САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

Виль Мария Юрьевна

Курсовая работа

Персонажи сериала «TeenWolf»

(Characters of TV-show «TeenWolf»)

Направление 01.03.02 «Прикладная математика и информатика»

Научный руководитель,
Старший преподаватель **Малинин К.А.**

Санкт-Петербург
2017

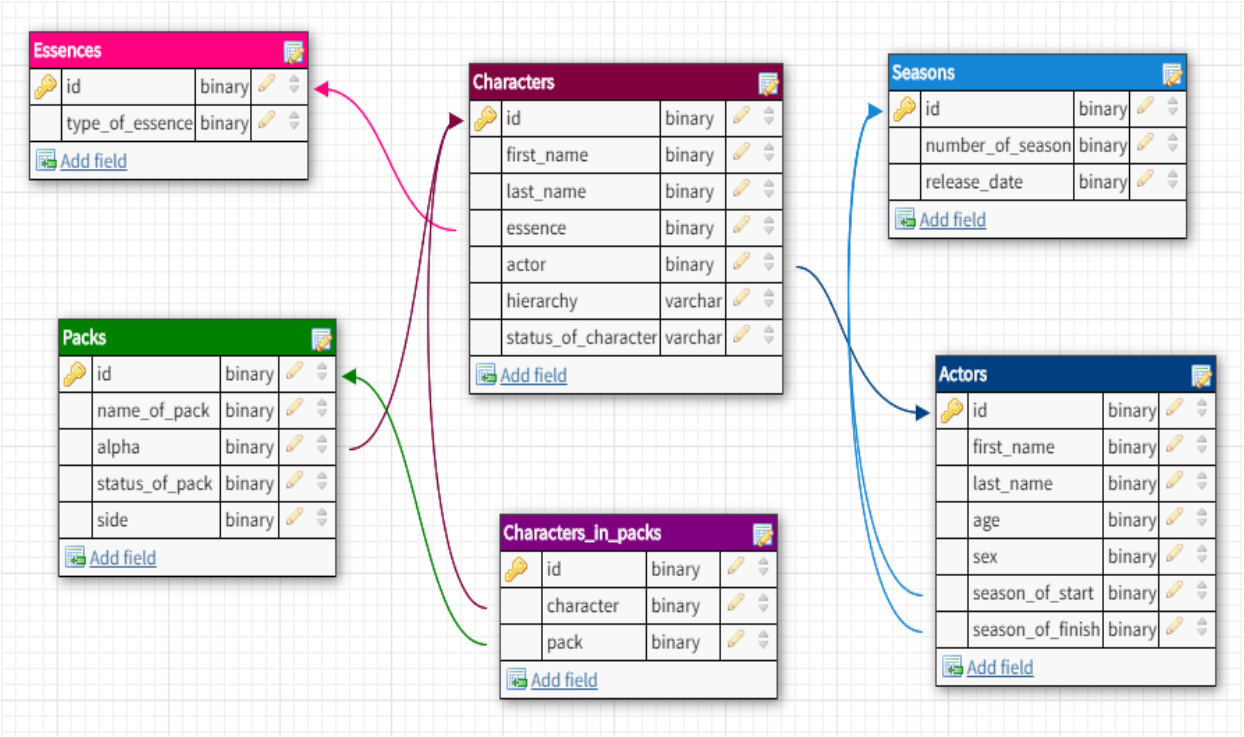
Содержание

Глава 1. Схема и описание базы данных

База данных основана на американском сериале «TeenWolf». «Волчонок» (англ. Teen Wolf) — молодёжный мистико-драматический телесериал, снятый в США по мотивам одноимённого фильма 1985 года по заказу канала MTV. Подробную информацию по нему можно найти, запросив в любом поисковой системе ключевые слова «TeenWolf», «Оборотень», «Волчонок».

В данной БД будут рассмотрены наиболее значимые, на мой взгляд, персонажи сериала и стаи, к которым они принадлежат.

Ниже приведена схема, отражающая таблицы, поля, а также все имеющиеся между ними связи. Типы данных на схеме указаны стандартные для веб-приложения DbDesigner, в котором эта схема была создана. Подробнее о типах данных будет написано позже.



