

Санкт-Петербургский государственный университет  
Факультет Прикладной математики - Процессов управления

ОТЧЕТ ПО ДИСЦИПЛИНЕ  
“Оптимизация баз данных”

Выполнили: Акопян Алексей, Ефимов Владислав, Мальцев Даниил

Преподаватель: Севрюков Сергей Юрьевич

Санкт-Петербург  
2022

# Начальные действия

Для последующей работы с базой данных PaymentData, которая генерируется посредством представленного скрипта, произведено ознакомление с формулами расчета баланса, скриптом создания базы данных, программными компонентами (функции и триггеры). На втором шаге была произведена генерация тестовых данных с помощью программного обеспечения RedGate, которое было выбрано благодаря своим возможностям (тонкая настройка, объем данных, тип данных) и благодаря возможности пользоваться пробным периодом.

Генерация данных производилась в 2 этапа:

На первом этапе вносились данные в таблицы Bank, Cashbox, Client, Employee, PaymentParticipant, Project, Supplier. Используемое программное обеспечение достаточно хорошо определило ограничения и подобрало соответствующие настройки для генерации данных большинства столбцов. Однако потребовалась ручная донастройка для некоторых столбцов. Среди важных изменений была установка NULL значений для столбцов с названием GCRecord. Остальные изменения носили скорее optionalный характер, так как не влияли на дальнейшие шаги при оптимизации базы данных. Баланс всех участников в таблице PaymentParticipant был установлен равным 0.

На втором этапе была заполнена таблица Payment. Непосредственно до и после генерации данных в таблицу Payment была произведена проверка балансов всех участников, чтобы удостовериться, в том, что после вставки новых записей в таблицу Payment происходит обновление балансов участников.

В таблицы Bank, Cashbox было добавлено по 100 записей, в таблицу Payment - 10 000 записей, в остальные по 1000. Такие оценка количества записей было сделана на основе коллективного обсуждения, логических рассуждений (банков и касс должно быть меньше чем клиентов, поставщиков, а платежей должно быть существенно больше чем клиентов).

Далее был разработан тест на корректность расчета баланса. Тест использует пример расчета, приведенный в файле *balance description* (таблица “Порядок платежей и расчёт балансов”). Соответственно, общий порядок действий выглядит следующим образом:

1. Выбор случайных идентификаторов банка, кассы, клиента и поставщика

2. Получение их балансов при помощи таблицы PaymentParticipant
3. Выбор категории платежа, соответствующей авансовому взносу на приобретение материалов (первая строка таблицы)
4. Вставка нового платежа в таблицу Payment с использованием значений из предыдущих пунктов
5. Подсчет и вывод разницы между значениями балансов до операции вставки и проверка ее соответствия значениям в таблице
6. Повтор шагов 2-6 для остальных строк таблицы.

Операция обновления балансов вынесена в отдельную хранимую процедуру set\_balances. Операция подсчета разницы балансов реализована в функции check\_balances\_diff.

## Задачи I уровня

В данном разделе стоит задача реализации индексов, повышающих производительность операций вставки и изменения платежей без модификации программных компонент.

В результате, был разработан код транзакции для операции вставки данных в таблицу dbo.Payment, а также код транзакции для операции изменения данных в таблице dbo.Payment, некластеризованные индексы для полей таблиц, которые использовались в триггере T\_Payment\_AI, срабатывающим после вставки или изменения платежа, в функциях dbo.F\_CalculatePaymentParticipantBalance, dbo.F\_CalculateBalanceByMaterial, dbo.F\_CalculateBalanceByWork, dbo.F\_CalculateProjectBalance.

Триггер работает следующим образом:

- Вызов триггера при вставке/изменении платежа в таблицу Payment;
- Обновление баланса у новых участников (плательщик - Payer и получатель - Payee) - функция dbo.F\_CalculatePaymentParticipantBalance;
- Обновление баланса у старых участников плательщик - Payer и получатель - Payee) - функция dbo.F\_CalculatePaymentParticipantBalance;
- Обновление баланса у новых объектов - функции dbo.F\_CalculateBalanceByMaterial, dbo.F\_CalculateBalanceByWork, dbo.F\_CalculateProjectBalance;
- Обновление баланса у старых объектов - функции dbo.F\_CalculateBalanceByMaterial, dbo.F\_CalculateBalanceByWork, dbo.F\_CalculateProjectBalance;

Было принято решение использовать некластеризованный индекс, так как, в отличие от кластеризованного индекса, листья некластеризованного индекса содержат только необходимые столбцы, а также указатель на строки с реальными данными в таблице, а также потому, что для одной таблицы можно определить более одного некластеризованного индекса, что позволяет добавить индекс только для необходимых атрибутов, использующихся в функциях, перечисленных выше.

## Основные тесты

В качестве метрики использовалось среднее время, затраченное на выполнение одной транзакции, подсчеты велись с помощью программного обеспечения SQLQueryStress. Было создано подключение к базе данных PaymentData и проведены тесты для операций вставки(100, 400 и 1000), а также для операций изменения(100, 400). Результаты представлены на рисунках ниже.



Рис. 1 Реализация операции вставки данных в таблицу Payment 100 итераций.



Рис. 2 Реализация операции вставки данных в таблицу Payment 400 итераций.

```

1 USE PaymentData;
2 BEGIN TRAN tranzac;
4-- Declare Variables that we take from other tables
5 DECLARE @payment_cat UNIQUEIDENTIFIER
6 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
7 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
8 DECLARE @payer_part UNIQUEIDENTIFIER
9 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
10 DECLARE @payee_part UNIQUEIDENTIFIER
11 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 SET @pay_cat_name VARCHAR
13 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
15-- Payment Insert Query
16 INSERT dbo.Payment
17 (Old, Amount, Category, Project, Justification, Comment,
18 Payer, Payee, OptimisticlockField, GCHRecord, CreateDate, CheckNumber, IsAuthorized, Number)
19 VALUES
20 (NEWID(),
21 '50000000',
22 @payment_cat,
23 @proj,
24 NULL,
25 @pay_cat_name,
26 '1988-05-21T12:58:46.480',
27 @payer_part,
28 @payee_part,
29 NULL,
30 NULL,
31 '1988-05-21T12:58:46.480',
32 '6542',
33 0,
34 '1234',
35 1);
36 ROLLBACK TRAN tranzac;

```

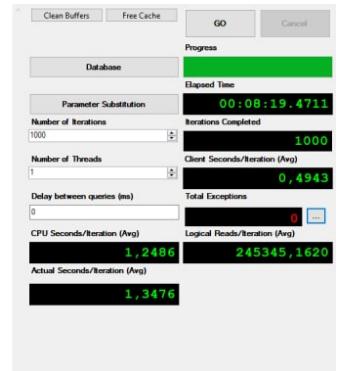


Рис. 3 Реализация операции вставки данных в таблицу Payment 400 итераций.

```

1--Transaction to UPDATE Payment Table
2 USE PaymentData;
4 BEGIN TRAN tranzac;
5-- Declare Variables that we take from other tables
6 DECLARE @payment_cat UNIQUEIDENTIFIER
7 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
8 DECLARE @proj UNIQUEIDENTIFIER
9 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
10 DECLARE @payer_part UNIQUEIDENTIFIER
11 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 DECLARE @payee_part UNIQUEIDENTIFIER
13 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
14 DECLARE @pay_cat_name VARCHAR
15 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
16
17 --Payment Update Query
18 UPDATE dbo.Payment
19 SET Old = NEWID(),
20 Amount = RAND()*500000,
21 Category = @payment_cat,
22 Justification = '1234',
23 Comment = @pay_cat_name,
24 Date = '1988-05-21T12:58:46.480',
25 Payer = @payer_part,
26 Payee = @payee_part,
27 OptimisticlockField = NULL,
28 GCHRecord = NULL,
29 CreateDate = '1988-05-21T12:58:46.480',
30 CheckNumber = '6542',
31 IsAuthorized = 0,
32 Number = '1234' WHERE dbo.Payment.Project=@proj;
33
34 ROLLBACK TRAN tranzac;

