Привет!

1 декабря, в 23:59 по московскому времени мы запускаем наш новый сервис - API хранилища истории сессий нашего онлайн кинотеатра «Фильмопоиск». Дату запуска сдвинуть нельзя, наш PR уже активно продвигает этот запуск. От тебя потребуется развернуть продуктовую инсталляцию этого сервиса.

Наш подрядчик "Horns&Hooves Soft inc" пишет для нас этот новый сервис. Неделю назад подрядчик провёл демонстрационную презентацию. На ней он показал *почти* корректно работающее приложение, и презентовал HTTP эндпоинт, который отвечает на **GET /ping** кодом **200**, если приложение работает корректно и кодом **500**, если нет.

Мы попросили внести небольшие изменения: нужно, чтобы запрос **GET /long\_dummy** в 75% случаев работал быстрее секунды, при этом нас устроит закешированный ответ не старше минуты. На презентации он работал дольше. Кроме того, подрядчик сообщил, что потребуется внести некоторые технологические изменения для повышения удобства эксплуатации, а так же починить несколько некритичных багов для повышения стабильности в работе.

Вчера должна была состояться приёмка, но подрядчик на связь не вышел и перестал отвечать на письма, сообщения и звонки. Нам удалось выяснить, что у подрядчика возникли серьёзные форс-мажорные обстоятельства. Скорее всего получится возобновить взаимодействие не раньше 2 декабря, то есть уже после согласованной даты запуска. Подрядчик не успел предоставить документацию к приложению, и не смог развернуть у нас своё приложение в срок, как ранее обещал. Тот стенд, на котором проводилась демонстрация, уже успели разобрать. К счастью, у нашего менеджера остался email с бинарником приложения, который использовали на демо.

<https://storage.yandexcloud.net/final-homework/bingo> – вот ссылка на этот бинарник.

Твоя задача развернуть отказоустойчивую инсталляцию приложения из имеющегося бинарника до даты запуска продукта. Планируется *стабильная* нагрузка в **60 RPS**, *пиковая* в **120 RPS**.

В эту пятницу, 24 ноября, выходит из отпуска наш тестировщик Петя, который работал с подрядчиком и умеет тестировать это приложение. Он сможет проверить твою и инсталляцию и подсказать, что с ней не так, чтобы тебе было удобнее готовиться к финальному запуску.

Петя интроверт, не любит живое общение, поэтому он обещал сделать автоматику и помогать тебе с помощью специального сервиса - <https://devops.yactf.ru>

Посредством этого сервиса он и будет принимать решение о том, насколько тебе удалось справиться с требованиями технического задания.

**Требования** (в порядке убывания важности):

- Отказоустойчивость: сервис должен быть развернут на **двух нодах**, отказ любой из них должен быть незаметен пользователю. Допускается просадка по RPS до *стабильного* значения в момент отказа любой из нод. При живости обеих нод, инсталяция обязана выдерживать *пиковую* нагрузку. Так же нужно обеспечить восстановление работоспособности любой отказавшей ноды быстрее, чем за минуту.

- Сервис должен переживать *пиковую* нагрузку в **120 RPS** в течение 1 минуты, *стабильную* в **60 RPS**.

- Запросы **POST /operation {"operation":1}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат быстрее, чем за **400 миллисекунд** в **90%** случаев при **120 RPS**, гарантируя **не более 1%** ошибок.

- Запросы **GET /db\_dummy** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат быстрее, чем за **400 миллисекунд** в **90%** случаев при **120 RPS**, гарантируя **не более 1%** ошибок.

Select pg\_sleep(0.1)

- Запросы **GET /api/movie/{id}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат быстрее, чем за **400 миллисекунд** в **90%** случаев при **120 RPS**, гарантируя **не более 1%** ошибок.

