Часть 1.

Задача: Реализовать индексы, повышающие производительность операций вставки и изменения платежей без модификации программных компонент

Исполнитель: Щеникова Снежана

Инструменты: SQL Server Profiler, Database Engine Tuning Advisor, Activity Monitor, Client Statistics, SQL Query Stress, запросы.

Перед началом реализации индексов необходимо было ознакомиться с существующими индексами (Таблица 1) и тем, как происходит расчет баланса.

Сущность	Индексы
accounttype	id
bank	Id,Accounttype_id
cashbox	Id,Accounttype_id
Client	id
employee	id
payment	1.id 2.category_id 3.project_id 4.payer_id 5.payee_id 6.gcrecord
Paymentcategory	Id, gcrecord
Paymentparticipant	id, gcrecord, objecttype
Project	id, client_id, manager_id, foreman_id, gcrecord
supplier	Id

Таблица 1. Существующие индексы

Принцип расчета баланса:

1.Insert Payment: заносятся данные в таблицу

2.Вызывается триггер **t_payment_ai**

- Вызывается триггер t_paymentparticipant_bu;

- Обновляется баланс в PaymentParticipant для новых получателя и плательщика и двух старых (UPDATE paymentparticipant);

3.Вызывается триггер **t_project_bu**

- Вносятся изменения в таблицу Project

Ради интереса было проведено тестирование на изменение производительности за счет удаления каких-либо индексов из данного списка. Но избавляться от индексов нужно с логической точки зрения, а не как попало. В связи с этим была рассмотрена статистика, которая показывает плотность индекса. Чем меньше плотность, тем лучше – это увеличивает избирательность, а, следовательно, и ценность построенного индекса. Плотность считается по формуле: $\frac{1}{\text{число записей}}$.

Делается это с помощью команды:

```
DBCC SHOW_STATISTICS ([<Сущность>], <Название индекса>)
```

