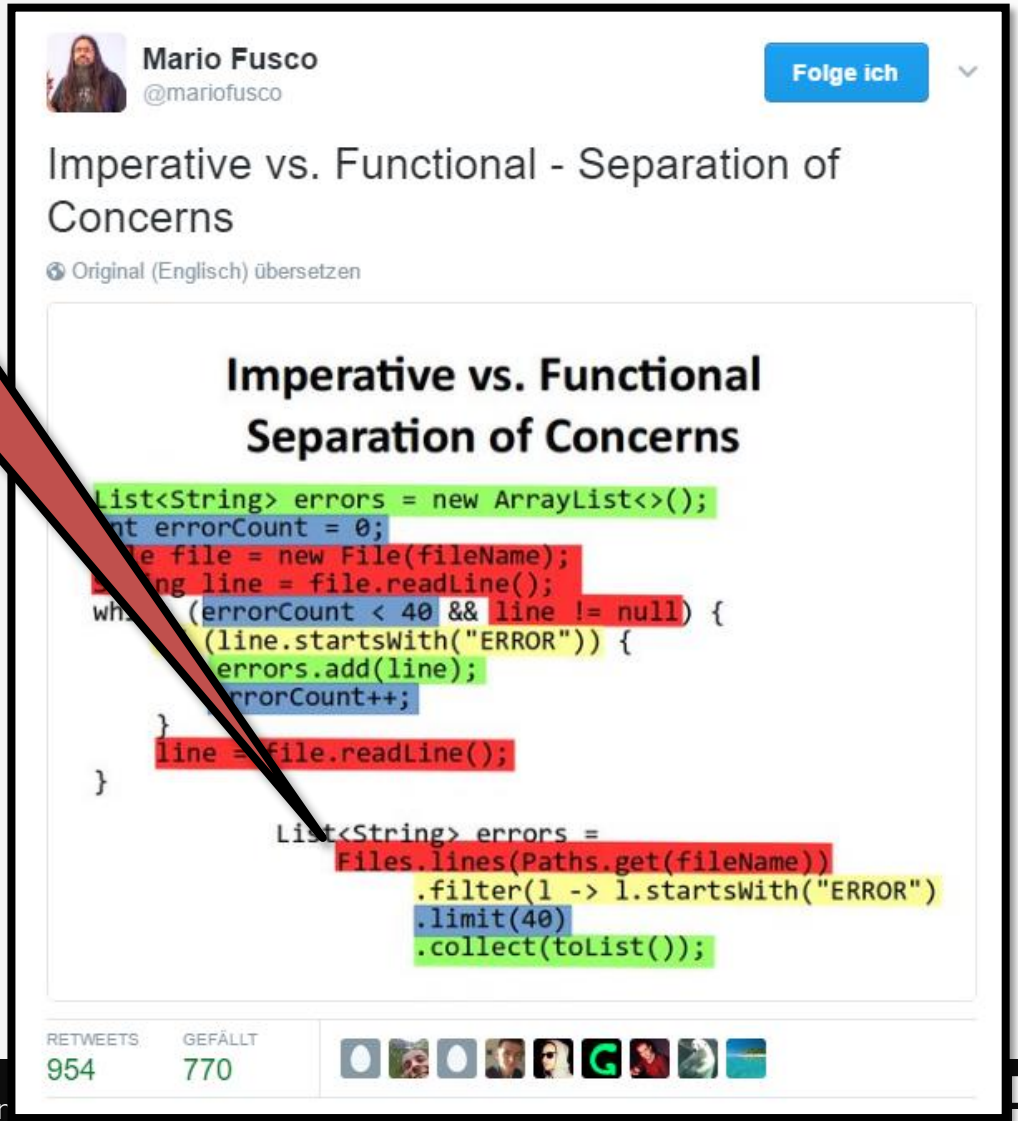
The Stream API looks a lot like SQL’s declarative style



Is Java 8 Streams “declarative” ?

FROM

The Stream API looks a lot like SQL’s declarative style

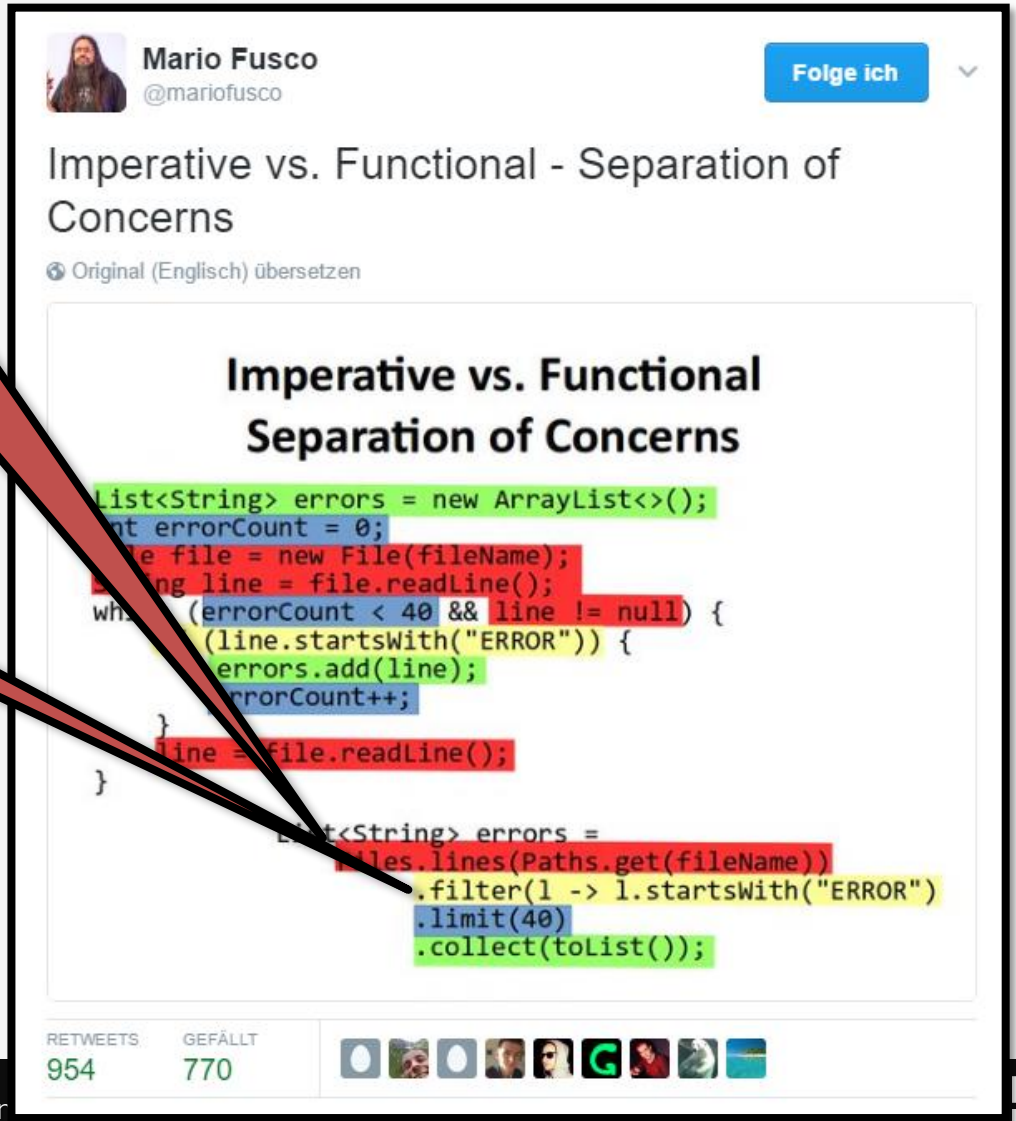


Is Java 8 Streams “declarative” ?

FROM

WHERE

T API
Id ke
SQL’s declar tive
style



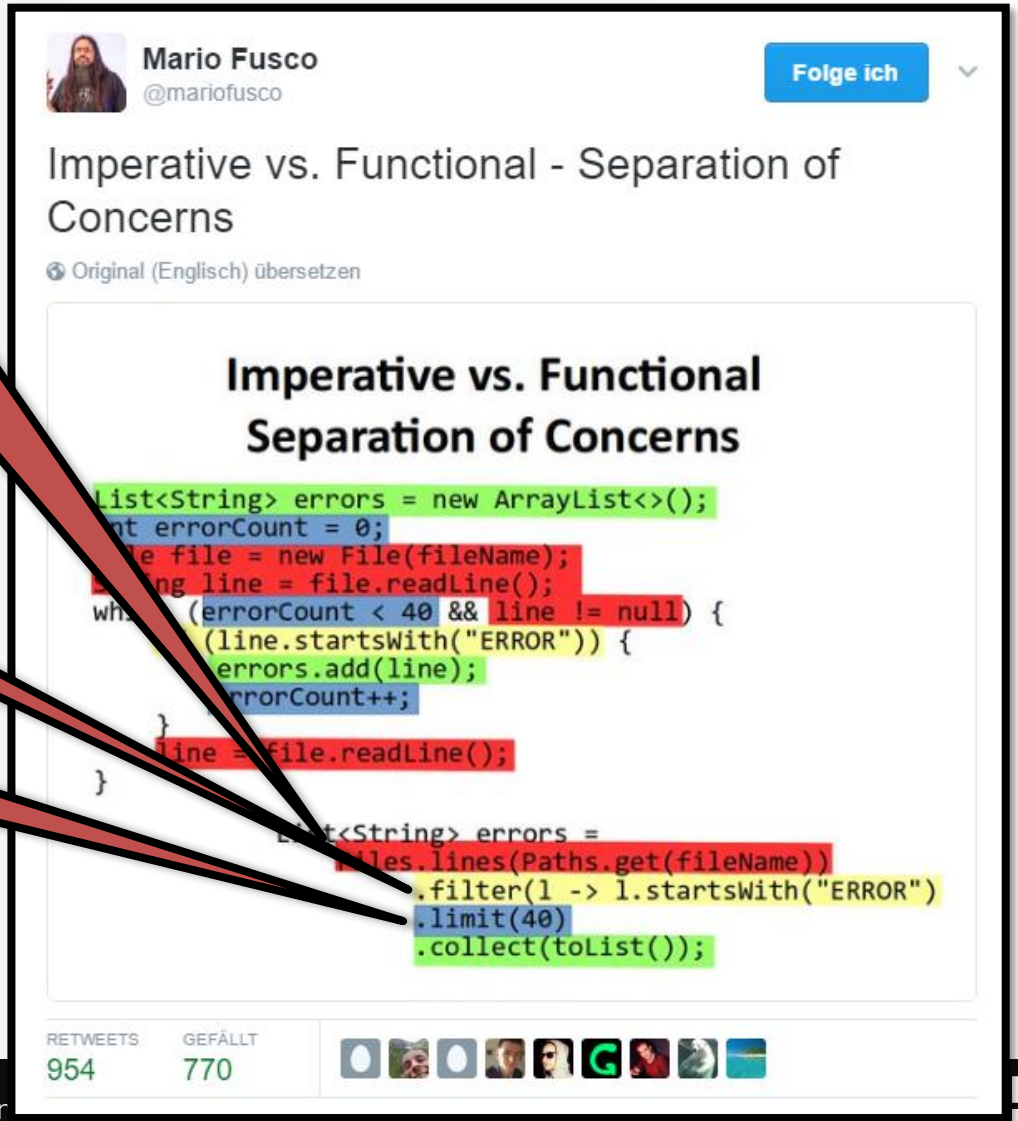
Is Java 8 Streams “declarative” ?

FROM

WHERE

FETCH NEXT

T API
Id ke
SQL’s declar tive



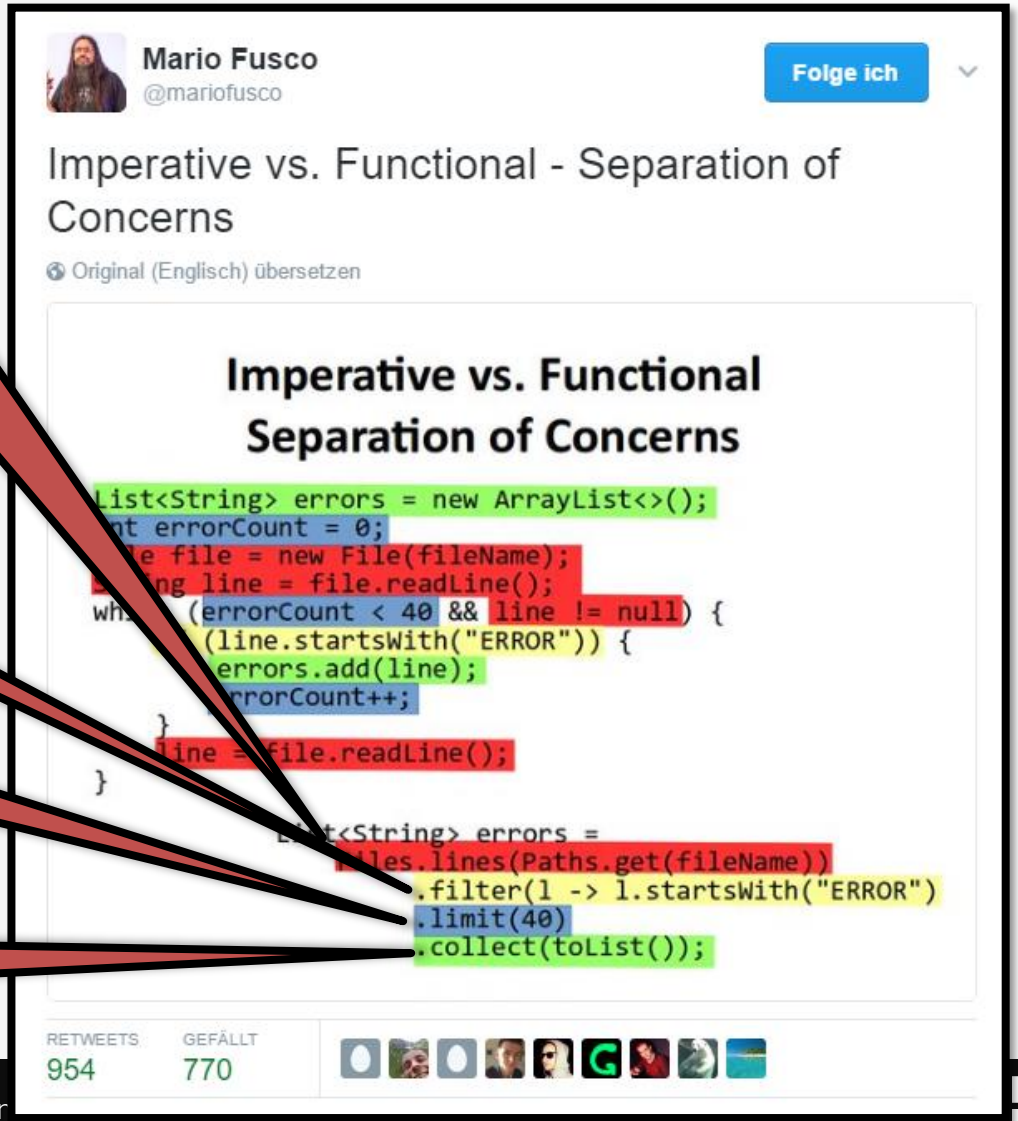
Is Java 8 Streams “declarative” ?