```

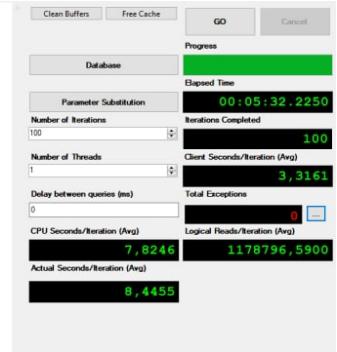


Рис. 4 Реализация операции изменения данных в таблицу Payment 100 итераций.

```

1--Transaction to UPDATE Payment Table
2 USE PaymentData;
4 BEGIN TRAN tranzac;
5-- Declare Variables that we take from other tables
6 DECLARE @payment_cat UNIQUEIDENTIFIER
7 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
8 DECLARE @proj UNIQUEIDENTIFIER
9 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
10 DECLARE @payer_part UNIQUEIDENTIFIER
11 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 DECLARE @payee_part UNIQUEIDENTIFIER
13 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
14 DECLARE @pay_cat_name VARCHAR
15 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
16
17 --Payment Update Query
18 UPDATE dbo.Payment
19 SET Old = NEWID(),
20 Amount = RAND()*500000,
21 Category = @payment_cat,
22 Justification = '1234',
23 Comment = @pay_cat_name,
24 Date = '1988-05-21T12:58:46.480',
25 Payer = @payer_part,
26 Payee = @payee_part,
27 OptimisticlockField = NULL,
28 GCHRecord = NULL,
29 CreateDate = '1988-05-21T12:58:46.480',
30 CheckNumber = '6542',
31 IsAuthorized = 0,
32 Number = '1234' WHERE dbo.Payment.Project=@proj;
33
34 ROLLBACK TRAN tranzac;

```

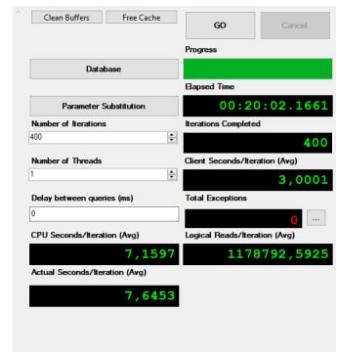


Рис. 5 Реализация операции изменения данных в таблицу Payment 400 итераций.

Далее, для реализации индексов был рассмотрен триггер T\_Payment\_AI. Если используются поля, в которых отсутствует индекс, то его, в таком случае, необходимо создать. При срабатывании данного триггера вызываются функции:

- dbo.F\_CalculatePaymentParticipantBalance – необходимо добавить индекс для полей PaymentCategory.NotInPaymentParticipantProfit, PaymentCategory.Name, AccountType.Name;
- dbo.F\_CalculateBalanceByMaterial – необходимо добавить индекс для полей Supplier.ProfitByMaterialAsPayer, Supplier.ProfitByMaterialAsPayee, AccountType.Name,

- |  |                                   |
|--|-----------------------------------|
| PaymentCategory.Name,<br>PaymentCategory.CostByMaterial; | PaymentCategory.ProfitByMaterial, |
|--|-----------------------------------|
- dbo.F\_CalculateBalanceByWork – необходимо добавить индекс для полей AccountType.Name, PaymentCategory.Name;
  - dbo.F\_CalculateProjectBalance – необходимо добавить индекс для полей PaymentCategory.Name;

В результате, были созданы следующие индексы: PCProfByMatInd, PCCostByMatInd, PCNPayPartInd, PCNameInd, ATNameInd, SuppProfByMatPayerInd, SuppProfByMatPayeeInd. Код представлен на рисунке ниже.

```
--Create Indexes
CREATE NONCLUSTERED INDEX PCProfByMatInd ON dbo.PaymentCategory (ProfitByMaterial)
CREATE NONCLUSTERED INDEX PCCostByMatInd ON dbo.PaymentCategory (CostByMaterial)
CREATE NONCLUSTERED INDEX PCNPayPartInd ON dbo.PaymentCategory (NotInPaymentParticipantProfit)
CREATE NONCLUSTERED INDEX PCNameInd ON dbo.PaymentCategory (Name)
CREATE NONCLUSTERED INDEX ATNameInd ON dbo.AccountType (Name)
CREATE NONCLUSTERED INDEX SuppProfByMatPayerInd ON dbo.Supplier (ProfitByMaterialAsPayer)
CREATE NONCLUSTERED INDEX SuppProfByMatPayeeInd ON dbo.Supplier (ProfitByMaterialAsPayee)
```

Рис. 6. Создание некластеризованных индексов.

Далее были заново проведены тесты для операций вставки данных в таблицу и изменений данных в таблице. Результаты представлены на рисунках ниже.



Рис. 7 Реализация операции изменения данных в таблицу Payment 100 итераций после добавления некластеризованных индексов.

```

1--Transaction to UPDATE Payment Table
2 USE PaymentData;
3
4 BEGIN TRAN tranzac;
5 -- Declare Variables that we take from other tables
6 DECLARE @payment_cat UNIQUEIDENTIFIER
7 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
8 DECLARE @proj UNIQUEIDENTIFIER
9 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
10 SET @proj_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
11 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
13 SET @pay_cat_name = 'VARCHAR';
14 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
15
16 --Payment Update Query
17 UPDATE dbo.Payment
18 SET Old = NEWID(),
19     Amount = RAND() * 500000,
20     Category = @pay_cat_name,
21     Project = @proj,
22     Justification = NULL,
23     Comment = @pay_cat_name,
24     Date = '1988-05-21T12:58:46.480',
25     Payer = @payer_part,
26     Payee = @payee_part,
27     OptimisticlockField = NULL,
28     OCRecord = NULL,
29     CreateDate = '1988-05-21T12:58:46.480',
30     CheckNumber = '6542',
31     IsAuthorized = 0,
32     Number = '1234' WHERE dbo.Payment.Project=@proj;
33
34 ROLLBACK TRAN tranzac;

