How to un-lame your (relational) database

Katarzyna Kittel
kasia.kittel@gmail.com

Why performance is important?

- Important factor of usability - response time
- Performance of database have big impact of performance of an application.
- When amount of data increase the performance may decrease dramatically - application may become inaccessible.

How to assure performance?

- Test your application on database level since the early stage of development
- Test queries
- Simulate data grow
- Simulate users

EXAMPLE

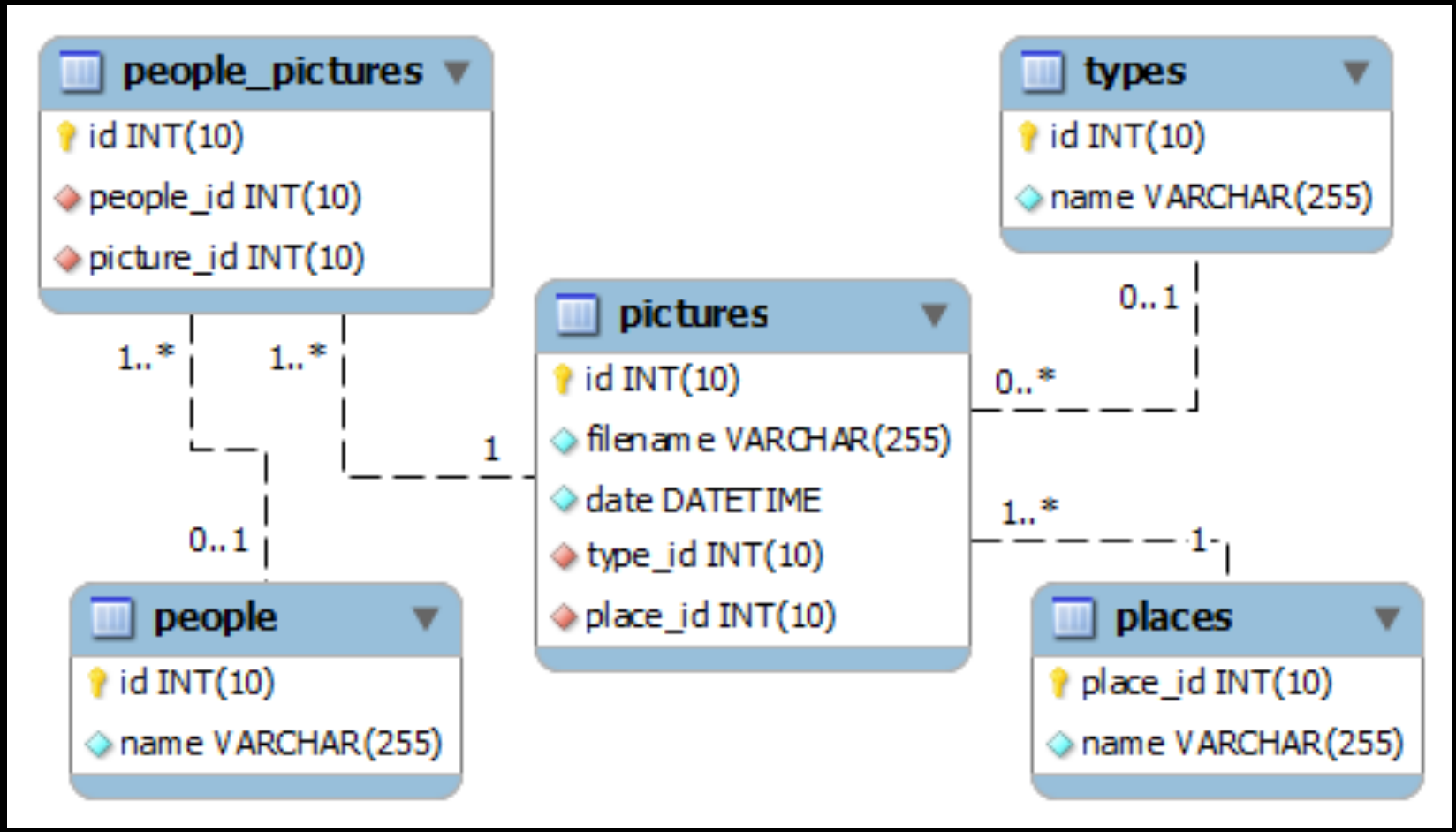
Sample application

Favorite serie fanpage. Pictures and characters...