FROM

WHERE

FETCH NEXT

GROUP BY



In a way, it's very similar

```
TABLE              : Stream<Tuple<..>>
SELECT             : map()
DISTINCT           : distinct()
JOIN               : flatMap()
WHERE / HAVING     : filter()
GROUP BY           : collect()
ORDER BY           : sorted()
UNION ALL          : concat()
```

See:
http://blog.jooq.org/2015/08/13/common-sql-clauses-and-their-equivalents-in-java-8-streams/

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = loadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyPerStore =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories()) {
    Map<LocalDate, BigDecimal> daily =
      dailyPerStore.computeIfAbsent(
        inventory.getStoreId(), k -> new HashMap());
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
  }
}
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
     .stream()
     .flatMap(f -> f.getInventories().stream())
     .flatMap(i -> i.getRentals().stream())
     .flatMap(r -> r.getPayments().stream())
     .collect(... /* amount per film and store */ ...);
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()                                          // Stream<Film>
    .flatMap(f -> f.getInventories().stream())
    .flatMap(i -> i.getRentals().stream())
    .flatMap(r -> r.getPayments().stream())
    .collect(... /* amount per film and store */ ...);
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()                                    // Stream<Film>
    .flatMap(f -> f.getInventories().stream()) // Stream<Inventory>
    .flatMap(i -> i.getRentals().stream())
    .flatMap(r -> r.getPayments().stream())
    .collect(... /* amount per film and store */ ...);
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()                                          // Stream<Film>
    .flatMap(f -> f.getInventories().stream())         // Stream<Inventory>
    .flatMap(i -> i.getRentals().stream())             // Stream<Rental>
    .flatMap(r -> r.getPayments().stream())
    .collect(... /* amount per film and store */ ...);
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()                                          // Stream<Film>
    .flatMap(f -> f.getInventories().stream())         // Stream<Inventory>
    .flatMap(i -> i.getRentals().stream())             // Stream<Rental>
    .flatMap(r -> r.getPayments().stream())            // Stream<Payment>
    .collect(... /* amount per film and store */ ...);
```

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()                                          // Stream<Film>
    .flatMap(f -> f.getInventories().stream())         // Stream<Inventory>
    .flatMap(i -> i.getRentals().stream())             // Stream<Rental>
    .flatMap(r -> r.getPayments().stream())            // Stream<Payment>
    .collect(... /* amount per film and store */ ...);
```

How to access «f» and «i» in the Collector?

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
     .stream()
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .collect(... /* amount per film and store */ ...);
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
    .stream()
    .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
    .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
    .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
    .collect(... /* amount per film and store */ ...);
```

Use some wrapper types and pray for stack allocation to happen

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
     .stream()
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .collect(
        Collectors.groupingBy(firp -> firp.getKey().getKey().getKey(),
          Collectors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                              .getStoreId(),
            Collectors.groupingBy(firp -> firp.getValue().getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.getValue().getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let’s try that Java algorithm again

But what the hell is this?
(ノ°□°)ノ ⌒ ┻━┻

```
Map<Film, Ma
    .stream
    .flatMap(f    -> f.getInventories().stream().ma                    1)))
    .flatMap(fi   -> fi.v2.getRentals().stream().ma                  (fi, r)))
    .flatMap(fir -> fir.v3.getPayments().stream()                -> entry(fir, p)))
    .collect(
       Collectors.groupingBy(firp -> firp.getKey().getKey().getKey(),
         Collectors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                              .getStoreId(),
           Collectors.groupingBy(firp -> firp.getValue().getPaymentDate(),
             Collectors.summingDouble(
               firp -> firp.getValue().getAmount().doubleValue()
             )
           )
         )
       )
    );
```

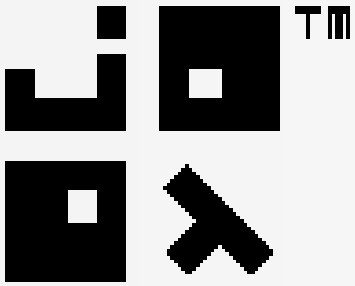
Let’s try that Java algorithm again

Well, it certainly looks declarative

We can do a bit better, though

Better Streams:

https://github.com/jOOQ/jOOL



We can do a bit better, though

```
Seq.seq(persons)
    .collect(
        count(),
        max(Person::getAge),
        min(Person::getHeight),
        avg(Person::getWeight)
    );
// (3, Optional[35],
//  Optional[1.69], Optional[70.0])
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
     .stream()
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .collect(
        Collectors.groupingBy(firp -> firp.getKey().getKey().getKey(),
          Collectors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                              .getStoreId(),
            Collectors.groupingBy(firp -> firp.getValue().getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.getValue().getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result = loadAllFilms()
     .stream()
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .collect(
        Collectors.groupingBy(firp -> firp.getKey().getKey().getKey(),
          Collectors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                              .getStoreId(),
            Collectors.groupingBy(firp -> firp.getValue().getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.getValue().getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .colle
        Col      rs.groupingBy(firp -> firp.getKey().getKey().getKey(),
          C      ors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                                .getStoreId(),
                      rs.groupingBy(firp -> firp.getValue().getPaymentDate(),
                      ors.summingDouble(
                        > firp.getValue().getAmount().doubleValue()
          )
     )
```

Use a Seq instead of a Stream

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .flatMap(f   -> f.getInventories().stream().map(i -> entry(f, i)))
     .flatMap(fi  -> fi.v2.getRentals().stream().map(r -> entry(fi, r)))
     .flatMap(fir -> fir.v3.getPayments().stream().map(p -> entry(fir, p)))
     .collect(
        Collectors.groupingBy(firp -> firp.getKey().getKey().getKey(),
          Collectors.groupingBy(firp -> firp.getKey().getKey().getValue()
                                              .getStoreId(),
            Collectors.groupingBy(firp -> firp.getValue().getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.getValue().getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals())
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Colle    rs.groupingBy(firp -> firp.v1.v1.v1,
          Col    ors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            C    tors.groupingBy(firp -> firp.v2.getPaymentDate(),
                 tors.summingDouble(
                      -> firp.v2.getAmount().doubleValue()
          )
        )
     );
```

Use SQL style CROSS APPLY instead of flatMap()

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())     // <Film, Inventory>
     .crossApply(fi -> fi.v2.getRentals())
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())     // <Film, Inventory>
     .crossApply(fi -> fi.v2.getRentals())    // <<F, I>, Rental>
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let’s try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())     // <Film, Inventory>
     .crossApply(fi -> fi.v2.getRentals())    // <<F, I>, Rental>
     .crossApply(fir -> fir.v2.getPayments()) // <<<F, I>, R>, Payment>
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

<<<Film, Inventory>, Rental>, Payment>

```
Map<Film, Map<Integer, Map<LocalDate, Do                    =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories()
     .crossApply(fi -> fi.v2.getRentals(
     .crossApply(fir -> fir.v2.getPaymen    ))
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals())
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> fi   v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().     eValue()
              )
            )
          )
        )
     );
```

<<Film, Inventory>, Rental>

Let's try that Java algorithm again

<Film, Inventory>

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> r      =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals())
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals())
     .crossApply(fir -> fir.v2.getPayments())
     .collect(
        Collectors.groupingBy(firp -> firp.v1.v1.v1,
          Collectors.groupingBy(firp -> firp.v1.v1.v2.getStoreId(),
            Collectors.groupingBy(firp -> firp.v2.getPaymentDate(),
              Collectors.summingDouble(
                firp -> firp.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Film

Let's try that Java algorithm again

```
Map<Film, Map<Integer, Map<LocalDate, Double>>> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals().stream()
                          .flatMap(r -> r.getPayments().stream()))
     .collect(
        Collectors.groupingBy(fip -> fip.v1.v1,
          Collectors.groupingBy(fip -> fip.v1.v2.getStoreId(),
            Collectors.groupingBy(fip -> fip.v2.getPaymentDate(),
              Collectors.summingDouble(
                fip -> fip.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Hm, never really needed the rentals

Let’s try that Java algorithm again

A little bit better now…

```
Map<Film, Ma
  Seq.seq(lo
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals().stream()
                       .flatMap(r -> r.getPaymen    eam()))
     .collect(
        Collectors.groupingBy(fip -> fip.v1.v1,
          Collectors.groupingBy(fip -> fip.v1.v2.getStoreId(),
            Collectors.groupingBy(fip -> fip.v2.getPaymentDate(),
              Collectors.summingDouble(
                fip -> fip.v2.getAmount().doubleValue()
              )
            )
          )
        )
     );
```

Let's try that Java algorithm again

```
Map<Tuple3<Film, Integer, LocalDate>, Double> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals().stream()
                           .flatMap(r -> r.getPayments().stream()))
     .groupBy(
        fip -> tuple(
          fip.v1.v1,
          fip.v1.v2.getStoreId(),
          fip.v2.getPaymentDate()
        ),
        Agg.sum(fip -> fip.v2.getAmount())
      );
```

Oh, not that bad, actually

┬─┬ノ(゜ -゜ノ)

Let’s try that Java algorithm again

```
Map<Tuple3<Film, Integer, LocalDate>, Double> result =
  Seq.seq(loadAllFilms())
     .crossApply(f -> f.getInventories())
     .crossApply(fi -> fi.v2.getRentals().stream()
                         .flatMap(r -> r.getPayments().stream()))
     .groupBy(
        fip -> tuple(
          fip.v1.v1,
          fip.v1.v2.getId(),
          fip.v2.getDate()
        ),
        Agg.sum(fip -> fip.v2.getAmount())
     );
```

Result type a bit nicer, too

Bad news is:

How much time did we spend on this problem now?

Bad news is:

Remember this slide? Was it all that hard?

```
SELECT
  title, store_id, payment_date,
  SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Everyone understands now…

SQL is a 4GL

This means great improvement of developer productivity

Everyone understands now…

How much time did I spend talking about SQL so far?

How much time on Java?

SQL is easier = SQL is more productive

What will I talk about?

- ~~SQL is awesome~~

- SQL is productive
- SQL is fast

What will I talk about?

- ~~SQL is awesome~~

- ~~SQL is productive~~

- SQL is fast

Suspense!

Best chapter:
Performance

But first

How does SQL work?

Business asking

How many films did each actor whose last name starts with «A» play in?

They want this

first_name character varying(45)	last_name character varying(45)	count bigint
KIRSTEN	AKROYD	34
CHRISTIAN	AKROYD	32
ANGELINA	ASTAIRE	31
KIM	ALLEN	28
CUBA	ALLEN	25
DEBBIE	AKROYD	24
MERYL	ALLEN	22

They want this

Starts with «A»

first_name character varying(45)	last_name character varying(45)	count bigint
KIRSTEN	AKROYD	34
CHRISTIAN	AKROYD	32
ANGELINA	ASTAIRE	31
KIM	ALLEN	28
CUBA	ALLEN	25
DEBBIE	AKROYD	24
MERYL	ALLEN	22

They want this

Starts with «A»

How many films

first_name character varying(45)	last_name character varying(45)	count bigint
KIRSTEN	AKROYD	34
CHRISTIAN	AKROYD	32
ANGELINA	ASTAIRE	31
KIM	ALLEN	28
CUBA	ALLEN	25
DEBBIE	AKROYD	24
MERYL	ALLEN	22

This is the query. But what really happens?

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
```

Observe: «LOGICAL». Not «ACTUAL»

Logical Operations order

1. Load tables
2. Apply predicates

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables
2. Apply predicates
3. Collect groups

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables
2. Apply predicates
3. Collect groups
4. Aggregate

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables
2. Apply predicates
3. Collect groups
4. Aggregate
5. Project

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Logical Operations order

1. Load tables
2. Apply predicates
3. Collect groups
4. Aggregate
5. Project
6. Order