Результаты

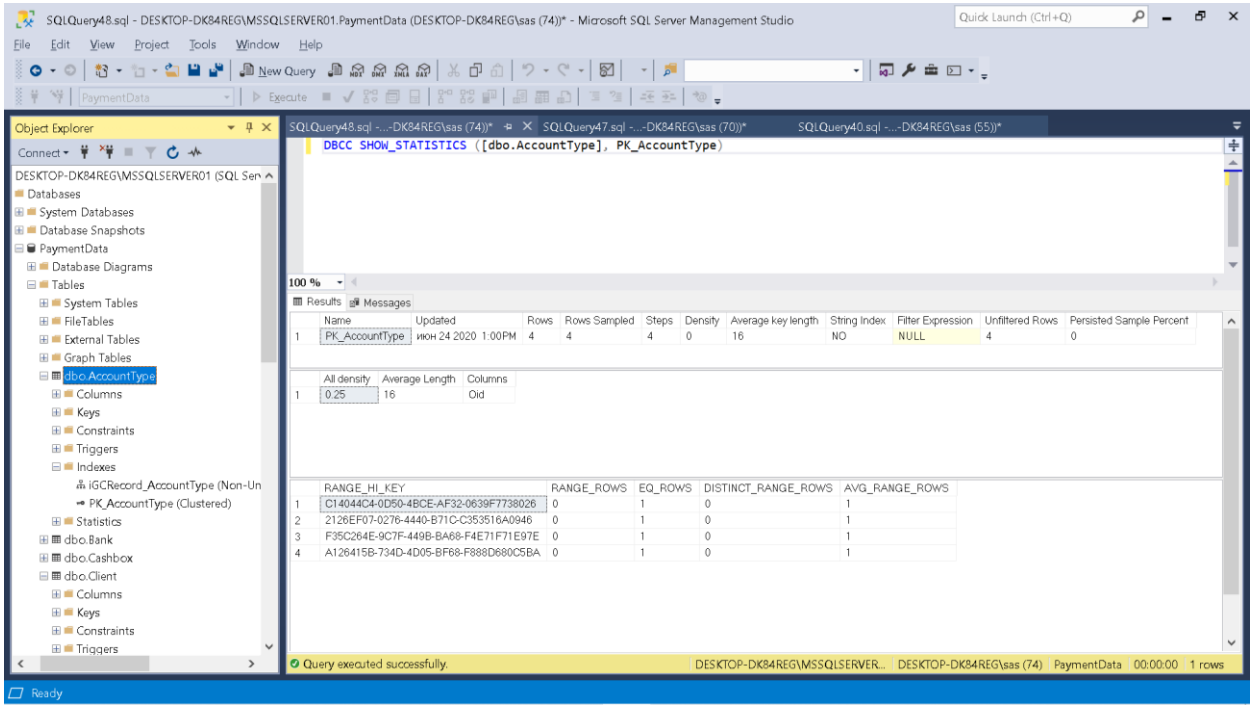

Рис. 1. Анализ индекса таблицы AccountType

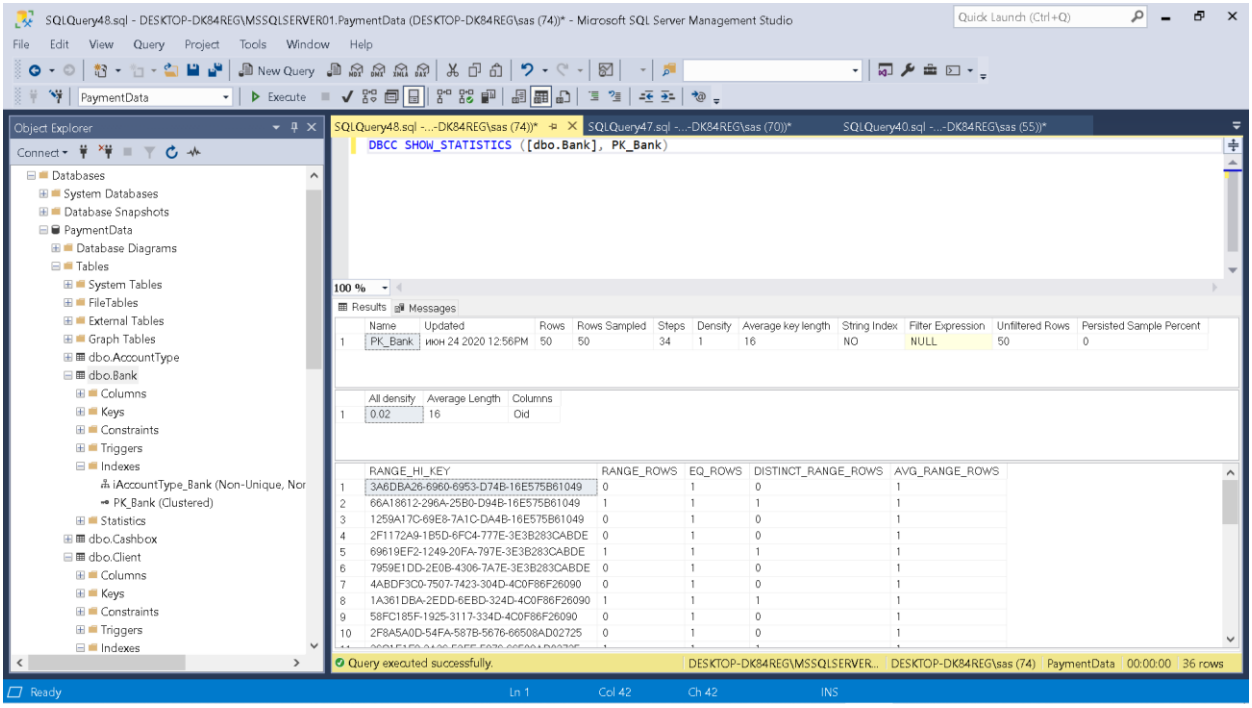

Рис. 2. Анализ индекса таблицы Bank

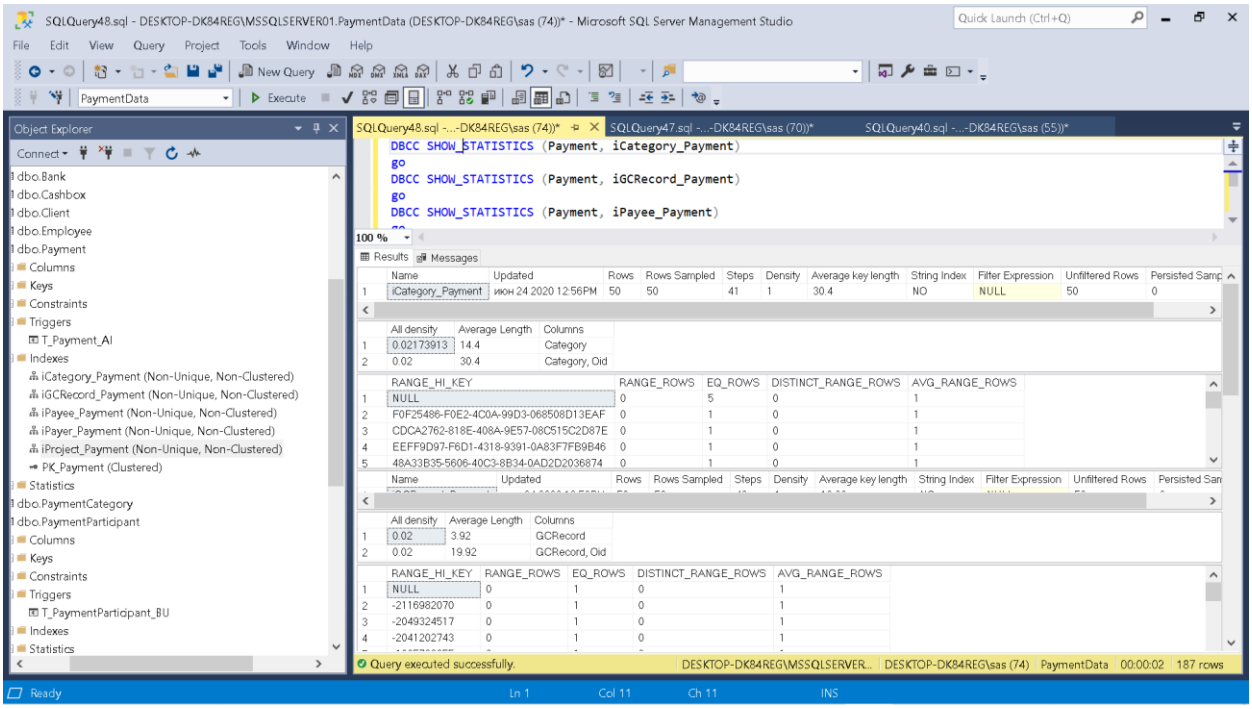

Рис. 3. Анализ индексов таблицы Payment

Такая работа была проделана для всех таблиц. В ходе анализа не было обнаружено плохих показателей, тем самым подтвердился тот факт, что существующие индексы являются полезными.

Далее была поставлена задача: определить, какие запросы к бд являются наиболее дорогими. Для этого использовался Activity Monitor, где можно увидеть запросы, скорость выполнения которых является замедленной. На Рис. 4 отображается результат анализа.

