1. **Цель эксперимента:**

Цель эксперимента заключается в измерении времени обновления записей в базе данных с их большим количеством (1000, 10000, 100000, 1000000).

1. **Проведение:**

Эксперимент проводился в тестовом методе. В нём заводилось два словаря, заполненные названиями столбцов и значениями этих столбцов. Один словарь предназначался для добавления в базу данных, второй для редактирования записи. Также был создан объект типа Random для генерации случайных id, существующих в таблице и ограниченных максимальным id, и лист для их записи, чтобы они не повторялись.

В итоге проводилось три типа манипуляций над записями:

* Добавление
* Изменение
* Удаление

Итак, алгоритм теста заключался в следующем:

1. Генерация случайного, неповторяющегося id;
2. Изменение записи в таблице по этому id;
3. Генерация следующего случайного id;
4. Удаление строки по этому id;
5. Добавление новой строки.
6. Повторение 5 предыдущих шагов 330 раз.

Число 330 было выбрано неслучайно. При проведении эксперимента с 1000 записями большее число повторений не позволило бы сгенерировать новый уникальный id, который существовал бы в таблице. Так как операций было 3, и для каждой требовался новый id, 330\*3=990. То есть почти 1000. Было решено взять не 333, а для 330 лишь для округления, так как 9 лишних операций вряд ли бы сильно повлияли на результат эксперимента. Также очевидно, что по мере увеличения записей нельзя увеличивать и количество проводимых операций, ибо это является противоположностью чистоты проводимых тестов.

Однако позже путём более тщательного изучения кода было выяснено, что уникальных id требовалось только на две операции: изменение и удаление. На добавление id генерируется автоматически. Из-за стадии обнаружения этой ошибки, а именно конец эксперимента, когда оставалось уже проанализировать полученные результаты, а также из-за длительности проведения тестов, было решено эксперимент не повторять. Также 990 каким-либо образом изменённых записей за один сеанс без обращения в базу данных – уже очень смелое предположение, вряд ли это уместно хоть при каких-либо обстоятельствах, поэтому увеличение числа изменённых записей вряд ли увеличит реальность и целесообразность эксперимента.

В итоге тестовый метод имеет следующий код:

public void InsertUpdateDelete()

{

IRepository<Debtor> repository = new Repository<Debtor>(new Debtor());

var table = repository.GetAllRecords();

var generatedIds = new List<int>();

var idGenerator = new Random();

var rowToInsert = new Dictionary<string, string>

{

{ "Category", "Физическое лицо" },

{ "Name", "Иван" },

{ "INN", "1234567891" },

{ "Adress", "ул. Крышкина, дом 15" },

{ "Phone", "89567854321" }

};

var rowToUpdate = new Dictionary<string, string>

{

{ "Category", "Физическое лицо1" },

{ "Name", "Иван1" },

{ "INN", "1234567892" },

{ "Adress", "ул. Крышкина, дом 151" },

{ "Phone", "89567854322" }

};

Func<int> funcIdGenerator = () => {

int randId = idGenerator.Next(1, 1000);

while (generatedIds.Contains(randId))

randId = idGenerator.Next(1, 1000);

generatedIds.Add(randId); return randId; };

for (int i = 0; i < 330; ++i)

{

int randId = funcIdGenerator();

var row = table.Rows[randId];

foreach (var element in rowToUpdate)

row[element.Key] = element.Value;

randId = funcIdGenerator();

table.Rows.RemoveAt(randId);

table.AddRow(rowToInsert);

}

repository.UpdateDB(table);

}

Объект класса Repository служит как раз для получения и обновления данных в БД.

Число в методе idGenerator.Next(1, 1000), а именно 1000 – верхняя граница случайного id. Изменяется с изменением количества записей. То есть от 1000 до 1000000.

Однако тут я не могу не позволить себе небольшое отступление. Совокупное количество проведённых тестов – 8. Первые 4 – с версией кода репозитория, в котором он после обновления БД снова к ней обращался и выбирал из неё все записи, тем самым обновляя таблицу у пользователя. Однако при увеличении количества записей количество времени на их выборку увеличивалось вплоть до 13,2 секунд при миллионе записей.

