**Tricolor RESTful API сервисы**

***Автор: Малоземов Станислав Анатольевич***

Магнитогорск 2022

**Содержание**

[1. Введение 4](#_Toc126835500)

[1.1. Выбор инфраструктуры 4](#_Toc126835501)

[2. Структура проекта 4](#_Toc126835502)

[2.1. Каталоги 4](#_Toc126835503)

[2.2. Внедрение зависимостей 4](#_Toc126835504)

[2.3. Поток данных 5](#_Toc126835505)

[2.4. Зависимости сервиса API 6](#_Toc126835506)

[3. Конфигурация 10](#_Toc126835507)

[4. Логирование 10](#_Toc126835508)

[4.1. Логирование хода выполнения 10](#_Toc126835509)

[4.2. Логирование запроса и ответа 10](#_Toc126835510)

[5. Обработка ошибок 12](#_Toc126835511)

[5.1. Обработка исключений 12](#_Toc126835512)

[5.2. Возврат бизнес-кодов ошибок 12](#_Toc126835513)

[6. Документирование API 12](#_Toc126835514)

[6.1. Подключение swagger 12](#_Toc126835515)

[6.2. 12](#_Toc126835516)

[7. Развёртывание 12](#_Toc126835517)

[7.1. Инструменты анализа производительности запросов 13](#_Toc126835518)

[7.1.1. Инструменты поиска плохих запросов: 13](#_Toc126835519)

[7.1.2. Инструменты анализа плохих запросов 14](#_Toc126835520)

[8. Основная часть 15](#_Toc126835521)

[8.1. Доступ к данным 15](#_Toc126835522)

[8.1.1. Таблицы 15](#_Toc126835523)

[8.1.2. Индексы 15](#_Toc126835524)

[8.1.3. Архитектура обработки запросов 19](#_Toc126835525)

[8.1.4. Типы физического соединения таблиц 22](#_Toc126835526)

[8.2. План выполнения запроса 24](#_Toc126835527)

[8.2.1. Предполагаемый план выполнения запроса 25](#_Toc126835528)

[8.2.2. Действительный план выполнения запроса 25](#_Toc126835529)

[8.2.3. Статистика активных запросов 25](#_Toc126835530)

[8.2.4. Включение плана выполнения 25](#_Toc126835531)

[8.2.5. Операторы плана выполнения запроса 26](#_Toc126835532)

[8.2.6. Статистика 28](#_Toc126835533)

[8.3. Статистика клиента 33](#_Toc126835534)

[8.4. Статистика по времени 34](#_Toc126835535)

[8.5. Поиск медленных SQL-запросов 35](#_Toc126835536)

[8.5.1. SQL Server Profiler 35](#_Toc126835537)

[8.5.2. XEvent 36](#_Toc126835538)

[8.5.3. DMV и DMF 37](#_Toc126835539)

[8.6. Анализ плана выполнения запроса 38](#_Toc126835540)

[8.6.1. Структура Плана запроса 38](#_Toc126835541)

[8.6.2. Свойства операторов в плане запроса 39](#_Toc126835542)

[8.6.3. Свойства окружения запроса 41](#_Toc126835543)

[8.6.4. Поиск неоптимальных операций 44](#_Toc126835544)

[8.7. Оптимизация запросов 46](#_Toc126835545)

[8.7.1. Создание статистики 46](#_Toc126835546)

[8.7.2. Обновление статистики 48](#_Toc126835547)

[8.7.3. Использование кэша планов запросов 50](#_Toc126835548)

[8.7.4. SARGArbility 52](#_Toc126835549)

[8.7.5. Соответствие типов параметров 52](#_Toc126835550)

[8.7.6. Дефрагментация индексов 54](#_Toc126835551)

[8.7.7. Общие рекомендации по оптимизации запросов 56](#_Toc126835552)

[8.7.8. Хинты 60](#_Toc126835553)

[9. Заключение 68](#_Toc126835554)

[10. Вопросы 68](#_Toc126835555)

[11. Список литературы 69](#_Toc126835556)

# Введение

Всем нужен RESTful API сервисы.

Проект в репозитории Tricolor в ветке CrmRestService

## Выбор инфраструктуры

Исходные положения:

* WCF устарел и больше не поддерживается Microsoft.
* Переход партнёров на Rest.
* Отказ от SOAP в пользу JSON.
* Возможный отказ от Microsoft CRM.
* Необходимость документирования информации о конечных точках API (Swagger).
* Поддержка клиентских сервисных контрактов.
* Логирование входящего запроса и ответа.
* На первом этапе (до отказа от Microsoft CRM) возможность взаимодействия с библиотеками Organization Service.
* Минимальные изменения в коде существующих API сервисов.
* Как можно более безболезненный переход на новые технологии после отказа от Microsoft CRM.
* Гибкая поддержка контрактов (в SOAP при добавлении параметра весь контракт становится недействительным)
* Версионность

**Проблемы:**

Набор коробочных библиотек Organization Service от Microsoft CRM не работает с платформой Net.Core с версией старше Net.Core 2.2.

Исходя из вышеперечисленного, принимается решение, что проект нового RESTful API сервиса будет создан на платформе Net.Core 2.2 с целевой реализацией на NET.Framework 6.2. Данный тип проекта обеспечит возможность создания RESTful API сервисов с одновременной поддержкой библиотеки Organization Service от Microsoft.

# Структура проекта

Проект STS.Tricolor.Xrm.Api представлен в решении Tricolor

## Каталоги

Описать каталоги НАДО

Группировка сервисов по контроллерам НАДО

## Внедрение зависимостей

Исходные положения

* Используем встроенный контейнер внедрения зависимостей .NET Core DI (Dependency Injection).
* Необходима реализация типов с помощью «сканирующих сборок» (данный функционал не реализован в NET.Core 2.0, но используется при регистрации типов шины Bus, библиотеки GMCS.Tricolor.XRM.Infrastructure.Esb)
* Соответствие времени жизни создаваемых обетов контейнеров DI Autofac и Net.Core приведены в таблице 1.0

|  |  |  |
| --- | --- | --- |
| Контейнер | | Время жизни |
| Autofac | Net.Core |
| InstancePerDependency (по умолчанию) | Transient | При каждом обращении к объекту |
| InstancePerLifetimeScope | Scoped | На время запроса |
| SingleInstance | Singleton | Хранит объект между запросами |

*Таблица №1. Соответствие времени жизни создаваемых обетов контейнеров DI Autofac и Net.Core*

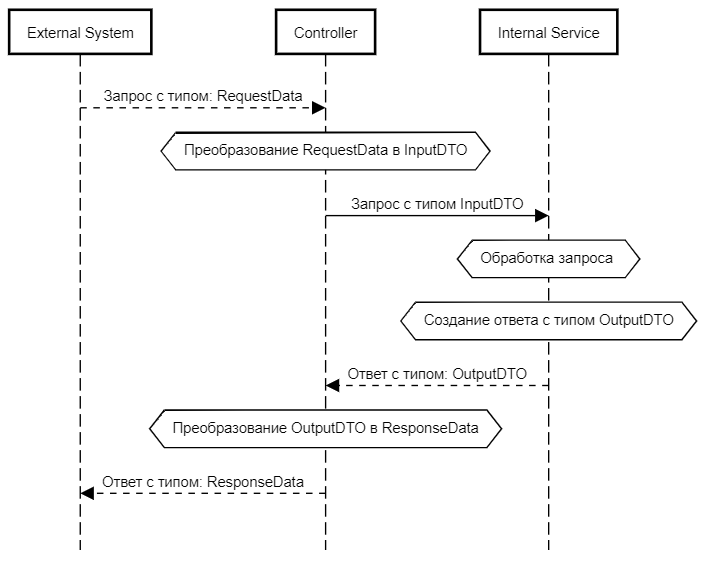
* **Проблема:** библиотека GMCS.CrmCore.IntegrationBus имеет зависимость от библиотеки Autofac 4.6.0.0
* **Решение:** Установка Nuget пакетов Autofac расширения (Autofac.Extensions.DependencyInjection" Version="8.0.0) для Net.Core, только в основном проекте Net.Core, библиотека GMCS.CrmCore.IntegrationBus имеет старую версию библиотеки Autofac 4.6.0.0 (переписывать не требуется).

## Поток данных

Поток данных при выполнении запроса к API сервису показан на рис. 1. Данные между модулями обработки передаются в 4-х различных типах контейнеров:

* RequestData
* InputDTO
* OutputDTO
* ResponseData

Данный способ передачи данных между модулями позволит отделить слой приложения от слоя бизнес-логики, что актуально при изменении типа приложения, например, переходе на Net.Core 3.0.



*Рисунок 1. Потоки данных при выполнении API запроса.*

## Зависимости сервиса API

Взаимосвязь компонентов сервиса представлена на рис. 2.

Приложение API зависит от библиотек бизнес-логики Domain.dll и Infrastructure.dll.

В библиотеке Domain.dll находятся определения типов, а в библиотеке Infrastructure.dll их реализация, данное распределение зависимостей позволяет сделать независимым уровень приложения API от уровня бизнес-логики.

**Проблема:** в настоящий момент приложение зависит от библиотеки Microsoft.Xrm.Sdk.dll, что усложнит переход на СРМ отличную от Microsoft Dynamics CRM.

**Причина:** В библиотеке Domain.dll имеется зависимость от типа Microsoft.Xrm.Sdk.Entity, являющегося составной частью библиотеки Microsoft.Xrm.Sdk.dll уровня доступа к данным. Данный класс представляет из себя не просто класс передачи данных, а также содержит в себе некоторую логику по извлечению данных из БД, поэтому отказаться от его использования при использовании библиотеки Microsoft.Xrm.Sdk.dll не представляется возможным.

На рисунке 3 показан состав библиотек ядра, используемых сервисом API в данный момент для работы с данными с использованием библиотеки СРМ Microsoft.Xrm.Sdk.dll.

На данном рисунке показано следующее:

* Работа с СРМ контекстом данных организована с помощью паттернов «Репозиторий» и «UnitOfWork».
* Интерфейсы определены в библиотеке GMCS.CrmCore.Domain.dll, а их реализации в библиотеке GMCS.CrmCore.Xrm.dll.
* Всё приложение связано через тип уровня доступа к данным Microsoft.Xrm.Sdk.Entity библиотеки Microsoft.Xrm.Sdk.dll.

**Предложения по решению**:

Содержат два варианта:

1. При дальнейшем использовании Microsoft Dynamics CRM оставляют данный функционал без изменения.
2. При переходе на работу с СРМ другого типа используют некоторую ОРМ (Object-Relational Mapping) позволяющую максимально сохранить сигнатуру клиентских методов, используемых для операций с БД. Для этих целей используемая ОРМ инкапсулируется в классах реализующими паттерны «Репозиторий» и «UnitOfWork» так же, как это выполнено в текущей реализации.

На рис. 4 показана диаграмма классов, реализующая взаимодействие с некоторой ОРМ системой с устранённой зависимостью от уровня доступа к данным. Характеристики системы:

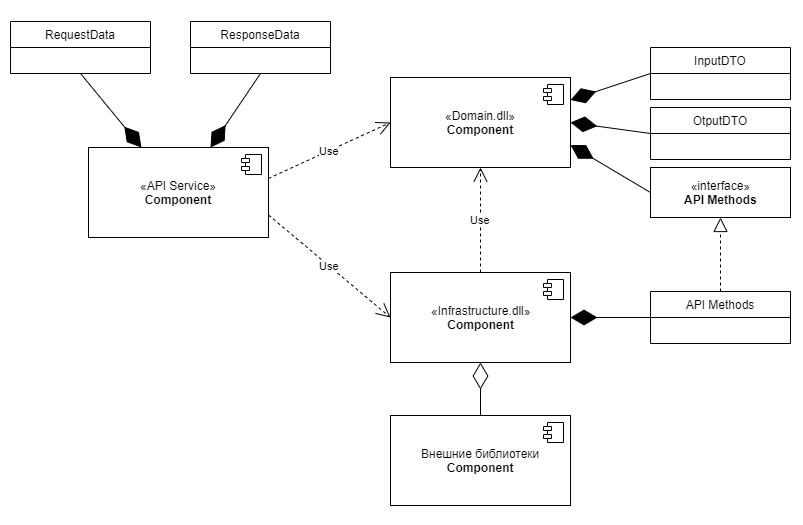
* Работа с СРМ контекстом данных организована с помощью паттернов «Репозиторий» и «UnitOfWork».
* Интерфейсы определены в библиотеке STS.CrmCore.Domain.dll, а их реализации в библиотеке STS.CrmCore.Xrm.dll.
* В библиотеке STS.CrmCore.Domain.dll определён тип Entity, являющийся базовым для типов передачи данных между слоями приложения, в данном случае с уровня бизнес-логики на уровень доступа к данным.