SELECT movies.id, movies.name, movies.year, movies.duration FROM movies WHERE movies.id IN (1) ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000;

postgres=# explain (analyze) SELECT movies.id, movies.name, movies.year, movies.duration FROM movies WHERE movies.id IN (1) ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000;

QUERY PLAN

---------------------------------------------------------------------------------------------------------------

Limit (cost=594.75..594.75 rows=1 width=33) (actual time=2.148..2.154 rows=1 loops=1)

-> Sort (cost=594.75..594.75 rows=1 width=33) (actual time=2.145..2.148 rows=1 loops=1)

Sort Key: year DESC, name

Sort Method: quicksort Memory: 25kB

-> Seq Scan on movies (cost=0.00..594.74 rows=1 width=33) (actual time=0.022..2.137 rows=1 loops=1)

Filter: (id = 1)

Rows Removed by Filter: 29018

Planning Time: 0.064 ms

Execution Time: 2.173 ms

(9 rows)

postgres=# create index idx\_movies\_id on movies(id);

CREATE INDEX

postgres=# explain (analyze) SELECT movies.id, movies.name, movies.year, movies.duration FROM movies WHERE movies.id IN (1) ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000;

QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------

Limit (cost=8.31..8.32 rows=1 width=33) (actual time=0.071..0.077 rows=1 loops=1)

-> Sort (cost=8.31..8.32 rows=1 width=33) (actual time=0.069..0.072 rows=1 loops=1)

Sort Key: year DESC, name

Sort Method: quicksort Memory: 25kB

-> Index Scan using idx on movies (cost=0.29..8.30 rows=1 width=33) (actual time=0.041..0.043 rows=1 loops=1)

Index Cond: (id = 1)

Planning Time: 0.234 ms

Execution Time: 0.117 ms

(8 rows)

- Запросы **GET /api/customer/{id}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат быстрее, чем за **400 миллисекунд** в **90%** случаев при **120 RPS**, гарантируя **не более 1%** ошибок.

SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers WHERE customers.id IN ($1) ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000","parameters: $1 = '1'

postgres=# explain (analyze)SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers WHERE customers.id IN (1) ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000;

QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=8901.28..8901.28 rows=1 width=52) (actual time=56.872..56.938 rows=1 loops=1)

-> Sort (cost=8901.28..8901.28 rows=1 width=52) (actual time=56.869..56.931 rows=1 loops=1)

Sort Key: surname, name, birthday DESC

Sort Method: quicksort Memory: 25kB

-> Gather (cost=1000.00..8901.27 rows=1 width=52) (actual time=0.319..56.916 rows=1 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Parallel Seq Scan on customers (cost=0.00..7901.17 rows=1 width=52) (actual time=27.404..46.249 rows=0 loops=3)

Filter: (id = 1)

Rows Removed by Filter: 166666

Planning Time: 0.096 ms

Execution Time: 56.966 ms

(12 rows)

create index idx\_cus\_id on customers(id);

postgres=# explain (analyze)SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers WHERE customers.id IN (1) ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000;

QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------

Limit (cost=8.45..8.46 rows=1 width=52) (actual time=0.062..0.068 rows=1 loops=1)

-> Sort (cost=8.45..8.46 rows=1 width=52) (actual time=0.060..0.063 rows=1 loops=1)

Sort Key: surname, name, birthday DESC

Sort Method: quicksort Memory: 25kB

-> Index Scan using idx\_cus\_id on customers (cost=0.42..8.44 rows=1 width=52) (actual time=0.049..0.051 rows=1 loops=1)

Index Cond: (id = 1)

Planning Time: 0.448 ms

Execution Time: 0.091 ms

(8 rows)

- Запросы **GET /api/session/{id}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат быстрее, чем за **400 миллисекунд** в **90%** случаев при **120 RPS**, гарантируя **не более 1%** ошибок.

SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id WHERE sessions.id IN (1) ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000","parameters: $1 = '1'

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id WHERE sessions.id IN (1) ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=58906.50..58906.51 rows=1 width=101) (actual time=622.441..624.028 rows=1 loops=1)

-> Sort (cost=58906.50..58906.51 rows=1 width=101) (actual time=622.438..624.022 rows=1 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id

Sort Method: quicksort Memory: 25kB

-> Nested Loop (cost=1000.71..58906.49 rows=1 width=101) (actual time=0.951..624.009 rows=1 loops=1)

-> Nested Loop (cost=1000.42..58898.18 rows=1 width=72) (actual time=0.936..623.987 rows=1 loops=1)

-> Gather (cost=1000.00..58889.72 rows=1 width=24) (actual time=0.909..623.954 rows=1 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Parallel Seq Scan on sessions (cost=0.00..57889.62 rows=1 width=24) (actual time=406.900..614.056 rows=0 loops=3)

Filter: (id = 1)

Rows Removed by Filter: 1666666

-> Index Scan using idx\_cus\_id on customers (cost=0.42..8.44 rows=1 width=52) (actual time=0.022..0.023 rows=1 loops=1)

Index Cond: (id = sessions.customer\_id)

-> Index Scan using idxxxxxx on movies (cost=0.29..8.30 rows=1 width=33) (actual time=0.013..0.015 rows=1 loops=1)

Index Cond: (id = sessions.movie\_id)

Planning Time: 0.574 ms

Execution Time: 624.072 ms

(18 rows)

postgres=# create index idxvbdhdj on movies(year asc, name asc);

CREATE INDEX

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id WHERE sessions.id IN (1) ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=58906.50..58906.51 rows=1 width=101) (actual time=430.137..432.174 rows=1 loops=1)

-> Sort (cost=58906.50..58906.51 rows=1 width=101) (actual time=430.134..432.168 rows=1 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id

Sort Method: quicksort Memory: 25kB

-> Nested Loop (cost=1000.71..58906.49 rows=1 width=101) (actual time=1.353..432.154 rows=1 loops=1)

-> Nested Loop (cost=1000.42..58898.18 rows=1 width=72) (actual time=1.345..432.140 rows=1 loops=1)

-> Gather (cost=1000.00..58889.72 rows=1 width=24) (actual time=1.331..432.119 rows=1 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Parallel Seq Scan on sessions (cost=0.00..57889.62 rows=1 width=24) (actual time=278.057..420.041 rows=0 loops=3)

Filter: (id = 1)

Rows Removed by Filter: 1666666

-> Index Scan using idx\_cus\_id on customers (cost=0.42..8.44 rows=1 width=52) (actual time=0.009..0.010 rows=1 loops=1)

Index Cond: (id = sessions.customer\_id)

-> Index Scan using idxxxxxx on movies (cost=0.29..8.30 rows=1 width=33) (actual time=0.005..0.007 rows=1 loops=1)

Index Cond: (id = sessions.movie\_id)

Planning Time: 0.560 ms

Execution Time: 432.219 ms

(18 rows)

postgres=# create index grsagn on cusmoters(id);

ERROR: relation "cusmoters" does not exist

postgres=# create index grsagn on customers(id);

CREATE INDEX

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id WHERE sessions.id IN (1) ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=58906.50..58906.51 rows=1 width=101) (actual time=434.191..434.757 rows=1 loops=1)

-> Sort (cost=58906.50..58906.51 rows=1 width=101) (actual time=434.189..434.751 rows=1 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id

Sort Method: quicksort Memory: 25kB

-> Nested Loop (cost=1000.71..58906.49 rows=1 width=101) (actual time=1.374..434.737 rows=1 loops=1)

-> Nested Loop (cost=1000.42..58898.18 rows=1 width=72) (actual time=1.365..434.721 rows=1 loops=1)

-> Gather (cost=1000.00..58889.72 rows=1 width=24) (actual time=1.338..434.688 rows=1 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Parallel Seq Scan on sessions (cost=0.00..57889.62 rows=1 width=24) (actual time=280.255..423.088 rows=0 loops=3)

Filter: (id = 1)

Rows Removed by Filter: 1666666

-> Index Scan using grsagn on customers (cost=0.42..8.44 rows=1 width=52) (actual time=0.022..0.024 rows=1 loops=1)

Index Cond: (id = sessions.customer\_id)

-> Index Scan using idxxxxxx on movies (cost=0.29..8.30 rows=1 width=33) (actual time=0.006..0.008 rows=1 loops=1)

