Программа краткого курса

## Контроль и повышение производительности в решениях аналитики и отчётности в условиях высоких нагрузок

Редакция 2021-08-24

Легенда:

Вы устроились работать разработчиком программного обеспечения. Вашрабочийстек: **C#, MS SQL Server, Net Core, Net framework**.

Вам передают на обслуживание старую систему (**legacy**) для подготовки специализированной отчетности. Система (**legacy**) реализована на следующем стеке**: MSSQLServer, ASP.net, Netframework 4.7.**

Данные отчеты направляются по защищенному каналу разным потребителям: ЦБ РФ, Министерство финансов и на анализ независимым аудиторам. Для каждого потребителясистема формирует один и те же данные, но в разных форматах. Каждый выгруженный отчет фиксируется статусом. Если отчет не принят, то формируется повторный отчет с обновленными данными. Если отчет принят условно, то формируется дополнение к ранее отправленному отчету, в котором хранятся данные, которые только дополняют ранее отправленные данные.

Перед Вами непростая задача. Текущее (**legacy**)приложение ввиду того, что создавалось на «ходу» имеет много недостатков, но основной недостаток – это производительность. Данные долго готовятся. Проверяются, а затем выгружаются. В случае, если коллектив не успел в срок, организация автоматически получает штраф.

Ваша задача: найти и устранить все «слабые» места в приложении. При этом, не возможно провести полную замену приложения ввиду большой и постоянной нагрузки, высоких требований и жестких регламентов.

|  |
| --- |
| **Оглавление**   1. Анализ текущего проекта и замеры производительности. 2. Проведение оптимизации базы данных без изменения структуры. 3. Проведение оптимизации приложения с использованием **micro ORM Dapper**. 4. Проведение оптимизации приложения с использованием многопоточного программирования. 5. Итоги. |

**Этап 1: Анализ текущего проекта и замеры производительности**

Проект представляет с собой высоконагруженную базу данных, которая разделена на три части: учет фактов, учет обработанных данных, учет выгруженной информации. Учет фактов состоит из таблицы **tblTransactionFacts** в которой содержатся все исходные банковские транзакции, которые подлежат контролю. Учет обработанных данных состоит из нескольких таблиц:

|  |  |
| --- | --- |
| refAccounts | Счета клиентов |
| refContracts | Контракты клиентов |
| refCustomers | Клиенты |
| tblTransactions | Подготовленные транзакции |
| refReportPeriods | Справочник периодов |

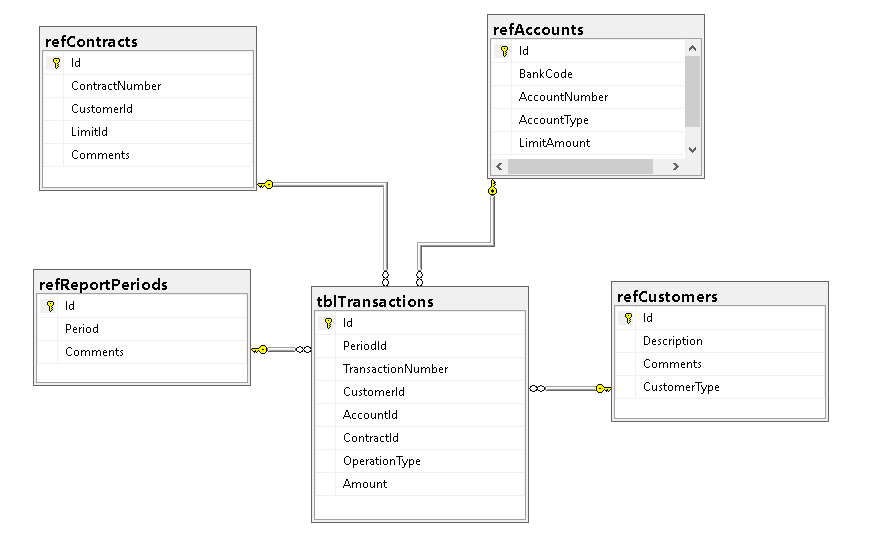


Рис. 1. Схема связей таблиц для учета обработанных данных

Учет выгруженной информации состоит из следующих таблиц:

|  |  |
| --- | --- |
| refReportPeriods | Таблица с периодами |
| repCustomerHistory | Таблица с информацией о выгрузке в разрезе каждого клиента. Для аудита |
| refRecipients | Справочник получателей информации |
| repCustomerTotals | Итоги выгрузки в разрезе получателя и клиента |
| refCustomers | Клиенты |