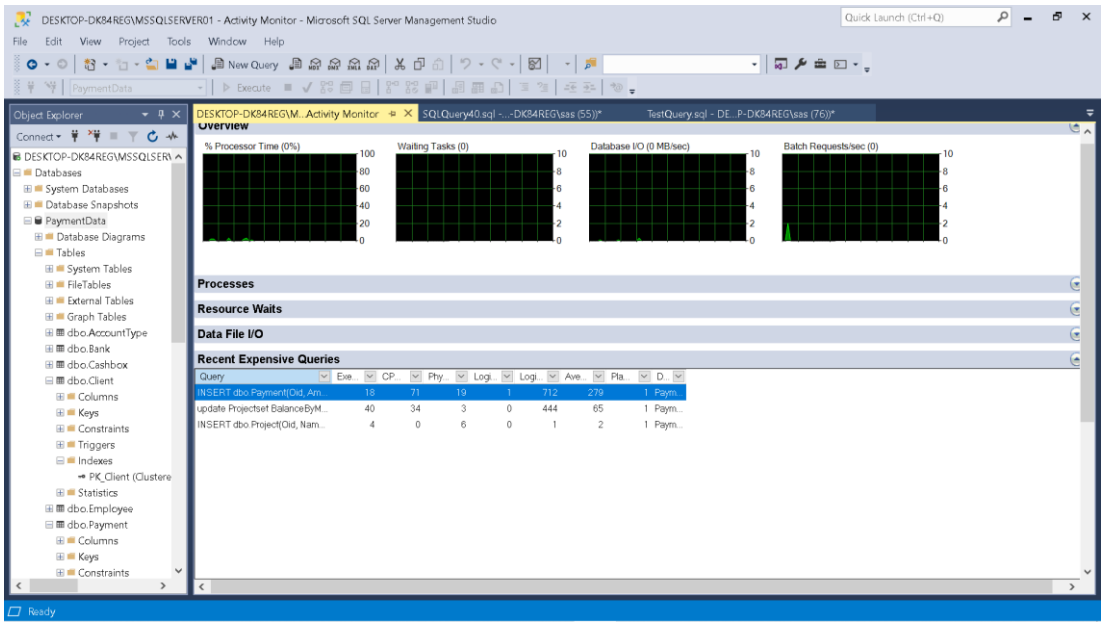

Рис.4. Результаты монитора активности

Данный анализ был проведен на данных размером 50 строк в таблице, однако далее, при работе с 5000 строками запрос по вычислению “BalanceByMaterial” также оказался дорогим (Рис. 5).

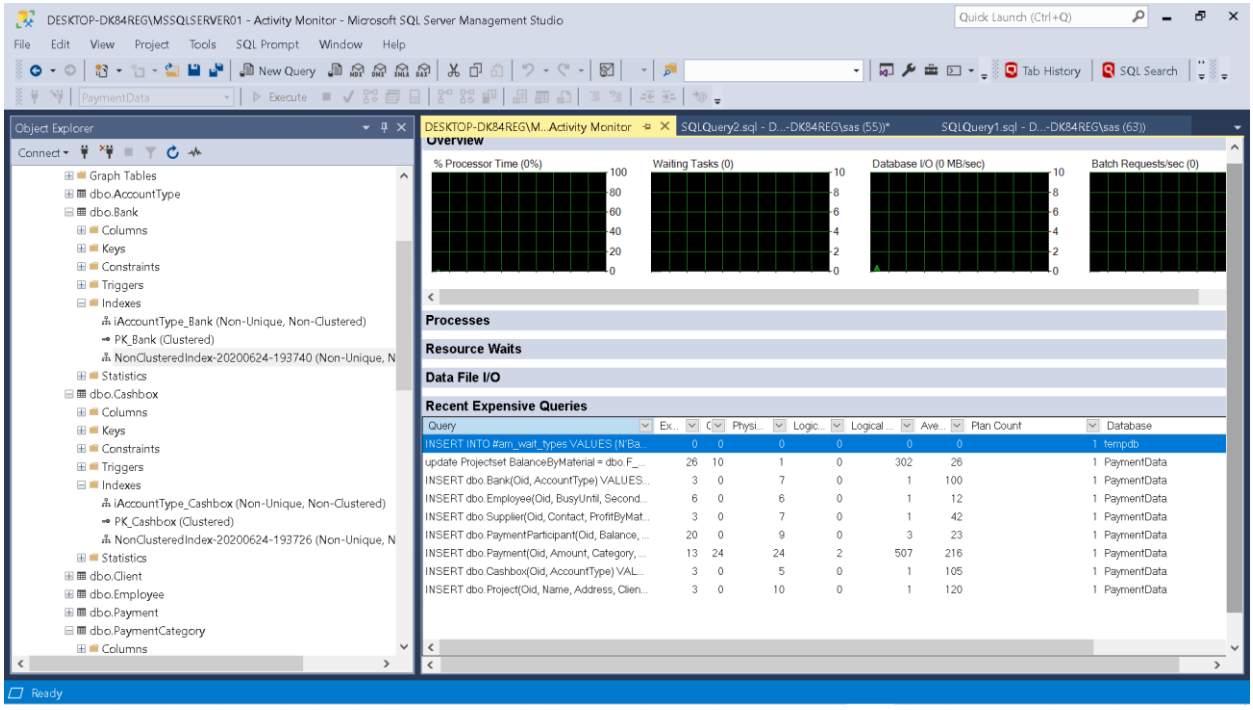

Рис. 5. Результат анализа производительности на 5000 строках

Можно заметить, что запросы по вставке также являются дорогими. Обусловлено это вычислением балансов.

Еще одним подходом для определения недостающих индексов была возможность отобразить их с помощью запроса

```
SET STATISTICS XML ON
```

Это было произведено для каждого запроса при расчете баланса. В результате в полученных xml файлах не было найдено тега <MissingIndexes/>.

Далее производился анализ с помощью SQL Server Profiler. Данный интерфейс позволяет создавать трассировки, управлять ими и получать результаты, которые можно в дальнейшем анализировать. Запрос был взят из данной статьи (https://docs.microsoft.com/ru-ru/sql/tools/sql-server-profiler/view-and-analyze-traces-with-sql-server-profiler?view=sql-server-ver15). В результате получилось подтвердить, что время исполнения скрипта превысило ожидаемое время (Рис. 6).