Index Cond: (id = sessions.movie\_id)

Planning Time: 0.606 ms

Execution Time: 434.816 ms

(18 rows)

postgres=# create index grsagnf on sessions(id);

CREATE INDEX

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id WHERE sessions.id IN (1) ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=25.23..25.23 rows=1 width=101) (actual time=2.134..2.145 rows=1 loops=1)

-> Sort (cost=25.23..25.23 rows=1 width=101) (actual time=2.131..2.139 rows=1 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id

Sort Method: quicksort Memory: 25kB

-> Nested Loop (cost=1.14..25.22 rows=1 width=101) (actual time=2.112..2.125 rows=1 loops=1)

-> Nested Loop (cost=0.85..16.90 rows=1 width=72) (actual time=2.104..2.111 rows=1 loops=1)

-> Index Scan using grsagnf on sessions (cost=0.43..8.45 rows=1 width=24) (actual time=2.090..2.091 rows=1 loops=1)

Index Cond: (id = 1)

-> Index Scan using grsagn on customers (cost=0.42..8.44 rows=1 width=52) (actual time=0.007..0.009 rows=1 loops=1)

Index Cond: (id = sessions.customer\_id)

-> Index Scan using idxxxxxx on movies (cost=0.29..8.30 rows=1 width=33) (actual time=0.006..0.007 rows=1 loops=1)

Index Cond: (id = sessions.movie\_id)

Planning Time: 3.700 ms

Execution Time: 2.188 ms

(14 rows)

- Запросы **GET /api/movie** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат гарантируя **не более 1%** ошибок. Требований по времени ответа нет, планируем делать не более одного такого запроса одновременно.

SELECT movies.id, movies.name, movies.year, movies.duration FROM movies ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000

postgres=# explain (analyze) SELECT movies.id, movies.name, movies.year, movies.duration FROM movies ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000;

QUERY PLAN

------------------------------------------------------------------------------------------------------------------------

Limit (cost=2673.18..2745.73 rows=29019 width=33) (actual time=60.878..118.375 rows=29019 loops=1)

-> Sort (cost=2673.18..2745.73 rows=29019 width=33) (actual time=60.875..82.405 rows=29019 loops=1)

Sort Key: year DESC, name, id DESC

Sort Method: quicksort Memory: 3164kB

-> Seq Scan on movies (cost=0.00..522.19 rows=29019 width=33) (actual time=0.009..19.192 rows=29019 loops=1)

Planning Time: 0.067 ms

Execution Time: 136.290 ms

(7 rows)

create index idx\_year\_name\_id on movies(year desc, name asc, id desc);

postgres=# explain (analyze) SELECT movies.id, movies.name, movies.year, movies.duration FROM movies ORDER BY movies.year DESC, movies.name ASC, movies.id DESC LIMIT 100000;

QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=0.41..1973.83 rows=29019 width=33) (actual time=0.024..67.639 rows=29019 loops=1)

-> Index Scan using idxxxxxxx on movies (cost=0.41..1973.83 rows=29019 width=33) (actual time=0.022..31.164 rows=29019 loops=1)

Planning Time: 0.263 ms

Execution Time: 85.975 ms

(4 rows)

- Запросы **GET /api/customer** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат гарантируя **не более** **1%** ошибок. Требований по времени ответа нет, планируем делать не более одного такого запроса одновременно.

SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000

postgres=# explain (analyze) SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000;

QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=3.88..11783.74 rows=100000 width=52) (actual time=0.133..436.842 rows=100000 loops=1)

-> Incremental Sort (cost=3.88..58903.17 rows=500000 width=52) (actual time=0.131..312.595 rows=100000 loops=1)

Sort Key: surname, name, birthday DESC, id DESC

Presorted Key: surname

Full-sort Groups: 2145 Sort Method: quicksort Average Memory: 29kB Peak Memory: 29kB

Pre-sorted Groups: 487 Sort Method: quicksort Average Memory: 28kB Peak Memory: 28kB