В базе данных используются типы перечислений (enum), созданные вручную:

- **Hierarchy**
  - ‘Alpha’
  - ‘Beta’
  - ‘Omega’
  - ‘True Alpha’

- **Sex**
  - ‘M’
  - ‘W’
- **Side**
  - ‘Dark’
  - ‘Light’
  - ‘both’
- **Status**
  - ‘Alive’
  - ‘Dead’
- **Status_of_pack**
  - ‘is’
  - ‘is not’

Теперь коротко о каждой из таблиц базы данных:

1. **Characters:** содержит основную информацию о персонаже.
   Поля:
   - **id:** порядковый номер персонажа, никак не определяет его место или приоритетность.
     PRIMARY KEY, type **serial.**
   - **First_name:** имя персонажа (может отсутствовать).
     Type **varchar (50).**
   - **Last_name:** фамилия персонажа (может отсутствовать).
     Type **varchar (50).**
   - **Essence:** информация о том, к какому классу существ принадлежит персонаж (классы существ содержатся в таблице **Essences (FOREIGN KEY)**). Не может принимать нулевое значение, т.е. класс существ определен для каждого персонажа.
     Type **integer.**
   - **Actor:** информация об актере, сыгравшем персонажа (таблица **Actors (FOREIGN KEY)**). Для каждого персонажа существует

актер, сыгравший его (один актер может играть нескольких персонажей).
Type **integer.**

- **Hierarchy:** информация о положении персонажа в иерархии сверхъестественных существ, определяется не для всех персонажей.
  Type **hierarchy**.
- **Status_of_character:** информация о том, жив или мертв персонаж на момент финальной серии. Определяется для всех персонажей.
  Type **status**.

2. **Actors:** содержит информацию об актерах, сыгравших тех или иных персонажей. (существует как минимум 1 актер, сыгравший 2 персонажей)

   Поля:

   - **id:** порядковый номер актера, никак не определяет его место или приоритетность.
     PRIMARY KEY, type **serial**.
   - **First_name:** имя актера. Поле не может быть пустым.
     Type **varchar (50)**.
   - **Last_name:** фамилия актера. Поле не может быть пустым.
     Type **varchar (50)**.
   - **Age:** возраст актера. Определен для всех актеров. Значения могут быть больше 16 и меньше 100.
     Type **integer**.
   - **Sex:** пол актера. Поле не может быть пустым.
     Type **sex**.
   - **Season_of_start:** сезон, в котором актер впервые приступил к съемкам (номера сезонов содержатся в таблице **Seasons (FOREIGN KEY)**). Поле не может быть пустым.
     Type **integer**.
   - **Season_of_finish:** сезон, в котором актер завершил свое участие в сериале (номера сезонов содержатся в таблице **Seasons (FOREIGN KEY)**). Поле может быть пустым, т.к. существуют актеры, которые на момент последней вышедшей серии не завершили свои съемки в сериале.
     Type **integer**.

3. **Packs:** содержит информацию о стае (семье или группировке), которые появлялись в сериале на протяжении всего времени.

   Поля:

   - **id:** порядковый номер стаи, никак не определяет ее место или приоритетность.
     PRIMARY KEY, type **serial.**

- **Name_of_pack:** название стаи. Поле не может быть пустым, уникально.
  Type **varchar (50)**.
- **Alpha:** персонаж, являющийся вожаком стаи, определяется не для всех стай (или главой семьи или группировки, таблица **Characters (FOREIGN KEY)**)
  Type **integer**.
- **Status_of_pack:** информация о том, существует ли стая на момент последней вышедшей серии. Поле не может быть пустым.
  Type **status_of_pack.**
- **Side:** информация о том, как стая позиционируется по сюжету, является ли враждебной, дружественной или же содержит как положительных, так и отрицательных персонажей (данный параметр субъективен).
  Type **side.**

4. **Essences:** содержит информацию о видах существ, представленных в сериале.

   Поля:

   - **id:** порядковый номер существа, никак не определяет его место или приоритетность.
     PRIMARY KEY, type **serial.**
   - **Type_of_essence:** название класса существ. Поле не может быть пустым, уникально.
     Type **varchar (50)**.

5. **Seasons:** содержит номера сезонов (1, 2, 3a, 3b, 4, 5a, 5b, 6a, 6b).

   Поля:

   - **id:** порядковый номер сезона в таблице (не путать с номером сезона).
   - **PRIMARY KEY,** type **serial.**
   - **Number_of_season:** номер сезона. Поле не может быть пустым, уникально.
     Type **varchar (3)**.
   - **Release_date:** дата выхода первой серии сезона. Поле не может быть пустым. Дата не может быть раньше, чем 4 июня 2011 года.
     Type **date.**

6. **Characters in pack:** содержит информацию о том, какой персонаж в каких семьях (стаях) состоит. Один персонаж может состоять в нескольких стаях. Стая может содержать несколько разных персонажей.

