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 настойки 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 опциональный характер, так как не влияли на дальнейшие шаги при оптимизации базы данных. Баланс всех участников в таблице PaymentParticipant был установлен равным 0.

В таблицы Bank, Cashbox было добавлено по 100 записей, в остальные по 1000.

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.

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.

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.

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

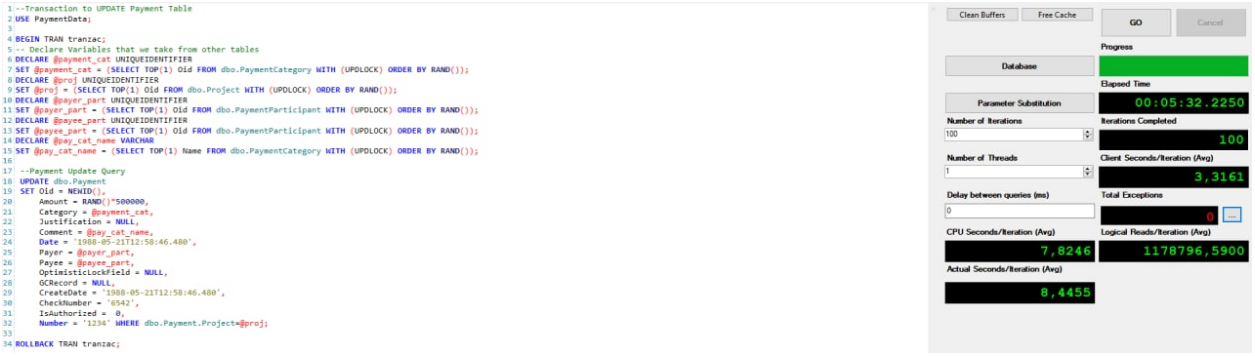

Рис. 4 Реализация операции изменения данных в таблицу Payment 100 итераций.

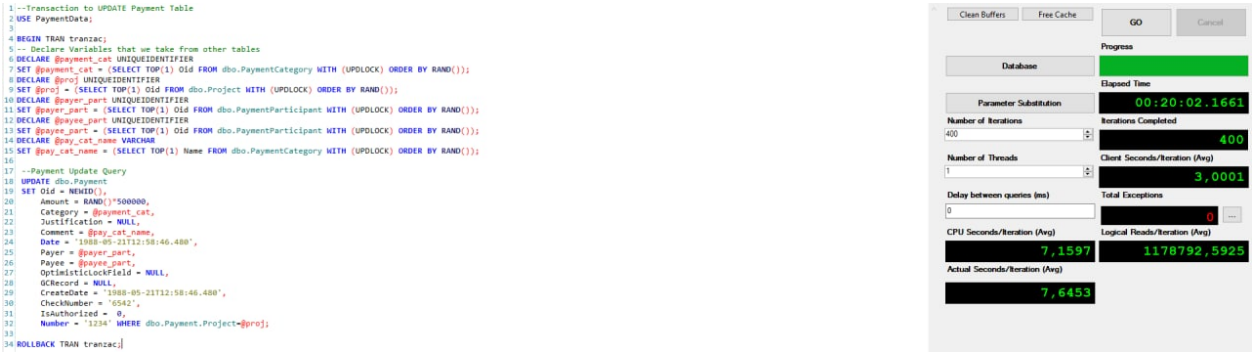

Рис. 5 Реализация операции изменения данных в таблицу Payment 400 итераций.

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, PaymentCategory.ProfitByMaterial, PaymentCategory.CostByMaterial;
- 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.