```

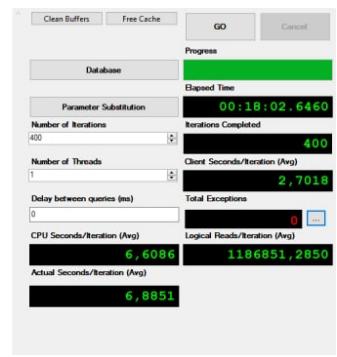


Рис. 8 Реализация операции изменения данных в таблицу Payment 400 итераций после добавления некластеризованных индексов.

```

1--Transaction to INSERT Payment Table
2 USE PaymentData;
3
4
5 BEGIN TRAN tranzac;
6 -- Declare Variables that we take from other tables
7 DECLARE @payment_cat UNIQUEIDENTIFIER
8 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
9 DECLARE @proj UNIQUEIDENTIFIER
10 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
11 SET @proj_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
13 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
14 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
15
16 --Payment Insert Query
17 INSERT dbo.Payment
18 (Old, Amount, Category, Project, Justification, Comment,
19 Date, Payer, Payee, OptimisticlockField, OCRecord, CreateDate, CheckNumber, IsAuthorized, Number)
20 VALUES (
21     NEWID(),
22     RAND() * 500000,
23     @payment_cat,
24     @proj,
25     @proj_part,
26     @payer_part,
27     @pay_cat_name,
28     '1988-05-21T12:58:46.480',
29     @payer_part,
30     @payee_part,
31     NULL,
32     NULL,
33     '1988-05-21T12:58:46.480',
34     '6542',
35     0,
36     '1234'
37 );
38
39 ROLLBACK TRAN tranzac;

```

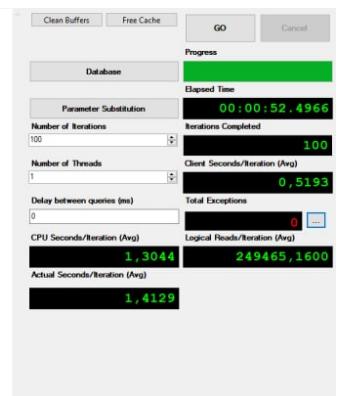


Рис. 8 Реализация операции вставки данных в таблицу Payment 100 итераций после добавления некластеризованных индексов.

```

1--Transaction to INSERT Payment Table
2 USE PaymentData;
3
4
5 BEGIN TRAN tranzac;
6 -- Declare Variables that we take from other tables
7 DECLARE @payment_cat UNIQUEIDENTIFIER
8 SET @payment_cat = (SELECT TOP(1) Old FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
9 DECLARE @proj UNIQUEIDENTIFIER
10 SET @proj = (SELECT TOP(1) Old FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
11 SET @proj_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
12 SET @payer_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
13 SET @payee_part = (SELECT TOP(1) Old FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
14 SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
15
16 --Payment Insert Query
17 INSERT dbo.Payment
18 (Old, Amount, Category, Project, Justification, Comment,
19 Date, Payer, Payee, OptimisticlockField, OCRecord, CreateDate, CheckNumber, IsAuthorized, Number)
20 VALUES (
21     NEWID(),
22     RAND() * 500000,
23     @payment_cat,
24     @proj,
25     @proj_part,
26     @payer_part,
27     @pay_cat_name,
28     '1988-05-21T12:58:46.480',
29     @payer_part,
30     @payee_part,
31     NULL,
32     NULL,
33     '1988-05-21T12:58:46.480',
34     '6542',
35     0,
36     '1234'
37 );
38
39 ROLLBACK TRAN tranzac;

```

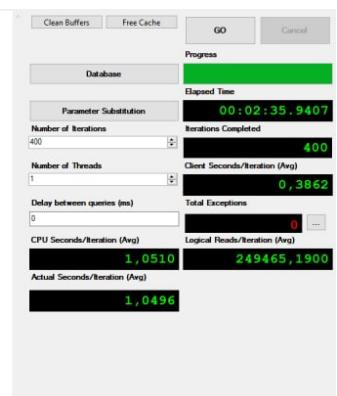


Рис. 8 Реализация операции вставки данных в таблицу Payment 400 итераций после добавления некластеризованных индексов.



Рис. 8 Реализация операции вставки данных в таблицу Payment 1000 итераций после добавления некластеризованных индексов.

Более детально результаты среднего времени на исполнение одной транзакции представлены в таблице ниже.

Название, количество итераций	Время перед добавлением индексов, с	Время после добавления индексов, с
Операция вставки		
100	1,3778	1,4129
400	1,2484	1,0496
1000	1,3476	1,0024
Операция изменения		
100	8,4455	8,1003
400	7,6453	6,8851

В результате, можно заметить, что при меньшем количестве операций операция вставки работает дольше, время увеличилось на 2.5%, возможно, данный феномен связан с техническим оснащением. Далее, при 400 итерациях прирост скорости операции вставки составляет около 16%, а при 1000 итерациях около 26%. Если говорить об операции изменения данных, то при 100 итерациях прирост составляет около 4%, а при 400 итерациях около 10%. Средний прирост для операции вставки данных в таблицу dbo.Payment, таким образом, составляет порядка 13%, а для операции изменения данных средний прирост составляет порядка 7%.

## Тесты с различными индексами

Далее были проведены тесты при добавлении только одного/некоторых из индексов. Сначала было проведено тестирование на вставку данных без индексов. Результаты представлены на рисунках ниже.

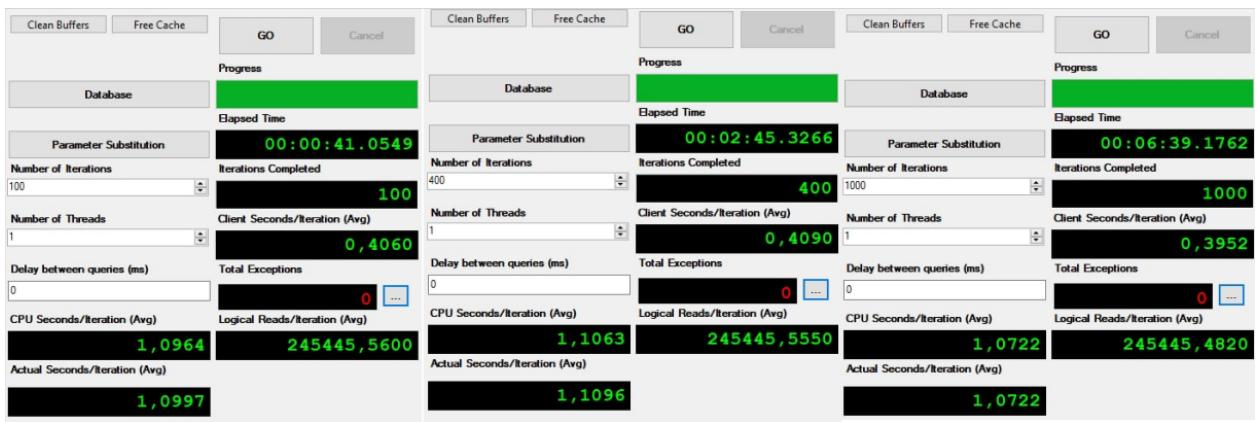


Рис.9. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций до добавления некластеризованных индексов.

Далее было принято решение вставить индекс в один столбец, AccountType.Name, так как он используется в большинстве функций.



Рис.10. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованного индекса для AccountType.Name.

Далее было принято решение вставить индекс в один столбец, PaymentCategory.Name, так как он также часто используется в функциях.



Рис.11. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованного индекса для PaymentCategory.Name.

Далее было принято решение вставить индекс во все столбцы таблицы PaymentCategory, которые используются в функциях.