-> Index Scan using idx\_cus\_sur on customers (cost=0.42..30554.51 rows=500000 width=52) (actual time=0.022..120.818 rows=100032 loops=1)

Planning Time: 0.089 ms

Execution Time: 498.785 ms

(9 rows)

postgres=# create index idx\_all on customers (surname asc, name asc, birthday desc, id desc);

postgres=# explain (analyze) SELECT customers.id, customers.name, customers.surname, customers.birthday, customers.email FROM customers ORDER BY customers.surname ASC, customers.name ASC, customers.birthday DESC, customers.id DESC LIMIT 100000;

QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=0.42..7949.01 rows=100000 width=52) (actual time=0.041..252.509 rows=100000 loops=1)

-> Index Scan using idx\_all on customers (cost=0.42..39743.34 rows=500000 width=52) (actual time=0.039..128.155 rows=100000 loops=1)

Planning Time: 0.354 ms

Execution Time: 313.977 ms

(4 rows)

- Запросы **GET /api/session** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат гарантируя **не более** **5%** ошибок. Требований по времени ответа нет, планируем делать не более одного такого запроса одновременно.

2023-11-24 11:54:22.762

SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000",,,"","parallel worker",4004,0

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=444285.01..501465.33 rows=100000 width=101) (actual time=26156.678..28103.570 rows=100000 loops=1)

-> Nested Loop (cost=444285.01..3303299.85 rows=4999998 width=101) (actual time=25770.906..27590.082 rows=100000 loops=1)

-> Gather Merge (cost=444284.59..1026616.75 rows=4999998 width=53) (actual time=25765.368..26205.125 rows=100001 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Sort (cost=443284.57..448492.90 rows=2083332 width=53) (actual time=25587.973..25620.507 rows=33811 loops=3)

Sort Key: movies.year DESC, movies.name, sessions.customer\_id, sessions.id DESC

Sort Method: external merge Disk: 111632kB

Worker 0: Sort Method: external merge Disk: 108120kB

Worker 1: Sort Method: external merge Disk: 107912kB

-> Hash Join (cost=884.93..82212.07 rows=2083332 width=53) (actual time=512.827..12972.673 rows=1666549 loops=3)

Hash Cond: (sessions.movie\_id = movies.id)

-> Parallel Seq Scan on sessions (cost=0.00..52681.32 rows=2083332 width=24) (actual time=3.708..4917.633 rows=1666666 loops=3)

-> Hash (cost=522.19..522.19 rows=29019 width=33) (actual time=508.855..508.860 rows=29019 loops=3)

Buckets: 32768 Batches: 1 Memory Usage: 2209kB

-> Seq Scan on movies (cost=0.00..522.19 rows=29019 width=33) (actual time=0.014..54.651 rows=29019 loops=3)

-> Index Scan using grsagn on customers (cost=0.42..0.45 rows=1 width=52) (actual time=0.010..0.011 rows=1 loops=100001)

Index Cond: (id = sessions.customer\_id)

Planning Time: 0.416 ms

JIT:

Functions: 36

Options: Inlining true, Optimization true, Expressions true, Deforming true

Timing: Generation 11.598 ms, Inlining 532.588 ms, Optimization 583.530 ms, Emission 432.494 ms, Total 1560.209 ms

Execution Time: 29060.220 ms

(24 rows)

postgres=# create index idxwtf on movies (year desc, name asc);

create index idxwtfd on customers(id);

postgres=# create index hbfrjkbjj on sessions(customer\_id);

CREATE INDEX

postgres=# create index hbfrjfkbjj on sessions(movie\_id);

CREATE INDEX

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=105.96..59011.63 rows=100000 width=101) (actual time=5.709..4551.870 rows=100000 loops=1)

-> Incremental Sort (cost=105.96..2945387.53 rows=4999997 width=101) (actual time=5.706..4426.790 rows=100000 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id, sessions.id DESC

Presorted Key: movies.year, movies.name

Full-sort Groups: 582 Sort Method: quicksort Average Memory: 39kB Peak Memory: 39kB