Sample application

- Every picture has a filename and a date when it was taken.
- Every picture may be associated a type and a place where it was taken.
- Additionally every character can be tagged on several pictures, and every picture can be tagged with several characters.

Sample database



(MySQL) JOINS in a nutshell

JOIN (or simple WHERE)

Choose pictures with theirs types (only if a picture has a type).

using WHERE:

```
select filename, name from pictures, types where
pictures.type_id=types.id;
```

or using JOIN:

```
select filename, name from pictures join types on
(pictures.type_id = types.id);
```

(MySQL) JOINS in a nutshell

LEFT JOIN

Imagine a situation when you need all pictures. Doesn't matter if a picture has a type. In such case you can use LEFT JOIN.

(That means, the query returns all rows from the left table, even if there are no matches in the right table).

```
select filename, name from pictures left join types on
(pictures.type_id = types.id);
```

(MySQL) JOINS in a nutshell

RIGHT JOIN

We can also do something opposite: chose all types and join the pictures that belongs to them:

```
select filename, name from pictures right join types on
(pictures.type_id = types.id);
```

In most cases we can use LEFT or RIGHT JOIN to get the same results (anyway it is recommended using LEFT JOIN).

```
select filename, name from types left join pictures on
(pictures.type_id = types.id);
```

(MySQL) JOINS in a nutshell

NATURAL JOIN

We can use it to join two (or more) tables using columns with the same names.

NATURAL JOIN works without any join condition.

(WARNING: if you have a habit to name primary key column “id”, MySQL may use this column to create the conjunction.)

```
select filename, name from pictures natural join
places;
```

(MySQL) JOINS in a nutshell

INNER and CROSS JOIN

In MySQL they are equivalent one to each other and produce exactly the same result as JOIN.

A bit more advanced queries

SELECT syntax

SELECT
FROM table_references
[**WHERE** where_condition]
[**GROUP BY** {col_name | expr | position}
[ASC | DESC], ... [WITH ROLLUP]]
[**HAVING** where_condition]
[**ORDER BY** {col_name | expr | position}
[ASC | DESC], ...]
[**LIMIT** {[offset,] row_count | row_count OFFSET offset}]

A bit more advanced queries

SELECT syntax

- **WHERE** indicates the condition or conditions that rows must satisfy to be selected.
- **GROUP BY** must be used with any aggregating functions to group the resultset by one or more columns.
- **HAVING** works only with GROUP BY and should be used only with aggregation functions (while the WHERE clause cannot).
- **ORDER BY** is used to order the result set by a desired column or columns.

A bit more advanced queries

Many-to-many relationship

Let's choose all pictures with a list of people tagged on each of them.

```
select filename, name
from people_pictures
join pictures on (people_pictures.
picture_id = pictures.id) join people on
(people_pictures.person_id=people.id);
```

A bit more advanced queries

Many-to-many relationship

The result set has many rows with the same filenames. Sometimes this can be a disadvantage. There is an easy way to show every picture with coma-separated list of tagged characters. For that we can use **group_concat** function with **GROUP BY** clause:

```
select filename, group_concat(name)
from people_pictures
join pictures on
(people_pictures.picture_id = pictures.id)
join people on
(people_pictures.person_id=people.id)
group by filename;
```

A bit more advanced queries

Many-to-many relationship

If we need to exclude some results on some conditions we can use **HAVING** clause. Let's choose only the pictures with at least three people tagged on them.

```
select filename
from people_pictures
join pictures on
(people_pictures.picture_id = pictures.id)
join people on
(people_pictures.person_id=people.id)
group by filename
having count(name)>2;
```