Современные систем ОРМ такие как:

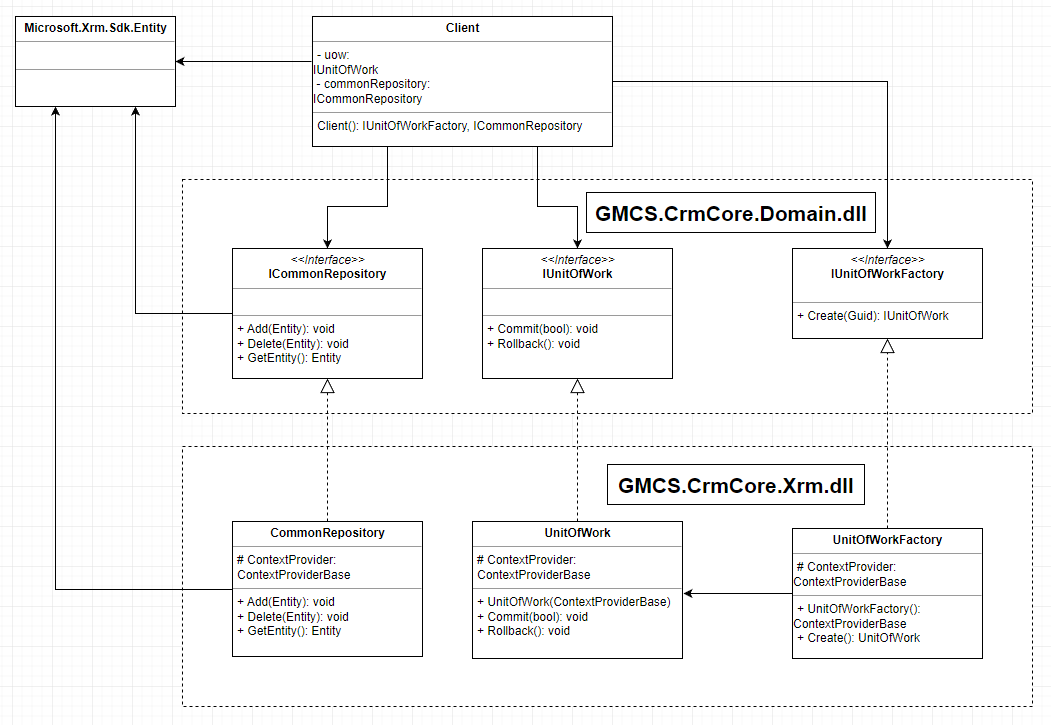
* Entity Framework
* Dapper
* NHibernate

Позволяют использовать для передачи данных между уровнями приложения те же типы, которые используются для отображения структуры БД на код приложения, их достоинство в том, что данные типы являются простыми типами передачи данных, и с высокой вероятностью могут быть использованы при изменении типа БД.

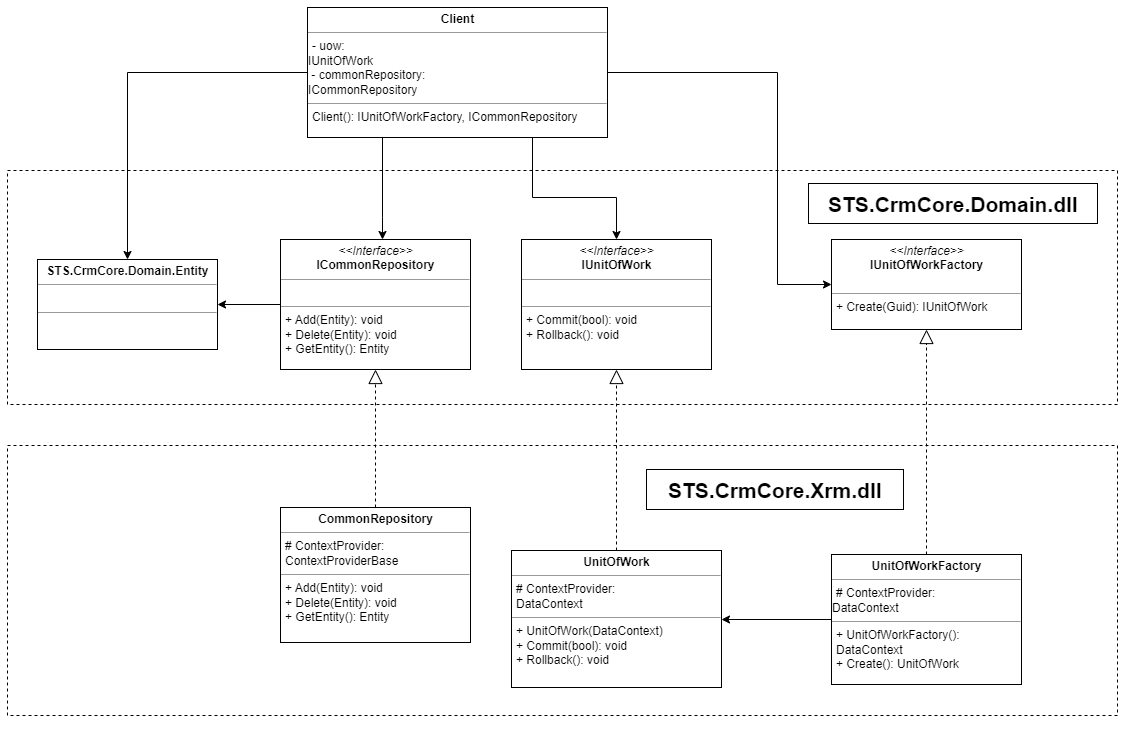
В данной работе выбор конкретной ОРМ не рассматривается.



*Рисунок 2. Взаимосвязь компонентов API сервиса.*



*Рисунок 3. Состав библиотек ядра, используемых сервисом API.*



*Рисунок 4. Предложение по изменению состава библиотек ядра.*

# Конфигурация

Конфигурация хранится в 2-х файлах:

* Web.config – настройки iis (настройки модуля AspNetCoreModuleV2 (мост между iis и Kestrel))
* App.config – строки подключения к источникам данных, необходимость использования данного файла вместо appsettings.json продиктована унаследованными библиотекой GMCS.CrmCore.Xrm.dll и особенностью Net.Core, хостирующемуся, в процессе dotnet.exe и использующему iis в качестве прокси-сервера (https://metanit.com/sharp/aspnet5/20.1.php).

Для использования строк подключения к источникам данным необходим рефакторинг библиотеки GMCS.CrmCore.Xrm.dll.

# Логирование

Анализ заголовка УБРАТЬ

Работа с перечислителями (Enum) УБРАТЬ

## Логирование хода выполнения

Создание экземпляра объекта логгера создаётся с помощью внедрения зависимости в конструкторе контроллера, настройка способа создания конкретного объекта производится в модуле настройки Autofac-контейнера. Создание и использование данного типа логгера не отличается от текущей реализации, благодаря этому логи процесса выполнения будут записываться в файл в формате таком же как и в существующем приложении.

## Логирование запроса и ответа

Логирование запроса и ответа происходит в три этапа:

1. В конвейере выполнения запроса с помощью компонента middleware «ResponseBodyEnabling» включается доступ к буферу, содержащему ответ на Http-запрос. Включение происходит путём создания временного буфера ответа и подмены существующего указателя на временный буфер.
2. В конвейере выполнения запроса с помощью компонента фильтра запроса «LogRequestCreator» перехватывается событие «OnResultExecuted», возникающее после создания ответа на запрос методом контроллера. В данном событии происходит создание содержимого лога, включающего в себя заголовки и тела запросов и ответов полученного запроса.
3. В конвейере выполнения запроса с помощью компонента middleware «ResponseBodyEnabling» содержимое временного буфера копируется в исходный буфер и возвращается указатель на исходный буфер.

Манипуляции п.п. 1 и 2 связаны с невозможностью чтения буфера ответа в Net.Core 2.0, смотри, например: https://www.thecodebuzz.com/read-request-response-netcore2-0/.

Процесс логирования запроса и ответа приведён на рисунке № 5.

Создание экземпляров логгера, их настройка и использование не отличается от соответствующего процесса в существующем функционале, благодаря этому логи методов группируются в файлах, имена которых соответствуют названиям сервисов.

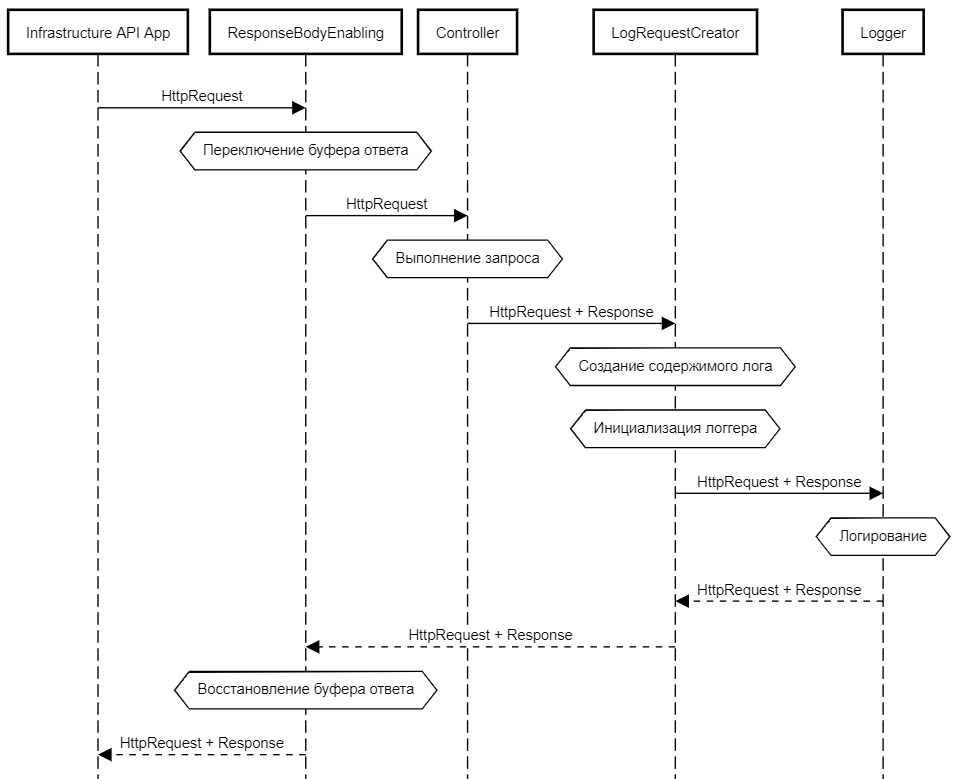


Рисунок 5. Процесс логирования запроса и ответа.

# Обработка ошибок

## Обработка исключений

никаких дополнительных обработок не происходит, при перехвате исключений их можно складывать в лог: CrmRestApi.txt

## Возврат бизнес-кодов ошибок

XrmApiException

# Документирование API

## Подключение swagger

* Устанавливают NuGet пакеты Swashbuckle.AspNetCore
* Регистрируют swagger в сервисах с помощью метода расширения AddSwaggerGen().
* Регистрируют промежуточные ПО swagger в конвейере приложения API.