Pre-sorted Groups: 582 Sort Method: quicksort Average Memory: 60kB Peak Memory: 60kB

-> Nested Loop (cost=1.27..2572606.96 rows=4999997 width=101) (actual time=2.226..4205.309 rows=100031 loops=1)

-> Nested Loop (cost=0.84..295924.31 rows=4999997 width=53) (actual time=2.213..1574.494 rows=100032 loops=1)

-> Index Scan using idxwtf on movies (cost=0.41..1909.83 rows=29019 width=33) (actual time=2.169..65.447 rows=584 loops=1)

-> Index Scan using hbfrjfkbjj on sessions (cost=0.43..8.41 rows=172 width=24) (actual time=0.017..2.345 rows=171 loops=584)

Index Cond: (movie\_id = movies.id)

-> Index Scan using idxfesfes on customers (cost=0.42..0.45 rows=1 width=52) (actual time=0.023..0.023 rows=1 loops=100032)

Index Cond: (id = sessions.customer\_id)

Planning Time: 14.320 ms

Execution Time: 4613.755 ms

(15 rows)

postgres=# create index hbfrjkggbjj on sessions(id desc);

CREATE INDEX

postgres=# explain (analyze) SELECT sessions.id, sessions.start\_time, customers.id, customers.name, customers.surname, customers.birthday, customers.email, movies.id, movies.name, movies.year, movies.duration FROM sessions INNER JOIN customers ON sessions.customer\_id = customers.id INNER JOIN movies ON sessions.movie\_id = movies.id ORDER BY movies.year DESC, movies.name ASC, customers.id, sessions.id DESC LIMIT 100000;

QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------------------------

Limit (cost=105.96..59011.63 rows=100000 width=101) (actual time=2.565..1657.038 rows=100000 loops=1)

-> Incremental Sort (cost=105.96..2945387.53 rows=4999997 width=101) (actual time=2.562..1532.542 rows=100000 loops=1)

Sort Key: movies.year DESC, movies.name, customers.id, sessions.id DESC

Presorted Key: movies.year, movies.name

Full-sort Groups: 582 Sort Method: quicksort Average Memory: 39kB Peak Memory: 39kB

Pre-sorted Groups: 582 Sort Method: quicksort Average Memory: 60kB Peak Memory: 60kB

-> Nested Loop (cost=1.27..2572606.96 rows=4999997 width=101) (actual time=0.063..1320.410 rows=100031 loops=1)

-> Nested Loop (cost=0.84..295924.31 rows=4999997 width=53) (actual time=0.052..560.530 rows=100032 loops=1)

-> Index Scan using idxwtf on movies (cost=0.41..1909.83 rows=29019 width=33) (actual time=0.030..1.666 rows=584 loops=1)

-> Index Scan using hbfrjfkbjj on sessions (cost=0.43..8.41 rows=172 width=24) (actual time=0.011..0.721 rows=171 loops=584)

Index Cond: (movie\_id = movies.id)

-> Index Scan using idxfesfes on customers (cost=0.42..0.45 rows=1 width=52) (actual time=0.004..0.005 rows=1 loops=100032)

Index Cond: (id = sessions.customer\_id)

Planning Time: 0.668 ms

Execution Time: 1718.913 ms

(15 rows)

- Запросы **POST /api/session**

**{"start\_time":"2021-12-09T16:12:52.059188Z","customer\_id":100,"movie\_id":500}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат гарантируя **не более 1%** ошибок. Требований по времени ответа и RPS нет.