   Поля:

   - **id:** порядковый номер связи, никак не определяет ее место или приоритетность.
     PRIMARY KEY, type **serial.**

   1. **character:** порядковый номер персонажа, никак не определяет его место или приоритетность.
      FOREIGN KEY, type **integer.**
   2. **Pack:** порядковый номер стаи, никак не определяет ее место или приоритетность.
      FOREIGN KEY, type **integer.**

Ссылка на репозиторий

Глава 2. Легкие запросы

1)Вывод списка имен и фамилий персонажей по параметру __status__, который может принимать значения 'Alive' и 'Dead'. То есть вывод имен песонажей, которые являются либо живыми, либо мертвыми. Отсортированно по фамилиям персонажей, в случае совпадения фамилий (или вообще отсутствия фамилии) сортировка производится по имени.

```
SELECT first_name, last_name
        FROM Characters
        WHERE status_of_character='__status__'
ORDER BY last_name, first_name;
```

Оптимизация:

Рассматриваемое значение параметра: __status__='Alive':

До оптимизации:

QUERY PLAN

```
 Sort  (cost=2.54..2.62 rows=34 width=14) (actual time=0.103..0.106 rows=34 loops=1)
   Sort Key: last_name, first_name
   Sort Method: quicksort  Memory: 26kB
   ->  Seq Scan on characters  (cost=0.00..1.68 rows=34 width=14) (actual time=0.015..0.028 rows=34 loops=1)
         Filter: (status_of_character = 'Alive'::status)
         Rows Removed by Filter: 20
 Planning time: 0.184 ms
 Execution time: 0.127 ms
```

Оптимизация:

Индексы, созданные по-умолчанию: characters_pkey

добавлен индекс characters_status_of_character_idx _

CREATE INDEX ON Characters(status_of_character);

После оптимизации:

QUERY PLAN

Sort (cost=2.54..2.62 rows=34 width=14) (actual time=0.116..0.119 rows=34 loops=1)

Sort Key: last_name, first_name

Sort Method: quicksort Memory: 26kB

-> Seq Scan on characters (cost=0.00..1.68 rows=34 width=14) (actual time=0.022..0.034 rows=34 loops=1)

Filter: (status_of_character = 'Alive'::status)

Rows Removed by Filter: 20

Planning time: 0.180 ms

Execution time: 0.145 ms

Индекс использован не был. Принудительно заставим использовать индекс, запретив Seq Scan:

SET enable_seqscan TO off;

QUERY PLAN

Sort (cost=13.60..13.68 rows=34 width=14) (actual time=0.164..0.168 rows=34 loops=1)

Sort Key: last_name, first_name

Sort Method: quicksort Memory: 26kB

-> Index Scan using characters_status_of_character_idx on characters (cost=0.14..12.74 rows=34 width=14) (actual time=0.027..0.043 rows=34 loops=1)

Index Cond: (status_of_character = 'Alive'::status)

Planning time: 0.203 ms

Execution time: 0.210 ms

Вывод: использование индекса только увеличивает cost и время выполнения запроса как для выборки (WHERE), так и для сортировки (ORDER BY).

2) Вывод списка имен и фамилий персонажей по двум условиям:

- Возраст персонажей входит в заданный промежуток, определяемый параметрами __age1__ (нижняя граница) и __age2__ (верхняя граница). Каждый из этих параметров может принимать значения от 16 до 100. Так же для корректной работы необходимо, чтобы параметр __age2__ был больше либо равен праметру __age1__ (в случае равенства будут выведены персонажи конкретного возраста)

- Пол персонажей либо только мужской, либо только женский. Определяется параметром __sex__ (возможные значения: 'M', 'W').

Список персонажей отсортирован по возрастанию возраста. Сортировка персонажей с одинаковым возрастом производится по имени.

```
SELECT first_name, last_name, age
        FROM Actors
        WHERE age>__age1__ and age<__age2__ and sex='__sex__'
ORDER BY age ASC, first_name;
```

Оптимизация:

Рассматриваемые значения параметров: __age1__=23; __age2__=45; sex='M';

До оптимизации:

QUERY PLAN

Sort (cost=2.30..2.34 rows=19 width=17) (actual time=0.073..0.075 rows=17 loops=1)

Sort Key: age, first_name

Sort Method: quicksort Memory: 26kB

-> Seq Scan on actors (cost=0.00..1.89 rows=19 width=17) (actual time=0.021..0.034 rows=17 loops=1)

Filter: ((age > 23) AND (age < 45) AND (sex = 'M'::sex))

Rows Removed by Filter: 34

Planning time: 0.254 ms

Execution time: 0.115 ms

Оптимизация:

Индексы, созданные по-умолчанию: actors_pkey

добавлен индекс actors_age_idx

CREATE INDEX ON ACTORS(age);

После оптимизации:

QUERY PLAN

Sort (cost=2.30..2.34 rows=19 width=17) (actual time=0.083..0.084 rows=17 loops=1)

Sort Key: age, first_name

Sort Method: quicksort Memory: 26kB

-> Seq Scan on actors (cost=0.00..1.89 rows=19 width=17) (actual time=0.030..0.043 rows=17 loops=1)

Filter: ((age > 23) AND (age < 45) AND (sex = 'M'::sex))

Rows Removed by Filter: 34

Planning time: 0.181 ms

Execution time: 0.117 ms

Индекс использован не был. Принудительно заставим использовать индекс, запретив Seq Scan:

SET enable_seqscan TO off;

QUERY PLAN

Sort (cost=13.29..13.33 rows=19 width=17) (actual time=0.103..0.105 rows=17 loops=1)

Sort Key: age, first_name

Sort Method: quicksort Memory: 26kB

-> Index Scan using actors_age_idx on actors (cost=0.14..12.88 rows=19 width=17) (actual time=0.054..0.063 rows=17 loops=1)

Index Cond: ((age > 23) AND (age < 45))

Filter: (sex = 'M'::sex)

Rows Removed by Filter: 15

Planning time: 0.196 ms

Execution time: 0.144 ms

Вывод: использование индекса только увеличивает cost и время выполнения запроса как для выборки (WHERE), так и для сортировки (ORDER BY). Кол-во Rows Removed by Filter уменьшилось.

Добавим еще один индекс actors_sex_idx

CREATE INDEX ON ACTORS(sex);

QUERY PLAN

Sort (cost=13.22..13.27 rows=19 width=17) (actual time=0.181..0.182 rows=17 loops=1)

Sort Key: age, first_name

Sort Method: quicksort Memory: 26kB

-> Index Scan using actors_sex_idx on actors (cost=0.14..12.82 rows=19 width=17) (actual time=0.114..0.131 rows=17 loops=1)

Index Cond: (sex = 'M'::sex)

Filter: ((age > 23) AND (age < 45))

Rows Removed by Filter: 13

Planning time: 0.211 ms

Execution time: 0.218 ms

Новый индекс использован, благодаря запрету на использование Seq Scan, но он также только увеличивает cost и время выполнения запроса как для выборки (WHERE), так и для сортировки (ORDER BY). Кол-во Rows Removed by Filter снова уменьшилось.

3) Вывод списка стай по параметрам __side__ (допустимые значения: 'Light', 'Dark', 'both') и __statusOfPack__ (допустимые значения: 'is', 'is not').

Т.е. вывод списка еще существующий или уже распавшихся стай, которые принадлежащих к одной из сторон.

```
SELECT name_of_pack
        FROM Packs
        WHERE side='__side__' and status_of_pack='__statusOfPack__'
```

Оптимизация:

Рассматриваемые параметры: __side__='Light'; __statusOfPack__='is';

До оптимизации:

QUERY PLAN

Seq Scan on packs (cost=0.00..1.23 rows=5 width=14) (actual time=0.019..0.022 rows=7 loops=1)

Filter: ((side = 'Light'::side) AND (status_of_pack = 'is'::status_of_pack))

Rows Removed by Filter: 8

Planning time: 0.238 ms

Execution time: 0.044 ms

Оптимизация:

Индексы, созданные по-умолчанию: packs_pkey

добавлен индекс packs_side_idx

CREATE INDEX ON Packs(side);

После оптимизации:

QUERY PLAN

Seq Scan on packs (cost=0.00..1.23 rows=5 width=14) (actual time=0.018..0.022 rows=7 loops=1)

Filter: ((side = 'Light'::side) AND (status_of_pack = 'is'::status_of_pack))

Rows Removed by Filter: 8

Planning time: 0.245 ms

Execution time: 0.042 ms

Индекс использован не был. Принудительно заставим использовать индекс, запретив Seq Scan:

SET enable_seqscan TO off;

QUERY PLAN

Index Scan using packs_side_idx on packs (cost=0.14..12.30 rows=5 width=14) (actual time=0.123..0.127 rows=7 loops=1)

Index Cond: (side = 'Light'::side)

Filter: (status_of_pack = 'is'::status_of_pack)

Rows Removed by Filter: 1

Planning time: 0.209 ms

Execution time: 0.165 ms

Вывод: использование индекса только увеличивает cost и время выполнения запроса для выборки (WHERE). Кол-во Rows Removed by Filter уменьшилось.

Создадим еще один индекс packs_status_of_pack_idx

CREATE INDEX ON PACKS(status_of_pack);

QUERY PLAN

Index Scan using packs_side_idx on packs (cost=0.14..12.30 rows=5 width=14) (actual time=0.016..0.019 rows=7 loops=1)

Index Cond: (side = 'Light'::side)

Filter: (status_of_pack = 'is'::status_of_pack)

Rows Removed by Filter: 1

Planning time: 0.883 ms

Execution time: 0.052 ms

Планировщик использует старый индекс, удалим его, чтобы посмотреть результаты работы нового:

DROP INDEX packs_side_idx;

QUERY PLAN

Index Scan using packs_status_of_pack_idx on packs (cost=0.14..12.33 rows=5 width=14) (actual time=0.056..0.062 rows=7 loops=1)

Index Cond: (status_of_pack = 'is'::status_of_pack)

Filter: (side = 'Light'::side)

Rows Removed by Filter: 3

Planning time: 0.386 ms

Execution time: 0.088 ms

Новый индекс также только увеличивает cost и время выполнения запроса как для выборки (WHERE). Кол-во Rows Removed by Filter увеличилось относительно первого индекса, и уменьшилось относительно начального запроса.