```
SELECT
  first_name, last_name, count(*)
FROM actor
JOIN film_actor USING (actor_id)
WHERE last_name LIKE 'A%'
GROUP BY first_name, last_name
ORDER BY count(*) DESC
```

Complete order

Logical operations order

1. FROM, JOIN, APPLY
2. WHERE
3. CONNECT BY (Oracle)
4. GROUP BY
5. Aggregations
6. HAVING
7. WINDOW
8. MODEL (Oracle)
9. SELECT
10. DISTINCT
11. UNION, INTERSECT, EXCEPT
12. ORDER BY
13. OFFSET
14. LIMIT
15. FOR UPDATE

Complete order

Logical operations order

Aggregate functions defined here

1. FROM, JOIN, APPLY
2. WHERE
3. CONNECT BY (Oracle)
4. GROUP BY
5. Aggregations
6. HAVING
7. WINDOW
8. MODEL (Oracle)
9. SELECT
10. DISTINCT
11. UNION, INTERSECT, EXCEPT
12. ORDER BY
13. OFFSET
14. LIMIT
15. FOR UPDATE

Complete order

Logical operations order

Window functions defined here

1. FROM, JOIN, APPLY
2. WHERE
3. CONNECT BY (Oracle)
4. GROUP BY
5. Aggregations
6. HAVING
7. WINDOW
8. MODEL (Oracle)
9. SELECT
10. DISTINCT
11. UNION, INTERSECT, EXCEPT
12. ORDER BY
13. OFFSET
14. LIMIT
15. FOR UPDATE

Complete order

Logical operations order

Column alias defined here

1. FROM, JOIN, APPLY
2. WHERE
3. CONNECT BY (Oracle)
4. GROUP BY
5. Aggregations
6. HAVING
7. WINDOW
8. MODEL (Oracle)
9. SELECT
10. DISTINCT
11. UNION, INTERSECT, EXCEPT
12. ORDER BY
13. OFFSET
14. LIMIT
15. FOR UPDATE

That explains a thing or two

What's the actual algorithm?

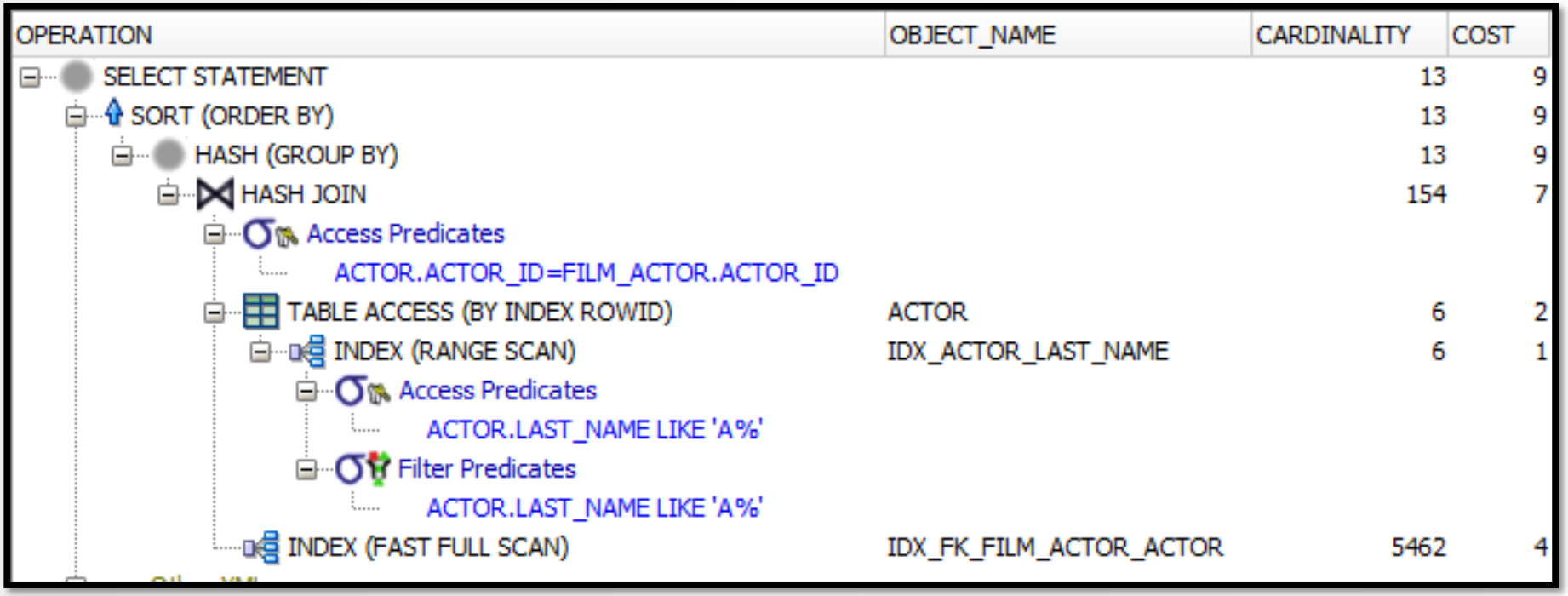
But that's not what really happens!

Any algorithm that produces the same result is acceptable.

What's the actual algorithm?

Can we see the actual algorithm?

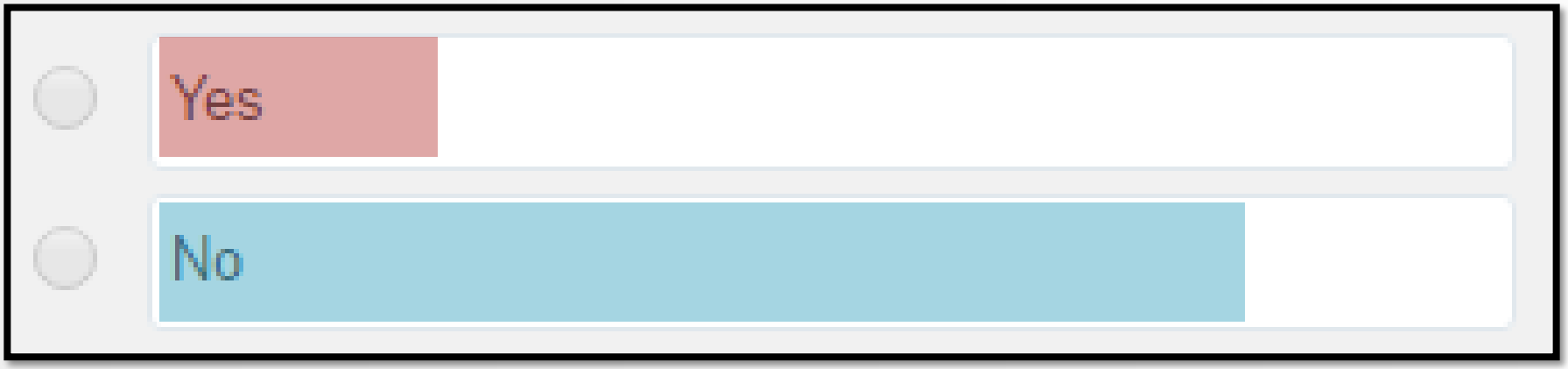
It’s called «execution plan»



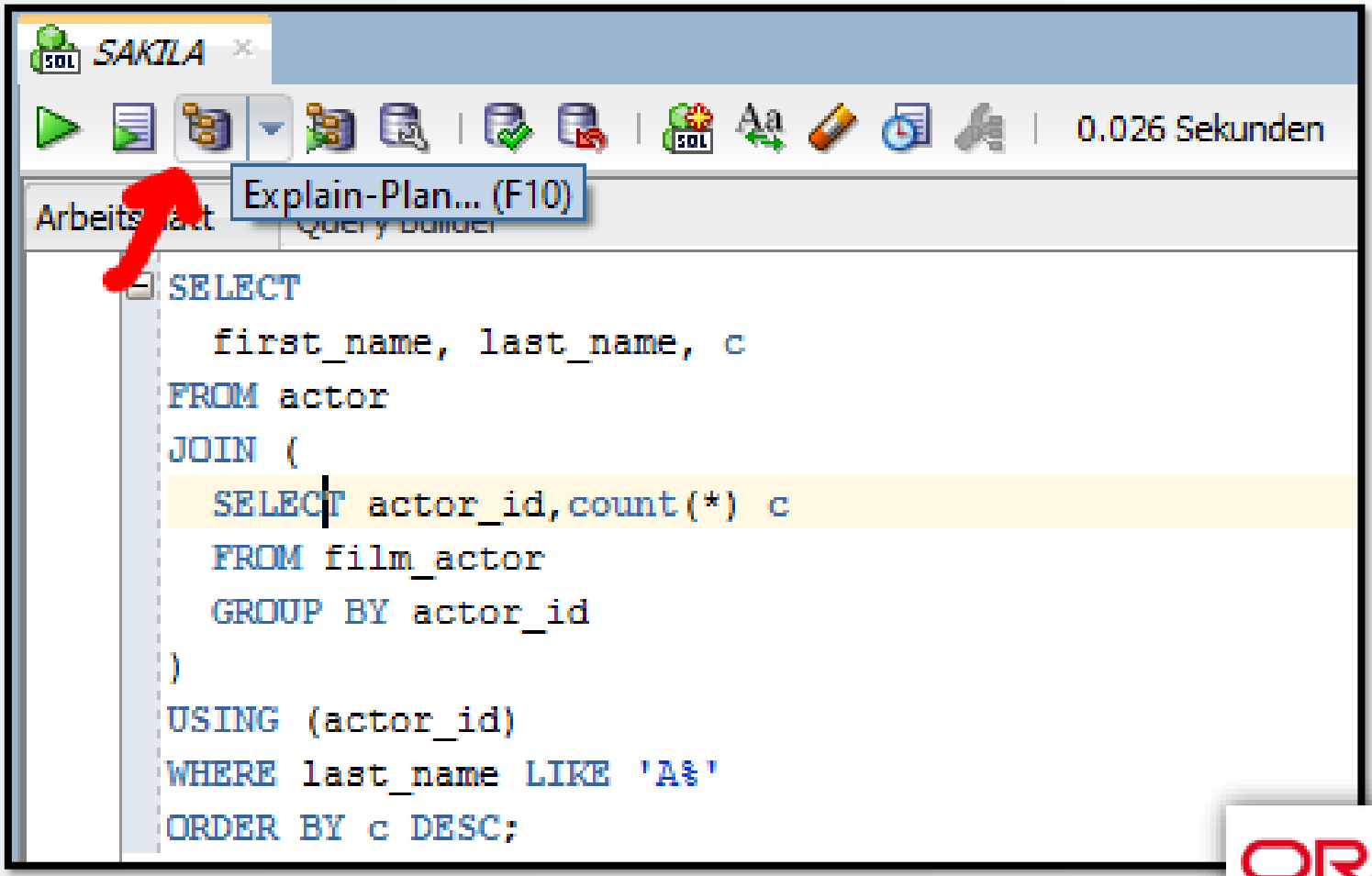
OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

Who has seen an execution plan?

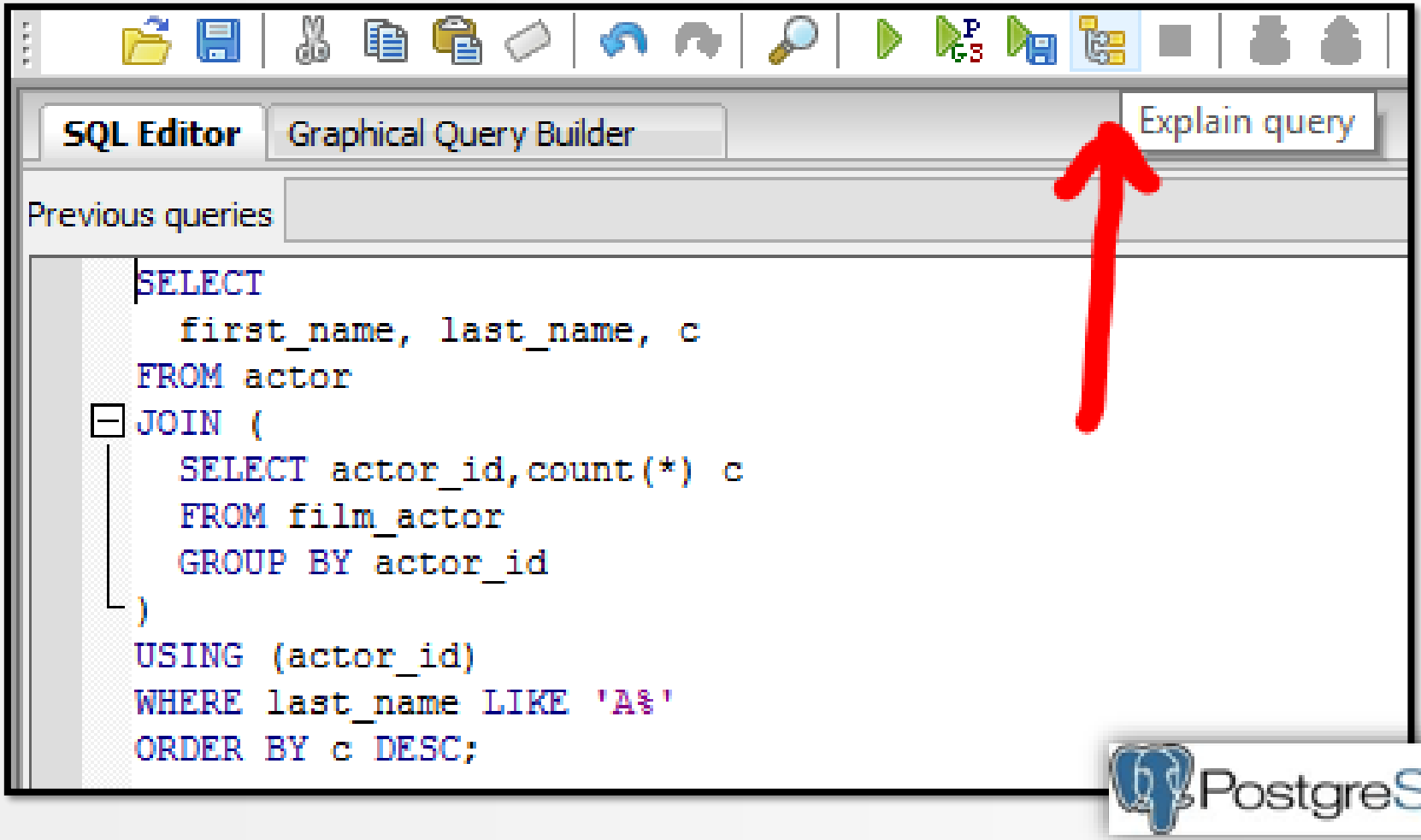
Who has seen an execution plan?



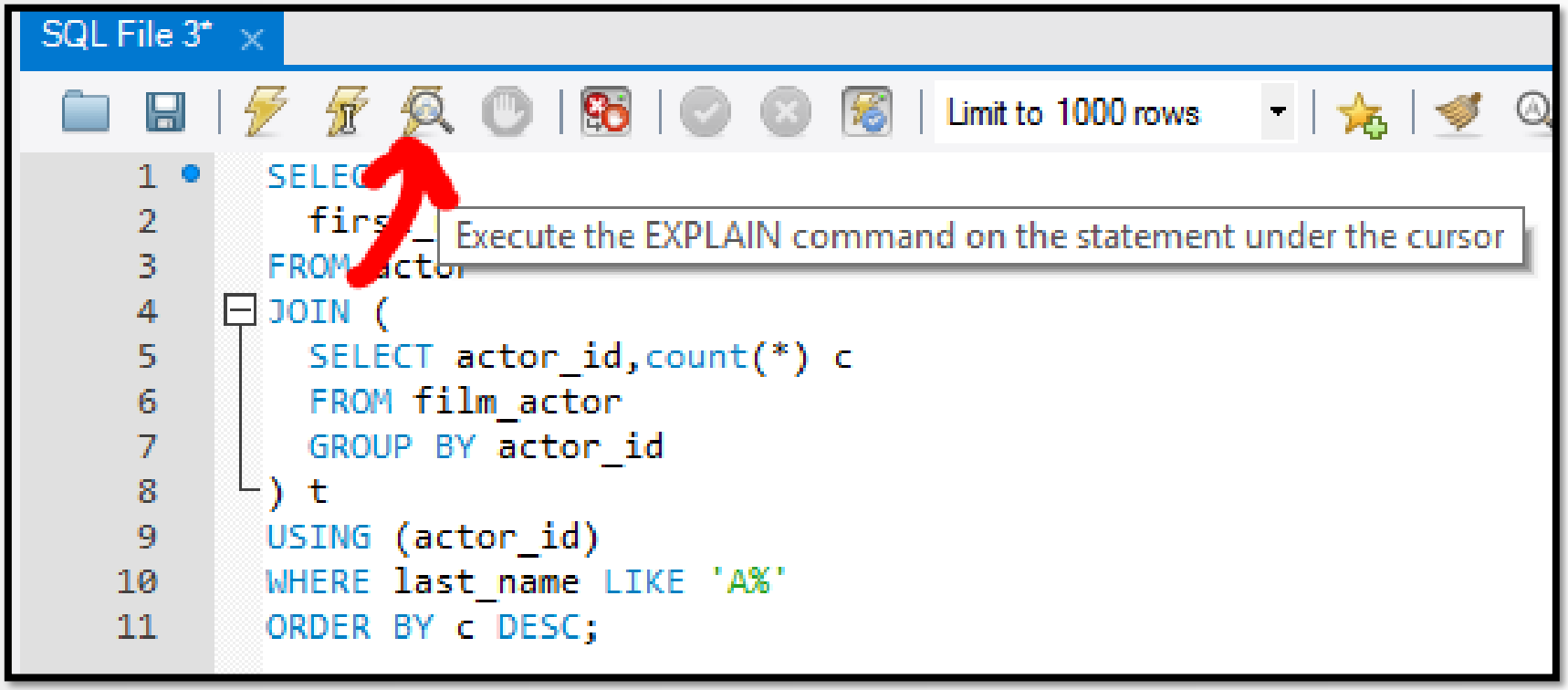
SQL Developer



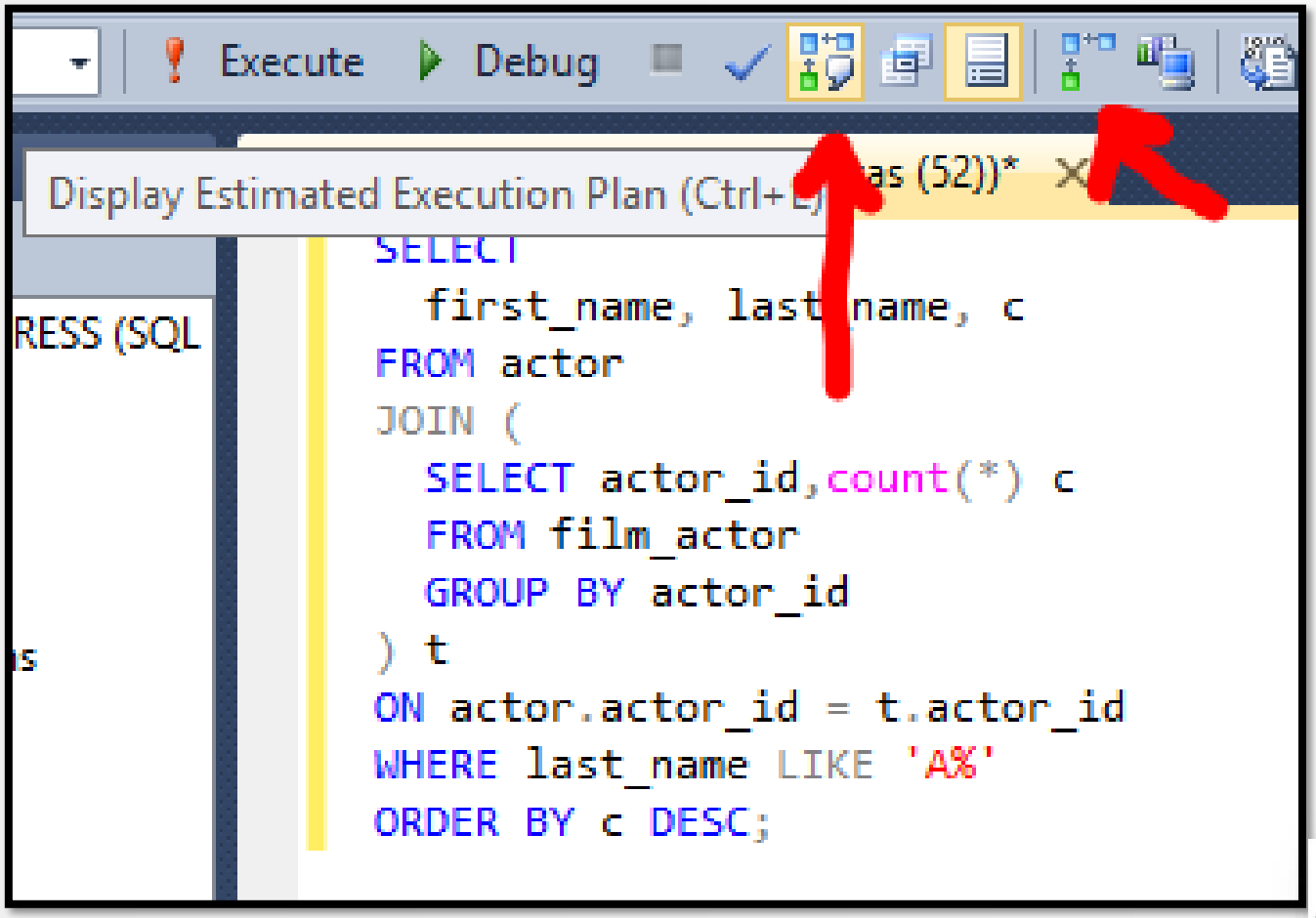
pgAdmin III