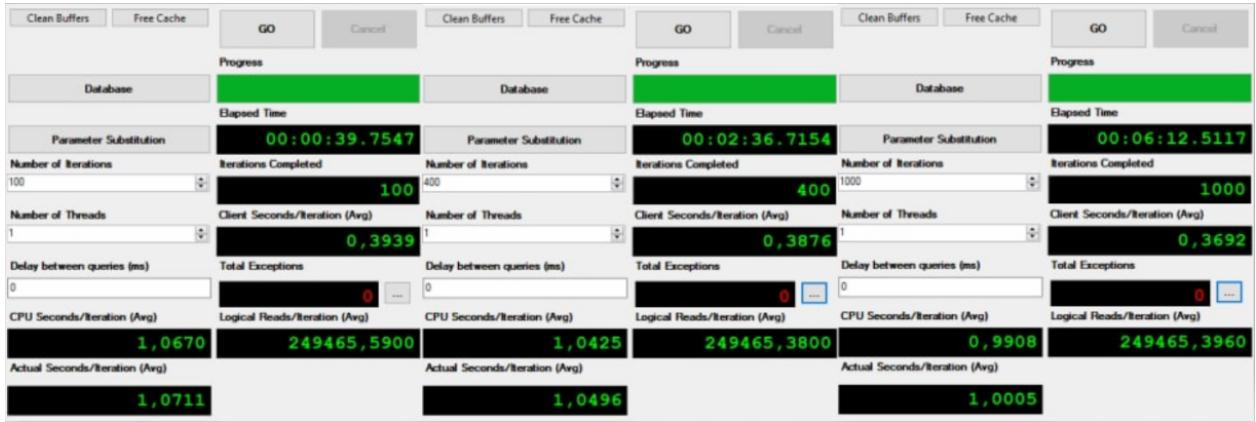


Рис.12. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованных индексов для PaymentCategory.Name, PaymentCategory.ProfitByMaterial, PaymentCategory.CostByMaterial.

Далее было принято решение вставить индекс во все столбцы таблицы Supplier, которые используются в функциях.

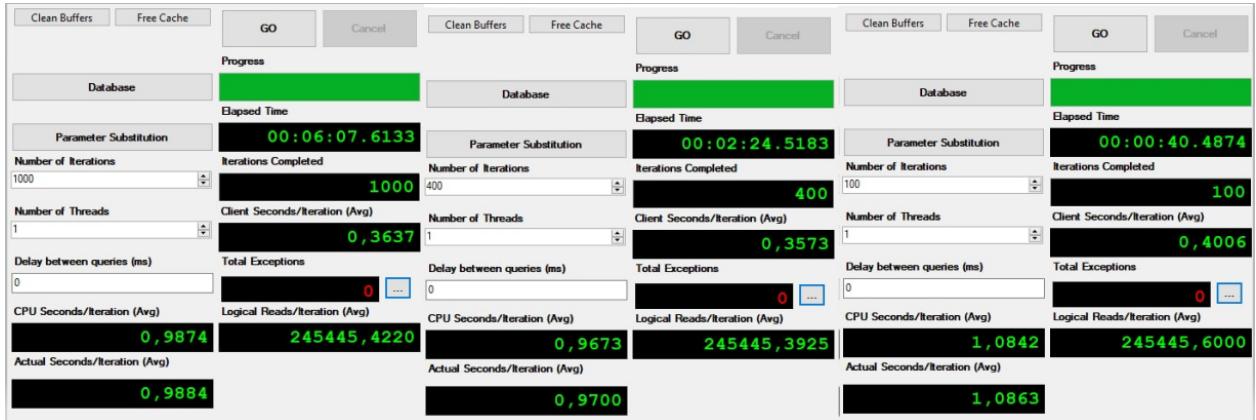


Рис.13. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованных индексов для Supplier.ProfitByMaterialAsPayer, Supplier.ProfitByMaterialAsPayee.

Далее была проведена проверка результатов при добавлении всех индексов во все таблицы, использующихся в функциях.

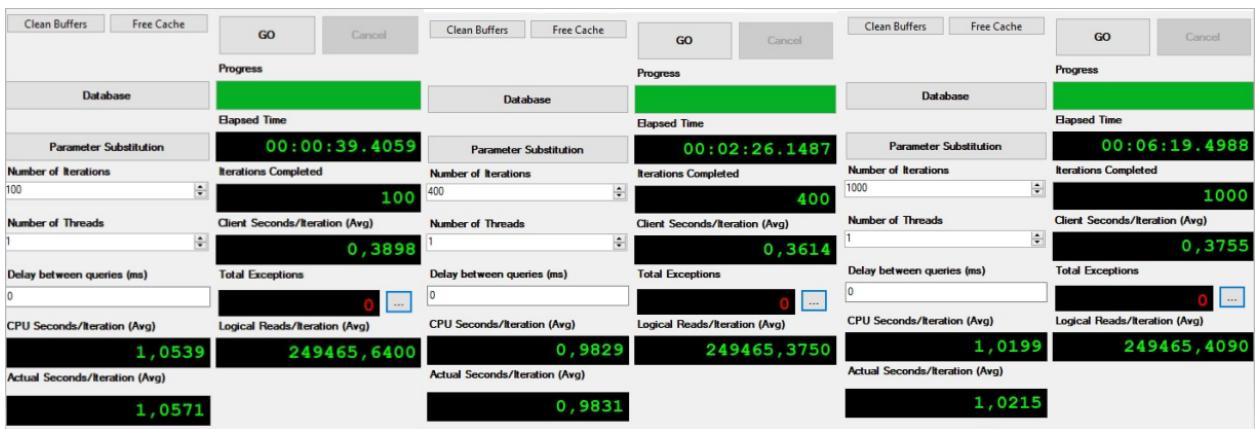


Рис.14. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованных индексов для всех предложенных столбцов.

Далее было принято решение проверить результаты при добавлении индексов в столбцы таблицы Supplier и столбец dbo.Project.Client, потому что сначала при вставке данных идет перерасчет балансов в таблице PaymentParticipant, а потом в таблице Project, в результате Project join'ится с другими таблицами и в таблице Project содержится много записей.

```
LEFT JOIN Project ON Payment.Project = Project.Oid
LEFT JOIN Client ON Project.Client = Client.Oid
```

Рис. 15. В функции F\_CalculateBalanceByMaterial столбцы, использующиеся для join'ов.

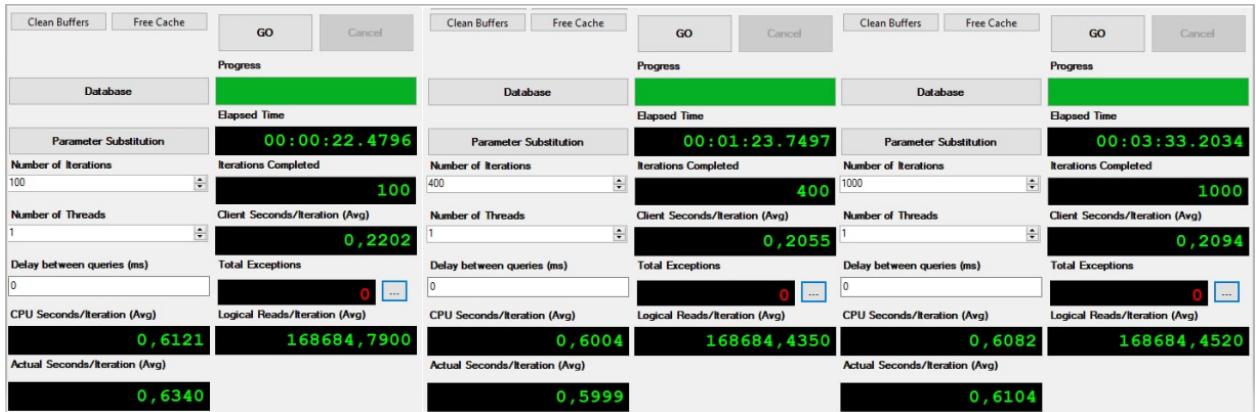


Рис. 16. Реализация операции вставки данных в таблицу Payment 100,400, 1000 итераций после добавления некластеризованных индексов для столбцов таблицы Supplier и Project.