Поэтому в следующих 4 тестах было решено вместо повторной выборки записей просто возвращать пустую таблицу, так как:

1. Id генерируется автоматически в DataTable путём автоинкрементации и не генерируется в самой БД, поэтому несостыковки в первичном ключе не может быть;
2. После обновления данных в БД вызывается метод DataTable.AcceptChanges(), который сохраняет все данные в этой таблице, так что эти данные абсолютно схожи с данными в БД.

В результате такого изменения метод репозитория занимался лишь обновлением данных в БД. Теперь при миллионе записей время обновления составляло лишь 3,7 секунд.

Теперь о том, что касается, собственно, самого времени сохранения изменений при их большом количестве.

Насколько я понял, метод DataAdapter не перезаписывает полностью всю таблицу при обновлении записей. Он проверяет так называемый «RowState» каждой строки в DataTable. При состоянии INSERT, UPDATE или DELETE он выполняет соответствующий запрос. То есть, если изменяется лишь одна запись, то он обновляет только её в самой БД. Поэтому время для обновления записей составляет лишь от ~3 до 3,7 секунд (в обновлённой версии репозитория). В конце будет приложен соответствующий комментарий из Microsoft Docs, которым я руководствовался.

Конечно, можно отдельно записывать каждую добавленную, изменённую или удалённую запись в некую коллекцию, а потом с её помощью отдельно написанными запросами манипулировать БД, но для этого нужно писать код, зная, что есть столь удобные и полезные инструменты, которые избавляют от этой необходимости (DataTable, SQLDataAdapter, SQLCommandBuilder).

Далее представлены 4 рисунка со сравнением времени работы метода обновления с последующей выборкой данных и без (две разные версии репозитория, слева с выборкой, справа без).

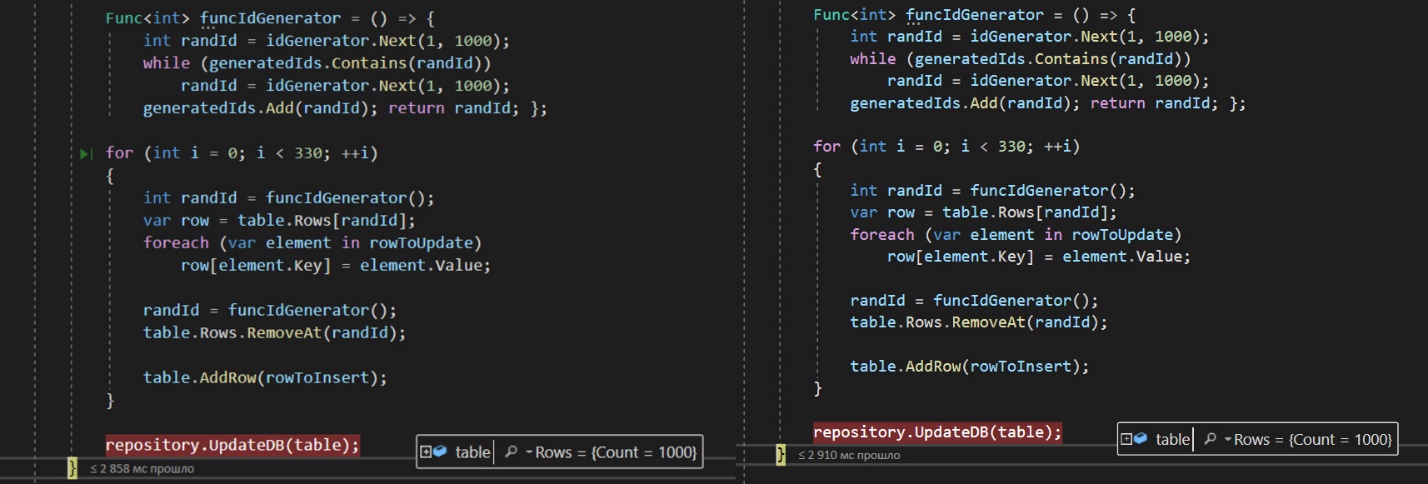


Рисунок 1 – Время работы методов обновления при 1000 записей (2 858 мс и 2 910 мс)