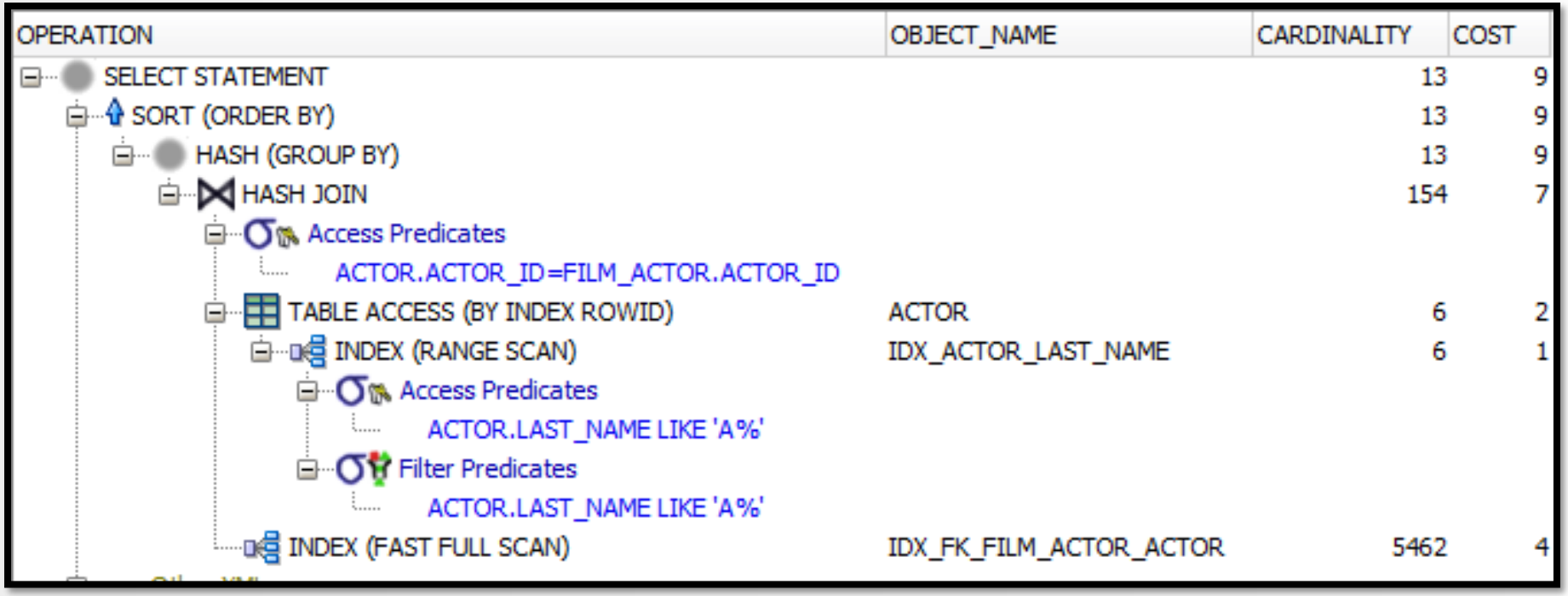
SQL Workbench



SQL Server Management Studio



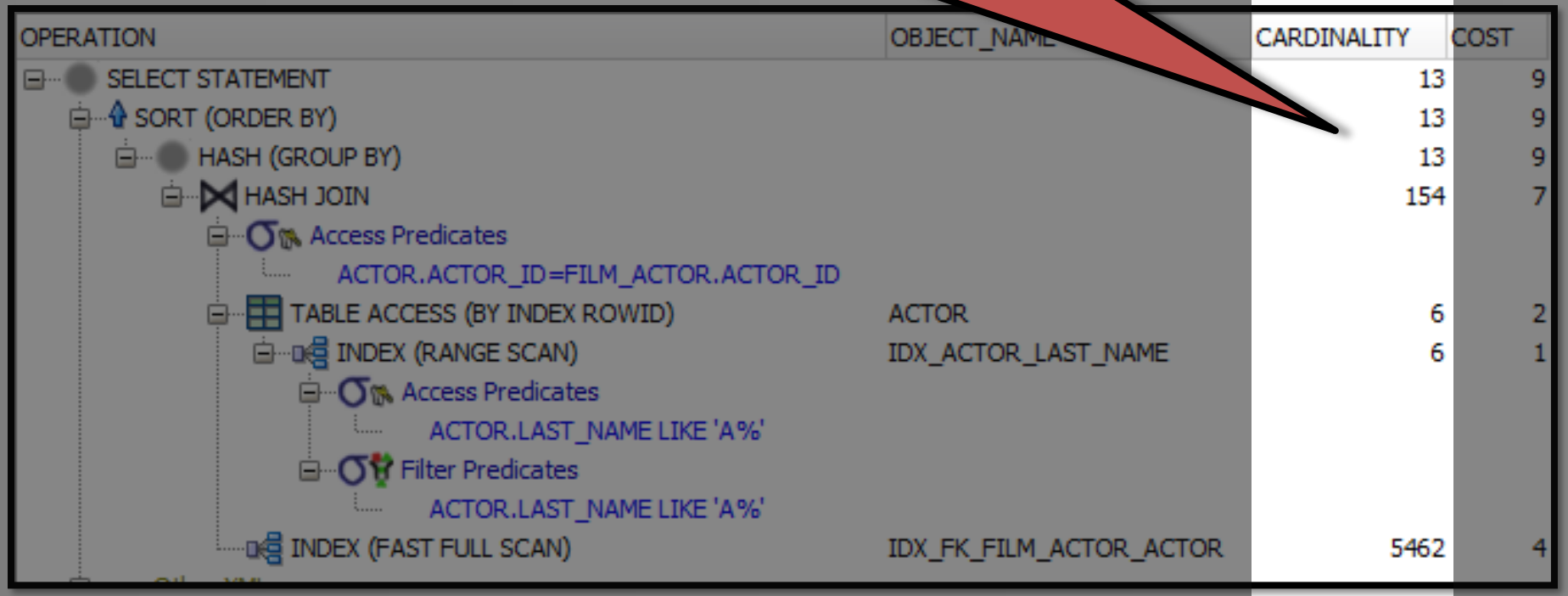
What's the actual algorithm?



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

What's the actual algorithm?

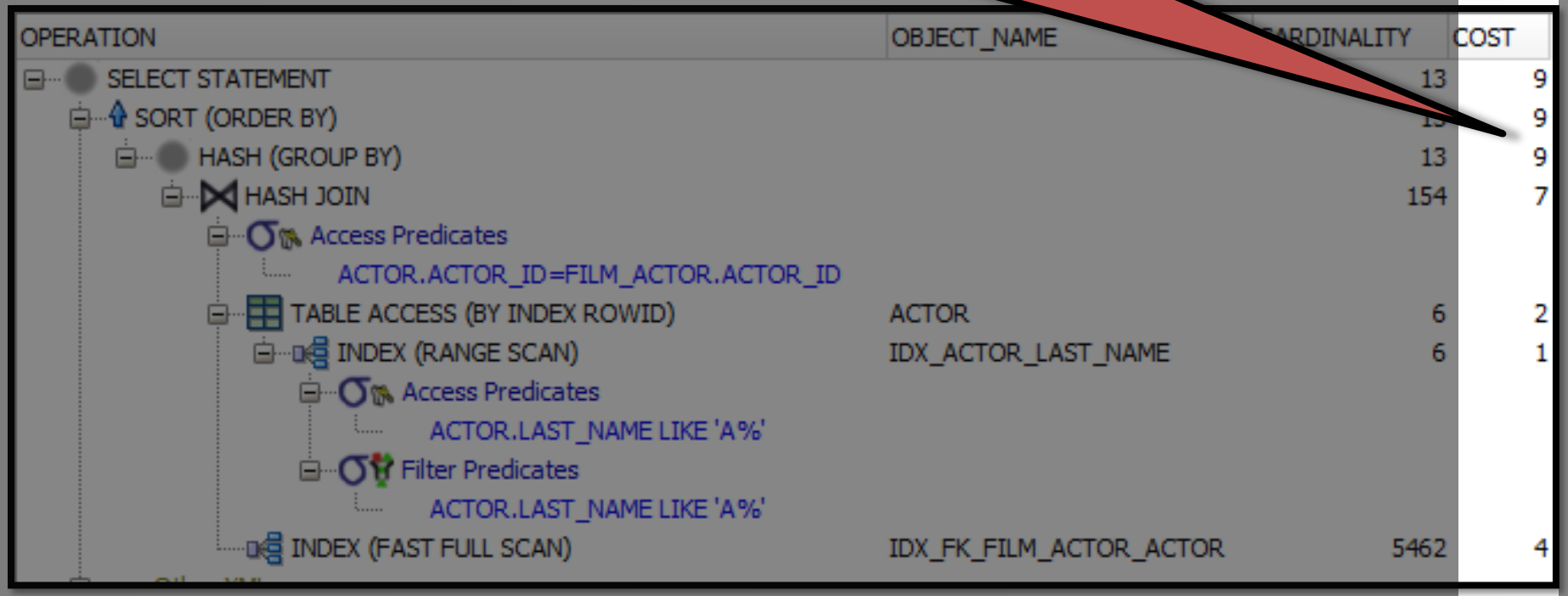
How many rows from operation



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

What's the actual algorithm?

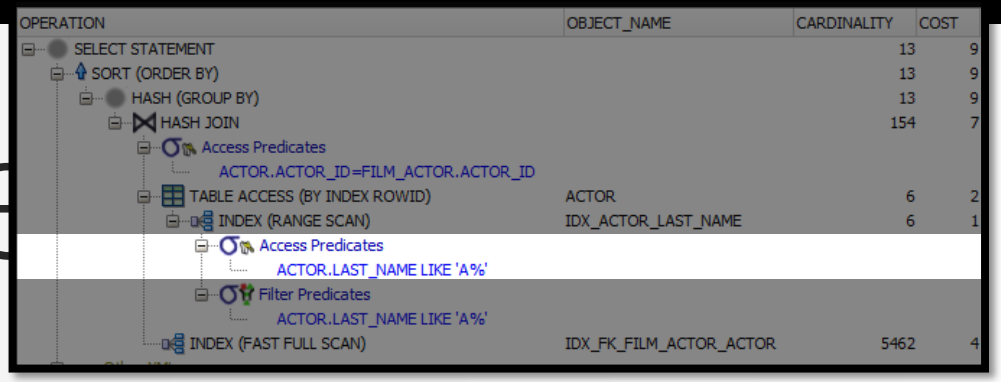
How much time does it take



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

What's the actual algorithm?

Imperative

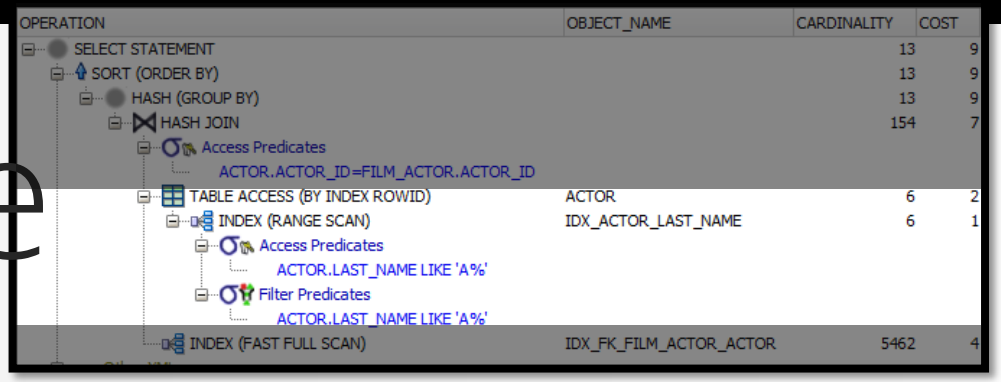


Actual operations order

1. Access ROWIDs from Index given the predicate

What's the actual algorithm?

Imperative



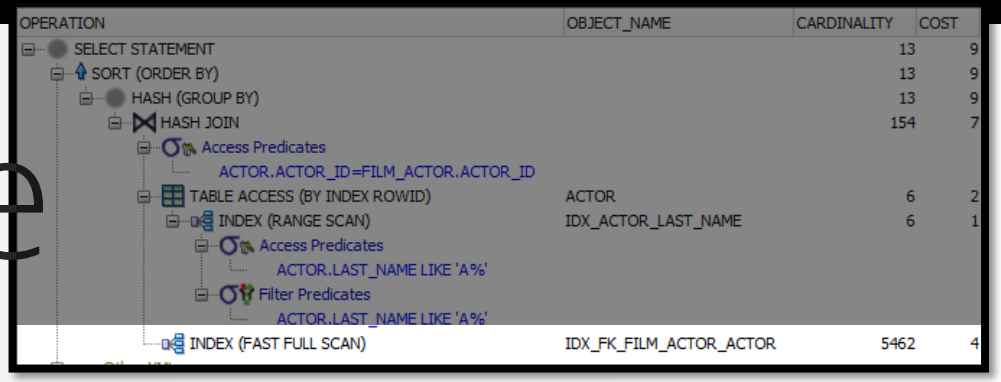
OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

Actual operations order

1. Access ROWIDs from Index given the predicate
2. Access other columns from the same row in the table

What's the actual algorithm?

Imperative



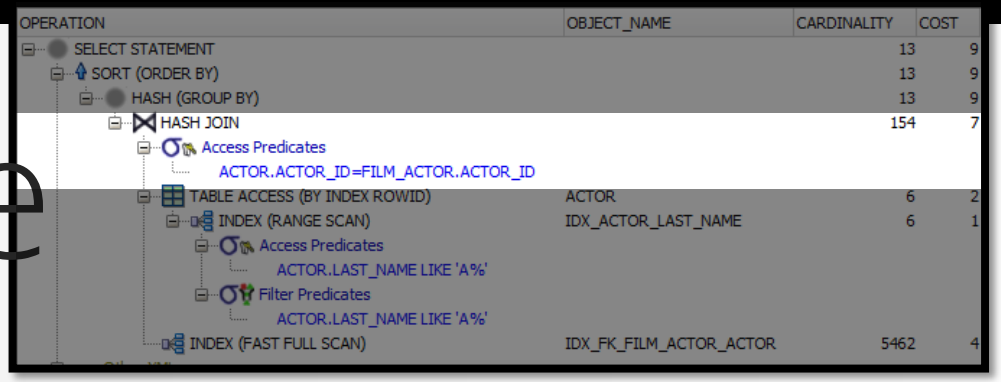
OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

Actual operations order

1. Access ROWIDs from Index given the predicate
2. Access other columns from the same row in the table
3. Load foreign key relationship index into memory

What's the actual algorithm?

Imperative

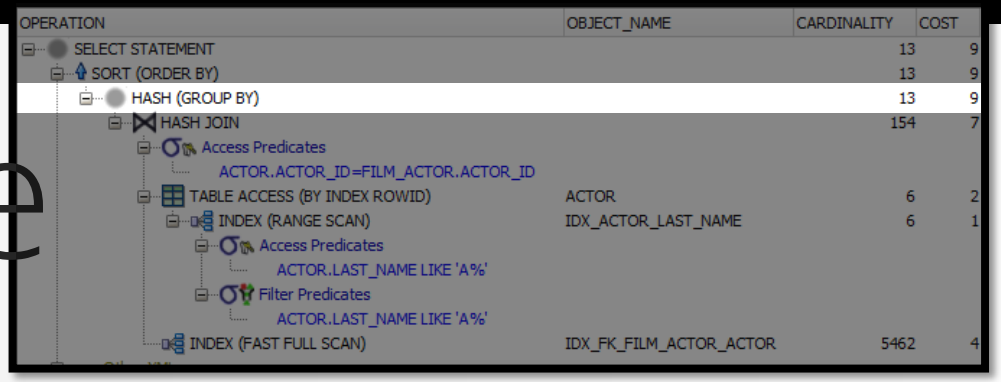


OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

Actual operations order

1. Access ROWIDs from Index given the predicate
2. Access other columns from the same row in the table
3. Load foreign key relationship index into memory
4. Put both data sets in hashmaps and join them

What's the actual algorithm?



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

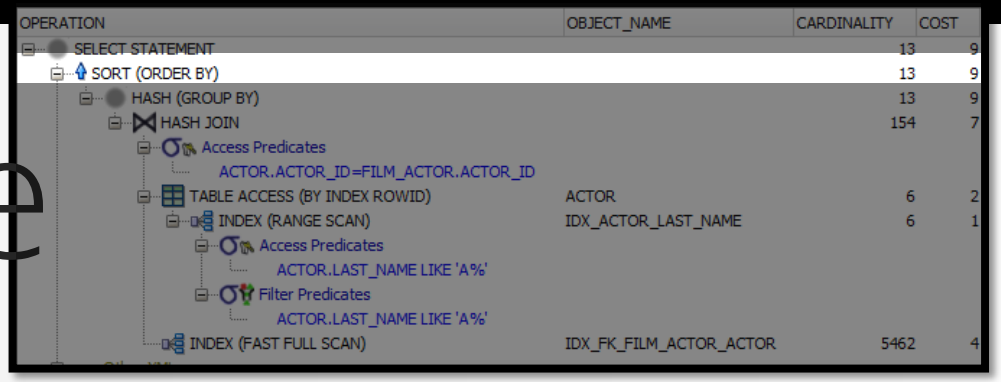
Imperative

Actual operations order

1. Access ROWIDs from Index given the predicate
2. Access other columns from the same row in the table
3. Load foreign key relationship index into memory