После чего, аналогичные тесты были проведены для операции изменения данных в таблице Payment.



Рис. 17. Реализация операции изменения данных в таблицу Payment 100 итераций без добавления некластеризованных индексов.

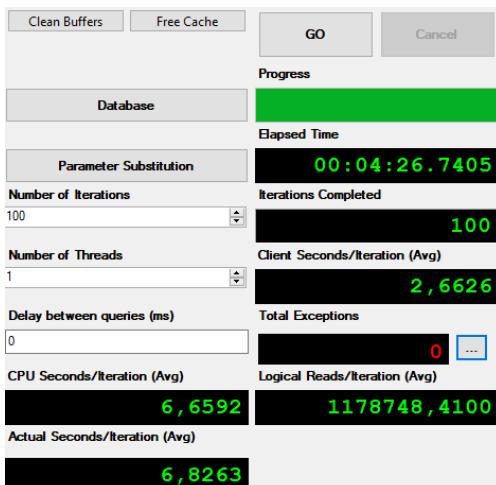


Рис. 18. Реализация операции изменения данных в таблицу Payment 100,400 итераций после добавления некластеризованных индексов для AccountType.Name

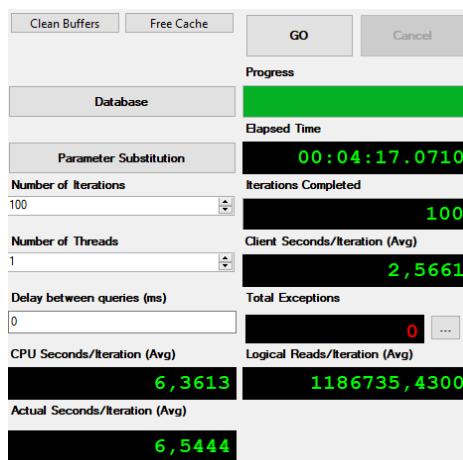


Рис. 19. Реализация операции изменения данных в таблицу Payment 100,400 итераций после добавления некластеризованных индексов для PaymentCategory.Name.

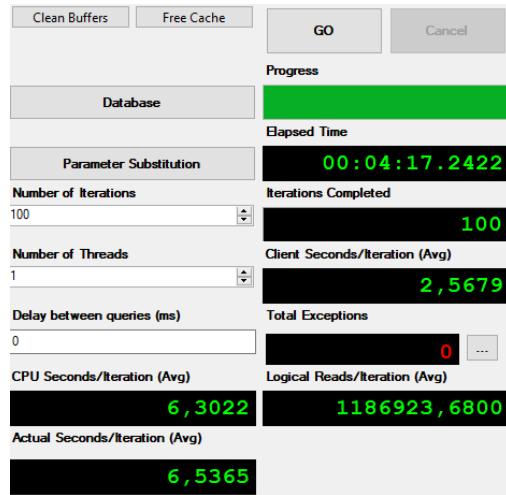


Рис.20. Реализация операции изменения данных в таблицу Payment 100,400 итераций после добавления некластеризованных индексов для PaymentCategory.Name, PaymentCategory.ProfitByMaterial, PaymentCategory.CostByMaterial.



Рис.21. Реализация операции изменения данных в таблицу Payment 100 итераций после добавления некластеризованных индексов для Supplier.ProfitByMaterialAsPayer, Supplier.ProfitByMaterialAsPayee.

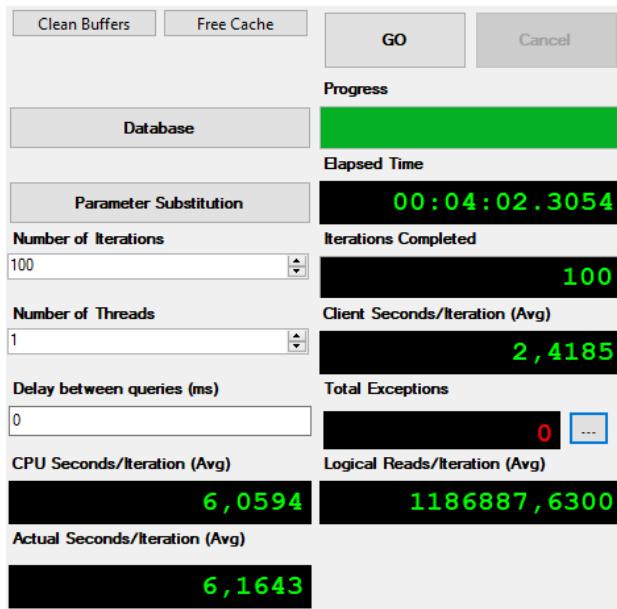


Рис.20. Реализация операции изменения данных в таблицу Payment 100 итераций после добавления всех некластеризованных индексов

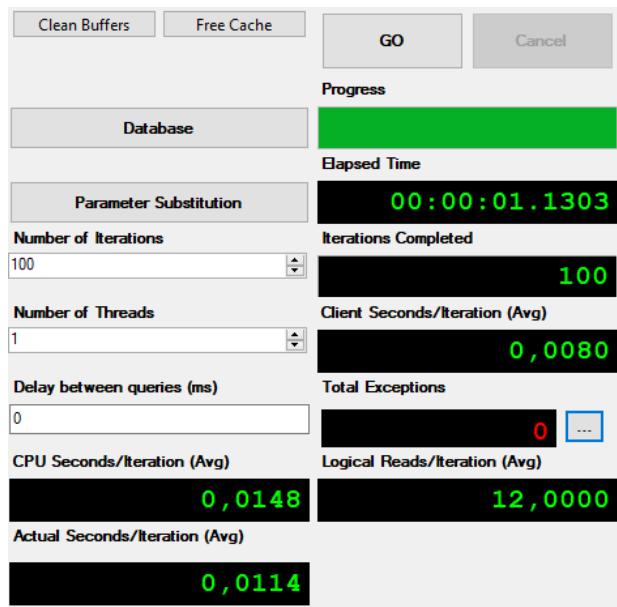


Рис.21. Реализация операции изменения данных в таблицу Payment 100 итераций после добавления некластеризованных индексов для столбцов таблицы Supplier и Project.