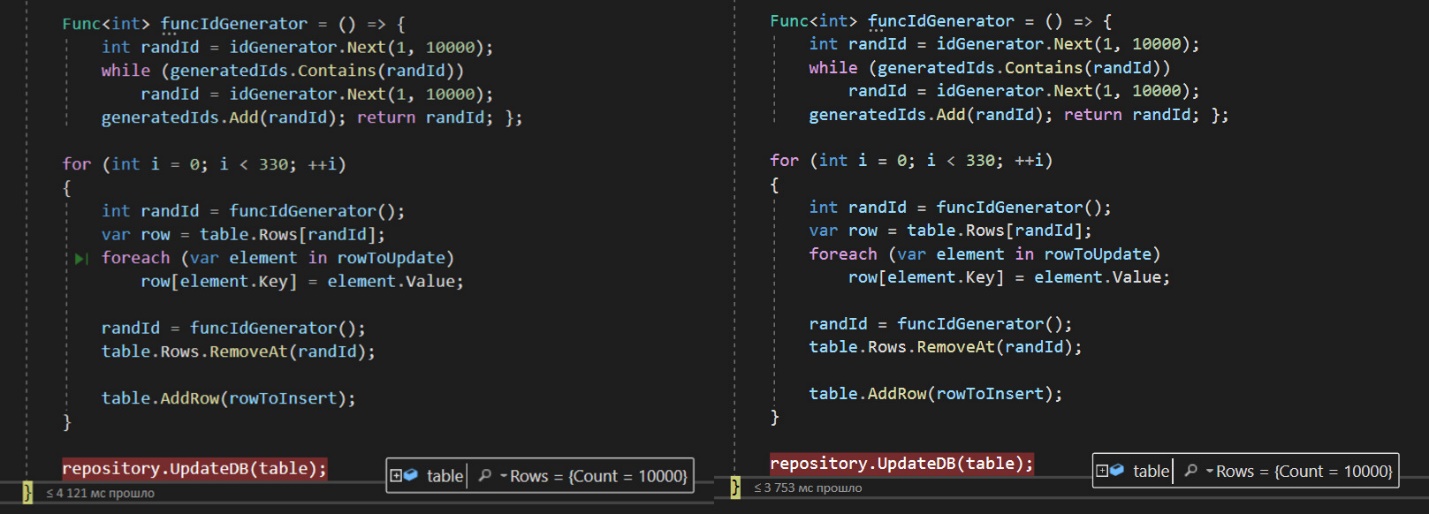


Рисунок 2 – Время работы методов обновления при 10000 записях (4 121 мс и 3 753 мс)