4. Put both data sets in hashmaps and join them
5. Create another hashmap to group rows

What's the actual algorithm?

Imperative



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

Actual operations order

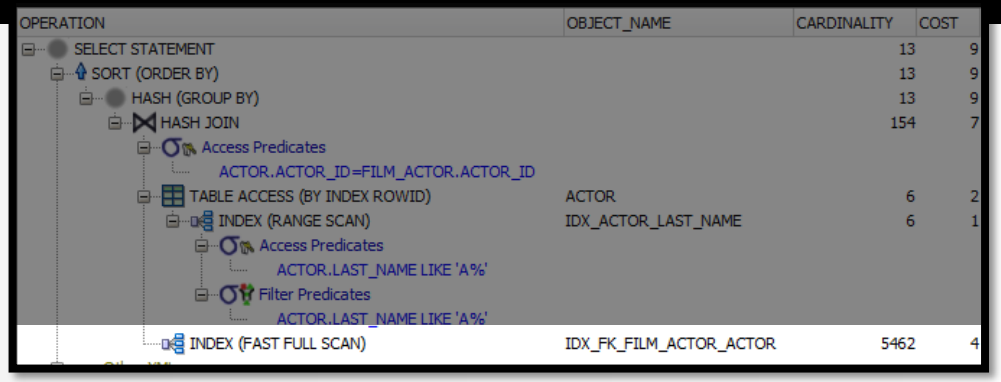
1. Access ROWIDs from Index given the predicate
2. Access other columns from the same row in the table
3. Load foreign key relationship index into memory
4. Put both data sets in hashmaps and join them
5. Create another hashmap to group rows
6. Sort the result

Huh!

Declarative vs Imperative

Would you have chosen the same algorithm?

Huh!



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		13	9
SORT (ORDER BY)		13	9
HASH (GROUP BY)		13	9
HASH JOIN		154	7
Access Predicates			
ACTOR.ACTOR_ID=FILM_ACTOR.ACTOR_ID			
TABLE ACCESS (BY INDEX ROWID)	ACTOR	6	2
INDEX (RANGE SCAN)	IDX_ACTOR_LAST_NAME	6	1
Access Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
Filter Predicates			
ACTOR.LAST_NAME LIKE 'A%'			
INDEX (FAST FULL SCAN)	IDX_FK_FILM_ACTOR_ACTOR	5462	4

E.g. Apparently, it's faster *in this case* to load 5462 rows into memory than seeking 6 counts individually…

Big O Notation

Hash join: O(N)

(N = film_actors (bigger table))

Nested loop join with indexes: O(log M x log N)

(M = actors, N = film_actors)

Big O Notation

Hash join: O(N)

(N = film_actors (bigger table))

Nested loop join with indexes:

O(log M x log N)

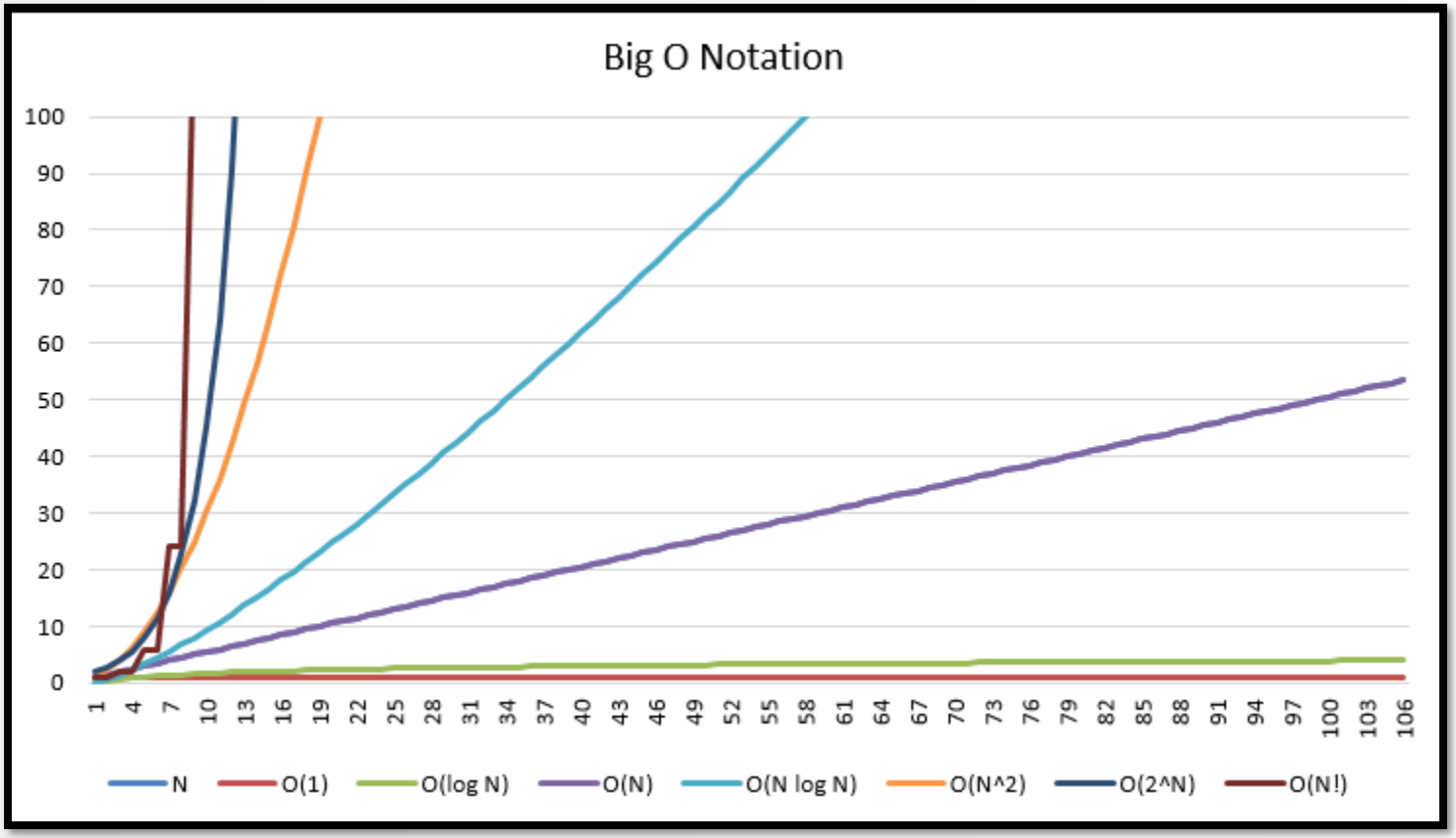
(M = actors, N = film_actors)

Which is better?

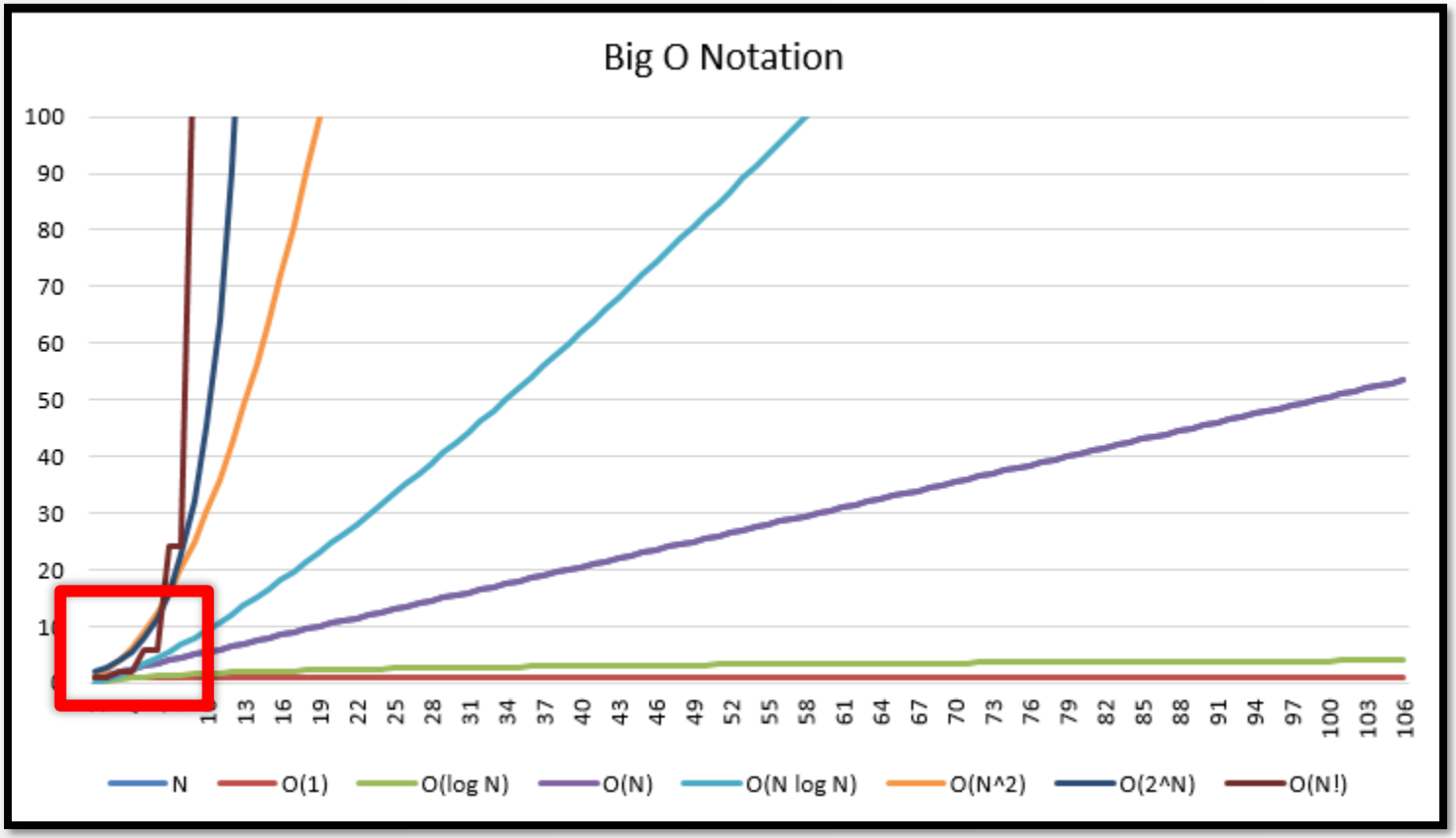
Well...

It depends!

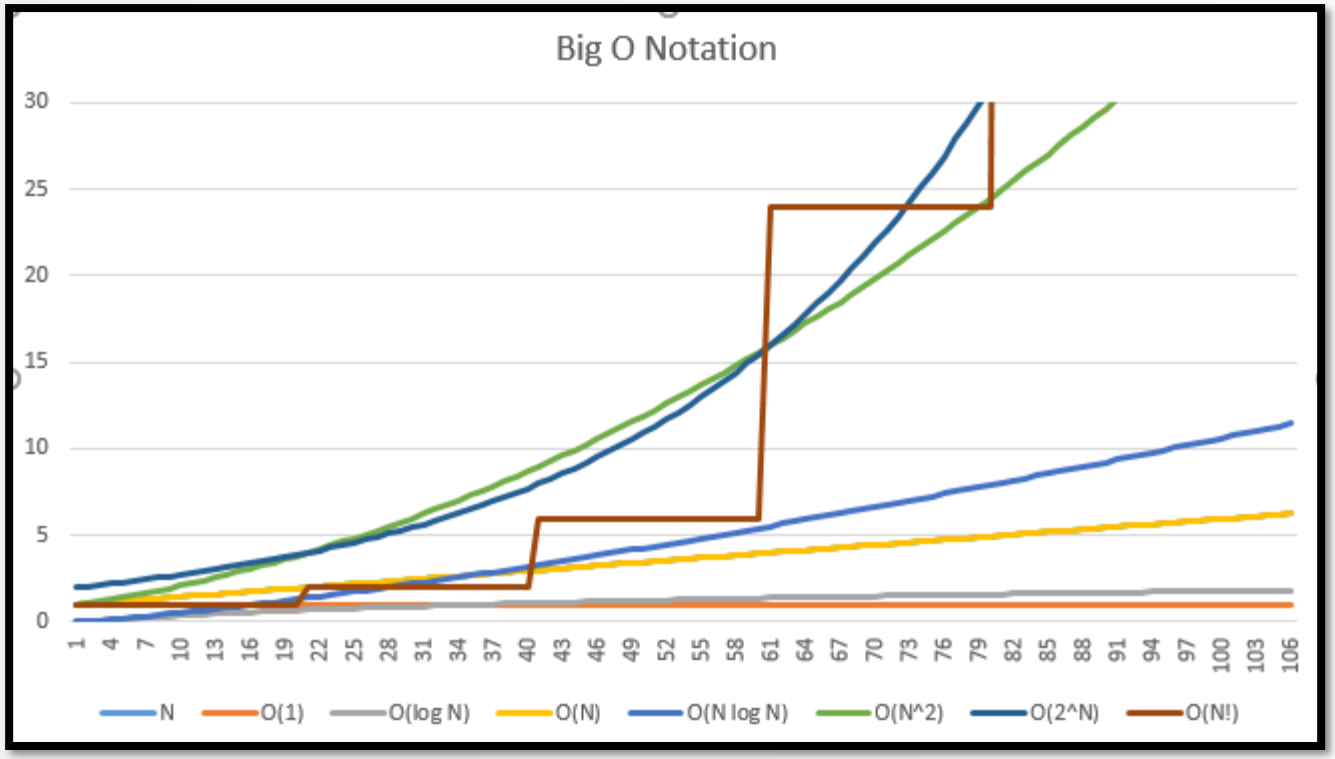
What you think you have



What you may really have

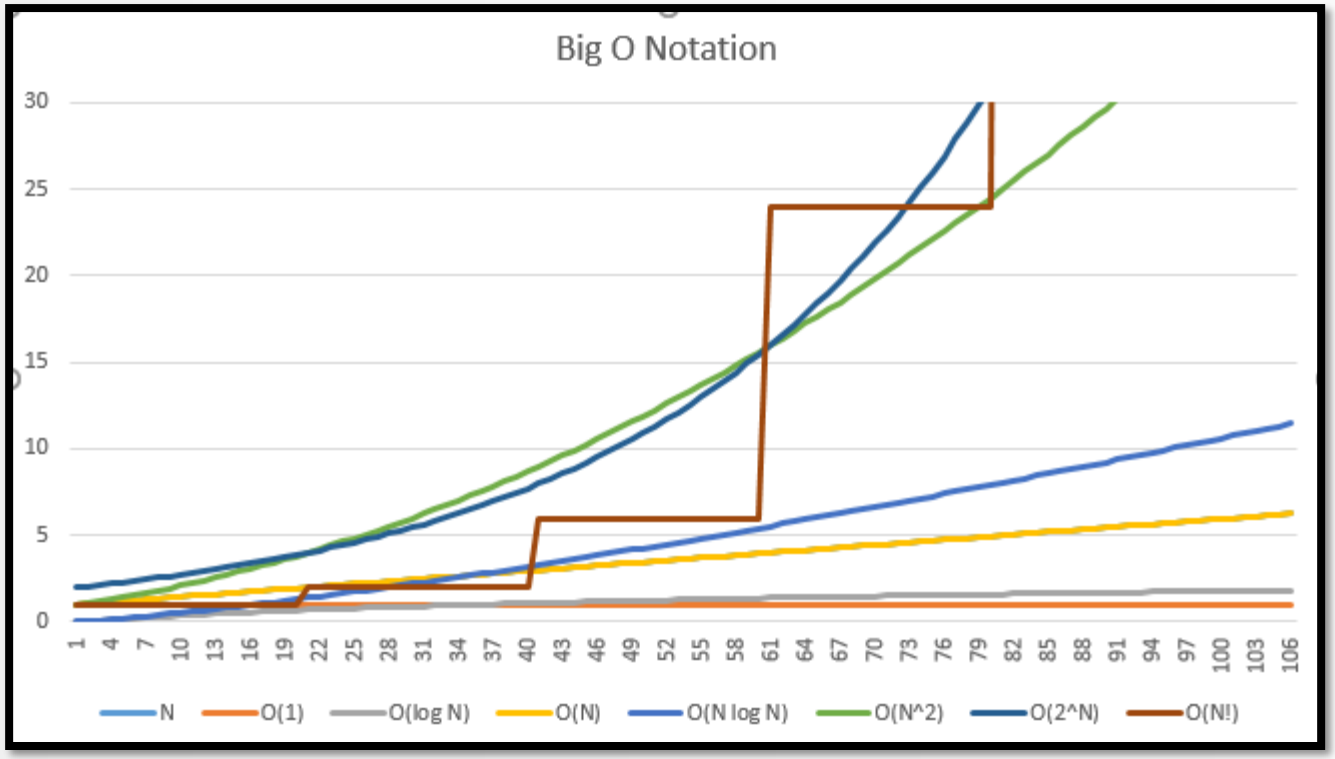


What you may really have



What you may really have

«Works on my machine» ಠ_ಠ



More info

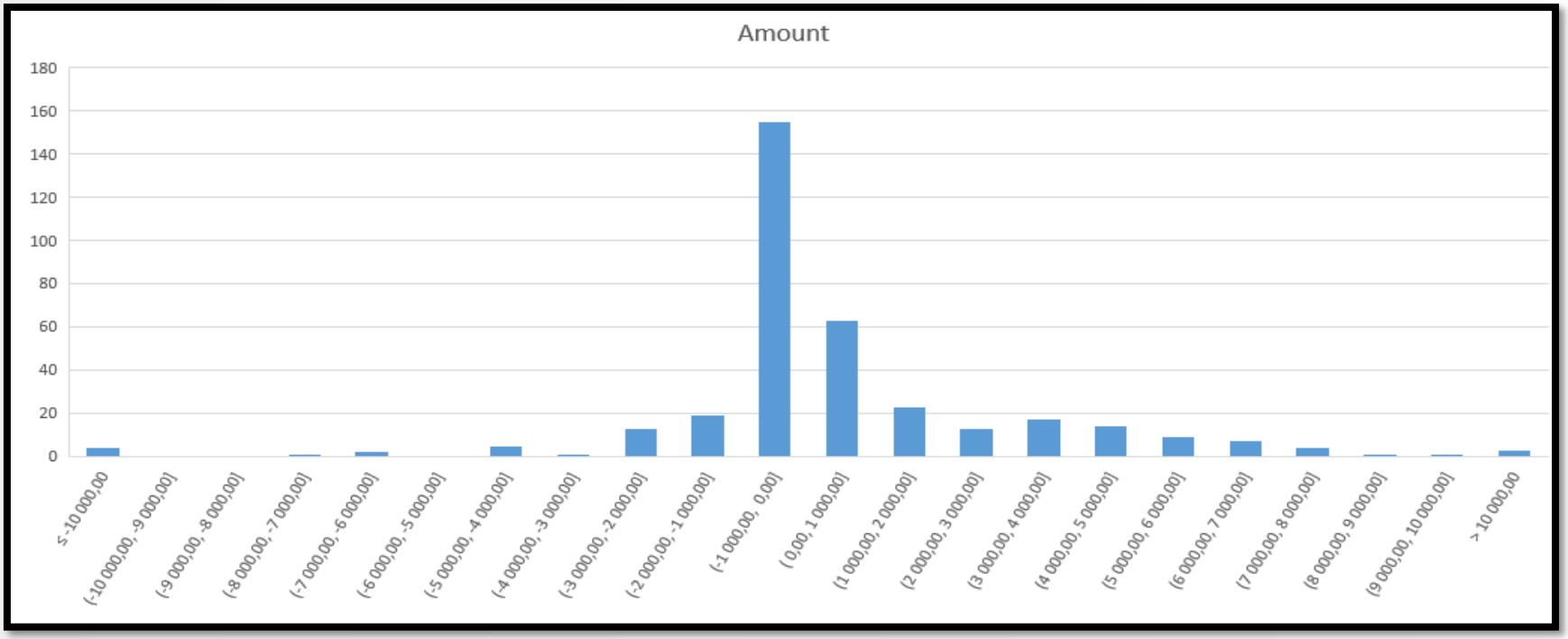
bigocheatsheet.com

Statistics

Here's an interesting thing to think about:

Histograms

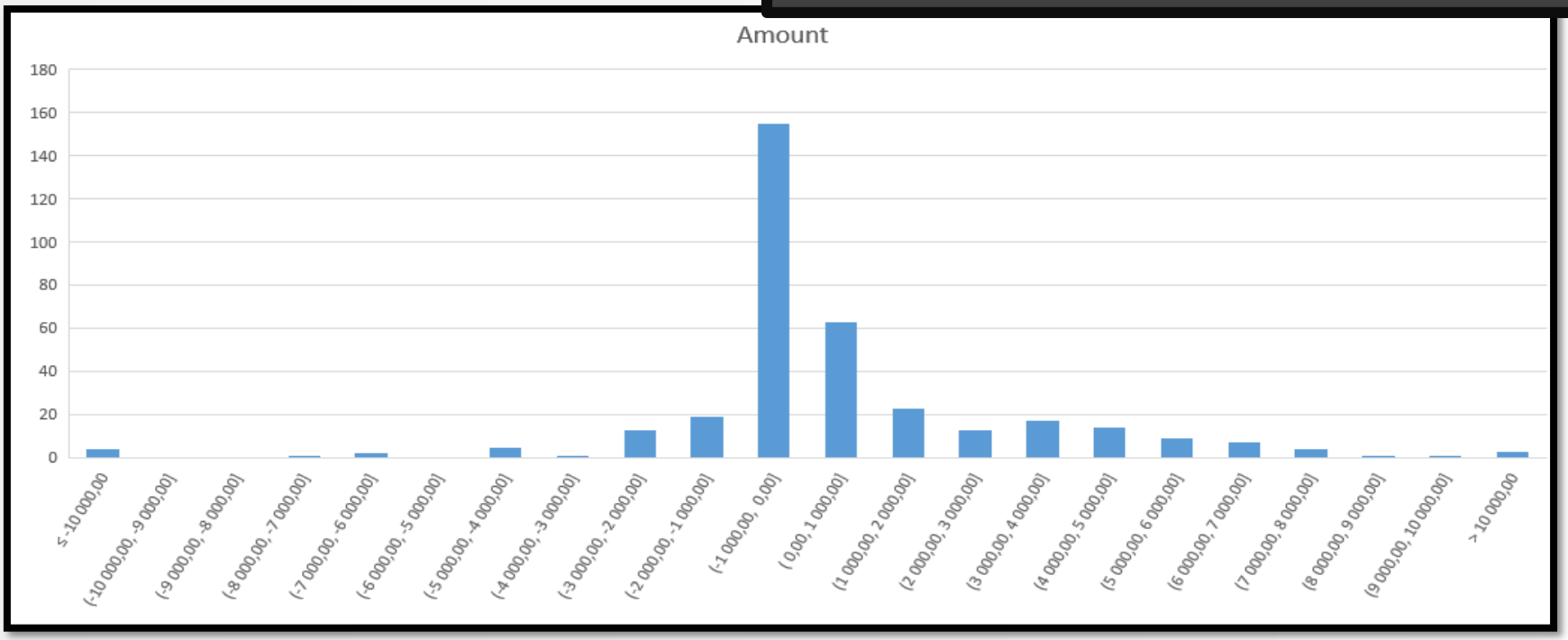
Histogram



Statistics

Histogram