Ниже приведена таблица с результатами сравнения и изменением среднего времени на одну транзакцию при введении отдельных некластеризованных индексов по сравнению с отсутствием некластеризованных индексов

Название, количество итераций	Время перед добавлением индексов, с	Время после добавления индексов, с	Изменение, %
Операция вставки		Индексы для AccountType.Name	Среднее 7,267788606
100	1,0997	1,0550	4,06474493
400	1,1096	0,9937	10,44520548
1000	1,0722	0,9940	7,293415408
Операция вставки		Индексы для PaymentCategory.Name	Среднее 7,439877354
100	1,0997	1,0171	7,511139402
400	1,1096	0,9843	11,29235761
1000	1,0722	1,0345	3,516135049
Операция вставки		Индексы для PaymentCategory.Name, PaymentCategory.ProfitByMaterial, PaymentCategory.CostByMaterial	Среднее 4,898416171
100	1,0997	1,0711	2,600709284
400	1,1096	1,0496	5,407354001
1000	1,0722	1,0005	6,687185227
Операция вставки		Индексы для Supplier.ProfitByMaterial AsPayer, Supplier.ProfitByMaterial AsPayee	Среднее 7,128999743
100	1,0997	0,9884	10,12094208
400	1,1096	0,9700	12,58111031
1000	1,0722	1,0863	-1,315053162
Операция вставки		Все индексы	Среднее 6,667627953
100	1,0997	1,0571	3,873783759
400	1,1096	0,9831	11,40050469
1000	1,0722	1,0215	4,728595411
Операция вставки		Индексы для Supplier.ProfitByMaterial AsPayer, Supplier.ProfitByMaterial AsPayee, Project.Client	Среднее 43,78456934

100	1,0997	0,6340	42,34791307
400	1,1096	0,5999	45,93547224
1000	1,0722	0,6104	43,0703227
Операция изменения данных		Индексы для AccountType.Name	
100	6,7267	6,8263	-1,480666597
Операция изменения данных		Индексы для PaymentCategory.Name	
100	6,7267	6,5444	2,710095589
Операция изменения данных		Индексы для PaymentCategory.Name, PaymentCategory.ProfitByMaterial, PaymentCategory.CostByMaterial	
100	6,7267	6,5365	2,82753802
Операция изменения данных		Индексы для Supplier.ProfitByMaterial AsPayer, Supplier.ProfitByMaterial AsPayee	
100	6,7267	6,6465	1,192263666
Операция изменения данных		Все индексы	
100	6,7267	6,1643	8,36071179
Операция изменения данных		Индексы для Supplier.ProfitByMaterial AsPayer, Supplier.ProfitByMaterial AsPayee, Project.Client	
100	6,7267	0,0114	99,83052611

Исходя из представленной выше таблицы, использование всех индексов, использованных в первой части первого задания не дает самого большого прироста.

Так, при операции вставки данных в таблицу Payment, в среднем, использование всех индексов дает прирост в примерно 6,7%, но если использовать некластеризованный индекс только для AccountType.Name, то прирост составит в среднем примерно 7,3%, если же некластеризованный индекс только для PaymentCategory.Name, то прирост составит в среднем примерно 7,4%, если использовать некластеризованный индекс для столбцов

`Supplier.ProfitByMaterialAsPayer`, `Supplier.ProfitByMaterialAsPayee`, то прирост составит в среднем примерно 7,1%. С другой стороны, использование всех некластеризованных индексов, применяемых в первой части, одновременно дает больший прирост по сравнению с использованием некластеризованных индексов только для столбцов `PaymentCategory.Name`, `PaymentCategory.ProfitByMaterial`, `PaymentCategory.CostByMaterial`, при использовании которых прирост в среднем составляет примерно 4,9%. Если же, помимо некластеризованных индексов из первой части задания (`Supplier.ProfitByMaterialAsPayer`, `Supplier.ProfitByMaterialAsPayee`) использовать также некластеризованный индекс для `Project.Client`, то прирост в среднем составляет примерно 43,8%.

Если говорить об операции изменения данных в таблице `Update`, то использование всех некластеризованных индексов из первой части задания, при 100 итерациях изменения данных строки, прирост составляет 8,4%, что гораздо выше в сравнении с другими вариантами. Единственное, если использовать некластеризованные индексы из первой части задания (`Supplier.ProfitByMaterialAsPayer`, `Supplier.ProfitByMaterialAsPayee`) использовать также некластеризованный индекс для `Project.Client`, то при 100 итерациях прирост составляет 99,8%.

## Вывод

Большой прирост при применении комбинации индексов для `Supplier.ProfitByMaterialAsPayer`, `Supplier.ProfitByMaterialAsPayee`, `Project.Client` можно объяснить тем, что эти таблицы являются самыми часто используемыми при вызове триггеров при получении нового платежа, а также, потому что данные таблицы содержат самое большое количество строк.

В целом, правильное использование индексов действительно позволяет получить прирост производительности при операциях вставки и изменения платежа в таблице `Payment`. Исходя из полученных результатов, лучшим вариантом является использование индексов для `Supplier.ProfitByMaterialAsPayer`, `Supplier.ProfitByMaterialAsPayee`, `Project.Client`.

# Задачи II уровня

## Задача 1.

*Дать оценку затрат на выполнения операций расчёта балансов в рамках транзакций создания и изменения платежа. Желательно представить количественную оценку, но допустимо и относительную (к примеру, "90% ресурсов и времени уходит на расчёт баланса"). Чем детальнее, тем лучше.*

Для оценки времени исполнения использовался встроенный в Microsoft SQL Server Management Studio инструмент Live Query Statistics. В скобках отражено относительное время исполнения.

**Вставка новой записи включала:**

1. Вставка новой записи в таблицу Payment (1%)
2. Обновление баланса у новых участников (22%):
  - 2.1. Вычисление функции F\_CalculatePaymentParticipantBalance для плательщика 5%
  - 2.2. Обновление баланса в таблице PaymentParticipant для плательщика 6%
  - 2.3. Вычисление функции F\_CalculatePaymentParticipantBalance для получателя 5%
  - 2.4. Обновление баланса в таблице PaymentParticipant для получателя 6%
3. Обновление баланса у старых участников (22%):
  - 3.1. Вычисление функции F\_CalculatePaymentParticipantBalance для плательщика 5%
  - 3.2. Обновление баланса в таблице PaymentParticipant для плательщика 6%
  - 3.3. Вычисление функции F\_CalculatePaymentParticipantBalance для получателя 5%
  - 3.4. Обновление баланса в таблице PaymentParticipant для получателя 6%

4. Обновление баланса у новых объектов (27%):

4.1. Вычисление функций F\_CalculateBalanceByMaterial, F\_CalculateBalanceByWork и F\_CalculateProjectBalance 8%

4.2. Обновление баланса в таблице Project 19%

5. Обновление баланса у старых объектов (27%):

5.1. Вычисление функций F\_CalculateBalanceByMaterial, F\_CalculateBalanceByWork и F\_CalculateProjectBalance 8%

5.2. Обновление баланса в таблице Project 19%

Схожие показатели получаются при **обновлении существующей записи** в таблице Payment, только сама операция обновления занимает 0% процессорного времени, а пункты 4.1 и 5.1 по 9%.

Видно, что обновление таблиц более длительная операция по сравнению с операцией расчетом баланса. Поэтому возможные сценарии оптимизации должны быть направлены в первую очередь на уменьшении времени, затрачиваемого на обновление балансов.

## Задача 2.

*Введём две роли пользователей: бухгалтер-оператор и бухгалтер-аналитик.*

*Оператор - занимается вводом и корректировкой платежей, не имеет доступа к данным о балансах.*

*Бухгалтер-аналитик - отслеживает балансы и заведённые платежи для принятия решения о предпринимаемых финансовых действиях (какие счета использовать, какие образовались долги и т.д.).*

*Предложить сценарий оптимизации механизмов расчёта. Сценарий должен допускать максимизацию скорости целевых изменений и допускать отложенное вычисление балансов (балансы и данные платежей должны быть согласованы в конечном счёте).*

При использовании данных двух ролей появляется возможность оптимизировать работу базы данных, используя механизм отложенного платежа. Роль оператора ограничивается только записью новых данных без доступа в данные о балансах, поэтому нет необходимости производить перерасчет балансов после каждого внесения оператором новой записи в таблицу Payment. Пересчет балансов требуется только перед началом

работы бухгалтера. В связи с этим были предложены 2 возможных сценария оптимизации работы базы данных:

1. Добавление в таблицу Payment нового столбца-флага

Данный столбец используется для обозначения новых внесенных записей, обработка которых отложена, а также обновленных записей. После обработки записи, значение флага меняется на противоположное.