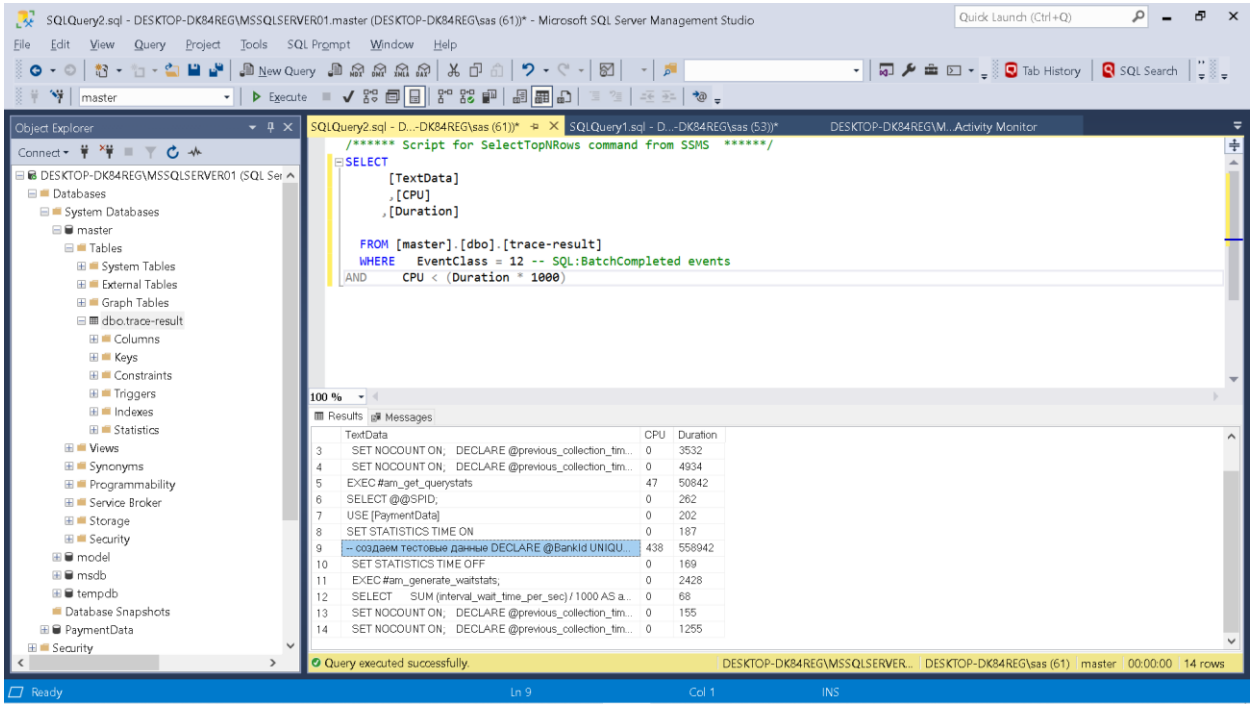

Рис 6. Результаты анализа работы SQL Server Profiler.

Помимо этого, был использован такой показатель, как “Object Execution Statistics”, что отображает время выполнения всех хранимых процедур и триггеров (Рис. 7).

Object No.*	Object Name	Object Type		Avg. CPU Time (ms.)	Total CPU Time (%)	# Avg. Logical Reads	# Avg. Logical Writes	# Avg. Logical IO	Total Logical IO (%)
1	[dbo]..F_CalculateBalanceBy Material	SQL Scalar-Function		0.56	0.15	46.00	0.00	46.00	0.22
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	SELECT @Profit = SUM(Payment.Amount)	1	1	0.34		26.00	0.00	26.00	
	SELECT @Cost = SUM(Payment.Amount)	1	1	0.22		20.00	0.00	20.00	
2	[dbo]..F_CalculateRemainder TheAdvance	SQL Scalar-Function		0.39	0.10	20.00	0.00	20.00	0.09
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	SELECT @Profit = SUM(Payment.Amount)	1	1	0.23		10.00	0.00	10.00	
	SELECT @Cost = SUM(Payment.Amount)	1	1	0.16		10.00	0.00	10.00	
3	[dbo]..T_PaymentParticipant_ BU	SQL Trigger		5.06	5.34	24.00	0.00	24.00	0.45
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	UPDATE [dbo].[PaymentParticipant]	4	1	**5.06**		24.00	0.00	24.00	
4	[dbo]..F_CalculateBalanceBy Work	SQL Scalar-Function		0.39	0.21	28.00	0.00	28.00	0.26
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	SELECT @Profit = SUM(Payment.Amount)	2	1	0.26		18.00	0.00	18.00	
	SELECT @Cost = SUM(Payment.Amount)	2	1	0.13		10.00	0.00	10.00	
5	[dbo]..F_CalculateProjectBala nce	SQL Scalar-Function		0.25	0.06	10.00	0.00	10.00	0.05
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	SELECT @Cost = SUM(Payment.Amount)	1	1	0.25		10.00	0.00	10.00	
6	[dbo]..T_Project_BU	SQL Trigger		31.62	16.67	4 529.50	63.50	4 593.00	43.43
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	UPDATE [Project] SET [Name] = inserted.Name	2	1	**31.62**		4 529.50	63.50	4 593.00	
7	[dbo]..T_Payment_AI	SQL Trigger		293.87	77.47	11 610.00	127.00	11 737.00	55.49
	SQL Statement	**# Executions (With Last**	**# Plans Generated**	**Avg. CPU Time (ms.)**		**# Avg. Logical Reads**	**# Avg. Logical Writes**	**# Avg. Logical IO**	
	update PaymentParticipant set Balance =	1	1	33.40		353.00	0.00	353.00	
	update PaymentParticipant set Balance =	1	1	10.33		99.00	0.00	99.00	
	update PaymentParticipant set Balance =	1	1	10.76		2.00	0.00	2.00	
	update PaymentParticipant set Balance =	1	1	10.73		2.00	0.00	2.00	
	update Project set BalanceByMaterial =	1	1	**195.31**		6 944.00	57.00	7 001.00	
	update Project set BalanceByMaterial =	1	1	33.33		4 210.00	70.00	4 280.00	

Рис. 7. Статистические данные по триггерам и процедурам

Отсюда были выделены запросы, на которые тратится больше CPU Time. Данная таблица оказалась полезна и для выполнения задания 1 части 2.

Инструмент Database Engine Tuning Advisor позволил изучить, какие поля из таблиц используются чаще всего в вычислениях, чтобы в дальнейшем на основе этого выбирать, что стоит использовать в качестве индексов (Рис.8).