A bit more advanced queries

Many-to-many relationship

We can also find all pictures with Ted.

```
select filename
from people_pictures
join people on
(people_pictures.person_id = people.id)
join pictures on
(people_pictures.picture_id=pictures.id)
where name = 'Ted';
```

A bit more advanced queries

Many-to-many relationship

At the end let's find the most tagged character.

```
select people.name
from people_pictures
join people on
(people_pictures.person_id = people.id)
group by person_id
order by count(person_id) desc
limit 1;
```

True story...

Real life (similar) example

Interface with two menus:

1. showing picture types and number of pictures for each type
2. showing characters and the number of pictures where they appear

Main content part shows 20 newest pictures.

True story...

Real life similar example

How we can assure good performance of the application?

- analyze, improve, test, ...

Prepare for stress

Real life similar example

Think which tables will grow faster and which should remain the same size.

Prepare for stress

Real life similar example

Estimate how the tables may grow:

20 000 users

average 5 pictures uploaded by a user

during 3 years

table *pictures*: 300 000 rows

average 3 characters on a pictures

table *people_pictures*: 900 000 rows.

Prepare for stress

Real life similar example

Make further assumption and generate sample data.

Some characters may appear on more pictures than others...

There may be more pictures taken in some particular places ...

(be careful with random generation with uniform distribution)

Prepare for stress

Real life similar example

Prepare queries. Think which queries will be used more often.

It is possible to use select cache (resultset cache)?

Prepare for stress

Why should we disconnect cache?

When a table changes the query cache flushes.

If there are tables that changes often we should disconnect cache to simulate real usage.

Prepare for stress

For the test use environment similar to the production environment.

same OS and OS version, same storage engines for tables...

Prepare for stress

Do benchmarking after every change on database or query.

Prepare for stress

Get a tool.

Let's slap MySQL

Mysqlslap

```
mysqlslap --user=root --password=root --create-schema=himym --concurrency=10 --iteration=3 --query=query3
```

--concurrency - the number of clients to simulate

--iteration - number of times to run the test

--create-schema - database to be tested

--query - query or file with queries

Let's slap MySQL

QUERY 1

```
select SQL_NO_CACHE types.name, count(pictures.id)
from types
left join pictures on pictures.type_id=types.id
group by types.name
union
select SQL_NO_CACHE 'other' as name, count(id)
from pictures
where type_id is NULL;
```

average time: 22.606 s

Let's slap MySQL

First improvement: customize data types

Keep the database as small as possible. This will save memory for data and the size of indexes.

eg.

- **TINYINT** instead of **INT** where possible (TINYINT occupy only 1 Byte since INT 4 Bytes)
- **VARCHAR** with good estimate of number of character instead of **CHAR**

Let's slap MySQL

QUERY 1 (again)

```
select SQL_NO_CACHE types.name, count(pictures.id)
from types
left join pictures on pictures.type_id=types.id
group by types.name
union
select SQL_NO_CACHE 'other' as name, count(id)
from pictures
where type_id is NULL;
```

Size of type.id and place.id is denomided to TINYINY.
Now the average time: 14.122 s

Let's slap MySQL

Second improvement: add indexes

Indexes should be created carefully. In most cases, added correctly, may dramatically speed up the queries. Generally speaking we should create indexes on these columns that we use for JOINs and clauses WHERE, ORDER BY and GROUP BY.

Test your queries after adding a new index.

Let's slap MySQL

QUERY 1 (again)

```
select SQL_NO_CACHE types.name, count(pictures.id)
from types
left join pictures on pictures.type_id=types.id
group by types.name
union
select SQL_NO_CACHE 'other' as name, count(id)
from pictures
where type_id is NULL;
```

Indexes on people_pictures.person_id, people_pictures.picture_id, pictures.type_id

Now the average time: **6.021s**

Let's slap MySQL

QUERY 2

```
select SQL_NO_CACHE filename, date
from pictures
where type_id = 5
order by date
limit 10 OFFSET 0;
```

We can observe even better improvement for simple select.