2. Использования вспомогательной таблицы PaymentDelayed.

При добавлении нового платежа новые записи вносятся не в таблицу Payment, а во временную таблицу PaymentDelayed, из которой затем накопленные записи одной транзакцией вносятся в основную таблицу и вызывают обновление балансов. После переноса накопленных транзакций записи во вспомогательной таблице должны быть удалены.

Перерасчет балансов может производиться с какой-то периодичностью или при накоплении какого-то наперед заданного количества необработанных записей.

### **Задача 3.**

*Оценить недостатки предлагаемого сценария с точки зрения потенциальных пользователей.*

С точки зрения реализации:

1. Первый подход требует изменения существующего кода (триггера, вызываемого после внесения записи в таблицу Payment).
2. Оба подхода подразумевают определения управления для вызова функций перерасчета балансов.

С точки зрения пользователей:

1. Оба подхода не накладывают никаких ограничений на работу оператора. С другой стороны, одновременная работа оператора и бухгалтера либо вообще не должна происходить, либо бухгалтер будет работать с неактуальными данными. В последнем случае требуется поиск оптимального соотношения между актуальностью информации и производительностью за счет выбора частоты обновления или максимального количества новых записей, которые могут быть необработанными.

Несущественный недостаток, если ограничиваться только 2 ролями. Если же, например, вводится третья роль пользователя, который хочет посмотреть свой долг или баланс, то потребуется намного больше обновлений и уже встанет вопрос целесообразности накопления необработанных платежей.

2. Необходимость запуска, возможно ручного, перерасчета балансов, которая может потребовать времени, если было внесено много записей, тем самым бухгалтер не сможет сразу получить актуальную информацию.

# Задачи III уровня

## Задача 1.

*Реализовать предложенный сценарий отложенного вычисления.*

В силу более легкой реализации для заданий 3 уровня был выбран второй сценарий с определением максимального количества вносимых необработанных записей. Выбор именно максимального количества значений позволяет избежать проблем с ручным вызовом переноса записей и перерасчета балансов.

Реализация данного сценария включала создание вспомогательной таблицы PaymentDelayed и триггера для нее, который после каждой вставки новой записи в данную таблицу вычисляет количество записей в таблице. При превышении максимального наперед заданного количества, происходит перенос всех новых записей в основную таблицу Payment.

В качестве максимального допустимого размера таблицы PaymentDelayed было выбрано **100 записей**, в силу того, что такое значение позволит посмотреть на производительность в разных соотношениях этого параметра с количеством вносимых записей. Помимо этого, такое значение кажется достаточно оптимальным с точки зрения частоты обновления записей.

Отдельное и более детальное исследование требуется, чтобы понять как происходит перенос записей при параллельном внесении записей (вопросы связанные с блокировками и целостностью данных). Здесь мы не будем уделять этим вопросам внимание.

Для проверки корректности работы был использован слегка модифицированный тест из начальной части. Модификация заключалась в добавлении цикла, который позволял заполнить таблицу PaymentDelayed до размера необходимого для срабатывания переноса данных.

Скрипт с реализацией данного сценария находится файле Task3.sql в папке DelayedCalculation.

## Задача 2.

*Дать оценку затрат на выполнение операций расчёта балансов в рамках транзакций создания и изменения платежа. Дать заключение о полученном росте производительности, если таковое будет наблюдаться.*

Для оценки затрат на выполнение операций расчета балансов был использован тот же инструмент, что и в первой части SQLQueryStress. Для сравнения с изначальной реализацией были рассмотрены следующие условия  
По количеству вносимых записей:

- Количество вносимых записей меньше порогового значения (10 записей), что означает, что новые записи не будут перенесены в основную таблицу
- Количество вносимых записей сопоставимо с пороговым значением (100 записей). Будет произведен только 1 перенос новых записей
- Количество вносимых записей больше порогового значения (400 записей). Будет произведено множество переносов новых записей по группам.

По величине задержки после каждого запроса:

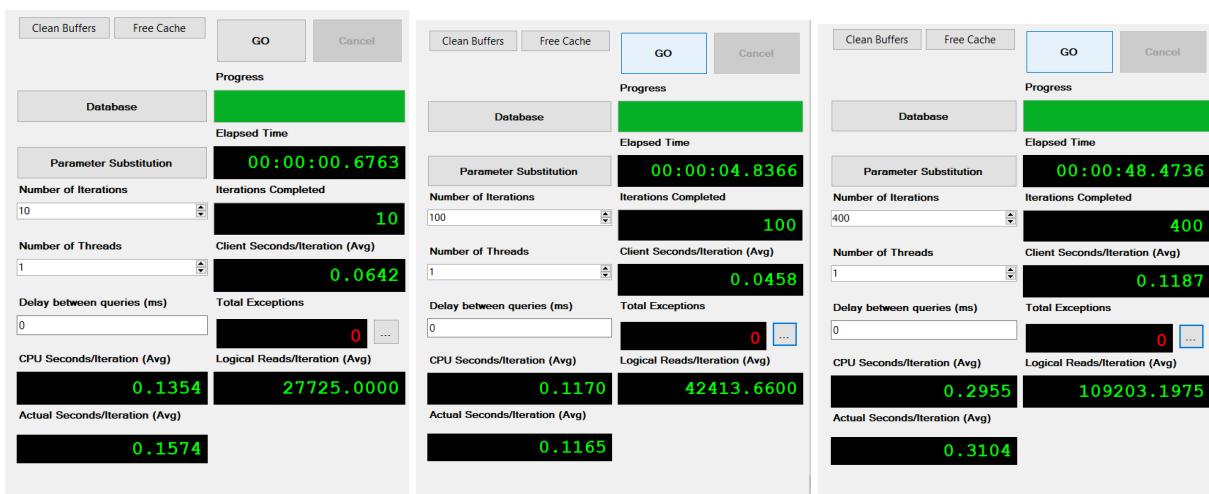
- Без задержки
- С задержкой 100 миллисекунд

Сравнение в условиях задержки после каждой итерации рассматривается для того, чтобы удостовериться, что перенос данных действительно будет произведен несколько раз (то есть не произойдет так, что сначала во вспомогательную базу данных будут добавлены все 400 записей, а затем будет произведен только 1 перенос).