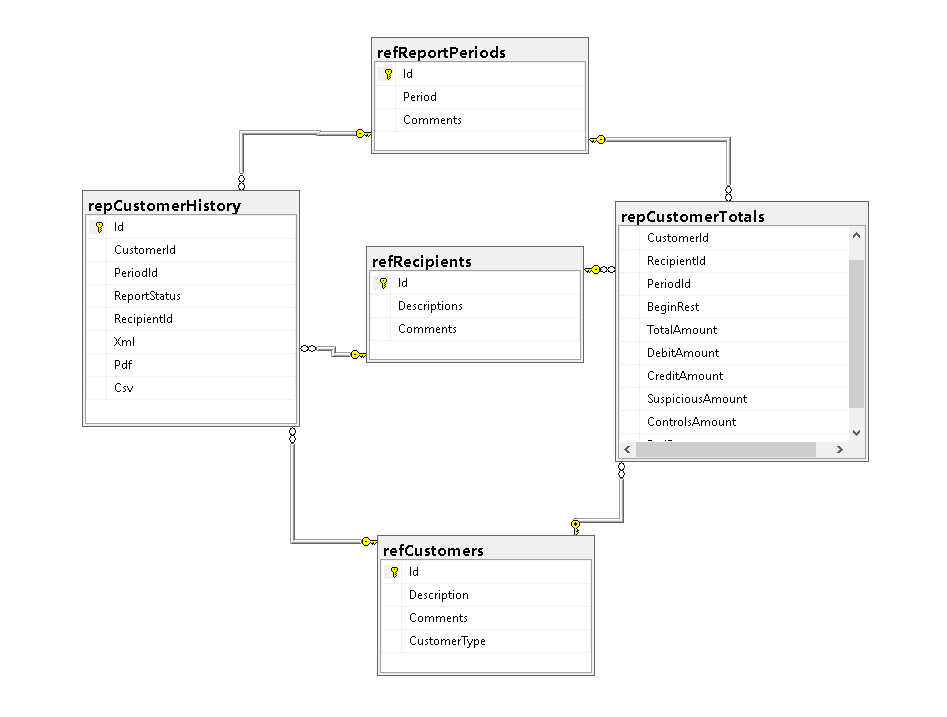


Рис. 2 Схема связей таблиц для учета выгруженной информации.

Исходные метрики:

|  |  |
| --- | --- |
| Количество записей в исходной таблице фактов | 3 129 977 |
| Разброс по периодам | 2018-08-19 00:00:00.000 2021-05-15 00:00:00.000 |
| Объем базы данных | ~15 ГГб |
| Выборка и обработка данных | |
| Получить список периодов | 6 сек. |
| Получить список записей для обработки за 1 месяц | 3,1 сек. |
| Подготовить данных за 10 дней | 1,1 мин. |
| Флаг наличия новых данных | 1,7 сек. |

**Этап 2. Оптимизация базы данных без изменения структуры**

В ходе анализа установлены несколько ключевых запросов. Данные операции выполняются часто и приводят к большим затратам ресурсов сервера. Пример алгоритма подготовки данных.