For this query adding an index on ***pictures.date*** improves the timing from:

6.408 to **0.039**.

Let's slap MySQL

EXPLAIN some important information

Join Type

- ALL - full table scan will occur - very bad indicator
- index, seems much better but works only with some storage engines
- ref - all rows with matching values are read

Let's slap MySQL

EXPLAIN some important information

Possible indexes

- show what indexes may be used
- if this column is NULL, there are no relevant indexes. In this case, you may be able to improve the performance of your query by examining the WHERE clause to check whether it refers to some column or columns that would be suitable for indexing.

Let's slap MySQL

EXPLAIN some important information

Key

The key column indicates the key (index) that MySQL actually decided to use. If MySQL decides to use one of the possible_keys indexes to look up rows, that index is listed as the key value.

BTW: MySQL may use the indexes more efficiently if they are the same type and size.

Let's slap MySQL

EXPLAIN some important information

Rows to examine

The rows column indicates the number of rows MySQL believes it must examine to execute the query.

MySQL analyze the key distribution to provide best optimization plan. By default they key distribution is assumed to be uniform

Use ANALYZE table to reinitiate this analyze on update table data.

Let's slap MySQL

EXPLAIN some important information

Extra info

Useful hints ex. unnecessary filesort or conditions that will never be fulfilled

Let's slap MySQL

EXPLAIN is your friend

Let's go back to QUERY 1:

select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
PRIMARY	types	ALL					5	Using temporary; Using filesort
PRIMARY	pictures	ref	index_type_id	index_type_id	2	himym3.types.id	6	(null)
UNION	pictures	ref	index_type_id	index_type_id	2	const	41539	Using where
UNION RESULT	<union1, 2>	ALL						(null)

Let's slap MySQL

Third improvement: get rid of UNION
and NULL columns if possible

Try to keep your queries simple.

NULL may occupy more space than other data types.

Why there is a filesort?

MySQL uses filesor by default for GROUP BY. Whe can turn it off using ORDER BY NULL.

Let's slap MySQL

QUERY 1 (version2)

```
select SQL_NO_CACHE types.name, count(pictures.id)
from types
left join pictures
on pictures.type_id=types.id
group by types.name
order by NULL;
```

average time: 5.919s

Let's slap MySQL

Fourth improvement: trick with procedures and triggers

Most of the execution time is taken to process the ***pictures*** table, while the one that is more interesting for this query is the ***types*** table.

To overcome this we can provide an extra column ***types._count.*** Every time a picture is added, deleted or updated this column will be updated.

Let's slap MySQL

Fourth improvement: trick with procedures and triggers

To keep the data coherent we can use procedures and triggers that works inside the database and are not depended on any external interface.

Since triggers are atomic operation we don't need to worry about data integrity.

Let's slap MySQL

QUERY 1 (version 3)

```
select name, _count from types;
```

average time **0.004s**

Let's slap MySQL

Same resultset - different queries.

Let's list all characters and number of pictures where they are tagged.

We can make it in at least two ways:

Let's slap MySQL

QUERY 3 (version 1)

```
select SQL_NO_CACHE people.name, count(people_pictures.
id) as pictures_count
from people_pictures
right join people
on (people.id=people_pictures.person_id)
group by people.name
order by pictures_count desc;
```

Let's slap MySQL

QUERY 3 (version 2)

```
select SQL_NO_CACHE people.name, count(people_pictures.
id) as pictures_count
from people_pictures
left join people
on (people_pictures.person_id = people.id)
group by people.name
order by pictures_count desc;
```

Let's slap MySQL

Surprise!