Смотри, например, здесь: https://kimsereyblog.blogspot.com/2017/11/swagger-for-asp-net-core-api-development.html.

## 

# Развёртывание

**Это надо ------------------**

На площадке необходимо установить dotnet-hosting-2.2.8-win.exe

Для пула приложения необходимо установить «Без управляемого кода»

Смотри: <https://metanit.com/sharp/aspnet5/20.1.php>

настройка конфигов на разные площадки + настройка компиляции Release и Debug

компиляция в разных режимах Debug и Release

свагеру всегда нужен файл: STS.Tricolor.Xrm.Api.xml

**Конец это надо ------------------**

С помощью встроенного DI контейнера

## Инструменты анализа производительности запросов

### Инструменты поиска плохих запросов:

#### Профайлер

SQL Profiler – это программа, входящая в MS SQL Server, которая предназначена для просмотра всех событий, которые происходят в SQL-сервере. Иначе говоря, она нужна для записи трассировки.

#### Query Store

«SQL Server Query Store» – средство для поиска проблемных запросов на базе. Сохраняет данные о выполняющихся на сервере запросах, и показывает информацию о том, какие запросы дольше выполняются, сильнее грузят систему и т.п.

#### Трассировка SQL

Трассировка - это деятельность, которая происходит в фоновом режиме на машине SQL Server, и которая перехватывает определенные события и данные, связанные с этими событиями.

Преимуществом трассировки по сравнению с Profiler является, то что она основана на sql запросах, что позволяет настраивать её из пользовательского приложения.

По сути: SQL Profiler является графической надстройкой для SQL Trace.

#### XEvent (Расширенные события)

XEvent — это система мониторинга производительности, позволяющая пользователям получать данные, необходимые для отслеживания и устранения неполадок в SQL Server.

**Внимание!** Трассировка SQL и SQL Server Profiler являются устаревшими. В будущих версиях MS SQL Server эти компоненты будут удалены. Вместо них использовать Расширенные события в SQL Server и SSMS XEvent Profiler.

Преимущества XEvent по сравнению с SQL Profiler и Трассировкой

* Значительно меньшее влияние на производительность при включенном сборе данных, причем имеются расширенные настройки, с помощью которых на это можно влиять.
* Настройки сбора данных (события, фильтры и др.) можно менять на активных сессиях, прямо во время сбора данных.
* Доступен хэш запросов, чтобы идентифицировать одинаковые тексты запросов.
* Можно настроить различные способы хранения логов, причем одновременно в нескольких вариантах.
* Встроенные инструменты в SQL Server Managment Studio и инструкции TSQL для работы с ними.
* Поддержка PowerShell

**Внимание!** Из-за существенного влияния на производительность использование SQL Profiler и Трассировки в производственной среде запрещено, использовать XEvent можно.

#### Динамические административные представления и функции (DMV, DMF)

DMV, DMF – Динамические административные представления и функции (Dynamic Management Views and Functions). Возвращают данные о состоянии сервера, которые могут использоваться для контроля исправности экземпляра сервера, диагностики проблем и настройки производительности.

DMV и DMF содержат данные, которые можно использовать для диагностики проблем и оптимизации производительности баз данных.

Вот неполный перечень доступной информации:

* Причины задержек выполнения запросов.
* Работа с индексами (отсутствующие, неиспользуемые, требующие больше всех операций ввода/вывода, часто используемые).
* Запросы с высокими издержками на ввод-вывод, с высоким использованием процессора.
* Запросы, выполняющиеся чаще всего.
* Запросы, страдающие от блокировок.

### Инструменты анализа плохих запросов

#### Использование статистики клиента

При запуске скрипта или запроса в редакторе Transact-SQL можно включить сбор статистики клиента. Показатели, собранные Статистикой клиента, позволяют измерять эффективность скрипта или сравнивать разные скрипты.

#### Использование статистики по времени

Для диагностики запросов, приводящих к высокой загрузке SQL Server, можно использовать статистику по времени, которая включается и выключается в SQL Server Management Studio (SSMS) на уровне сеансов.

#### Использование анализа кода

С помощью анализа кода можно обнаруживать потенциальные проблемы в скриптах, например, ошибки проектирования, именования и проблемы с производительностью. Правила для проектов баз данных организованы в заранее заданные наборы правил, которые указывают на конкретные области, и можно включить или отключить любое правило на вкладке Анализ кода на странице свойств Свойства проекта.

#### Использование планов выполнения

План выполнения запроса – это набор конкретных действий, выполнение которых приведет SQL запрос к итоговому результату.

Посмотрев на план запроса и проанализировав его, мы можем предпринять те или иные действия для увеличения быстродействия запроса.

Иными словами, план выполнения запроса в основном нам требуется для решения проблем с производительностью наших SQL инструкций.

В SSMS имеется возможность просматривать и анализировать планы выполнения запроса.

# Основная часть

## Доступ к данным

Как при решении алгебраической задачи необходимо чёткое понимание арифметических операций. Так и при анализе производительности SQL-запроса необходимо понимать, как база данных обращается к данным в отдельных таблицах и как она соединяет данные из нескольких таблиц, для того чтобы научиться комбинировать эти операции для создания оптимального плана выполнения.

### Таблицы

Таблицы представляют собой основу реляционных баз данных. Реляционная теория описывает таблицы как абстрактные объекты, не придавая значения порядку составляющих их строк и столбцов. Однако физически таблицы существуют на дисках серверов базы данных. Типы физического размещения данных на дисках бывают различными, но в общем и целом можно сказать, что данные хранятся в неупорядоченном виде – в виде кучи, что в конечном итоге приводит к неоптимальному чтению селективных выборок данных.

Для получения доступа к конкретной записи базы данных, каждой записи присевается уникальный идентификатор – RID (Row ID).

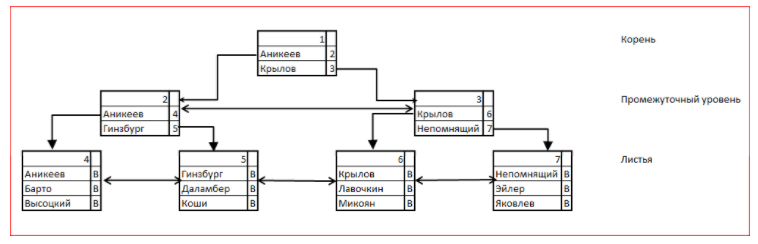
### Индексы

Для упорядоченного доступа к данным кучи используются индексы. Индекс – это вспомогательная структура, позволяющая повышать производительность запросов за счет снижения количества операций ввода-вывода, необходимых для поиска запрошенных данных. Если таблица не имеет индекса, то для выборки некоторых данных SQL-серверу придётся последовательно перебрать все записи таблицы, данная операция называется сканированием таблицы.

#### В-дерево

Индекс базы данных организован в виде структуры – B-дерева.

B-tree означает balanced tree, «сбалансированное дерево». Индекс состоит из страниц (узлов), расположенных в иерархическом порядке и связанных друг с другом. Корневой уровень содержит одну страницу, которая является точкой входа для поиска. Узлы промежуточного уровня (уровень ветвей) содержат значения ключей, ссылки на страницы следующего уровня (уровень листьев), и ссылки на соседей. При поиске по индексу находится последнее значение, которое не превосходит искомое, и происходит переход на соответствующую страницу. На последнем, листьевом уровне дерева перечислены все ключи, и для каждого из них указана ссылка (bookmark) на данные таблицы. В качестве ссылки может использоваться RID (для кучи), или кластеризованный индекс. На следующем рисунке буквы B обозначают ссылки на строки таблицы.



*Рисунок 1. Упрощенная схема организации индекса.*

#### Основной недостаток индекса

Индекс очень хорошо работает при поиске небольшого количества значений, но плохо работает при прохождении больших диапазонов. Поэтому если оптимизатор ожидает, что поиск по индексу даст достаточно большой диапазон, то вместо поиска по индексу, он выберет полное сканирование таблицы.

Для устранения данного недостатка предлагается:

* Для индекса использовать колонки, содержащие как можно больше уникальных значений
* Использовать составные индексы

#### Простой индекс

Простой индекс определяется только по одной колонке таблицы. Чтобы индекс использовался оператором SQL, ссылка на эту колонку должна быть включена в предложение WHERE данного оператора.

#### Составной индекс

Составной индекс – это индекс, определенный более чем по одной колонке. Доступ к составному индексу может осуществляться с помощью одного или нескольких индексных ключей.

Для запросов, включающих составной индекс, не требуется помещать все индексные ключи в предложение WHERE оператора SQL, но имеет смысл использовать более одного ключа. Например, если индекс создается по колонкам a, b и c какой-либо таблицы, то доступ к этому индексу можно осуществлять с помощью оператора SELECT, содержащего выражение ( a АND b АND c ), или ( a АND b ), или a. Конечно, использование более ограничивающего предложения WHERE, содержащего, например, выражение a АND b АND c, обеспечит более высокую производительность.

При использовании составного индекса запрос всегда должен содержать первый столбец индекса.

#### Кластеризованный индекс

Кластеризованный (Clustered) – это индекс, который хранит данные таблицы в отсортированном, по значению ключа индекса, виде. У таблицы может быть только один кластеризованный индекс, так как данные могут быть отсортированы только в одном порядке. По возможности каждая таблица должна иметь кластеризованный индекс, если у таблицы нет кластеризованного индекса, такая таблица называется «кучей». Кластеризованный индекс создается автоматически при создании ограничений PRIMARY KEY (первичный ключ) и UNIQUE, если до этого кластеризованный индекс для таблицы еще не был определен. В случае создания кластеризованного индекса для таблицы (кучи), в которой есть некластеризованные индексы, то после создания все их необходимо перестроить.

#### Некластеризованный индекс

Некластеризованный (Nonclustered) – это индекс, который содержит значение ключа и указатель на строку данных, содержащую остальные данные записи. У таблицы может быть несколько некластеризованных индексов. Создаваться некластеризованные индексы могут как на таблицах с кластеризованным индексом, так и без него. Именно этот тип индекса используется для повышения производительности часто используемых запросов, так как некластеризованные индексы обеспечивают быстрый поиск и доступ к данным по значениям ключа.

#### Покрывающий индекс

К недостатку некластеризованного индекса можно отнести, то, что сам индекс не содержит данных, а только ссылку на запись с данными (это может быть либо кластеризованный индекс, либо RID). Поэтому если в предложении «SELECT» присутствуют столбцы отличные от индексируемого столбца, то общее время выборки данных возрастёт, на время получения данных по ссылке. Для устранения данного недостатка можно использовать «Покрывающий индекс».

Покрывающий индекс – это индекс, содержащий все столбцы, используемые в запросе, но не все они являются индексируемыми. В терминологии MS SQL Server, данный индекс называется – «Индекс с включёнными столбцами». Благодаря использованию данного индекса оптимизатор запросов может найти все значения столбцов в индексе, при этом не обращаясь в класстеризованный индекс, или кучу за дополнительными данными, что приводит к меньшему числу дисковых операций ввода-вывода.