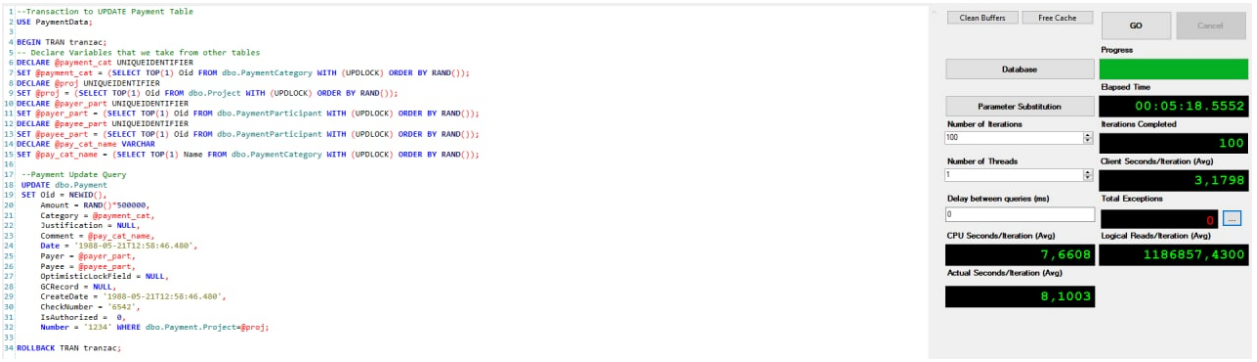

Рис. 7 Реализация операции изменения данных в таблицу Payment 100 итераций после добавления некластеризованных индексов.

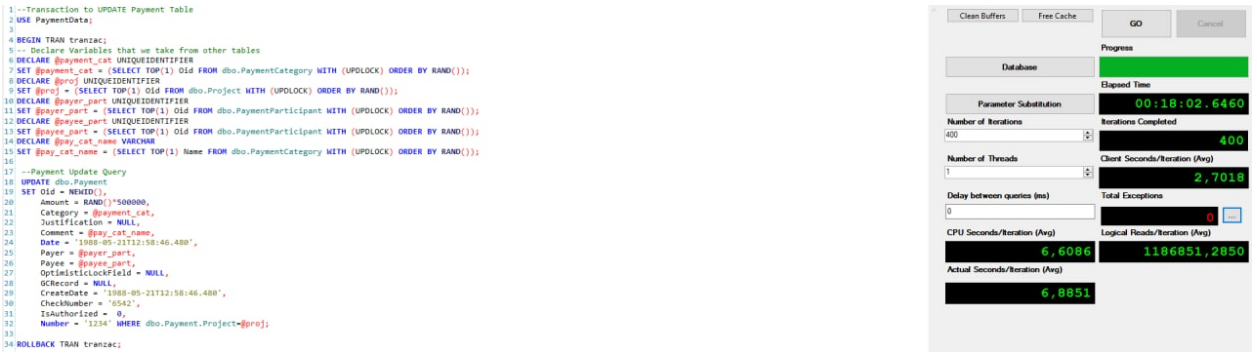

Рис. 8 Реализация операции изменения данных в таблицу Payment 400 итераций после добавления некластеризованных индексов.

```
--Transaction to INSERT Payment Table
USE PaymentData;


BEGIN TRAN tranzac;
-- Declare Variables that we take from other tables
DECLARE @payment_cat UNIQUEIDENTIFIER
SET @payment_cat = (SELECT TOP(1) Oid FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
DECLARE @proj UNIQUEIDENTIFIER
SET @proj = (SELECT TOP(1) Oid FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payer_part UNIQUEIDENTIFIER
SET @payer_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payee_part UNIQUEIDENTIFIER
SET @payee_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @pay_cat_name VARCHAR
SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
-- Payment Insert Query
INSERT dbo.Payment
(Oid, Amount, Category, Project, Justification, Comment,
Date, Payer, Payee, OptimisticLockField, GCRecord, CreateDate, CheckNumber, IsAuthorized, Number)
VALUES (
NEWID(),
RAND()*500000,
@payment_cat,
@proj,
NULL,
@pay_cat_name,
'1988-05-21T12:58:46.480',
@payer_part,
@payee_part,
NULL,
NULL,
'1988-05-21T12:58:46.480',
'6542',
0,
'1234'
);
ROLLBACK TRAN tranzac;
```