Query 3 version 1: **10.559**
Query 3 version 2: **19.812**

10.559 vs. **19.812**

Let's slap MySQL

EXPLAIN Query 3 (version 1)

select type	table	type	possibl keys	key	key_len	ref	rows	Extra
SIMPLE	people	ALL					11	Using temporary; Using filesort
SIMPLE	people_ pictures	ref	index_per son_id	index_pe rson_id	1	himym4. people.id	16200	(null)

Let's slap MySQL

EXPLAIN Query 3 (version 2)

select type	table	type	possible keys	key	ref	rows	Extra
SIMPLE	people_ pictures	ALL				900010	Using temporary; Using filesort
SIMPLE	people	eq_ref	PRIMARY	PRIMAR Y	himym4. people_pictures. person_id	1	(null)

Let's slap MySQL

And the winner is.... ;)

```
select SQL_NO_CACHE people.name, count(people_pictures.
id) as pictures_count
from people
straight_join people_pictures on (people.
id=people_pictures.person_id)
group by people.name
order by pictures_count desc;
```

Let's slap MySQL

Why this happened?

- MySQL considers many factor to calculates optimization plan
- different optimization plan for left and right join
- number of rows for join is just an educated guess

  (use ANALYZE TABLE table - to analyze key distribution for table with data - this may improve optimization plan)
- use STRAIGHT_JOIN to force join order

Let's slap MySQL

Similar example - QUERY 4 version 1

```
select SQL_NO_CACHE filename, date
from pictures
join people_pictures
on (people_pictures.picture_id = pictures.id)
where person_id = 7
order by date
limit 10 OFFSET 0;
```

Let's slap MySQL

Similar example - QUERY 4 version 2

```
select SQL_NO_CACHE filename, date
from pictures
straight_join people_pictures
on (people_pictures.picture_id = pictures.id)
where person_id = 7
order by date
limit 10 OFFSET 0;
```

Let's slap MySQL

Similar example - QUERY 4 version 1 vs version 2

average execution time for 10 concurrent clients

2.660s vs. 0.226s

Let's slap MySQL

QUERY 5

```
select SQL_NO_CACHE filename, group_concat(name)
from people_pictures
left join pictures
on (people_pictures.picture_id=pictures.id)
join people
on (people_pictures.person_id=people.id)
group by filename
order by date
limit 1,20;
```

time for single query: 23.186s

Let's slap MySQL

QUERY 5

```
select SQL_NO_CACHE filename, group_concat(name)
from pictures
straight_join people_pictures
on (people_pictures.picture_id=pictures.id)
join people
on (people_pictures.person_id=people.id)
group by filename
order by date
limit 20;
```

time for single query: 8.030

Let's slap MySQL

QUERY 5

In fact we just need to get 20 newest pictures...

We could use where clause with subquery that will choose first 20 newest pictures...

but...

MySQL doesn't support LIMIT in subqueries..

Let's slap MySQL

QUERY 5

(In Postgres such query would look like this:)

```
select filename, array_agg(name)
from people_pictures
join people
on (people_pictures.person_id = people.id)
join pictures
on (people_pictures.picture_id = pictures.id)
where picture_id in
( select id from pictures order by date limit 20 )
group by filename;
```

Let's slap MySQL

Query 5

or we can divide it to two queries and join it in our program code:

Let's slap MySQL

Query 5

```
$query = 'select id from pictures order by date desc limit 1, 20;';
$res=$db->query($query);

$pictures_array=array();

foreach($res as $row){
    foreach($row as $value){
        $pictures_array[]=$value;
    }
}

$pictures = implode(', ', $pictures_array);

$query = "select picture_id, group_concat(name) from people_pictures
join people on (people_pictures.people_id=people.id) where picture_id
in ($pictures) group by picture_id";
$res=$db->query($query);
```

Let's slap MySQL

Conclusions

As we can see there are many factors that influence the performance of the database but also many solutions for improving the execution time.

- analyse design of your DB
- check if your indexes works
- run Explain for slow performing queries
- don't hesitate to use stored procedures
- go beyond the DB if necessary

- QUESTIONS?

- OBRIGADA!