#### Полнотекстовой индекс

Полнотекстовый (Full-text) – это специальный тип индекса, который обеспечивает эффективную поддержку сложных операций поиска слов в символьных строковых данных. Процесс создания и обслуживания полнотекстового индекса называется «заполнением». Существует такие типы заполнения как: полное заполнение и заполнение на основе отслеживания изменений. По умолчанию SQL сервер полностью заполняет новый полнотекстовый индекс сразу после его создания, но на это может потребоваться значительный объем ресурсов, в зависимости от размеров таблицы, поэтому есть возможность откладывать полное заполнение. Заполнение на основе отслеживания изменений используется для обслуживания полнотекстового индекса после его первоначального полного заполнения;

#### Особенности индексов

* В процессе индексирования сервер базы данных создает дополнительные таблицы, в которых хранятся отсортированные данные.
* Запросы на вставку, обновление и удаление данных выполняются дольше, поскольку серверу базы данных требуется создавать новые индексы и реорганизовать существующие.
* Базы данных используют только один индекс на таблицу в запросе, поэтому при большом количестве индексов на таблице оптимизатор запроса может выбрать менее эффективный индекс.
* Для индекса, как и для колонки таблицы, существует понятие плотности распределения. Если каждое значение в индексе/колонке уникальное, то плотность будет 1/N, например, если в таблице 100 записей, то плотность будет равна 1%. Теория говорит, что чем меньше плотность, тем лучше – это увеличивает избирательность, а, следовательно, и ценность построенного индекса.
  + Например, если колонка содержит только 3 значения, плотность распределения будет равна 33.3%, что показывает бесполезность построения индекса по данному полю. Индексы занимают место на диске и в оперативной памяти и отнимают быстродействие. В идеале, самый лучший индекс имеет плотность распределения равную единице, деленной на количество записей в таблице — все записи уникальны.
  + При построении индексов, необходимо обращать внимание на плотность распределения – если она превышает 10%, то индекс можно считать бесполезным. Сканирование по таблице в таком случае будет более эффективным.
  + Для того что бы получить плотность распределения необходимо выполнить команду dbcc show\_statistics по заданной таблице и столбцу.

### Архитектура обработки запросов

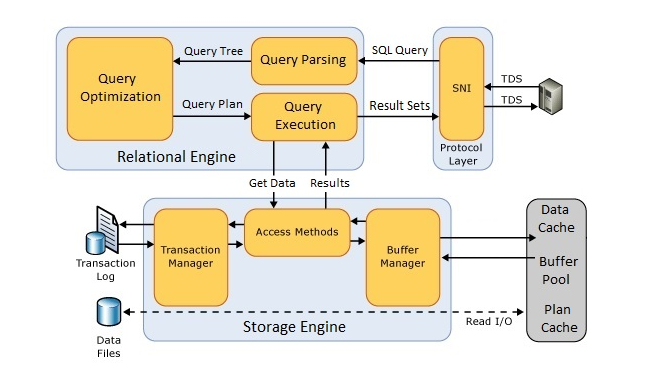
Перед тем, как вернуть результат SQL Server выполняет много различных операций.

Внутри SQL Server работает сложный механизм, работу которого обеспечивают несколько специально созданных так называемых «движков», каждый из которых отвечает за определенный участок работы.

На рисунке 2 представлена упрощенная схема обработки запроса в MS SQL Server.

За обработку SQL запросов в Microsoft SQL Server отвечает движок, который называется – Relational Engine.

На входе данный движок принимает текст SQL запроса, а на выходе отдает данные, которые были запрошены с помощью этого SQL запроса.



*Рисунок 2. Процесс обработки SQL запроса в Microsoft SQL Server.*

Relational Engine включает несколько этапов обработки SQL запроса. Можно выделить 3 основных, глобальных этапа:

* Query Parsing
* Query Optimization
* Query Execution

#### Query Parsing

Целью данного этапа является получение дерева логических операторов, либо возвращение готового плана выполнения запроса, если таковой имеется в кэше планов запроса.

#### Query Optimization

Query Optimization – это как раз тот самый, «Оптимизатор запросов», основной функцией которого является построение плана выполнения запроса.

На данном этапе будет получен оптимальный план выполнения запроса, при котором результат возвращается быстрее всего и задействовано меньше всего ресурсов.

Во время своей работы оптимизатор запроса не перебирает бесконечно все возможные планы выполнения запроса, а после нескольких попыток, отдаёт план максимально отвечающий заранее заданным критериям. Данная методика реализована для того, чтобы поиск плана выполнения запроса не занимал больше времени, чем само выполнение запроса.

Фазы оптимизации:

* Simplification
  + Удаление ненужных соединений
  + Разворачивание подзапросов в соединения (если это возможно)
  + Условия фильтрации могут быть перемещены в начало дерева запроса, чтобы отфильтровать данные как можно раньше
* Trivial Plan Optimization
  + Этап поиска тривиального плана, т.е. если запрос может быть решен единственным способом, то значит, запрос удовлетворяет условию тривиального плана и никакие правила оптимизации применять не стоит.
* Full Optimization
  + Search 0
    - На этом этапе оптимизатор ищет хороший план за минимальное время. Но данный этап может быть пропущен, и оптимизатор сразу может перейти к следующему этапу, если запрос не удовлетворяет определенным условиям.
  + Search 1
    - На данном этапе используются дополнительные правила преобразования. Если после генерации плана на этой стадии, план все еще недостаточно хорош, то данная стадия повторяется с целью поиска параллельного плана. После чего два плана сравниваются, и для оценки выбирается лучший из них. Если этот лучший план все еще не проходит внутренние пороги оптимизатора, то управление переходит к следующей фазе.
  + Search 2
    - Это самый последний этап оптимизации, на котором в любом случае будет найден тот или иной план выполнения запроса.

На этапе Query Optimization перед тем, как передать найденный, т.е. итоговый план запроса на выполнение, этот план помещается в кэш планов, для случаев, если этот же запрос в ближайшее время будет использован повторно.

#### Query Execution

Query Execution предназначен для реализации плана выполнения запроса.

Для этого Query Execution запрашивает у подсистемы хранилища (Storage Engine) данные из базовых таблиц, которые требуются для формирования результирующего набора данных.

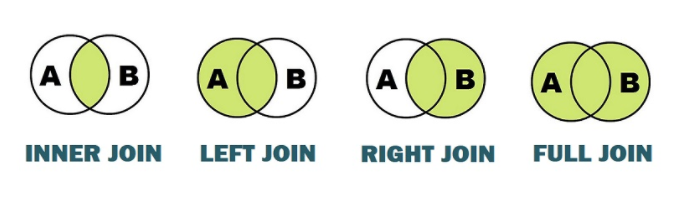
Затем он преобразует эти данные в формат результирующего набора данных и возвращает этот набор клиенту.

Все действия, связанные с блокировками, с записями в файл данных и журнал транзакций, выполняются на стороне подсистемы хранилища, т.е. в Storage Engine.

### Типы физического соединения таблиц

В языке T-SQL существуют следующие виды логического соединения таблиц (смотри рис. 3):

* INNER JOIN – внутреннее соединение
* LEFT JOIN – левое внешнее соединение
* RIGHT JOIN – правое внешнее соединение
* FULL JOIN – полное внешнее соединение
* CROSS JOIN – перекрестное соединение



*Рисунок 3. Типы логического соединения таблиц.*

На физическом уровне в Microsoft SQL Server эти соединения реализуются с помощью специальных алгоритмов:

* Nested Loops
* Merge Join
* Hash Match

На рисунке 4 показаны обозначения операторов физического соединения таблиц в плане запроса.

#### Nested Loops Join

Nested Loops Join – это оператор вложенных циклов.

Принцип работы Nested Loops следующий: SQL Server берет первое значение из первой таблицы (она называется внешней) и сравнивает его последовательно со всеми значениями во второй таблице (она называется внутренней), если находит соответствие, то запись включается в итоговый набор данных. Когда значение из первого набора данных сравнилось со всеми значениями из второго набора, то берётся второе значение первого набора и снова происходит сравнение со всеми значениями из второго набора и так происходит до тех пор, пока каждое значение из первой таблицы, т.е. внешней, не будет сравнено с каждым значением из второй таблицы, т.е. внутренней.



*Рисунок 4. Операторы физического соединения таблиц и их обозначение в плане запроса.*

Эффективность данного вида соединения повышается, если данные внутренней таблицы отсортированы по соединяемому столбцу, например, если по нему создан индекс.

Тип физического соединения таблиц Nested Loops обычно возникает, когда мы соединяем наборы данных, где один из наборов имеет небольшой размер, а другой набор данных сравнительно большой и индексирован по соединяемым столбцам. Nested Loops встречается достаточно часто, так как является самой быстрой операцией соединения на небольшом объеме данных.

**Внимание!** Если два набора данных имеют достаточно большие размеры, то данный способ соединения будет крайне неэффективен.

#### Merge Join

Merge – соединение слиянием.

Данный тип физического соединения данных является самым быстрым, и хорошо подходит для больших наборов данных, однако, он требует, чтобы оба набора данных были отсортированы, например, есть индексы по соединяемым столбцам.

Принцип работы данного типа соединения следующий: SQL Server получает первые строки из каждого набора входных данных и сравнивает их. Затем он продолжает сравнение следующих строк из второго набора, до тех пор, пока значения соответствуют значению из первого набора данных. Как только значения больше не совпадают, SQL Server переходит к следующей строке в первом наборе и продолжает выполнять сравнения.

Данный алгоритм эффективен, потому что SQL Server не должен возвращаться и читать какие-либо строки несколько раз, т.е. чтение данных происходит только один раз.

Однако алгоритм становится менее эффективен, когда в наборах существуют повторяющиеся значения, т.е. когда происходит соединение слиянием «многие ко многим».

В таких случаях SQL Server записывает любые повторяющиеся значения из второй таблицы во временную таблицу в базе данных tempdb и выполняет сравнения там.

#### Hash Match

Hash Match – хэш-соединение.

Алгоритм соединения включает 2 фазы:

* Build
* Probe

В первой фазе «Build» строится хэш-таблица при помощи вычисления хэш-значения для каждой строки одного набора данных (обычно меньшего из двух). Эти хэши вычисляются на основе ключей соединения входных данных и затем сохраняются вместе со строкой в ​​хеш-таблице.

На втором этапе SQL Server для каждой строки другого набора данных, с помощью той же хэш-функции, вычисляет хэш-значение и осуществляет поиск совпадений по хэш-таблице. Если он находит совпадение для этого хеша, то затем он проверяет, действительно ли совпадают ключи соединения между строкой в хеш-таблице и строкой из второй таблицы (ему необходимо выполнить эту проверку из-за потенциальных хеш-коллизий).

Стоит отметить, что иногда могут возникать ситуации, когда на этапе «Build» хеш-таблица не может быть сохранена полностью в памяти. В таких случаях SQL Server сохраняет некоторую часть данных в памяти, а остальную часть перенаправляет в tempdb.

Способ физического соединения данных Hash Match возникает, когда обрабатываются большие, несортированные и неиндексированные наборы данных, при этом данный оператор делает это достаточно эффективно.

## План выполнения запроса

Как уже указывалось выше, план выполнения запроса – это то, как именно будет выполняться пользовательский запрос, т.е. как именно будет осуществляться доступ к исходным данных, в каком порядке, какие конкретные методы будут использоваться для извлечения данных из каждой таблицы, какие конкретные методы будут использованы для вычислений, фильтрации, статистической обработки и сортировки данных.

Планы выполнения могут быть «Предполагаемыми» и «Действительными», кроме того можно наблюдать за выполнением плана запроса в режиме «реального времени».

### Предполагаемый план выполнения запроса

Предполагаемый – это план, созданный оптимизатором запросов на основе оценок. При создании предполагаемого плана выполнения сам запрос и в целом пакеты языка Transact-SQL не выполняются, поэтому такой план не содержит фактических метрик использования ресурсов. Вместо этого предполагаемый план отображает наиболее вероятный план выполнения запроса.

### Действительный план выполнения запроса

Действительный – это план, созданный оптимизатором запросов после фактического выполнения запроса. Иными словами, план становится доступным после выполнения SQL инструкции. Поэтому такой план отображает фактические метрики использования ресурсов.

### Статистика активных запросов

Статистика активных запросов (Live Query Statistics) – это план, который создаётся в режиме реального времени. Такой план доступен во время выполнения SQL запроса и обновляется каждую секунду. С помощью данного плана удобно обнаруживать медленно выполняющиеся операторы запроса.