4) Вывод списка имен, фамилий персонажей либо мужского, либо женского пола с помощью параметра __sex__ (допустимые значения: 'M', 'W'). Список отсортирован по фамилиям.

```
SELECT last_name, first_name
        FROM Actors
        WHERE sex='__sex__'
ORDER BY last_name ASC;
```

Оптимизация:

Рассматриваемый параметр:__sex__='W';

До оптимизации:

QUERY PLAN

```
 Sort  (cost=2.10..2.15 rows=21 width=13) (actual time=0.096..0.098 rows=21 loops=1)
   Sort Key: last_name
   Sort Method: quicksort  Memory: 26kB
   ->  Seq Scan on actors  (cost=0.00..1.64 rows=21 width=13) (actual time=0.028..0.037 rows=21 loops=1)
         Filter: (sex = 'W'::sex)
         Rows Removed by Filter: 30
 Planning time: 0.182 ms
 Execution time: 0.124 ms
```

Оптимизация:

Индексы, созданные по-умолчанию: actors_pkey

индекс actors_sex_idx уже создан.

С оптимизацией:

QUERY PLAN

Sort (cost=2.10..2.15 rows=21 width=13) (actual time=0.107..0.109 rows=21 loops=1)

Sort Key: last_name

Sort Method: quicksort Memory: 26kB

-> Seq Scan on actors (cost=0.00..1.64 rows=21 width=13) (actual time=0.032..0.040 rows=21 loops=1)

Filter: (sex = 'W'::sex)

Rows Removed by Filter: 30

Planning time: 0.217 ms

Execution time: 0.138 ms

Индекс использован не был. Принудительно заставим использовать индекс, запретив Seq Scan:

QUERY PLAN

Sort (cost=8.97..9.02 rows=21 width=13) (actual time=0.079..0.081 rows=21 loops=1)

Sort Key: last_name

Sort Method: quicksort Memory: 26kB

-> Index Scan using actors_sex_idx on actors (cost=0.14..8.51 rows=21 width=13) (actual time=0.018..0.024 rows=21 loops=1)

Index Cond: (sex = 'W'::sex)

Planning time: 0.160 ms

Execution time: 0.114 ms

Вывод: использование индекса увеличивает cost для выборки (WHERE) и сортировки (ORDER BY), но уменьшает время выполнения.

Глава 3. Средние запросы

1) Запрос отображает какой актер каких персонажей сыграл. Выводится как таблица, в которой именам и фамилиям актеров сопоставляются имена и фамилии персонажей. Отсортировано по именам и фамилиям актеров.

```
SELECT a.first_name,
        a.last_name,
        c.first_name AS first_name_of_character,
        c.last_name AS last_name_of_character
FROM Actors a INNER JOIN Characters c
            ON a.ID = c.Actor
ORDER BY a.first_name, a.last_name;
```

Оптимизация:

До оптимизации:

```
                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Sort  (cost=5.98..6.12 rows=54 width=27) (actual time=0.383..0.387 rows=54 loops=1)
   Sort Key: a.first_name, a.last_name
   Sort Method: quicksort  Memory: 29kB
   ->  Hash Join  (cost=2.15..4.43 rows=54 width=27) (actual time=0.071..0.093 rows=54 loops=1)
         Hash Cond: (c.actor = a.id)
         ->  Seq Scan on characters c  (cost=0.00..1.54 rows=54 width=18) (actual time=0.020..0.025 rows=54 loops=1)
         ->  Hash  (cost=1.51..1.51 rows=51 width=17) (actual time=0.034..0.034 rows=51 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 11kB
               ->  Seq Scan on actors a  (cost=0.00..1.51 rows=51 width=17) (actual time=0.010..0.015 rows=51 loops=1)
```

Planning time: 0.527 ms

Execution time: 0.434 ms

Оптимизация:

Индексы, созданные по-умолчанию: actors_pkey; characters_pkey

добавлен индекс characters_actor_idx

CREATE INDEX ON Characters(Actor);

После оптимизации:

QUERY PLAN

Sort (cost=5.98..6.12 rows=54 width=27) (actual time=0.255..0.258 rows=54 loops=1)

Sort Key: a.first_name, a.last_name

Sort Method: quicksort Memory: 29kB

-> Hash Join (cost=2.15..4.43 rows=54 width=27) (actual time=0.067..0.088 rows=54 loops=1)

Hash Cond: (c.actor = a.id)

-> Seq Scan on characters c (cost=0.00..1.54 rows=54 width=18) (actual time=0.016..0.018 rows=54 loops=1)