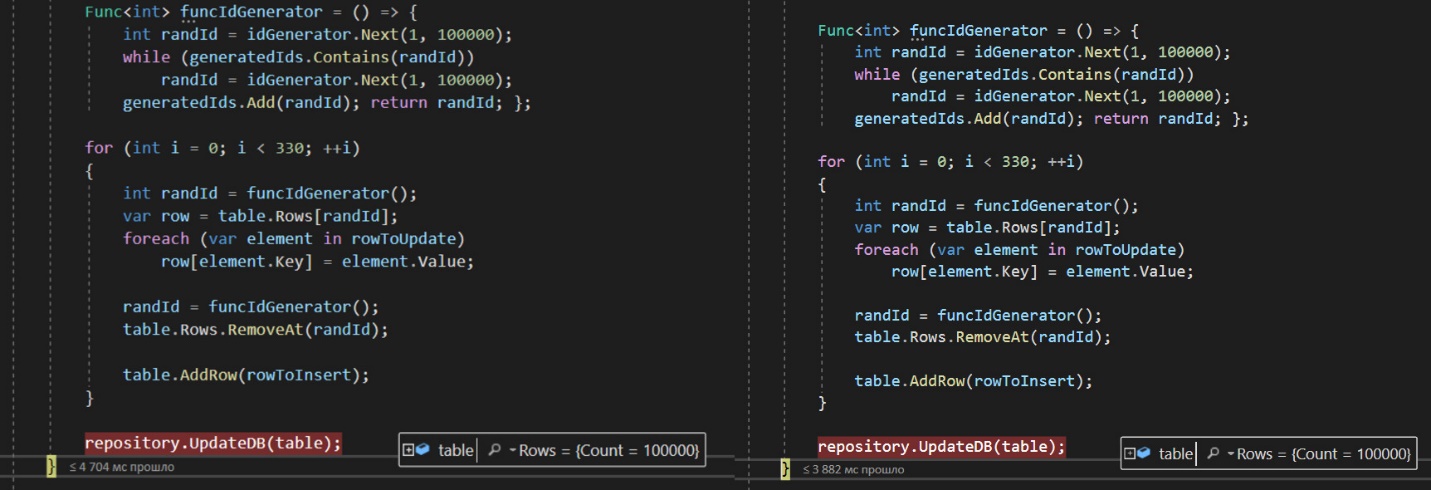


Рисунок 3 – Время работы методов обновления при 100000 записях (4 704 мс и 3 882 мс)

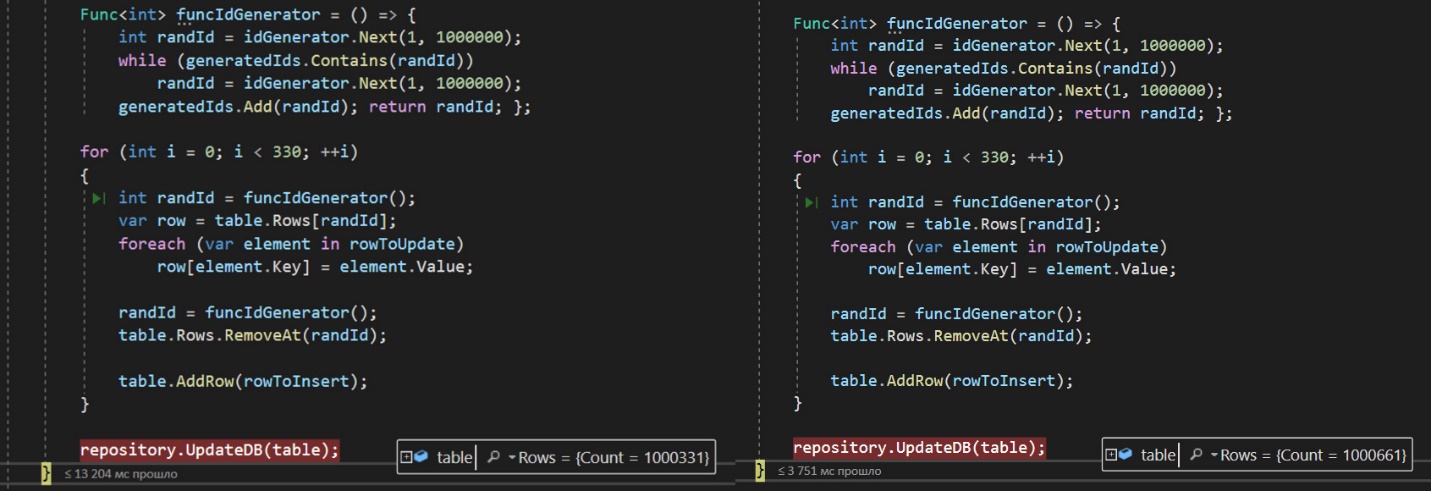


Рисунок 4 – Время работы методов обновления при 1000000 записей (13 204 мс и 3 751 мс)

Также хотелось бы заметить, что при увеличении количества записей в базе данных вторая версия метода обновления работает почти с тем же временем, чуть ли не в пределах погрешности.

Комментарий на Microsoft Docs:

Обновление выполняется отдельно по строкам. Для каждой вставленной, измененной и удаленной строки метод Update определяет тип изменения, которое было выполнено над ним (INSERT, UPDATE или DELETE). В зависимости от типа изменения, шаблон команд Insert, Update, или Delete выполняется для распространения измененной строки в источник данных. Когда приложение вызывает метод Update, DataAdapter проверяет свойство RowState и последовательно выполняет необходимые инструкции INSERT, UPDATE или DELETE для каждой строки в зависимости от порядка индексов, настроенных в DataSet. Например, Update может выполнить инструкцию DELETE, за которой следует инструкция INSERT, а затем другую инструкцию DELETE из-за упорядочения строк в DataTable.

Ссылка на источник: <https://docs.microsoft.com/ru-ru/dotnet/api/system.data.common.dataadapter.update?view=netframework-4.8>