### Включение плана выполнения

Планы выполнения могут быть представлены как в графическом виде, так и в виде результата выполнения SQL-запроса. Включение планов выполнения может быть произведено с помощью интерфейса SSMS и команд SQL:

* Предполагаемый план:
  + Кнопка: «Показать предполагаемый план выполнения»
  + SQL-инструкции: SET SHOWPLAN\_XML ON, SET SHOWPLAN ON, SET SHOWPLAN\_XML OFF, SET SHOWPLAN OFF
* Действительный план:
  + Кнопка: «Включить действительный план выполнения»
  + SQL-инструкции: SET STATISTICS XML ON, SET STATISTICS ON, SET STATISTICS XML OFF, SET STATISTICS OFF
* Статистика активных запросов
  + Кнопка: «Включить статистику активных запросов» (Include Live Query Statistics)

### Операторы плана выполнения запроса

Чтобы создать план выполнения запроса, который показывает, как именно будет достигнут результат выполнения SQL инструкции, оптимизатор запросов использует операторы, которые описывают конкретные действия.

Операторы плана запроса делятся на логические и физические:

**Логические операторы** – описывают операции реляционной алгебры, используемые для обработки инструкции. Другими словами, логические операторы описывают на концептуальном уровне, какие действия следует совершить;

**Физические операторы** – реализуют действия, описанные логическими операторами. Иными словами, физический оператор является действием или процедурой, которая выполняет операцию. Например, некоторые физические операторы обращаются к данным в таблицах, а другие физические операторы выполняют соединения данных, вычисления, статистическую обработку, проверку целостности и другие действия.

**Результатом плана выполнения запроса** является дерево физических операторов, которое как раз и описывает, как именно SQL Server будет выполнять SQL инструкцию, т.е. как именно будет достигнут результат этой SQL инструкции.

На рисунке № 5 приведены физические операторы, которые наиболее часто

встречаются в плане выполнения запроса.

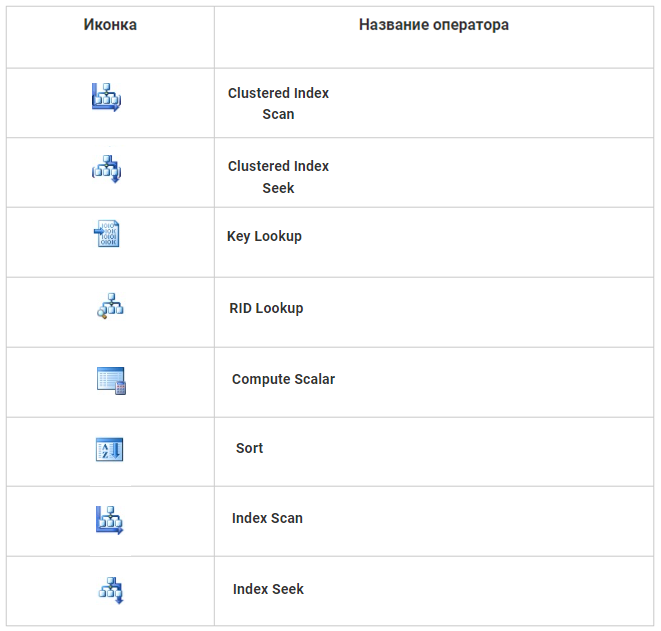
**Clustered Index Scan** – оператор сканирует (просматривает весь) кластеризованный индекс. Если существуют условия, то возвращаются только строки, удовлетворяющие условию.

**Clustered Index Seek** – этот оператор использует поисковые возможности индексов для получения строк из кластеризованного индекса, т.е. выполняет поиск в кластеризованном индексе.

Предикат поиска SEEK данного оператора содержит имя кластеризованного индекса. Подсистема хранилища использует этот индекс для обработки только тех строк, которые удовлетворяют данному предикату. Также может включаться предикат WHERE, в котором подсистема хранилища вычисляет выражение для всех строк, удовлетворяющих предикату SEEK, но это не является обязательным.

Если возвращает небольшое число строк, то считается хорошим по производительности, так как представляет собой прямой доступ SQL Server к требуемым строкам данных. Однако, если он возвращает большое число строк, то его производительность будет хуже.

**Примечание:** Предика́т — это утверждение, высказанное о субъекте. Субъектом высказывания называется то, о чём делается утверждение. Предикат в программировании — выражение, использующее одну или более величину с результатом логического типа.



*Рисунок 5. Обозначение физических операторов плана выполнения запроса.*

**Index Scan** – оператор предназначен для сканирования всех записей некластеризованного индекса. Если в аргументах оператора присутствует необязательный предикат WHERE, то возвращаются только те строки, которые удовлетворяют условию, указанному в этом предикате.

**Index Seek** – данный оператор выполняет поиск в некластеризованном индексе.

Предикат поиска SEEK данного оператора содержит имя некластеризованного индекса. Подсистема хранилища использует этот индекс для обработки только тех строк, которые удовлетворяют данному предикату. Также может включаться предикат WHERE, в котором подсистема хранилища вычисляет выражение для всех строк, удовлетворяющих предикату SEEK, но это не является обязательным.

Поиск в индексе является более эффективной операцией, чем сканирование индекса, однако если в запросе запрашивается большая часть данных индекса, то гораздо быстрее будет один раз просканировать индекс, чем осуществлять поиск каждого значения. Таким образом, Index Seek не всегда эффективнее, чем Index Scan, SQL Server сам определяет, что выбрать в том или ином случае на основе внутреннего порогового значения.

**Key Lookup** – Данный оператор выполняет поиск данных в кластеризованном индексе. Возникает он, например, тогда, когда происходит получение данных из некластеризованного индекса, однако один из столбцов, указанных в запросе, отсутствует в этом некластеризованном индексе, т.е. в данном случае SQL Server по ключу обращается в кластеризованный индекс за недостающими данными. В большинстве случаев можно выиграть в производительности, избавившись от этого оператора, например, создав покрывающий индекс.

**RID Lookup** – Этот оператор похож на Key Lookup, однако он выполняет поиск данных не в кластеризованном индексе, а в таблице «куче». Иными словами, если Вы видите данный оператор, значит у Вас есть таблица «куча», что в большинстве случаев является менее эффективным способом хранения данных, чем их хранение в кластеризованном индексе. Что бы избавится от этого оператора необходимо создать кластеризованный индекс.

**Compute Scalar** – данный оператор вычисляет выражение и выдает скалярную величину. Затем эту величину можно вернуть в качестве результата или использовать в запросе, например, в предикате фильтра или соединения.

**Sort** – оператор Sort сортирует входящие строки. Сортировка является достаточно трудоемкой операцией, поэтому лучше ее избегать, например, это можно достигнуть путем создания индекса с ключевыми столбцами, перечисленными в том же самом порядке, который использует оператор сортировки.

### Статистика

Статистика – это статистические сведения о распределении значений в одном или нескольких столбцах таблицы или индексированного представления. Статистику собирает SQL-Server по мере выполнения запросов.

Оптимизатор запросов использует статистику для оценки кардинальности – числа строк в результатах запроса. Такая оценка кардинальности позволяет оптимизатору запросов создать оптимальный план запроса. Например, в зависимости от предикатов оптимизатор запросов может использовать оценку кардинальности, чтобы выбрать оператор Index Seek вместо оператора Index Scan, который потребляет больше ресурсов, если благодаря этому повысится производительность запроса. Статистическая информация сгруппирована по разделам, которые представлены ниже.

#### Разделы статистики

Статистика включает 3 раздела.

**Заголовок**

Содержит общую информацию о статистике

**Плотность**

Плотность – это показатель, который отражает, сколько уникальных значений содержится в столбце или наборе столбцов.

Иными словами,

Плотность = 1 / Число различных значений столбца (столбцов)

Оптимизатор запросов использует плотности для улучшения оценки размерности для запросов, которые возвращают данные нескольких столбцов из одной таблицы или индексированного представления. По мере повышения плотности избирательность значения повышается.

Плотность содержит расширенную информацию о статистике, называемую «Вектор плотностей».

**Гистограмма**

Гистограмма – это распределение данных в первом ключевом столбце объекта статистики.

Она измеряет частоту появления каждого различающегося значения в наборе данных. Оптимизатор запросов вычисляет гистограмму для значений столбца в первом ключевом столбце объекта статистики, выбирая значения столбцов путем статистической выборки строк или полного просмотра всех строк в таблице или представлении. Если гистограмма создается на основе выбранного набора строк, то сохраняемые итоговые значения количества строк и количества различающихся значений являются приблизительными и не всегда выражаются целыми числами.

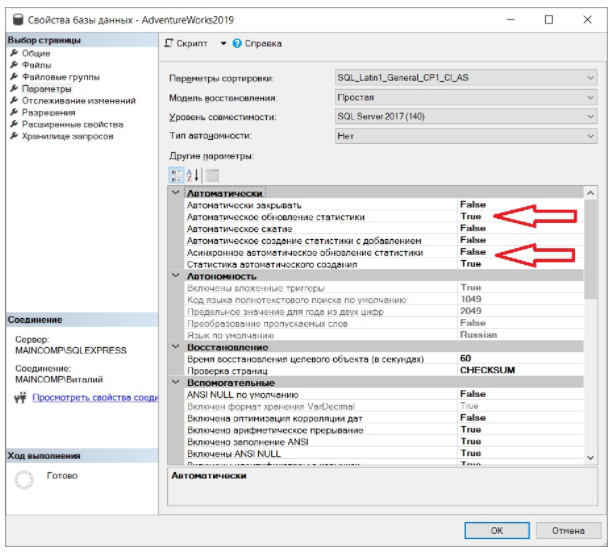
#### Информация о статистике

Информацию о статистике можно получить, например, с помощью команды DBCC SHOW\_STATISTICS. В качестве параметра необходимо передать имя таблицы или индексированного представления, а также вторым параметром — имя индекса, статистики или столбца, для которого отображаются статистические данные.

#### Параметры статистики

В Microsoft SQL Server предусмотрено три параметра, которые влияют на условия и методы создания и обновления статистики. Эти параметры задаются только на уровне базы данных.

На рисунке № 6 показано окно SQL Server Management Studio в котором можно изменить параметры статистики, также эти параметры можно изменить с помощью команды ALTER DATABASE SET.



*Рисунок 6. Изменение параметров статистики.*

* Параметр AUTO\_CREATE\_STATISTICS

Данный параметр отвечает за автоматическое создание статистики, и если параметр включен, а он включен по умолчанию, то оптимизатор запросов при необходимости создает статистику по отдельным столбцам в предикате запроса, чтобы улучшить оценку кардинальности для плана запроса.

Параметр AUTO\_CREATE\_STATISTICS не определяет, создается ли статистика для индексов, для индексов статистика создаётся всегда.

* Параметр AUTO\_UPDATE\_STATISTICS

Данный параметр отвечает за автоматическое обновление статистики. Если параметр включен, а он по умолчанию включен, оптимизатор запросов определяет, устарела ли статистика, и при необходимости автоматически обновляет ее, если она используется в запросе.

Это действие также называется повторной компиляцией статистики. Статистика становится устаревшей после изменений данных в результате операций вставки, обновления, удаления или слияния.

Оптимизатор запросов определяет, когда статистика может оказаться устаревшей, подсчитывая операции модификации строк с момента последнего обновления статистики и сравнивая количество изменений строк с пороговым значением. Пороговое значение зависит от кардинальности таблицы, что можно определить, как количество строк в таблице или индексированном представлении.

В общей практике принято считать, что пороговое значение, при котором SQL Server будет считать статистику устаревшей и обновит ее, это примерно 20% от количества строк в таблице.

Оптимизатор запросов проверяет наличие устаревшей статистики перед компиляцией запроса и до выполнения кэшированного плана запроса. Перед компиляцией запроса оптимизатор запросов с помощью столбцов, таблиц и индексированных представлений в предикате запроса определяет статистические данные, которые могли устареть. Перед выполнением кэшированного плана запроса компонент Database Engine проверяет, ссылается ли план запроса на актуальную статистику.

* Параметр AUTO\_UPDATE\_STATISTICS\_ASYNC

Обновление статистики может выполняться синхронно или асинхронно. Данный параметр определяет, какой режим обновления статистики использует оптимизатор запросов: синхронный или асинхронный. По умолчанию параметр асинхронного обновления статистики отключен, соответственно, оптимизатор запросов обновляет статистику синхронно.