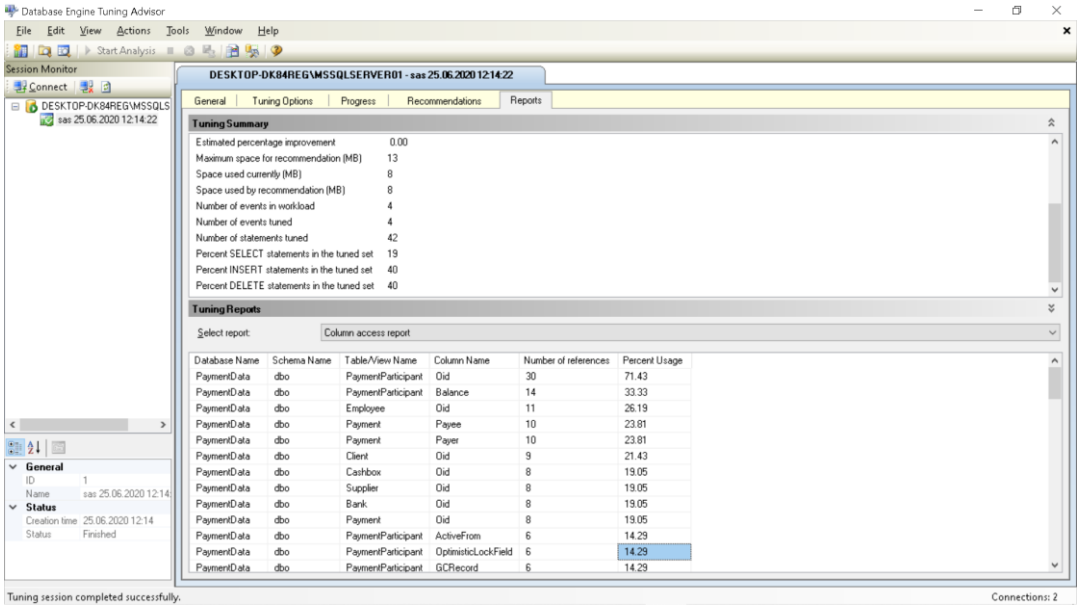

Рис. 8. Таблица частоты использования полей в запросах

Далее был исследован вручную запрос «INSERT dbo.Payment …», в котором производились вычисления баланса. Во внутренних запросах были определены поля, которые используются в части «WHERE». В запросе по обновлению таблицы “PaymentParticipant” были выделены следующие поля:

```
SupplierPayer.ProfitByMaterialAsPayer, CashboxAccountType.Name,
BankAccountType.Name, PaymentCategory.ProfitByMaterial,
SupplierPayer.CostByMaterialAsPayer, PaymentCategory.CostByMaterial
```

К этим полям были добавлены некластерные индексы. Был проведен анализ на Monitor Activity. На Рис. 9 представлен результат ДО добавления новых индексов, на Рис.10 – после добавления.