```
-- Predicate 1
WHERE amount BETWEEN -2000 AND 2000

-- Predicate 2
WHERE amount BETWEEN 5000 AND 9000
```



Statistics

Histogram

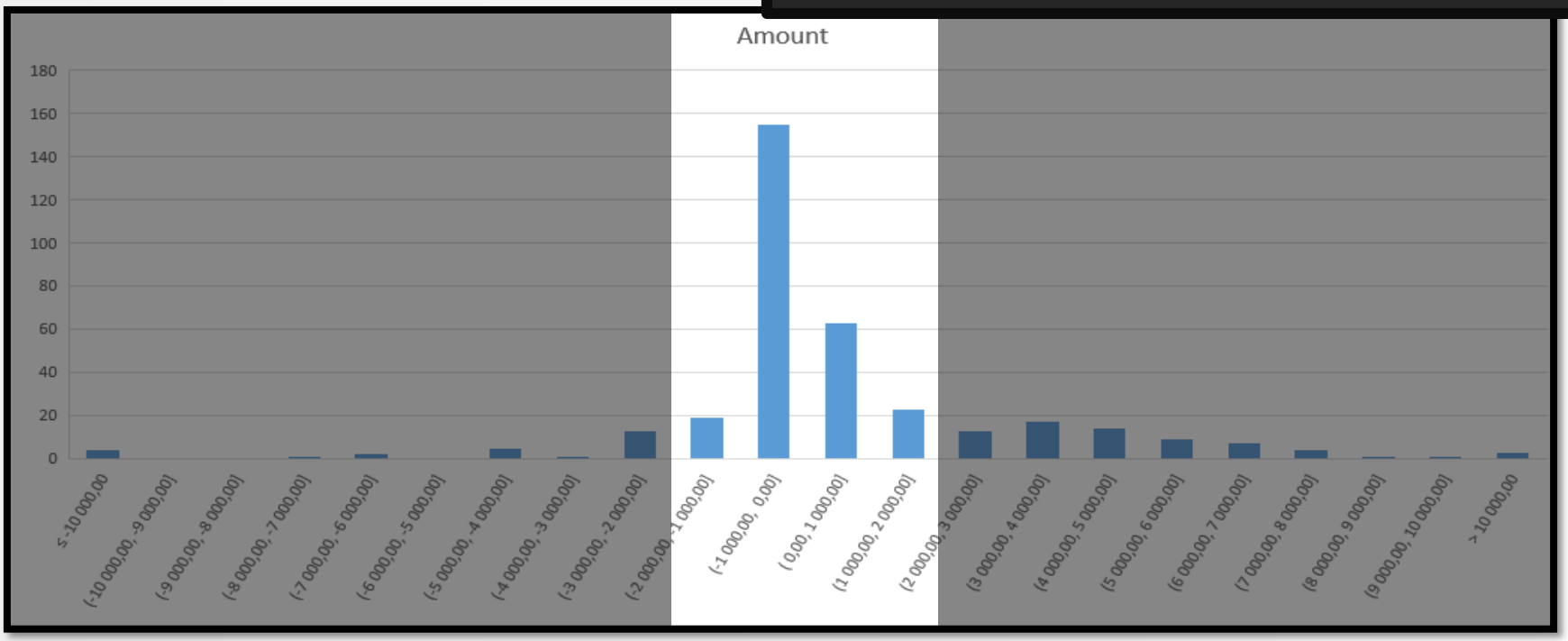
```
-- Predicate 1
WHERE amount BETWEEN -2000 AND 2000