При синхронном обновлении статистики запросы всегда компилируются и выполняются с актуальной статистикой. Когда статистика устарела, оптимизатор запросов ожидает обновленную статистику перед компиляцией и выполнением запроса.

При асинхронном обновлении статистики запросы компилируются с существующей статистикой, даже если она устарела. Если статистика при компиляции запроса является устаревшей, оптимизатор запросов может выбрать неоптимальный план запроса. Как правило, статистика обновляется вскоре после этого. Запросы, которые компилируются после обновления статистики, будут использовать обновленную статистику.

В общих случаях рекомендуется использовать синхронную статистику, так как это позволяет гарантировать актуальность статистики.

Асинхронная статистика рекомендуется для достижения более прогнозируемого времени ответа на запросы, иными словами, этот параметр надо включать только в том случае, когда ваши запросы ждут синхронного обновления статистики слишком часто, и это вызывает проблемы производительности, например, в следующих случаях:

* + - Приложение часто выполняет один и тот же запрос, схожие запросы или схожие кэшированные планы запроса. Асинхронное обновление статистики может обеспечить более прогнозируемое время ответа на запрос по сравнению с синхронным, так как оптимизатор запросов может выполнять входящие запросы, не ожидая появления актуальной статистики. Это устраняет задержку в некоторых запросах, при этом не влияет на другие запросы.
    - Если в приложении установлено небольшое время ожидания ответа от сервера баз данных, то возможны случаи, когда ожидание синхронной статистики может вызвать ошибку таймаута.

Обновление асинхронной статистики выполняется с помощью фонового процесса. Когда запрос готов к записи обновленной статистики в базу данных, он пытается получить блокировку изменения схемы для объекта метаданных статистики. Если другой сеанс уже удерживает блокировку того же объекта, обновление асинхронной статистики блокируется до тех пор, пока не появится возможность получить блокировку изменения схемы. Таким образом, для систем с очень частыми компиляциями запросов и частыми обновлениями статистики использование асинхронной статистики может повысить вероятность проблем с блокировками.

#### Отфильтрованная статистика

Кроме обычной статистки, которая создается по полной таблице, существует также отфильтрованная статистика.

Отфильтрованная статистика – это статистика, созданная для определенного подмножества данных.

Отфильтрованная статистика может улучшить план выполнения запроса по сравнению со статистикой по полной таблице и тем самым повысить производительность запросов, которые выполняют выборку из четко определенных подмножеств данных, так как отфильтрованная статистика использует предикат фильтра для выбора подмножества данных, включенных в статистику.

## Статистика клиента

Как указывалось, выше, при запуске скрипта или запроса в редакторе Transact-SQL можно включить сбор статистики клиента. Статистика клиента может быть включена для отображения в окне «Статистика клиента», для этого необходимо выполнить команды графического редактора SSMS:

Включение: Запрос –> Включить статистику клиента.

После выполнения запроса в окне «Статистика клиента» можно будет увидеть следующие статистики:

* Количество операторов INSERT, DELETE или UPDATE, которые были выполнены в результате запроса.
* Число строк, на которые повлияли операторы INSERT, DELETE или UPDATE, которые были выполнены как часть вашего запроса.
* Количество операторов SELECT, которые были выполнены через соединение как часть выполнения вашего запроса. В это число входят операторы FETCH для извлечения строк из курсоров.
* Число строк, выбранных при выполнении запроса. Это число отражает все строки, сгенерированные операторами Transact-SQL, даже те, которые фактически не использовались вызывающей стороной (если, например, вы отменяете выполнение). Это число также включает операторы FETCH для извлечения строк из курсоров.
* Количество пользовательских транзакций, запущенных как часть выполнения вашего запроса, включая откаты.
* Количество раз, когда соединение отправляло команды серверу и получало ответ как часть выполнения запроса.
* Количество байтов, отправленных клиентом экземпляру SQL Server во время выполнения запроса.
* Число байтов, полученных клиентом от экземпляра SQL Server во время выполнения запроса.
* Совокупное время, затраченное клиентом на выполнение кода во время выполнения запроса.
* Совокупное количество времени (в миллисекундах), которое клиент потратил на обработку во время выполнения запроса, включая время, которое клиент потратил на ожидание ответов от сервера, а также время, потраченное на выполнение кода.
* Время (в миллисекундах), которое клиент провел, ожидая ответа от сервера.

## Статистика по времени

Отображение статистики по времени включается и выключается в SQL Server Management Studio (SSMS) на уровне сеансов с помощью следующей команды:

* set statistics io on
* set statistics time on

При этом в окне «Сообщения» можно увидеть следующие статистики:

* Число просмотров (Количество операций поиска или сканирования, с целью получения всех значений при построении окончательного набора данных для вывода).
* Число просмотров равно 0, если используется уникальный или кластеризованный индекс первичного ключа и происходит поиск только одного значения. Например, WHERE Primary\_Key\_Column = <value>.
* Число просмотров равно 1, если при поиске одного значения используется неуникальный кластеризованный индекс, который определен в столбце непервичного ключа. Это позволяет проводить проверку на наличие повторяющихся значений ключа, который вы ищете. Например, WHERE Clustered\_Index\_Key\_Column = <value>.
* Число просмотров равно N, где N — количество различных операций поиска или сканирования, начатых по левую или правую сторону от конечного уровня после обнаружения значения ключа с помощью ключа индекса.
* Логические операции чтения
* Число страниц, считанных из кэша данных.
* Физические операции чтения
* Число страниц, считанных с диска.
* Операции упреждающего чтения
* Число страниц, помещенных в кэш для запроса.

Постоянный вывод статистики в окне «Сообщения», во время выполнения запроса, можно включить можно включить в окне настроек запроса, с помощью:

ПКМ –> Параметры запроса –> Дополнительно

Для этого вида статистики важным показателем является количество операций логического чтения (*logical reads*). На другие виды чтения воздействуют внешние факторы, на которые разработчик повлиять не может.

При настройке запросов SQL, цель должна состоять в том, чтобы получить число логических операций чтения как можно более низкой. Чем меньше логических операций чтения, тем меньше время выполнения запроса.

## Поиск медленных SQL-запросов

### SQL Server Profiler

Для поиска долго выполняющихся запросов достаточно использовать два события: RPC:Completed, SQL:BatchCompleted, эти события фиксируют все внешние sql-вызовы к серверу. Они возникают, как видно из названия (Completed), после окончания обработки запроса.

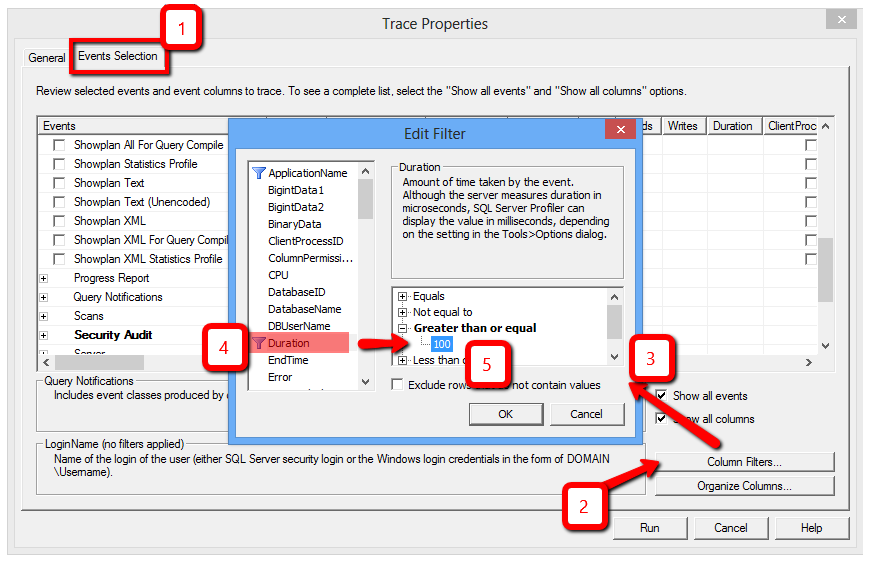
#### Запуск

В среде SSMS выполнить: Инструменты --> SQL Server Profiler

#### Настройка поиска медленных запросов

На рисунке № 7 показан порядок настройки SQL Server Profiler для фильтрации медленных запросов. Порядок настройки, следующий:

1. Создать новый файл трассировки и перейдите в раздел «Выбор событий» на вкладке «Свойства трассировки».
2. Нажать кнопку «Пользовательские фильтры».
3. Подождать, пока не появится окно редактирования фильтра.
4. Выбрать «Продолжительность» из списка слева.
5. Настроить необходимые условия.



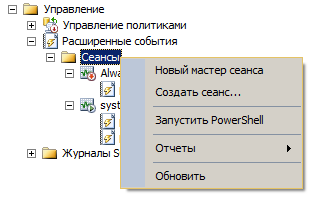
*Рисунок 7. Настройка SQL Server Profiler для поиска медленных запросов.*

### XEvent

На рисунке № 8 показано окно, в котором можно перейти к настройке расширенных событий, добавить новый сеанс события можно несколькими способами:

* с помощью мастера,
* с помощью полной настройки вручную с нуля
* с помощью скрипта T-SQL.

Для поиска медленных запросов достаточно отслеживать два события: rpc\_completed, sql\_batch\_completed. Фильтр можно настраивать по полю: task\_execution\_time.



*Рисунок 8. Начальное окно настроек сеанса расширенных событий.*

### DMV и DMF

Существует два динамических административных представления (DMV - Dynamic Management Views), полезных при оценке показателей процессора и продолжительности запросов SQL Server: sys.dm\_exec\_query\_stats и sys. dm\_exec\_procedure\_stats.

Первое представление показывает статистику выполнения запросов, какой запрос сколько раз выполнялся и сколько ресурсов на это потрачено.

Второе — это представление о выполненных хранимых процедурах.

В остальном эти представления очень похожи по структуре и результатам. Каждое из них возвращает информацию о выполнении с минимальным/максимальным/средним и последним значениями всех показателей, такими как:

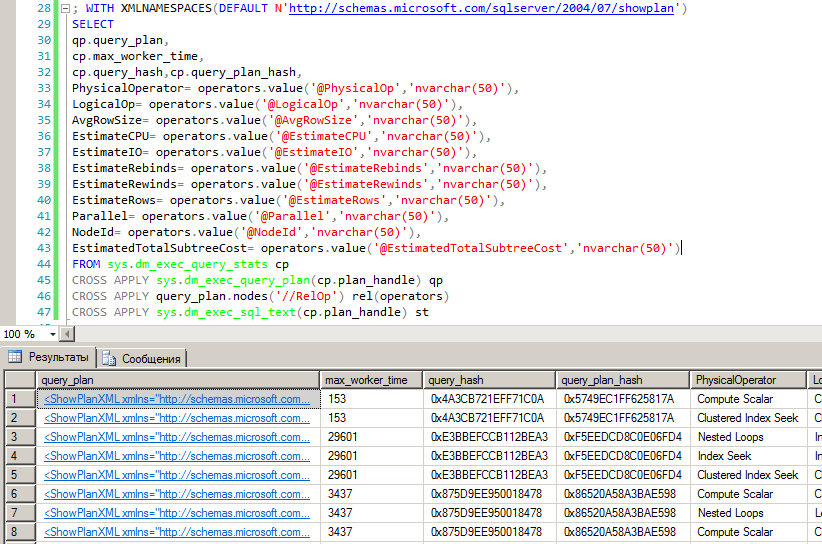
количество циклов ввода-вывода,

загрузки процессора

времен выполнения.

На рисунке № 9 приведён пример запроса, и его результаты, к системному представлению статистики выполнения запросов, представляющий не только запросы и их планы выполнения, но и операторы планов с их статистикой.

Для получения медленных запросов можно проанализировать, например, поле max\_worker\_time системного представления sys.dm\_exec\_query\_stats.