-> Hash (cost=1.51..1.51 rows=51 width=17) (actual time=0.035..0.035 rows=51 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 11kB

-> Seq Scan on actors a (cost=0.00..1.51 rows=51 width=17) (actual time=0.009..0.017 rows=51 loops=1)

Planning time: 0.841 ms

Execution time: 0.309 ms

Индекс использован не был. Принудительно заставим использовать индекс, запретив Seq Scan:

QUERY PLAN

Sort (cost=28.21..28.35 rows=54 width=27) (actual time=0.186..0.191 rows=54 loops=1)

Sort Key: a.first_name, a.last_name

Sort Method: quicksort Memory: 29kB

-> Merge Join (cost=0.28..26.66 rows=54 width=27) (actual time=0.008..0.056 rows=54 loops=1)

Merge Cond: (a.id = c.actor)

-> Index Scan using actors_pkey on actors a (cost=0.14..12.91 rows=51 width=17) (actual time=0.003..0.007 rows=51 loops=1)

-> Index Scan using characters_actor_idx on characters c (cost=0.14..12.95 rows=54 width=18) (actual time=0.002..0.012 rows=54 loops=1)

Planning time: 0.278 ms

Execution time: 0.223 ms

Вывод: использование индекса увеличивает cost для склейки (INNER JOIN) и сортировки (ORDER BY), но уменьшает время выполнения запроса.

2) Вывод имен, фамилий и типа существ персонажей по заданному типу существ с помощью параметра __type__ (допустимые значения: 'WereWolf', 'Chimera', 'Kitsune', 'Human', 'Kanima', 'Jaguar', 'Advisor', 'Darak', 'Hellhound', 'Banchee', 'Hunter').

Отсортировано по именам персонажей.

```
SELECT c.first_name, c.last_name, e.type_of_essence
        FROM Characters c JOIN Essences e
                ON c.essence = e.ID WHERE type_of_essence='__type__'
ORDER BY c.first_name;
```

Оптимизация:

Рассматриваемые параметры: __type__='Human';

До оптимизации:

QUERY PLAN

Sort (cost=3.00..3.01 rows=5 width=21) (actual time=0.085..0.085 rows=3 loops=1)

Sort Key: c.first_name

Sort Method: quicksort Memory: 25kB

-> Hash Join (cost=1.15..2.94 rows=5 width=21) (actual time=0.048..0.057 rows=3 loops=1)

Hash Cond: (c.essence = e.id)

-> Seq Scan on characters c (cost=0.00..1.54 rows=54 width=18) (actual time=0.019..0.023 rows=54 loops=1)

-> Hash (cost=1.14..1.14 rows=1 width=11) (actual time=0.015..0.015 rows=1 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

-> Seq Scan on essences e (cost=0.00..1.14 rows=1 width=11) (actual time=0.010..0.012 rows=1 loops=1)

Filter: ((type_of_essence)::text = 'Chimera'::text)

Rows Removed by Filter: 10

Planning time: 0.316 ms

Execution time: 0.121 ms

Оптимизация:

Индексы, созданные по-умолчанию: characters_pkey; essences_pkey

добавлены индексы essences_type_of_essence_

characters_essence_idx

CREATE INDEX ON Characters(Essence);

CREATE INDEX ON Essences(type_of_essence);

После оптимизации:

QUERY PLAN

Sort (cost=3.00..3.01 rows=5 width=21) (actual time=0.083..0.083 rows=3 loops=1)

Sort Key: c.first_name

Sort Method: quicksort Memory: 25kB

-> Hash Join (cost=1.15..2.94 rows=5 width=21) (actual time=0.045..0.052 rows=3 loops=1)

Hash Cond: (c.essence = e.id)

-> Seq Scan on characters c (cost=0.00..1.54 rows=54 width=18) (actual time=0.012..0.015 rows=54 loops=1)

-> Hash (cost=1.14..1.14 rows=1 width=11) (actual time=0.014..0.014 rows=1 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

-> Seq Scan on essences e (cost=0.00..1.14 rows=1 width=11) (actual time=0.009..0.010 rows=1 loops=1)

Filter: ((type_of_essence)::text = 'Chimera'::text)

Rows Removed by Filter: 10

Planning time: 0.524 ms

Execution time: 0.119 ms

Индексы использован не были. Принудительно заставим использовать индексы, запретив Seq Scan:

SET enable_seqscan TO off;

QUERY PLAN

Sort (cost=16.49..16.50 rows=5 width=21) (actual time=0.047..0.048 rows=3 loops=1)

Sort Key: c.first_name

Sort Method: quicksort Memory: 25kB

-> Nested Loop (cost=0.28..16.43 rows=5 width=21) (actual time=0.025..0.027 rows=3 loops=1)

-> Index Scan using essences_type_of_essence_idx on essences e (cost=0.14..8.15 rows=1 width=11) (actual time=0.019..0.019 rows=1 loops=1)

Index Cond: ((type_of_essence)::text = 'Chimera'::text)

-> Index Scan using characters_essence_idx on characters c (cost=0.14..8.23 rows=5 width=18) (actual time=0.002..0.004 rows=3 loops=1)

Index Cond: (essence = e.id)

Planning time: 0.392 ms

Execution time: 0.088 ms

Вывод: использование индекса увеличивает cost сортировки(ORDER BY), но уменьшает для склейки (JOIN), а также уменьшает время выполнения запроса.