-- Predicate 2
WHERE amount BETWEEN 5000 AND 9000
```

Do both predicates profit from indexing?

Statistics

Histogram

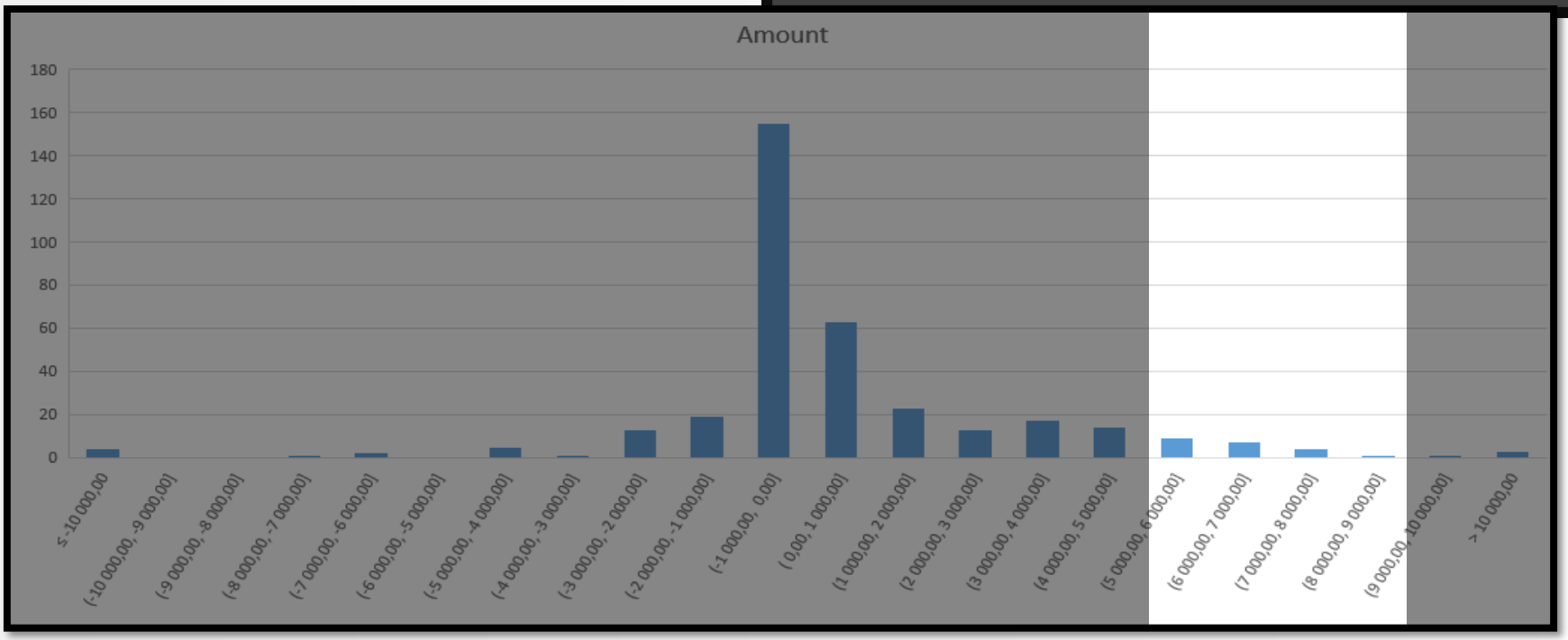
```
-- Not very selective predicate
WHERE amount BETWEEN -2000 AND 2000

-- Very selective predicate
WHERE amount BETWEEN 5000 AND 9000
```



Statistics

Histogram

```
-- Not very selective predicate
WHERE amount BETWEEN -2000 AND 2000

-- Very selective predicate
WHERE amount BETWEEN 5000 AND 9000
```



Statistics

Accessing the index is only worth the pain if it helps reduce the result set

Statistics

Accessing the index is only worth the pain if it helps reduce the result set

Otherwise, just read from the table

Statistics

The database knows all these things.

Statistics

The database knows all these things.

- How many rows are returned from a table?

Statistics

The database knows all these things.

- How many rows are returned from a table?
- From a query?

Statistics

The database knows all these things.

- How many rows are returned from a table?
- From a query?
- How much does it cost to access the disk?

Statistics

The database knows all these things.

- How many rows are returned from a table?
- From a query?
- How much does it cost to access the disk?
- The cache?

Statistics

The database knows all these things.

- How many rows are returned from a table?
- From a query?
- How much does it cost to access the disk?
- The cache?
- How often is this query run?

Statistics

The database knows all these things.

- How many rows are returned from a table?
- From a query?
- How much does it cost to access the disk?
- The cache?
- How often is this query run?
- How often is the table accessed?

Statistics

The database knows all these things.

- How much memory do we have?

Statistics

The database knows all these things.

- How much memory do we have?
- How many processes are running in parallel right now?

Statistics

The database knows all these things.

- How much memory do we have?
- How many processes are running in parallel right now?
- Is the operating system doing anything?

Statistics

The database knows all these things.

- How much memory do we have?
- How many processes are running in parallel right now?
- Is the operating system doing anything?
- How does the query perform with this bind variable?

Statistics

The database knows all these things.

- How much memory do we have?
- How many processes are running in parallel right now?
- Is the operating system doing anything?
- How does the query perform with this bind variable?
- Are we running on an HDD or SSD or Flash or RAM?

Statistics

The database knows all these things.

- How much memory do we have?
- How many processes are running in parallel right now?
- Is the operating system doing anything?
- How does the query perform with this bind variable?
- Are we running on an HDD or SSD or Flash or RAM?
- How many locks do we have on our rows?

Expression trees

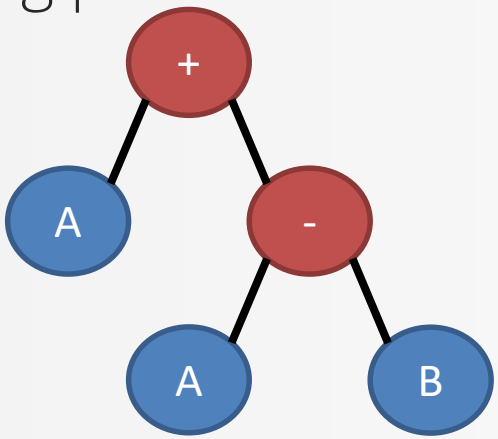
Everything has a cost

How does SQL work?

1. A parsed SQL string produces an expression tree

How does SQL work?

1. A parsed SQL string produces an expression tree



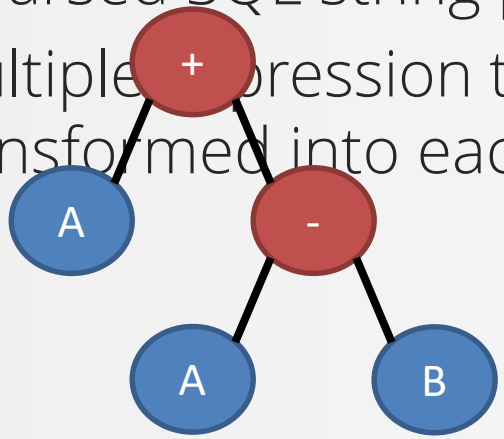

```
A + (A - B)
```

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other

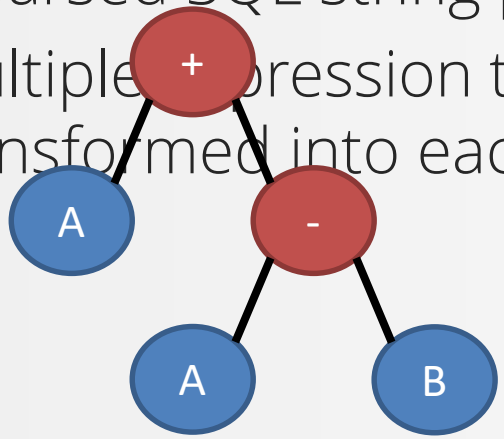


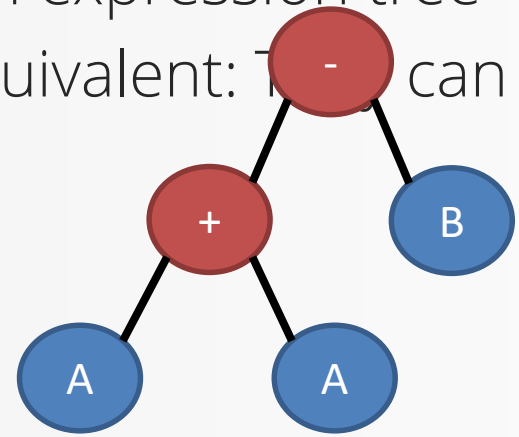

```
A + (A - B)
```

Expression trees

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other



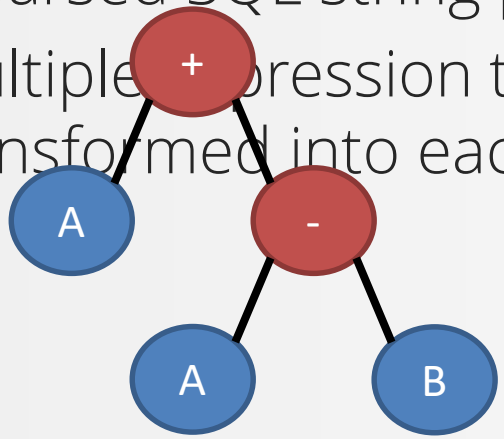


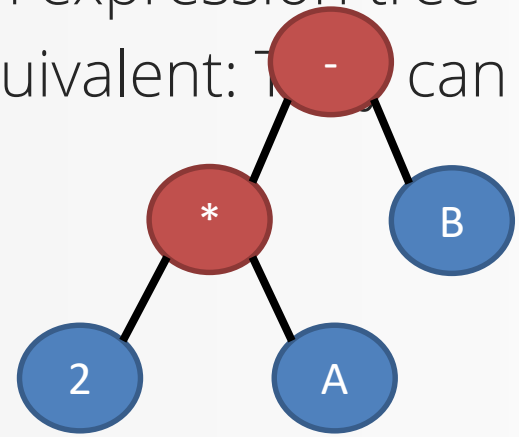

```
A + (A - B)
```

```
(A + A) - B
```

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other






```
A + (A - B)
```

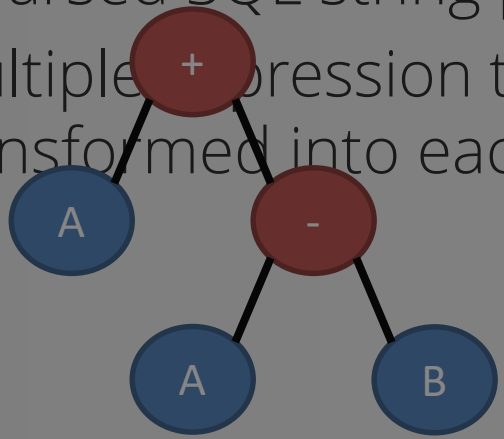
```
(2 * A) - B
```

Expression trees

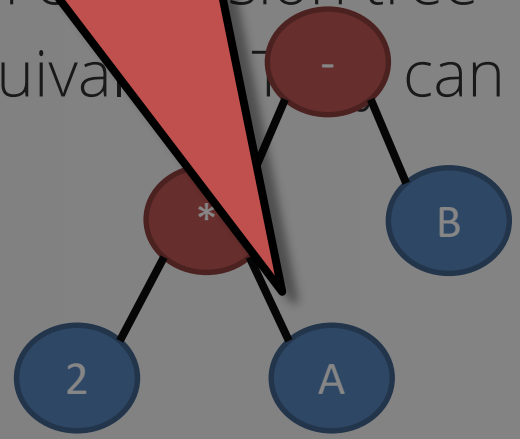
E.g.: Accessing A is super expensive

How does S work?

1. A parsed SQL string produces an e sion tree
2. Multiple pression trees are equiva T can be transformed into each other



A + (A - B)



(2 * A) - B

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other
3. Each expression tree has a corresponding execution plan and an associated, estimated cost

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other
3. Each expression tree has a corresponding execution plan and an associated, estimated cost
4. The «best» plan (= lowest estimated cost) is chosen

How does SQL work?

1. A parsed SQL string produces an expression tree
2. Multiple expression trees are equivalent: They can be transformed into each other
3. Each expression tree has a corresponding execution plan and an associated, estimated cost
4. The «best» plan (= lowest estimated cost) is chosen
5. The actual execution cost is compared with the estimate to improve future estimates

Expression trees

When's the last time you wrote Java code that did this?

How does SQL work?

1. A pa string produces an expression tree
2. Multi ression trees are equivalent: They can be trans d into each other
3. Each e ession tree has a corresponding execution plan a an associated, estimated cost
4. The «b t» plan (= lowest estimated cost) is chosen
5. The actual execution cost is compared with the estimate to improve future estimates

Expression trees

Your estimates aren't even done in production!!

How does SQL work?

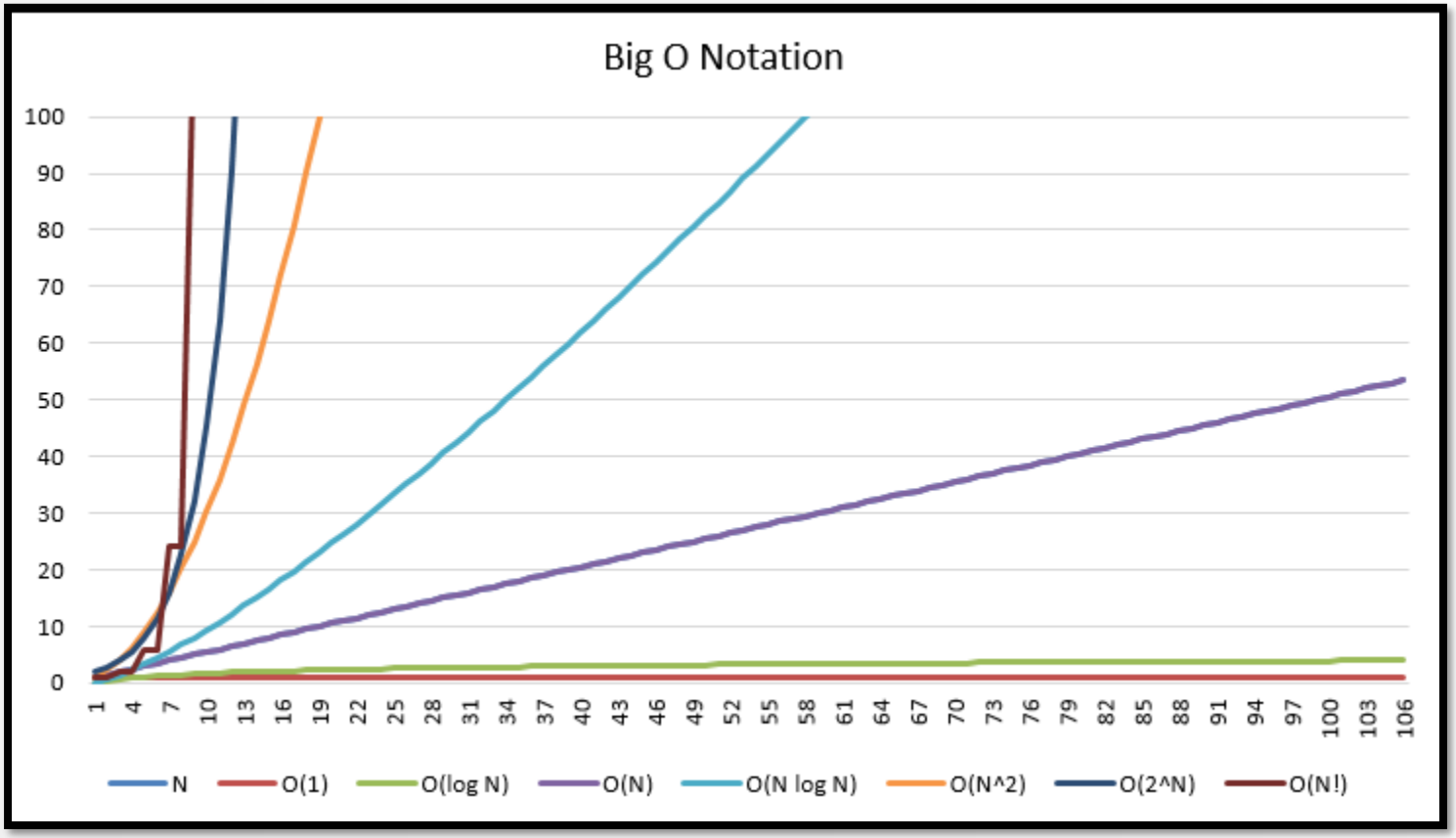
1. A pa string produces an expression tree
2. Multi ression trees are equivalent: They can be trans d into each other
3. Each e ession tree has a corresponding execution plan a an associated, estimated cost
4. The «b t» plan (= lowest estimated cost) is chosen
5. The actual execution cost is compared with the estimate to improve future estimates

Expression trees

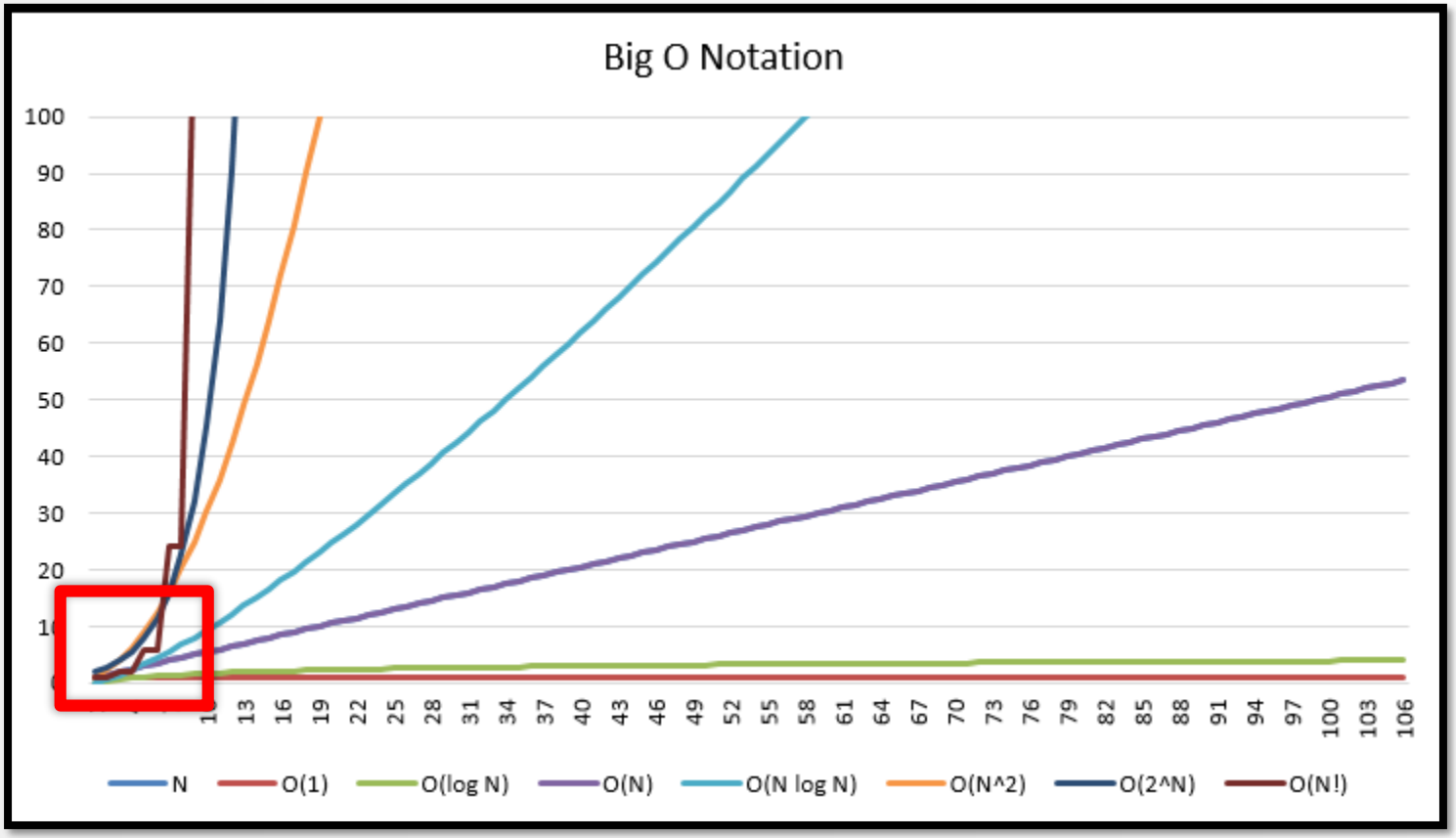
Special treat:

Oracle 12c can abort and switch execution plans «in flight». This is called «Adaptive Execution Plans».

Production is here



Your estimate / dev environment is here



What will I talk about?

- ~~SQL is awesome~~

- ~~SQL is productive~~

- SQL is fast

What will I talk about?

- ~~SQL is awesome~~

- ~~SQL is productive~~

- ~~SQL is fast~~

Why do I talk about SQL?

Now that you know all of this...

What do you prefer?

This?

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyPerStore =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories()) {
    Map<LocalDate, BigDecimal> daily =
      dailyPerStore.computeIfAbsent(
        inventory.getStoreId(), k -> new HashMap());
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
  }
}
```