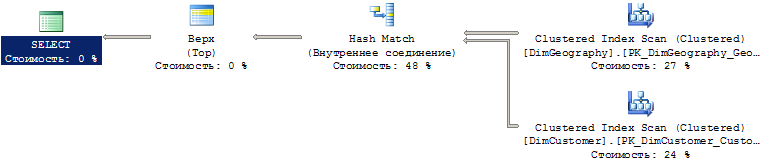


*Рисунок 9. Пример запроса к системному представлению sys.dm\_exec\_query\_stats.*

## Анализ плана выполнения запроса

### Структура Плана запроса

На рисунке № 10 показан графический план выполнения запроса



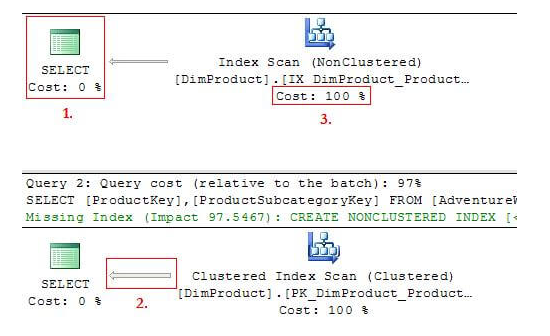
*Рисунок 10. Графический план выполнения запроса.*

Как уже указывалось, план выполнения состоит из разных операторов, где каждый оператор имеет один выход, который называется набором результатов. Операторы могут иметь один или несколько входов, например, операторы соединения с двумя входами. Каждый набор результатов (выход) будет входом для следующей операции, пока не завершится выполнение запроса SQL. Поэтому поток данных можно представить, как связь между операторами справа налево.

Каждый оператор в плане запроса представляет физическую операцию, созданную анализатором запросов и выполненную ядром базы данных. Операторы соединены линиями, показывающими поток данных. Их следует читать справа налево, чтобы стрелки указывали налево. Более толстые линии между операциями указывают на то, что между операторами передается больше строк.

На рисунке № 11 показаны следующие элементы:

1. Оператор в плане запроса.
2. Относительно более толстая линия, представляющая основной поток данных.
3. Процент ресурсов, используемых для выполнения оператора.



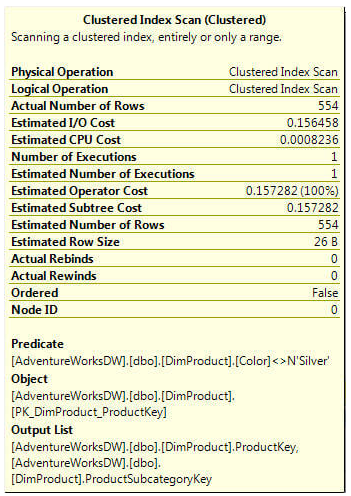
*Рисунок 11. Элементы плана выполнения запроса.*

### Свойства операторов в плане запроса

Для того, чтобы увидеть свойства элемента плана запроса необходимо навести, или кликнуть мышкой на него.

На рисунке № 12 показаны свойства оператора сканирования кластерного индекса.

* Physical Operation (физическая операция): выполненная физическая операция.
* Logical Operation (логическая операция): логическая операция, выполняемая физической операцией.
* Actual Number of Rows (фактическое количество строк): фактическое количество строк, возвращаемых оператором.
* Estimated I/O Cost (приблизительная стоимость ввода-вывода): это значение показывает использование ввода-вывода. Это относительное значение, поэтому по нему можно только понять насколько затратная данная операция по сравнению с другими. Чем больше число, тем больше ресурсов используется.
* Estimated CPU Cost (приблизительная стоимость ЦП): это значение показывает использование ЦП. Это также относительное значение. То есть показывает величину относительно стоимости всего плана.
* Number of Executions (количество выполнений): сколько раз операция была выполнена во время выполнения запроса.
* Estimated Number of Executions (предполагаемое количество выполнений): величина рассчитанная планировщиком при построении плана запроса.



*Рисунок 12. Свойства оператора сканирования кластерного индекса.*

* Estimated Operator Cost (ориентировочная стоимость оператора): именно она показана под значком оператора в плане запроса. Это общая сумма предполагаемых затрат на ввод-вывод и ЦП, поэтому это также относительное значение.
* Estimated Subtree Cost (ориентировочная стоимость поддерева): это общая стоимость этого оператора и всех предшествующих операторов.
* Estimated Number of Rows (предполагаемое количество строк): приблизительное количество строк, возвращаемых запросом. Эта значение рассчитывается планировщиком при построении плана выполнения основано на статистике.
* Estimated Row Size (Предполагаемый размер строки): предполагаемый размер строки, возвращаемых запросом. Это значение также расчётное и основано на статистике.
* Actual Rebinds (фактические повторные привязки): это значение показывает, сколько раз был вызван метод Init() для всего оператора, или другими словами оператор сброшен на начало (запуск с начала всего цикла поиска).
* Actual Rewinds (фактическая перемотка назад): это значение показывает, сколько раз был вызван метод Init() для внутреннего цикла.
* Ordered (отсортировано): это логическое значение, показывающее, упорядочены ли строки в операции.
* NodeID: каждый оператор в плане выполнения имеет уникальный NodeID. Это порядковый номер. Если он равен нулю, то фактическое количество строк и расчетное количество строк сопоставимы.
* Cached plan size (размер занимаемый планом в кэше): это объем памяти, зарезервированной для этого плана запроса в кэше процедур. Это свойство отображается только у оператора Result.

### Свойства окружения запроса

В зависимости от типа оператора в нём может содержаться дополнительная информация о свойствах среды, в которой выполняется запрос. В таблице № 1 приведены сведения о дополнительной информации, которую можно найти в свойствах операторов запроса в графическом плане выполнения запроса.

В частности, из свойств операторов запроса можно почерпнуть следующую информацию:

* Версия сервера

Если открыть план как XML, в самом первом элементе, который описывает план, можно увидеть версию сервера в свойстве Build

* Число строк в таблице

Данная информация содержится в операторах доступа к данным (Scan или Seek) в свойстве TableCardinality.

* Контекст

План запроса сохраняет значимые настройки окружения, для которых был построен для того, чтобы их посмотреть, нужно выбрать в плане корневой элемент и открыть свойство Set Options.

* Количество CPU

Можно узнать число процессоров, доступных оптимизатору, для этого, нужно открыть свойство OptimizerHardwareDependentProperties -> Esti-matedAvailableDegreeOfParallelism и умножить его на 2. Исключение, если доступен всего один процессор, умножать на 2 не нужно.

* Версия Cardinality Estimator

Начиная с SQL Server 2014 доступны несколько версий механизма оценки числа строк (Cardinality Estimator) оптимизатором, от этого механизма зависит стоимость операторов и большинство решений оптимизатора при выборе плана. Версию Cardinality Estimator можно посмотреть в свойстве корневого оператора CardinalityEstimationModelVersion.

* Время выполнения запроса и ожидания

Начиная с SQL Server 2016 SP1, в действительном плане запроса доступна информация о времени выполнения и времени процессора, для этого в корневом элементе нужно раскрыть свойство QueryTimeStats и посмотреть значения CpuTime и ElapsedTime.

* Сведения о включенных флагах трассировки

Флаги трассировки в SQL Server являются своеобразными «переключателями» поведения сервера с поведения по умолчанию, на какое-либо другое, отличающееся. Начиная с SQL Server 2014 SP2 и 2016 SP1, информация о включенных флагах трассировки доступна в свойстве TraceFlags корневого элемента, она может включать до 100 одновременно включенных флагов в момент построения плана.

* Информация о сливе данных в tempdb

Некоторые операторы плана, например, такие как Sort или Hash Match, требуют память во время выполнения запроса, однако, количество памяти рассчитывается в момент компиляции запроса. В силу разных причин (например, неверной оценки числа предполагаемых строк), количество памяти может быть посчитано неверно. Если памяти было выделено меньше, чем нужно для выполнения, сервер вынужден сливать данные в tempdb, что замедляет выполнение запроса. Предупреждение о такой ситуации появилось в 2012 сервере, но начиная с SQL Server 2012 SP3, 2014 SP2, 2016 диагностическая информация была расширена и теперь включает в себя объем слитых и прочитанных данных, так что можно оценить степень бедствия и принять адекватные меры.

|  |  |
| --- | --- |
|  | Версия сервера |
|  | Число строк в таблице (Table Cardinality) |
|  | Контекст |
|  | Количество CPU |
|  | Версия Cardinality Estimator |
|  | Время выполнения запроса и ожидания |
|  | Флаги трассировки |
|  | Информация о сливе данных в tempdb |

*Таблица 1. Дополнительная информация о плане запроса.*

### Поиск неоптимальных операций

При анализе плана запроса для его оптимизации следует искать наиболее ресурсоемкие и неоптимальные операции. Следующие артефакты, обнаруженные на плане, требуют более глубокого изучения:

* Estimated Operator Cost (ориентировочная стоимость оператора). Высокое значение этого свойства может иногда показать, где находится главная проблема плана выполнения. Иногда – поскольку, стоимость плана выполнения не всегда точна. Операторы с высокой стоимостью могут стать хорошей отправной точкой, для анализа всего плана.
* Жирные линии – большое количество строк, возвращаемое оператором, особенно стоит обращать внимание на большое расхождение расчётного (предполагаемого) и фактического количества строк, возвращаемого оператором.
* Scan – сканирование (полный просмотр) индекса при малом количестве возвращаемых строк.
* Seek – поиск в индексе при большом количестве возвращаемых строк.
* Поиск по ключу кластерного индекса – такая ситуация случается, когда в некластеризованном индексе недостаточно колонок. Решением может быть создание покрывающего индекса.
* Количество выполнений операции – большое количество выполнений операции при малом количестве возвращаемых ей строк.
* Операция сортировки – это очень ресурсоёмкая операция, а учитывая, то, что данные в индексах хранятся в отсортированном виде необходимо по возможности строить запросы так, чтобы избежать этой операции.
* Предупреждения, отмеченные иконкой с желтым восклицательным знаком в углу иконки оператора. Данный значок сигнализирует о проблемах с оператором, это могут быть:
  + операторы, сбрасывающие данные в tempdb
  + неявные преобразования, которые SQL Server должен производить для выполнения сравнения (что потенциально препятствует использованию индекса)
  + превышение из-за недооценки сервером, объёма памяти необходимого для выполнения запроса.
* Nested Loops – это оператор вложенных циклов с двумя входами если на оба его входа поступает большое количество данных, то работа оператора становится крайне неэффективной.
* Рекомендации по индексам – если SQL Server считает, что производительность запроса можно улучшить с помощью индекса, он сообщит об этом, поместив соответствующий текст зеленого цвета вверху плана выполнения. Нужно учитывать, что в данной рекомендации SQL Server может предлагать не все колонки необходимые для оптимального индекса.

|  |  |
| --- | --- |
|  | Estimated Operator Cost (ориентировочная стоимость оператора) |
|  | Предупреждения, отмеченные иконкой с желтым восклицательным знаком в углу иконки оператора. |
|  | Seek – поиск в индексе при большом количестве возвращаемых строк |
|  | Scan – сканирование (полный просмотр) индекса при малом количестве возвращаемых строк. |
|  | Поиск по ключу кластерного индекса. |
|  | Операция сортировки – это очень ресурсоёмкая операция |
|  | Большие наборы данных на обоих входах оператора вложенных циклов Nested Loops |
|  | Рекомендации по индексам |

*Таблица 2. Поиск неоптимальных операторов плана запроса.*

## Оптимизация запросов

### Создание статистики

Поскольку оптимизатор запросов использует статистику для оценки кардинальности – числа строк в результатах запроса, то отсутствие статистики может привести к тому, что оптимизатор построит не оптимальный план запроса.