3) Вывод всех существующих/распавшихся стай и всех мертвых/живых персонажей, принадлежащих этим стаям.

Отбор стай по критерию существования осуществляется с помощью параметра __statusOfPack__ (допустимые значения: 'is', 'is not'). Отбор живых или мертвых персонажей осуществляется с помощью параметра __status__, который может принимать значения 'Alive' и 'Dead'. Сортировка происходит по названию стаи и имени персонажа.

```
SELECT p.name_of_pack, c.first_name, c.last_name
        FROM Characters c JOIN Characters_in_packs cp ON c.ID=cp.character JOIN Packs p
        ON p.ID=cp.pack
       WHERE p.status_of_pack='__statusOfPack__' and c.status_of_character='__status__'
ORDER BY p.name_of_pack, c.first_name;
```

Оптимизация:

Рассматриваемые параметры: __statusOfPack__='is'; __status__='Alive';

До оптимизации:

QUERY PLAN

Sort (cost=6.85..6.92 rows=28 width=28) (actual time=0.266..0.269 rows=35 loops=1)

Sort Key: p.name_of_pack, c.first_name

Sort Method: quicksort Memory: 27kB

-> Hash Join (cost=3.41..6.18 rows=28 width=28) (actual time=0.120..0.153 rows=35 loops=1)

Hash Cond: (cp.pack = p.id)

-> Hash Join (cost=2.10..4.43 rows=42 width=18) (actual time=0.059..0.080 rows=46 loops=1)

Hash Cond: (cp."character" = c.id)

-> Seq Scan on characters_in_packs cp (cost=0.00..1.66 rows=66 width=8) (actual time=0.013..0.018 rows=66 loops=1)

-> Hash (cost=1.68..1.68 rows=34 width=18) (actual time=0.033..0.033 rows=34 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 10kB

-> Seq Scan on characters c (cost=0.00..1.68 rows=34 width=18) (actual time=0.009..0.019 rows=34 loops=1)

Filter: (status_of_character = 'Alive'::status)

Rows Removed by Filter: 20

-> Hash (cost=1.19..1.19 rows=10 width=18) (actual time=0.046..0.046 rows=10 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

-> Seq Scan on packs p (cost=0.00..1.19 rows=10 width=18) (actual time=0.035..0.039 rows=10 loops=1)

Filter: (status_of_pack = 'is'::status_of_pack)

Rows Removed by Filter: 5

Planning time: 0.530 ms

Execution time: 0.431 ms

Оптимизация:

Индексы, созданные по-умолчанию: characters_pkey; packs_pkey; characters_in_packs_pkey.

добавлены индексы characters_in_packs_character_id

characters_in_packs_pack_id

CREATE INDEX ON Characters_in_packs(pack);

CREATE INDEX ON Characters_in_packs(Character);

ранее были созданы индексы:

characters_status_of_character_idx;

packs_status_of_pack_idx;

После оптимизации:

QUERY PLAN

Sort (cost=6.85..6.92 rows=28 width=28) (actual time=0.269..0.272 rows=35 loops=1)

Sort Key: p.name_of_pack, c.first_name

Sort Method: quicksort Memory: 27kB

-> Hash Join (cost=3.41..6.18 rows=28 width=28) (actual time=0.085..0.125 rows=35 loops=1)

Hash Cond: (cp.pack = p.id)

-> Hash Join (cost=2.10..4.43 rows=42 width=18) (actual time=0.048..0.076 rows=46 loops=1)

Hash Cond: (cp."character" = c.id)

-> Seq Scan on characters_in_packs cp (cost=0.00..1.66 rows=66 width=8) (actual time=0.007..0.014 rows=66 loops=1)

-> Hash (cost=1.68..1.68 rows=34 width=18) (actual time=0.031..0.031 rows=34 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 10kB

-> Seq Scan on characters c (cost=0.00..1.68 rows=34 width=18) (actual time=0.009..0.018 rows=34 loops=1)

Filter: (status_of_character = 'Alive'::status)

Rows Removed by Filter: 20

-> Hash (cost=1.19..1.19 rows=10 width=18) (actual time=0.026..0.026 rows=10 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

-> Seq Scan on packs p (cost=0.00..1.19 rows=10 width=18) (actual time=0.016..0.020 rows=10 loops=1)