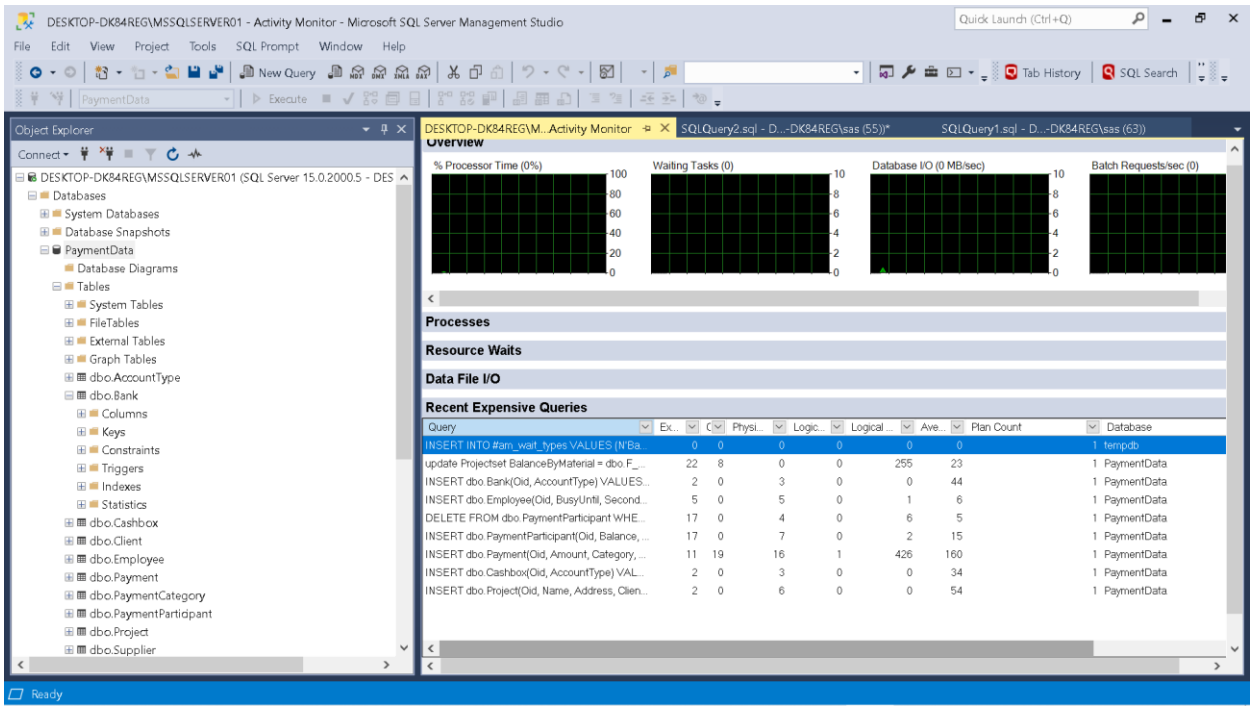

Рис. 9. Дорогие запросы до добавления новых индексов

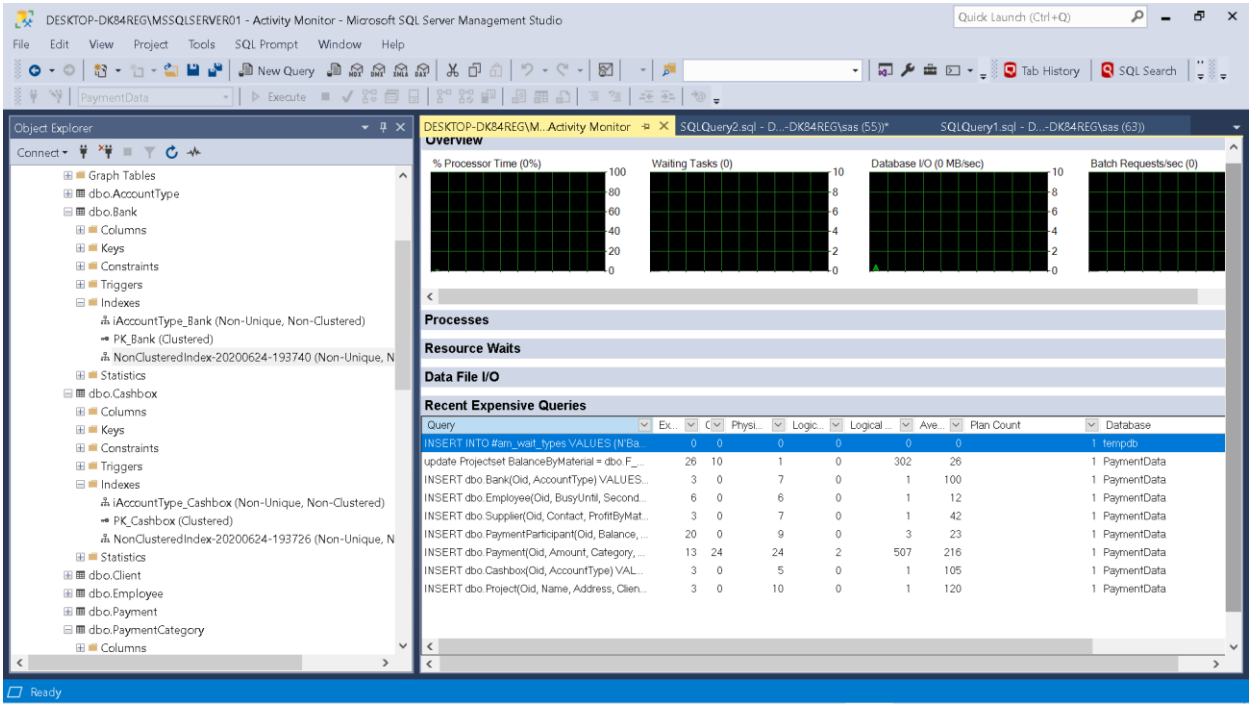

Рис. 10. Дорогие запросы после добавления индексов

По рисункам видно, что производительность ухудшилась (было 22мс, стало 26 мс).

Далее был проведен эксперимент, по добавлению лишь части из перечисленных индексов. Здесь для анализа использовалось расширение Client Statistics (Рис. 11).

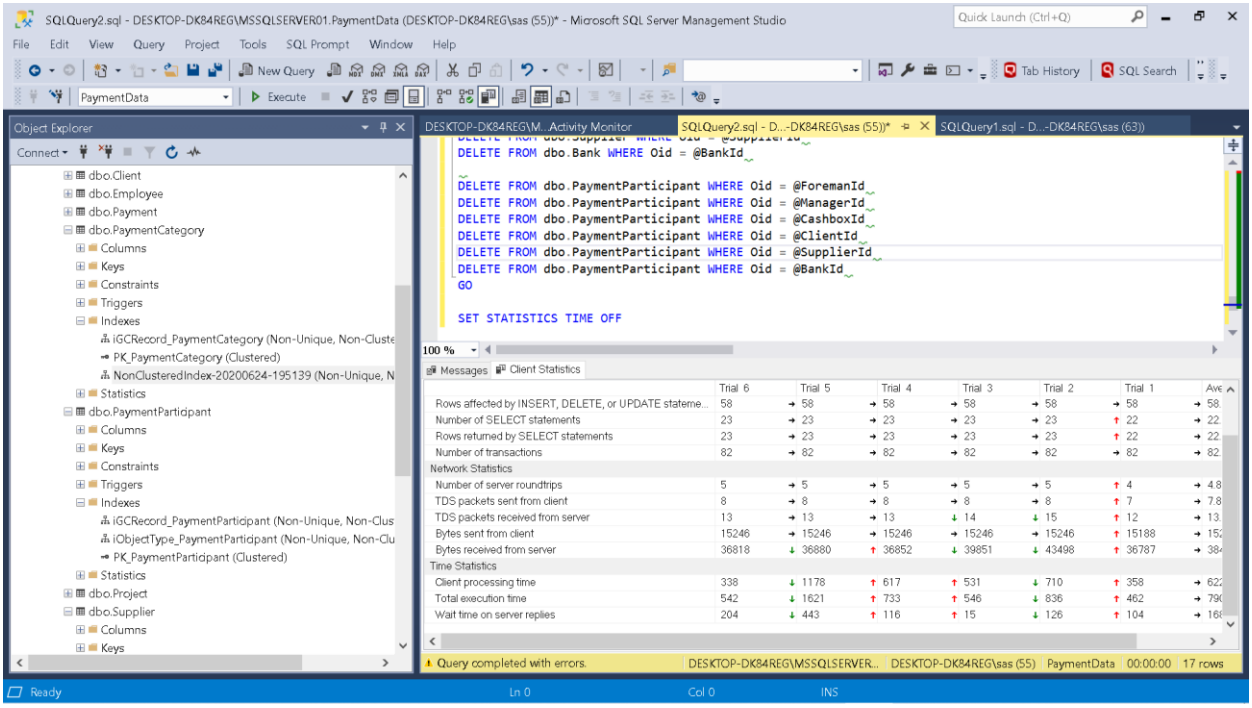

Рис. 11. Эксперимент по добавлению части индексов

Описание:

Trial 1: нет новых индексов

Trial 2: индексы в PaymentParticipant, Bank, Cashbox, Supplier

Trial 3: убран индекс из Supplier

Trial 4: убран индекс из PaymentCategory

Trial 6: возвращен индекс в PaymentCategory

Вывод: добавление индексов даже таким способом не улучшает производительность.

Далее был изучен запрос по обновлению таблицы Project (так как тоже занимает много времени). Был добавлен индекс к полю PaymentCategory.Name. По результатам (Рис.12) видно, что ничего не улучшилось.