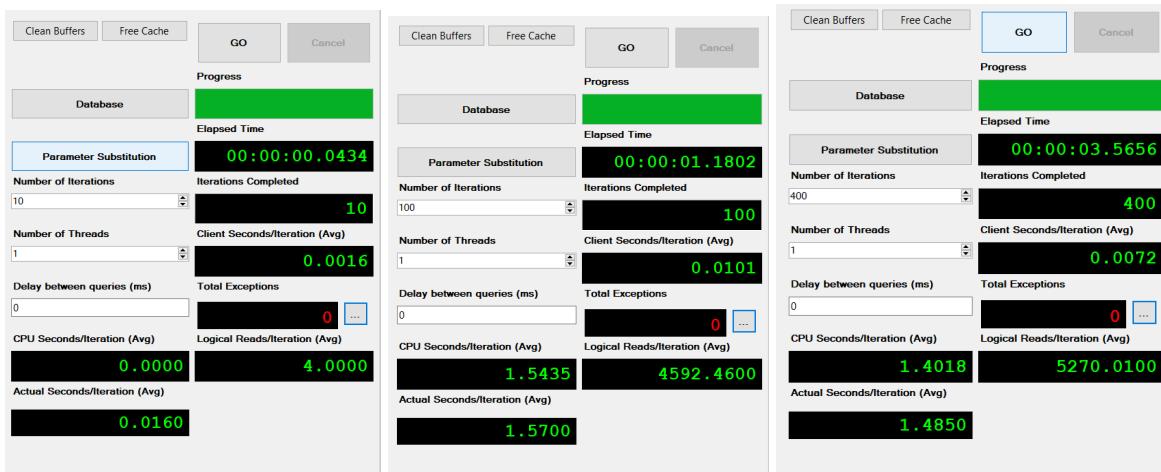
## Результаты:

### Без ожидания после каждой итерации

#### В Payment (изначальная реализация)



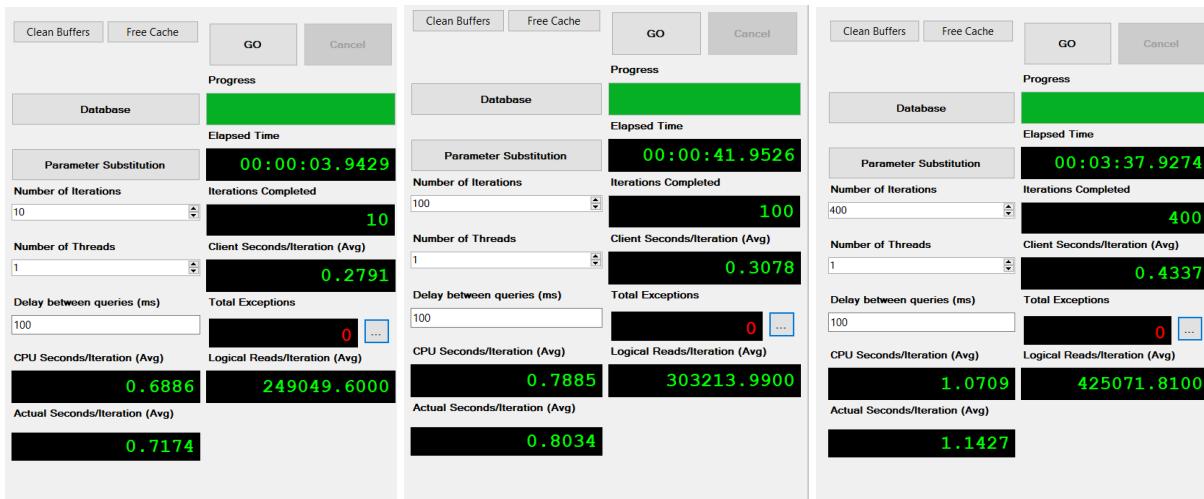
#### В PaymentDelayed (предлагаемая реализация)



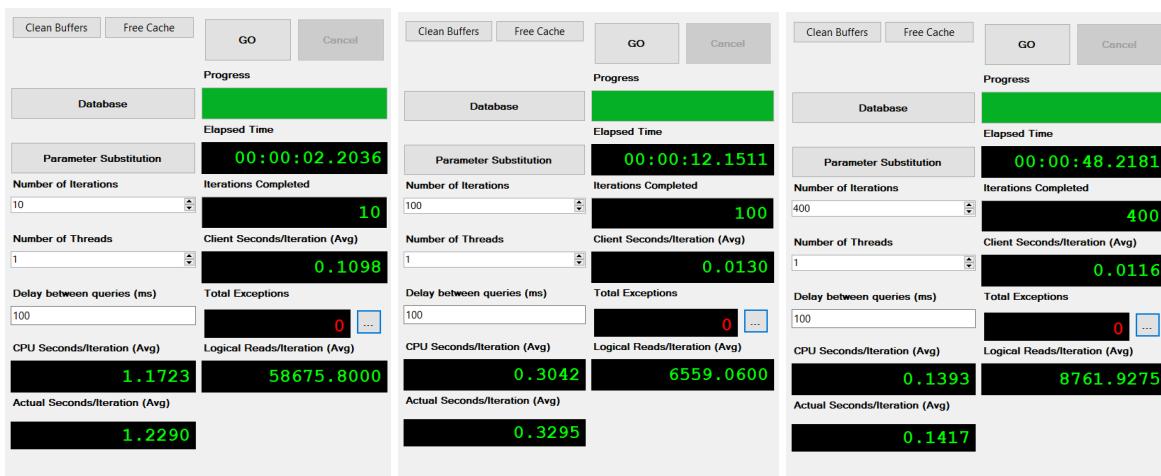
## С ожиданием после каждого запроса (100 миллисекунд)

Чтобы получить чистое время работы с учетом задержек, необходимо отнять  $0.01 * n$  секунд от каждого результата.  $n$  - количество итераций.

## В Payment (изначальная реализация)



## В PaymentDelayed (предлагаемая реализация)



Как видно по результатам рост производительности действительно имеет место быть, причем во всех случаях этот рост существенный. Сделать выводы о постоянной относительном приросте производительности сложно, так как полное время для изначального сценария растет нелинейно по отношению к количеству запросов, а для предлагаемого сценария уже прослеживается такая линейность.

Если же сравнивать попарно для условий, когда количество вставленных записей равно 100 (то есть равно максимальному размеру достаточному, чтобы произошел один перерасчет балансов), то заметен прирост производительности **примерно в 4 раза**. Аналогично для 100 записей с задержками.

### **Задача 3.**

*Дать оценку возникшим проблемам и недостаткам, сравнить их с оценками, сделанными при проектировании сценария оптимизации (указать, что сбылось, что неожиданно проявилось и т.д.).*

1. При реализации предложенного сценария возникли вопросы связанные с обновлением платежей. Сценарий со вспомогательной таблицей `PaymentDelayed` не предполагает возможности оптимизации производительности при обновлении записей. Обновление уже перенесенной записи приведет к вызову функции пересчета балансов. В данном случае необходима более детальное погружение в предметную область, например, необходимо понять, как оператор может менять записи. Напомним, что оператор не имеет доступа к данным и не может их читать.  
\* Реализация первого сценария с дополнительным столбцом в таблице `Payment` решала бы этот вопрос.
2. Все также сохраняется вопрос об оптимальности выбора максимально размера вспомогательной таблицы.
3. Необходимость занесения данных в таблицу `PaymentDelayed` может несколько запутать нового пользователя и ему придется разобраться в структуре базы данных. Однако этот недостаток можно устраниТЬ изолированием внутренностей от пользователей, предоставлением им понятного интерфейса для работы, либо же сменой названий таблиц.

### **Задача 4.**

*Дать заключение о преимуществах и недостатках выполненной оптимизации (превосходят ли полученные преимущества те недостатки, которые возникли после оптимизации).*

Выявленный прирост производительности, как нам кажется, существенно перевешивает выявленные недостатки. По крайней мере, возможность

обновления старых записей все также сохранилась, и время обновления будет таким же, как и при изначальной реализации.