Filter: (status_of_pack = 'is'::status_of_pack)

Rows Removed by Filter: 5

Planning time: 0.944 ms

Execution time: 0.364 ms

Индексы использованы не были. Принудительно заставим использовать индексы, запретив Seq Scan:

QUERY PLAN

Sort (cost=40.52..40.59 rows=28 width=28) (actual time=0.207..0.210 rows=35 loops=1)

Sort Key: p.name_of_pack, c.first_name

Sort Method: quicksort Memory: 27kB

-> Hash Join (cost=12.81..39.85 rows=28 width=28) (actual time=0.045..0.100 rows=35 loops=1)

Hash Cond: (cp.pack = p.id)

-> Merge Join (cost=0.28..26.89 rows=42 width=18) (actual time=0.014..0.057 rows=46 loops=1)

Merge Cond: (c.id = cp."character")

-> Index Scan using characters_pkey on characters c (cost=0.14..13.09 rows=34 width=18) (actual time=0.009..0.023 rows=34 loops=1)

Filter: (status_of_character = 'Alive'::status)

Rows Removed by Filter: 20

-> Index Scan using characters_in_packs_character_idx on characters_in_packs cp (cost=0.14..13.13 rows=66 width=8) (actual time=0.002..0.012 rows=64 loops=1)

-> Hash (cost=12.40..12.40 rows=10 width=18) (actual time=0.015..0.015 rows=10 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

-> Index Scan using packs_pkey on packs p (cost=0.14..12.40 rows=10 width=18) (actual time=0.003..0.008 rows=10 loops=1)

Filter: (status_of_pack = 'is'::status_of_pack)

Rows Removed by Filter: 5

Planning time: 0.598 ms

Execution time: 0.284 ms

Вывод: использование индексов увеличивает cost почти везде, но уменьшает время выполнения.

Глава 4. Сложные запросы

1) Запрос выводит имена и фамилии актеров, которые уже завершили свое участие в сериале, и подсчитывает количество сезонов, в которых актер снялся. Список отсортирован, в первую очередь, по возрастанию кол-ва сезонов, при совпадении сортировка ведется по именам и фамилиям.

```
SELECT first_name,
            last_name,
            s.number_of_season AS season_of_start,
            s1.number_of_season AS season_of_finish,
            (a.season_of_finish-a.season_of_start+1) AS seasons_count
FROM
        (SELECT last_name, first_name, season_of_start, season_of_finish
          FROM actors
          WHERE season_of_finish IS NOT NULL) AS a
INNER JOIN seasons s
                ON (a.season_of_start=s.id)
INNER JOIN seasons s1
                ON (a.season_of_finish=s1.id)
ORDER BY seasons_count, first_name, last_name;
```

2) Запрос выводит имена и фамилии актеров, которые сыграли более, чем одного персонажа, подсчитывает и кол-во и выводит имена и фамилии этих персонажей. Список отсортирован по кол-ву сынгранных персонажей. Далее сортировка ведется по именам и фамилиям актеров.

```
SELECT a.first_name AS name_of_actor,
a.last_name AS surname_of_actor,
count_of_characters,
ch.first_name AS name_of_character,
ch.last_name AS surname_of_character
FROM
        (SELECT a.id,
                a.first_name,
                a.last_name,
                COUNT (ch.id) AS count_of_characters
         FROM actors a RIGHT JOIN characters ch
                                        ON ch.actor=a.id
         GROUP BY a.id, a.first_name, a.last_name) AS a
RIGHT JOIN characters ch
                ON ch.actor=a.id WHERE count_of_characters>1
ORDER BY count_of_characters, name_of_actor, surname_of_actor;
```

3) Запрос выводит имена и фамилии персонажей, которые фигурировали в определенном сезоне (в данном случае 3b), и принадлежат к определенному виду существ (в данном случае "WereWolf"). И сезон, и существо можно задать параметром, конкретные значени выбраны для простоты. В случае сезона, сравнению подлежит не его номер, а его ID, так как они не всегда совпадают. Список отсортирован по именам и фамилиям персонажей.

```
SELECT ch.first_name, ch.last_name
FROM
        (SELECT ch.first_name, ch.last_name, essence, a.season_of_start,
a.season_of_finish
         FROM
                (SELECT first_name,
                        last_name,
                        type_of_essence AS essence,
                        actor
                 FROM characters ch INNER JOIN essences es
                                ON (ch.essence=es.id
                                AND es.type_of_essence='WereWolf')) AS ch
INNER JOIN actors a
ON ch.actor=a.id) AS ch
INNER JOIN seasons s
                ON ch.season_of_start=s.id WHERE (ch.season_of_start<4 AND
(4<ch.season_of_finish OR ch.season_of_finish IS NULL))
ORDER BY ch.first_name, ch.last_name;
```