Messages | Client Statistics

	Trial 2	Trial 1	Average
Client Execution Time	12:15:50	12:15:28	
Query Profile Statistics			
Number of INSERT, DELETE and UPDATE statements	82 →	82 →	82.0000
Rows affected by INSERT, DELETE, or UPDATE stateme...	58 →	58 →	58.0000
Number of SELECT statements	23 →	23 →	23.0000
Rows returned by SELECT statements	23 →	23 →	23.0000
Number of transactions	82 →	82 →	82.0000
Network Statistics			
Number of server roundtrips	5 →	5 →	5.0000
TDS packets sent from client	8 →	8 →	8.0000
TDS packets received from server	14 ↑	13 →	13.5000
Bytes sent from client	15310 →	15310 →	15310.0000
Bytes received from server	40526 ↑	37048 →	38787.0000
Time Statistics			
Client processing time	708 ↑	340 →	524.0000
Total execution time	734 ↑	479 →	606.5000
Wait time on server replies	26 ↓	139 →	82.5000

Рис. 12. Результаты добавления индекса по Payment.Name

В ходе работы я ознакомилась с тем, какие индексы вводились коллегами за предыдущие года. С использованием нагрузочного тестирования (инструмент SQL Query Stress) была проведена следующая работа:

1.Нагрузочного тестирование по 1000 итераций на 5 потоков без использования новых индексов (Рис. 13).

2.Нагрузочное тестирование с использованием индексов по `dbo.Supplier (CostByMaterialAsPayer)`, `dbo.Supplier (ProfitByMaterialAsPayee)`, `dbo.Supplier (ProfitByMaterialAsPayer)`, `dbo.PaymentCategory (NotInPaymentParticipantProfit)`, `dbo.PaymentCategory (CostByMaterial)`, `dbo.PaymentCategory (ProfitByMaterial)`, `dbo.PaymentCategory (Name)`, `dbo.AccountType (Name)` (Рис. 14).

3.Нагрузочное тестирование с использованием составных индексов (Рис. 15):

```
dbo.Supplier (CostByMaterialAsPayer) + dbo.Supplier (ProfitByMaterialAsPayee) + dbo.Supplier
(ProfitByMaterialAsPayer);

dbo.PaymentCategory (NotInPaymentParticipantProfit) + dbo.PaymentCategory (CostByMaterial) +
dbo.PaymentCategory (ProfitByMaterial) + dbo.PaymentCategory (Name);
```