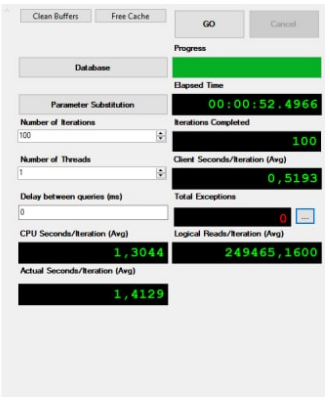

Рис. 8 Реализация операции вставки данных в таблицу Payment 100 итераций после добавления некластеризованных индексов.

```
--Transaction to INSERT Payment Table
USE PaymentData;


BEGIN TRAN tranzac;
-- Declare Variables that we take from other tables
DECLARE @payment_cat UNIQUEIDENTIFIER
SET @payment_cat = (SELECT TOP(1) Oid FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
DECLARE @proj UNIQUEIDENTIFIER
SET @proj = (SELECT TOP(1) Oid FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payer_part UNIQUEIDENTIFIER
SET @payer_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payee_part UNIQUEIDENTIFIER
SET @payee_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @pay_cat_name VARCHAR
SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
-- Payment Insert Query
INSERT dbo.Payment
(Oid, Amount, Category, Project, Justification, Comment,
Date, Payer, Payee, OptimisticLockField, GCRecord, CreateDate, CheckNumber, IsAuthorized, Number)
VALUES (
NEWID(),
RAND()*500000,
@payment_cat,
@proj,
NULL,
@pay_cat_name,
'1988-05-21T12:58:46.480',
@payer_part,
@payee_part,
NULL,
NULL,
'1988-05-21T12:58:46.480',
'6542',
0,
'1234'
);
ROLLBACK TRAN tranzac;
```

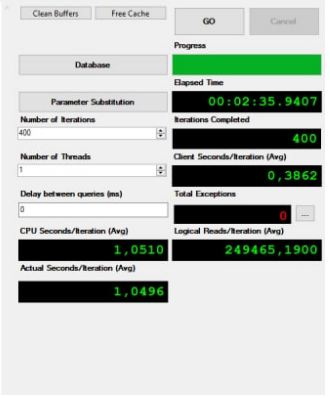

Рис. 8 Реализация операции вставки данных в таблицу Payment 400 итераций после добавления некластеризованных индексов.

```
--Transaction to INSERT Payment Table
USE PaymentData;


BEGIN TRAN tranzac;
-- Declare Variables that we take from other tables
DECLARE @payment_cat UNIQUEIDENTIFIER
SET @payment_cat = (SELECT TOP(1) Oid FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
DECLARE @proj UNIQUEIDENTIFIER
SET @proj = (SELECT TOP(1) Oid FROM dbo.Project WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payer_part UNIQUEIDENTIFIER
SET @payer_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @payee_part UNIQUEIDENTIFIER
SET @payee_part = (SELECT TOP(1) Oid FROM dbo.PaymentParticipant WITH (UPDLOCK) ORDER BY RAND());
DECLARE @pay_cat_name VARCHAR
SET @pay_cat_name = (SELECT TOP(1) Name FROM dbo.PaymentCategory WITH (UPDLOCK) ORDER BY RAND());
-- Payment Insert Query
INSERT dbo.Payment
(Oid, Amount, Category, Project, Justification, Comment,
Date, Payer, Payee, OptimisticLockField, GCRecord, CreateDate, CheckNumber, IsAuthorized, Number)
VALUES (
NEWID(),
RAND()*500000,
@payment_cat,
@proj,
NULL,
@pay_cat_name,
'1988-05-21T12:58:46.480',
@payer_part,
@payee_part,
NULL,
NULL,
'1988-05-21T12:58:46.480',
'6542',
0,
'1234'
);
ROLLBACK TRAN tranzac;
```

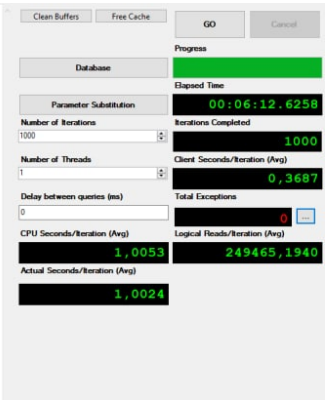

Рис. 8 Реализация операции вставки данных в таблицу Payment 1000 итераций после добавления некластеризованных индексов.

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%.

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 данным о балансах, поэтому нет необходимости производить перерасчет балансов после каждого внесения оператором новой записи в таблицу Payment. Перерасчет 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 новый 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.