2023-11-24 15:18:53.000 UTC,"postgres","postgres",4656,"127.0.0.1:48920",6560ba88.1230,1113,"INSERT",2023-11-24 15:00:24 UTC,3/5398,0,LOG,00000,"execute stmtcache\_6: INSERT INTO sessions (start\_time,customer\_id,movie\_id) VALUES ($1,$2,$3) ON CONFLICT DO NOTHING RETURNING ""id""","parameters: $1 = '2021-12-09 16:12:52.059188', $2 = '100', $3 = '500'",,,,,,,,"","client backend",,0

postgres=# explain (analyze) INSERT INTO sessions (start\_time,customer\_id,movie\_id) VALUES ('2021-12-09 16:12:52.059188', '100', '500') ON CONFLICT DO NOTHING RETURNING "id";

QUERY PLAN

-------------------------------------------------------------------------------------------------

Insert on sessions (cost=0.00..0.01 rows=1 width=24) (actual time=0.078..0.083 rows=1 loops=1)

Conflict Resolution: NOTHING

Tuples Inserted: 1

Conflicting Tuples: 0

-> Result (cost=0.00..0.01 rows=1 width=24) (actual time=0.007..0.008 rows=1 loops=1)

Planning Time: 0.040 ms

Execution Time: 0.106 ms

(7 rows)

- Запросы **DELETE /api/session/{id}** должны возвращать незакешированный ответ. Сервер должен обрабатывать такие запросы и отдавать результат гарантируя **не более 1% ошибок**. Требований по времени ответа и RPS нет.

DELETE FROM sessions WHERE id IN ($1)","parameters: $1 = '5'",,,,,,,,"","client backend",,0

- **Задача со звёздочкой**: сделать так, чтобы сервис работал на отдельном домене по https протоколу, и по http без редиректа на https (допускается самоподписанный сертификат).

- **Задача со звёздочкой**: сделать http3.

- **Задача со звёздочкой**: сделать так, чтобы запросы **GET /long\_dummy** возвращали ответ не старше 1 минуты и отвечали быстрее, чем за 1 секунду в 75% случаев.

- **Задача со звёздочкой**: желательно обеспечить наблюдаемость приложения: графики RPS и ошибок по каждому эндпоинту.

- **Задача со звёздочкой**: автоматизировать развёртывание при помощи devops инструментов, с которыми вы успели познакомиться ранее.

Немножко комментариев и рекомендаций:

- Для нас важно получить работающую инсталляцию, и менее важно как эта работоспособность была обеспечена. Мы засчитываем только рабочие решения. В качестве решения помимо эндпоинта мы хотим увидеть, как эта инсталляция была развернута. В минимальном варианте это должно быть сочинение на тему "как я это развернул руками". В идеале же, это должен быть репозиторий, который при помощи различных devops инструментов способен развернуть всю инсталляцию по одной команде. Мы не ожидаем от всех участников идеального варианта, но автоматизированные решения при прочих равных, мы оценим выше неавтоматизированных. При этом работающее неавтоматизированное решение будет засчитано, а неработающее автоматизированное - нет;

- Бинарник умеет в развернутой базе данных создавать необходимые для работы таблицы и наполнять их тестовыми данными. Нужно его только об этом правильно попросить, уверен вы сможете разобраться как. Наличие в базе этих тестовых данных - одно из необходимых условий успешного выполнения задания. Вы можете в рамках тестов создавать некоторые сущности в базе и их же удалять. Однако не стоит удалять ничего из изначально там заведённого - это может не понравиться автоматике, которая будет считать Ваш рейтинг;

- Важно убедиться в том, что на сервере настроено корректно точное время. Таймзона не имеет значения. В противном случае некоторые наши проверки решат, что вы считерили, добавив кеширование там, где его быть не должно;

- В конфигурации есть параметр **student\_email** - тут должен быть ваш email, который вы использовали при регистрации на тренировки - это важно для проверки задания;

- Хэлсчек в приложении есть, живёт на **GET /ping** и свою функцию выполняет;

- Можешь на своё усмотрение упаковать бинарник в docker image, либо этого не делать;

- Для тех, кто разворачивается в Яндекс.Облаке - разворачивайтесь в одной АЗ;

- Для самостоятельной проверки RPS и времени ответа своей инсталяции можете пользовать wrk / ab;

- Для SSL, http3 и кеширования мы рекомендуем использовать reverse proxy, которые это поддерживают, например nginx;

- Некоторый софт не умеет отдавать метрики в формате prometheus, для такого софта сбоку разворачивают что-то, что умеет собрать и отдать за него эти метрики. Гуглить по запросу "<название софта> export metrics".