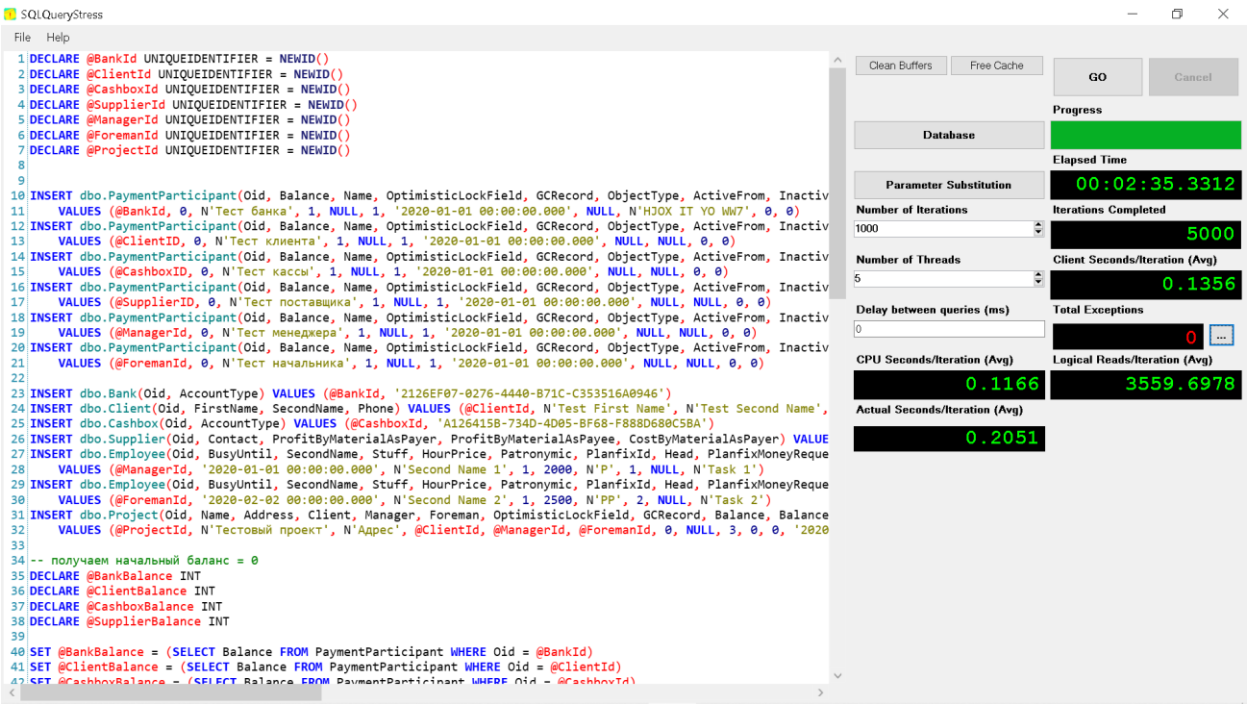

Рис. 13. Результаты нагрузочного тестирования без новых индексов

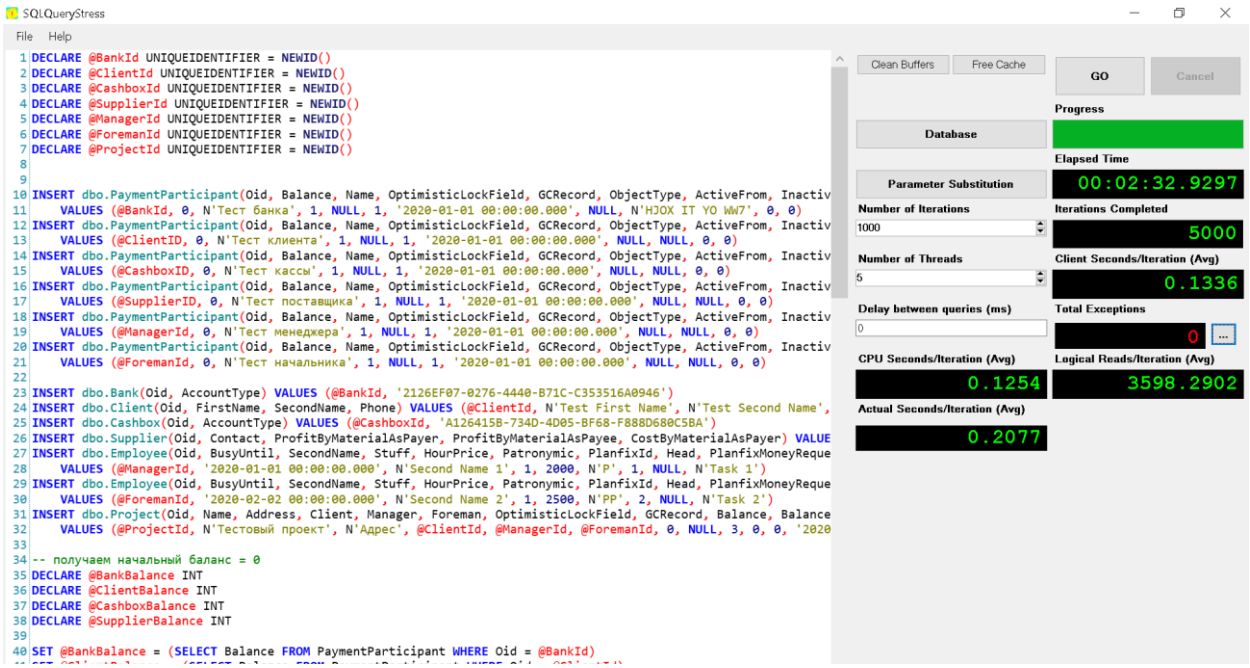

Рис. 14. Результаты нагрузочного тестирования с новыми индексами

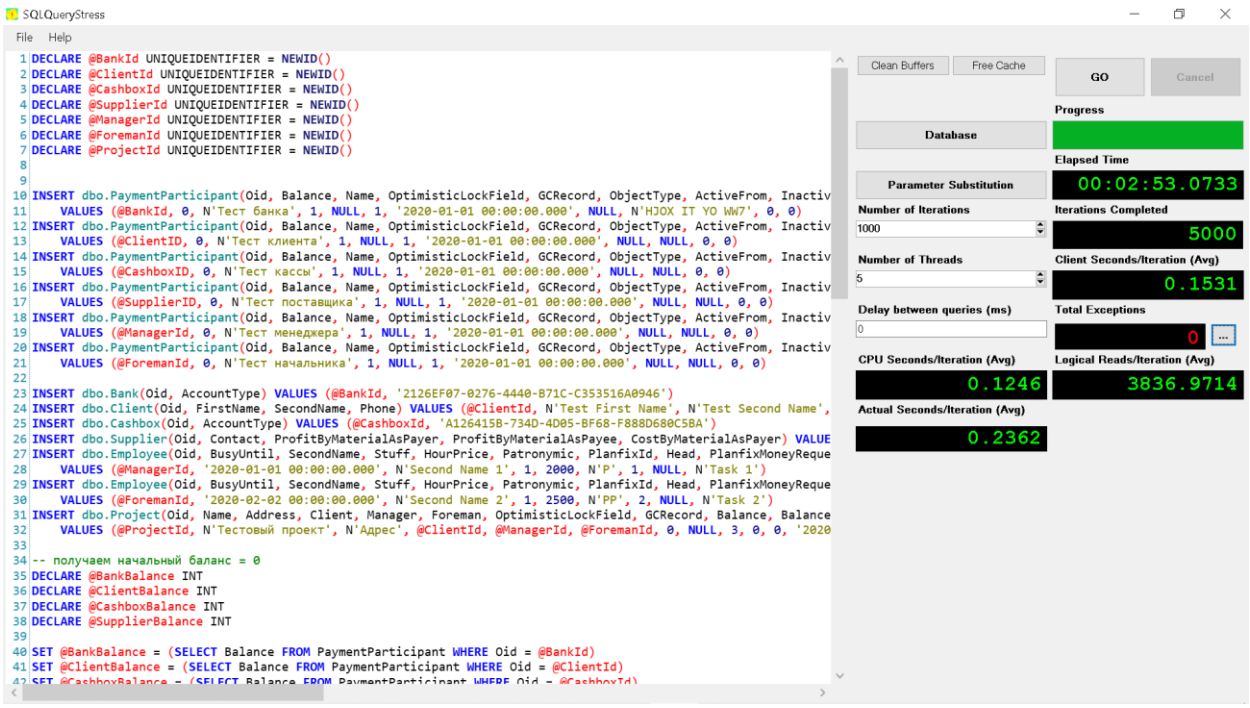

Рис. 15. Результаты нагрузочного тестирования с составными индексами

Результаты:

Вариант	CPU (Avg)	Elapsed time	Actual query time
Нет индексов новых	0.1166	2:35.332 минуты	0.2051
Новые индексы (7 шт.)	0.1254	2:32.9297 минуты	0.2077
Составные индексы	0.1246	2:53.0733 минуты	0.2362

Таблица 2. Сравнение результатов нагрузочного тестирования

Можно отметить, что общее время тестирования при добавлении новых индексов уменьшилось, однако среднее время обращения к процессору и среднее время выполнения всего скрипта увеличилось.

В ходе работы была попытка анализировать изменение плана выполнения запроса в зависимости от добавления индексов, но для каждого запроса в транзакции планы оказались очень большими, что вызвало сложности в анализе.

Вывод: В ходе анализа было принято решение, что добавление индексов не приводит к оптимизации производительности. Обусловлено это тем, что при INSERT происходит изменение индексов, их реорганизация, что приводит к плохому результату при работе с большими данными. Здесь чем больше индексов, тем хуже. Поэтому логичнее оставить существующие индексы по первичным и внешним ключам.

Статьи, что помогли мне делать анализ:

1. https://habr.com/ru/post/336586/
2. https://habr.com/ru/post/310328/
3. http://wikie.lexema.ru/index.php/Анализ_запросов_с_помощью_SQL_Profiler
4. http://ts-soft.ru/blog/sql-optimization-1