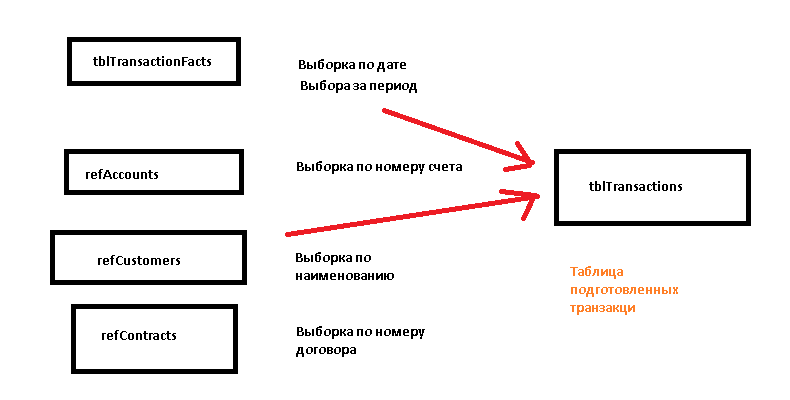


Рис. 3. Схема работы алгоритма для подготовки данных

Описание:

* Определить блок новых данных

Для каждой записи таблицы **tblTransactionFacts** формируется 4 операции:

* подобрать (добавить) счет
* подобрать (добавить) клиента,
* подобрать (добавить) контракт

Для определения ключевых запросов сформируем набор модульных тестов с определенными критериям и произведем запуск тестов. Далее, проведем анализ наиболее длительных тестов.

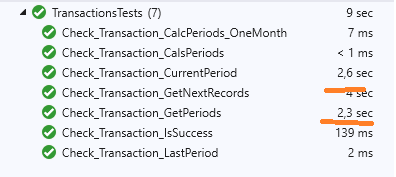


Рис. 4. Изображение замера работы модульных тестов.

Операции по оптимизации. Для проведения оптимизации сформируем копию исходной базы данных – **legacyFinish** и изменим строку соединения с базой данных.

|  |  |  |  |
| --- | --- | --- | --- |
| № | Операция | Результат | Итог |
| 1 | Сжать базу данных и перестроить индексы  PerformanceDbStep1.sql | |  |  | | --- | --- | | Выборка и обработка данных | | | Получить список периодов | 1,6 сек. | | Получить список записей для обработки за 1 месяц | ~5 сек. | | Подготовить данных за 10 дней | ~1 мин. | | Флаг наличия новых данных | 32 мс. | |  |
| 2 | Проверить выборку данных в таблице фактов. Запускаем Profiler. Анализируем запрос, который подготовил EF. Сформируем индекс  PerformanceDbStep2.sql | |  |  | | --- | --- | | Выборка и обработка данных | | | Получить список периодов | 115 мс. | | Получить список записей для обработки за 1 месяц | ~5 сек. | | Подготовить данных за 10 дней | ~1 мин. | | Флаг наличия новых данных | 30 мс. | |  |
| 3 | Проверим выборку данных при построении данных. Сформируем индексы для таблиц. | |  |  | | --- | --- | | Выборка и обработка данных | | | Получить список периодов | 24 мс. | | Получить список записей для обработки за 1 месяц | 4,5 сек. | | Подготовить данных за 10 дней | 43 сек | | Флаг наличия новых данных | 21 мс. | |  |

Задания:

1. Провести дальнейший анализ исходной базы данных на предмет соответствия критериям [нормальной формы.](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%84%D0%BE%D1%80%D0%BC%D0%B0)
2. Написать T-SQL скрипт для удаления данных из таблицы tblTransaction за указанный период с использованием транзакции.
3. Написать T-SQL скрипт для обработки исходных транзакций аналог реализация метода Build класса BuildManager

**Этап 3. Проведение оптимизации приложения с использованием micro ORM Dapper**

Предпосылки для применения micro ORM Dapper (либо замена на прямые SQL запросы).

* Массовые операции удаления;
* Сложные разовые SQL запросы;

Выделим реализацию бизнес процесса в отдельный интерфейс IlogicBuildTransactions. Реализуем данный интерфейс logicBuildTransactions и создадим дополнительные модульные тесты. Далее, сделаем замер каждой части модульного теста и получим результат для каждой отдельной части бизнес процесса.

|  |
| --- |
|  |

Проверяем, работу ORM при удалении большого объема данных при помощи ORM EF. Выполняется в два этапа: выборка данных и удаление по одной записи.