Or, this?

```
SELECT
  title, store_id, payment_date, SUM(amount)
FROM film
JOIN inventory USING (film_id)
JOIN rental USING (inventory_id)
JOIN payment USING (rental_id)
GROUP BY film_id, store_id, payment_date
ORDER BY title, store_id, payment_date
```

Why do I talk about SQL?

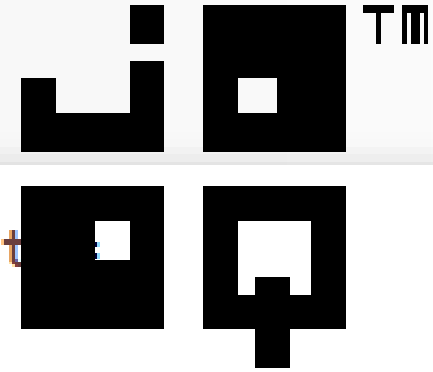
Why do I talk about SQL?

SQL is the only ever successful, mainstream, and general-purpose 4GL (Fourth-Generation Programming Language)

And it is awesome!

Why doesn’t anyone else talk about SQL?

Can I write SQL in Java?

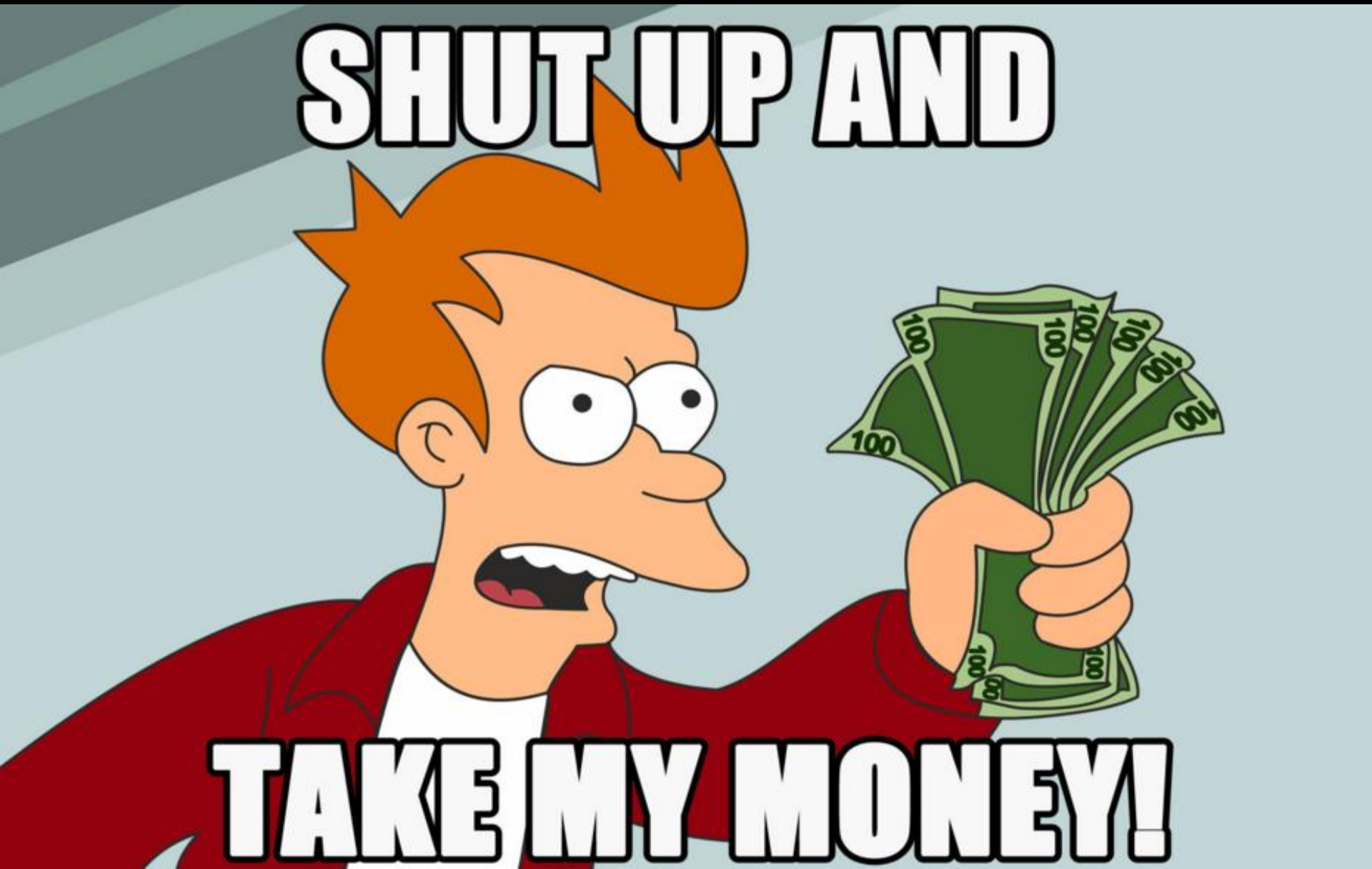


```
Result<Record2<String, String>> result =
dsl().select(
        ACTOR.FIRST_NAME,
        ACTOR.LAST_NAME)
     .from(ACTOR)
     .join(FILM_ACTOR)
     .on(ACTOR.ACTOR_ID.eq(FILM_ACTOR.ACTOR_ID))
     .where(ACTOR.FIRST_NAME.like("A%"))
     .fetch();
```

Can I write SQL in Java? – Yes. With jOOQ

Can I write SQL in Java? – Yes. With jOOQ

Can I write SQL in Java? – Yes. With jOOQ



So what's the key takeaway?

1. Can you do it in the database?

So what's the key takeaway?

1. Can you do it in the database? Yes

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database?

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)

http://www.jooq.org/training

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (… after visiting my 2 day SQL training)
3. Can you do it in *your* database?

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (… after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database?

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? No

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? No

JUST KIDDING!

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? Yes

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? Yes
5. Do long talk titles attract attention?

How Modern SQL Databases Come up with Algorithms that You Would Have Never Dreamed Of

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? Yes
5. Do long talk titles attract attention? Yes

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? Yes
5. Do long talk titles attract attention? Yes
6. Will this talk ever end?

So what's the key takeaway?

1. Can you do it in the database? Yes
2. Can *you* do it in the database? Yes
   (... after visiting my 2 day SQL training)
3. Can you do it in *your* database? Yes
   (... unless you're using MySQL)
4. *Should* you do it in the database? Yes
5. Do long talk titles attract attention? Yes
6. Will this talk ever end? Yes

If you haven't had enough

Google «10 SQL Tricks»

and find the other talk's transcript

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

Thank you

Check out our trainings:

http://www.jooq.org/training

Coordinates

- Blog: http://blog.jooq.org (excellent Java SQL content)
- Twitter: @JavaOOQ / @lukaseder (more lame jokes)
- E-Mail: lukas.eder@datageekery.com
- Bank account: CH57 8148 7000 0SQL AWSM 7

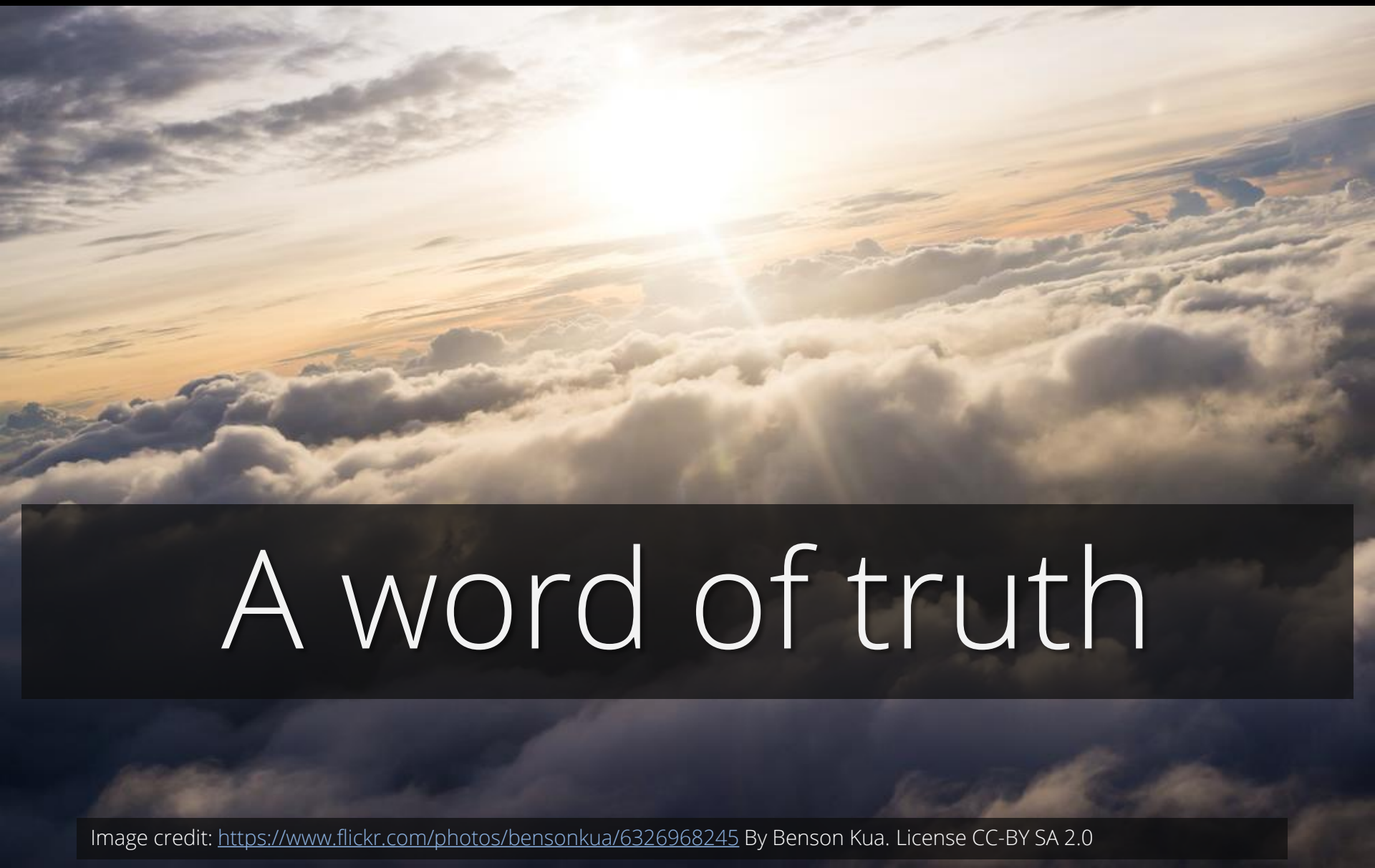
Why this question?

Your question was expected

Why this question?

But what about unit tests?

Testing SQL



Testing SQL

You don’t need tests for SQL queries!

It’s easy, stateless, side-effect free, declarative and just works – ask a functional programmer

Testing SQL

You do need tests for state transfer (writes)

But that is not related to SQL. You have to integration-test state transfer with any language!

This is why you test things. Too many bugs!

```
Map<Film, Map<Integer, Map<LocalDate, BigDecimal>>> result = ...;
List<Film> films = nPlusOneLoadAllFilms(); // TODO
for (Film film : films) {
  Map<Integer, Map<LocalDate, BigDecimal>> dailyPerStore =
    result.computeIfAbsent(film, k -> new HashMap<>());
  for (Inventory inventory : film.getInventories()) {
    Map<LocalDate, BigDecimal> daily =
      dailyPerStore.computeIfAbsent(
        inventory.getStoreId(), k -> new HashMap());
    for (Rental rental : inventory.getRentals())
      for (Payment p : rental.getPayments())
        daily.compute(
          p.getPaymentDate(),
          (k, v) -> v == null ? p.getAmount() : p.getAmount().add(v));
  }
}
```