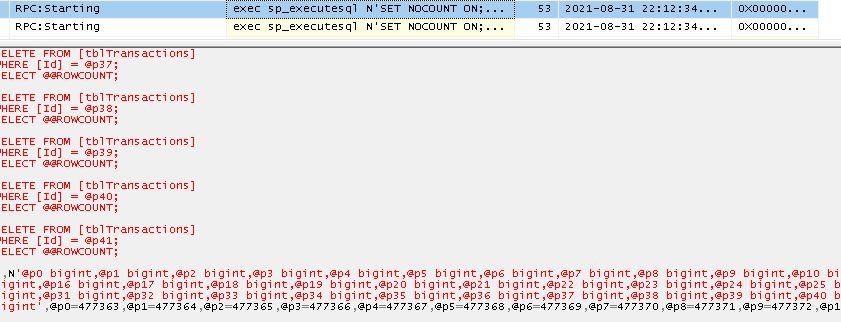


Рис. 5. Вид работы ORM при удалении записей из таблицы tblTransactions

Сделаем реализацию от класса LogicBuildTransactions и дополнительный модульный тест.

|  |  |
| --- | --- |
| public class LogicBuildTransactionsWithDapper: LogicBuildTransactions  {  public override bool RemoveRecord(ITransactionsPeriod period)  {  var dbPeriod = GetPeriod(period);  var cs = @"Server=LEGACYSERVER\SQLEXPRESS;Database=LegacyFinish;Trusted\_Connection=No;Connection Timeout=500;User Id=sa;Password=123456;";  using var connect = new SqlConnection(cs);  connect.Open();  var sql = $"Delete from tblTransactions where PeriodId = {dbPeriod.Id}";  var version = connect.ExecuteScalar(sql);  return true;  }  } | 30 vs 17 сек. |

Вопросы:

1. Есть ли еще возможность увеличить производительность операции подготовки данных для отчета с использованием ORM Dapper?

|  |
| --- |
| public bool Build(ITransactionsPeriod period) |

1. Есть ли возможность заменить типовой ORM EF на micro ORM Dapper?

**Этап 4. Проведение оптимизации приложения с использованием многопоточного программирования**

Для реализации многопоточного процесса нам необходимо переопределить бизнес класс

BuildManager . Для этого в классе данный метод отметим как виртуальный и создадим новый класс.

Так же, необходимо в базовом классе сделать основные свойства доступным для наследников. Заменим метод доступа private на protected

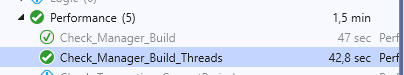
|  |
| --- |
| protected ITransactions \_transactions;  protected IlogicBuildTransactions \_logic; |

Объединим этапы 1,2,3,4 в одну группу, а этап 5 в другую группу т.к. этап 5 (добавления новый транзакций) должен быть запущен только после выполнения этапов 1 – 4.

Далее, вызов каждой операции включим в отдельный поток и запустим 2 группы потоков.

|  |
| --- |
| // 1 Удалить старые записи  var removeRecord = Task.Run (() => \_logic.RemoveRecords(period));  // 2. Добавим новых клиентов  var addCustomers = Task.Run (() => \_logic.AddCustomers(period));  // 3 Добавим счета  var addAccounts = Task.Run(() => \_logic.AddAccounts(period));  // 4 Добавим контракты  var addContracts = Task.Run(() => \_logic.AddContracts(period));  // 5 Добавим транзации  var addTransactions = Task.Run(() => \_logic.AddTransactions(period)); |
| // Запускаем задачи  Task.WhenAll(firtsGroup.ToArray()).GetAwaiter().GetResult();  Task.WhenAll(secondGroup.ToArray()).GetAwaiter().GetResult(); |

Результат выполнения тестов. Прирост производительности ~8%



Задачи:

1. В примере, дополнить класс BuildManagerThreads исползуя класс для работы с LogicBuildTransactionsWithDapper
2. Написать класс BuildManager с внедрением интерфейса IlogicBuildTransactions таким образом, чтобы один класс BuildManager мог работать с разными реализациями.