И хотя создание статистики для индексов и столбцов задано по умолчанию (если включен параметр AUTO\_CREATE\_STATISTICS) бывают случаи, когда план выполнения запроса можно улучшить, создав дополнительную статистику. Такая дополнительная статистика может фиксировать статистическую корреляцию, которую не учитывает оптимизатор запросов при создании статистики для индексов или отдельных столбцов. Например, план запроса можно улучшить путем использования отфильтрованной статистики по подмножеству строк данных или статистики по нескольким столбцам предиката запроса.

Создавать статистику вручную можно в следующих случаях:

* Предикат запроса содержит несколько коррелирующих столбцов, которые еще не включены в один индекс;
* Запрос выполняет выборку из подмножества данных;
* Для запроса отсутствует статистика;
* Помощник по настройке ядра СУБД рекомендует создание статистики.

#### Предикат запроса содержит несколько коррелирующих столбцов

Если предикат запроса содержит несколько столбцов, между которыми есть связи и зависимости, то статистика по нескольким столбцам может улучшить план запроса. Статистика по нескольким столбцам содержит статистику корреляции между столбцами, называемую плотностью, которая недоступна в статистике по отдельным столбцам. Плотность может повысить точность оценки количества элементов, если результаты запроса зависят от связей между данными из нескольких столбцов.

Если столбцы уже принадлежат одному индексу, то объект статистики по нескольким столбцам уже существует и его не нужно создавать вручную. В противном случае можно вручную создать статистику инструкцией CREATE STATISTICS. Стоит отметить, что на поддержание индекса расходуется больше системных ресурсов по сравнению с объектом статистики. Если приложению не нужен индекс по нескольким столбцам, можно сэкономить системные ресурсы, создав объект статистики и не создавая индекс.

#### Запрос выполняет выборку из подмножества данных

Когда оптимизатор запросов создает статистику по отдельным столбцам и индексам, она создается по значениям во всех строках. Если запросы выполняют выборку из подмножества строк и в этом подмножестве присутствует уникальное распределение данных, то отфильтрованная статистика может улучшить планы запросов. Отфильтрованную статистику можно создать с помощью инструкции CREATE STATISTICS с предложением WHERE, в котором определить выражение предиката фильтра.

#### Для запроса отсутствует статистика

Если в результате ошибки или другого события оптимизатору запросов не удается создать статистику, он формирует план запроса, не используя статистику. SQL Server помечает статистику как отсутствующую и пытается восстановить ее перед следующим выполнением запроса.

Потерянная статистика отображается в виде предупреждения на графическом отображении плана выполнения запроса в среде SQL Server Management Studio.

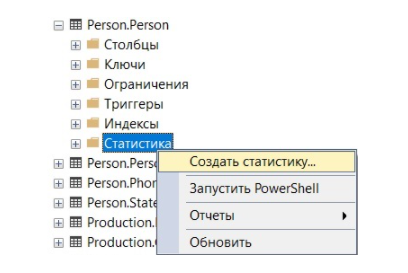
В данном случае необходимо:

* Проверить, что включены параметры AUTO\_CREATE\_STATISTICS и AUTO\_UPDATE\_STATISTICS;
* Проверить, что база данных доступна на запись, т.е. она не в состоянии «Только для чтения». Дело в том, что если база данных доступна только для чтения, новый объект статистики сохранить нельзя;
* Если предыдущие пункты выполнены, но статистика все равно не создается, можно создать статистику вручную с помощью инструкции CREATE STATISTICS.

#### Команды создания статистики

Создание и удаление статистики может быть произведено с помощью команд:

* Команды T-SQL: CREATE STATISTICS и DROP STATISTICS
* sp\_createstats – системная хранимая процедура, которая вызывает инструкцию CREATE STATISTICS для создания статистики по отдельным столбцам для столбцов, которые еще не являются первым столбцом в объекте статистики.
* Из графического интерфейса SSMS – на рисунке № 13 показан пример создания статистики, для этого необходимо в обозревателе объектов щелкнуть правой кнопкой мыши по контейнеру «Базы данных -> Таблицы -> Имя таблицы -> Статистика» и выбрать «Создать статистику».



*Рисунок 13. Создание статистики для таблицы Person.*

### Обновление статистики

Использование актуальной статистики является залогом того, что оптимизатор построит правильный план запроса. Триггером, вызывающим автоматическое обновление статистики, служит количество изменений данных в результате операций вставки, обновления, удаления или слияния.

Но дело в том, что достижение порогового значения по количеству изменений, не всегда происходит вовремя.

В общей практике принято считать, что пороговое значение, при котором SQL Server будет считать статистику устаревшей и обновит ее, это примерно 20% от количества строк в таблице. Например, если в таблице 100 000 строк, SQL Server обновит статистику если изменится 20 000 строк. Таким образом, может возникнуть ситуация, что Вы разово изменили достаточно большое количество данных, но это значение меньше порогового, и статистика, соответственно, автоматически не обновляется, и как следствие оптимизатор запросов может строить менее эффективные планы, по сравнению с планами, которые он мог бы построить, основываясь на актуальной статистике.

А если говорить про большие таблицы, то здесь даже до 2016 версии существовала серьезная проблема, связанная с тем, что у больших таблиц статистика обновлялась крайне редко, а то и вовсе не обновлялась, так как, например, если в таблице было несколько миллиардов записей, то по этому алгоритму статистика должна обновится, когда изменятся сотни миллионов строк, что, будет случаться крайне редко. После 2016 алгоритм поиска порогового значения был усовершенствован.

Тем не менее существуют рекомендации, когда необходимо обновить статистику:

* Запросы выполняются медленно
* После операций обслуживания
* После операций вставки в ключевые столбцы

#### Запросы выполняются медленно

Запросы выполняются медленно, т.е. если время ответа на запросы велико или непрогнозируемое, то следует убедиться, что для запросов есть актуальная статистика, и только после этого продолжить диагностику;

#### После операций обслуживания

После операций обслуживания, которые изменяют распределение данных, таких как усечение таблицы или массовая вставка большого количества строк (в процентном отношении). Это может предотвратить задержки в обработке запросов, вызванные ожиданием автоматического обновления статистики. Такие операции, как перестроение и реорганизация индекса, не изменяют распределение данных, и поэтому после выполнения таких операций не нужно обновлять статистику. Но обязательно стоит отметить, что перестроение индекса вызывает обновление статистики, это является побочным эффектом повторного создания индекса. При этом в случае реорганизации индекса, статистика не обновляется.

#### После операций вставки в ключевые столбцы

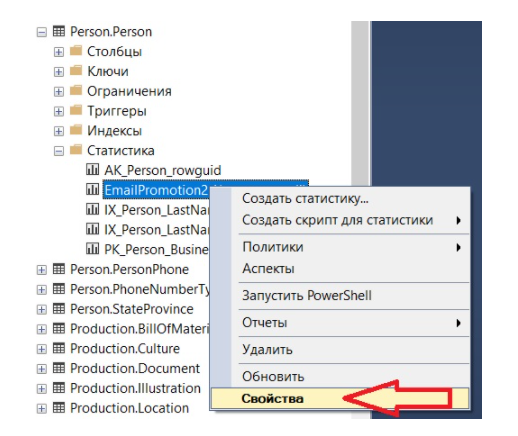
После операций вставки в ключевые столбцы, отсортированные по возрастанию или по убыванию. Для статистики по ключевым столбцам, отсортированным по возрастанию или убыванию (например, столбец IDENTITY или столбцы отметок реального времени), может понадобиться выполнять обновление чаще, чем это делает оптимизатор запросов. Операции вставки добавляют новые значения в столбцы, отсортированные по возрастанию или по убыванию.

#### Команды обновления статистики

Обновить статистику можно:

* с помощью среды SQL Server Management Studio
* с помощью инструкций языка Transact-SQL. При этом параметр AUTO\_UPDATE\_STATISTICS рекомендуется все равно оставлять включенным, чтобы оптимизатор запросов продолжал регулярно обновлять статистику.
* Чтобы определить время последнего обновления статистики, можно использовать функции sys.dm\_db\_stats\_properties или STATS\_DATE

На рисунке № 14 показан просмотр сведений о статистике в интерфейсе SSMS.



*Рисунок № 14. Просмотр сведений о статистике в окне SSMS.*

**Внимание!** Обновление статистики вызывает перекомпиляцию запросов. Поэтому обновлять статистику слишком часто не рекомендуется, необходимо найти компромисс между выигрышем в производительности за счет усовершенствованных планов запросов и потерей времени на перекомпиляцию запросов.

### Использование кэша планов запросов

#### Повторное использование плана запроса

На этапе «разбора» плана выполнения запроса движок Relational Engine, проверяет кэш планов запросов, и если находит там подходящий план, то нового построения плана (рекомпиляции) не происходит, найденный план сразу выдаётся на следующий этап конвейера обработки запроса, что приводит к значительному сокращению времени обработки запроса.

Критериями повторного использования плана запроса без его рекомпиляции могут служить:

* Полное соответствие запросов, вплоть до регистра символов, нового и находящегося в кэше.
* Использование параметризированных хранимых процедур.
* При использовании динамических запросов необходимо использовать оператор EXEC SP\_EXECUTESQL вместо EXECUTE.
* Отказ от литеральных запросов, поскольку при динамической сборке в коде приложения они будут отличаться параметрами.

#### Рекомпиляция запросов

Запрос полученный из кэша запросов не всегда является оптимальным, это происходит, если значения параметров сильно отличаются по селективности. Например, в случае если для одного значения параметра выбирается 99% строк таблицы, а для второго 1% — серверу может быть выгодно использовать разные планы. Один план будет более эффективен для большего числа строк, второй для меньшего.

Если для построения плана запроса используется запрос с параметром для которого возвращается небольшое количество, то будет построен план выгодный для получения небольшого числа строк. Когда значение параметра изменится так, что процедура должна будет вернуть гораздо больше строк, план останется «старым», эффективным для небольшого числа строк.

Для решения данной проблемы можно применить следующие решения:

* Использовать хинты:
  + OPTIMIZE FOR @ПАРМЕТР – создает план, использующий любое указанное значение параметра. Будет оптимизировать запросы для указанного параметра, для остальных будет использован этот план запроса.
  + OPTIMIZE FOR UNKNOWN – будет использовать план, который генерируется на основании средней статистики распределения для этого столбца/индекса. Зачастую это приводит к среднему или плохому плану.
  + OPTION(RECOMPILE) или WITH RECOMPILE – это постоянная рекомпиляция, без использования кэша планов запросов, можно указать данную опцию как для всего запроса, так и только для конкретных инструкций. Несмотря на это использовать её следует, только, если вызываемая процедура используется редко.
* Использование нескольких процедур – метод похож на метод с использованием хинта OPTIMIZE FOR, только вместо хинта в коде хранимой процедуры необходимо предусмотреть разные ветки кода (процедуры) для разных параметров.
* Использование локальных переменных – при этом способе, параметр хранимой процедуры, не используется непосредственно, а сначала сохраняется в локальную копию. Этот метод похож на метод с использованием хинта OPTIMIZE FOR UNKNOWN – также всегда используется одинаковый посредственный план.

### SARGArbility

В реляционных базах данных термин SARGable означает, что условие (или предикат, предложение WHERE) в запросе считается допустимым, если механизм СУБД может использовать индекс для ускорения выполнения запроса. Термин получен из сокращения Возможности поиска аргументов (Search Arguments) или SARGs.

SARG означает, что вместо сканирования всего индекса (или всей таблицы) для получения необходимого значения оптимизатор будет оценивать возможность использования некластеризованного индекса для столбца, который указан как аргумент для операции SEEK (поиска).

Выражениями, которые не относятся к категории SARG (иногда их называют non-sargable), являются, например:

Выражения, которые включают предложение LIKE с подстановочным знаком в начале искомой строки (WHERE lastName LIKE ‘%SMITH%’).

Использование функций на столбцах

WHERE CONVERT(CHAR(10), CreateDate,121) = ‘2020-03-22’.

Эти запросы с выражениями, отличными от SARGable, обычно обнаруживаются путем анализа планов выполнения если в операторах индексов используется сканирование (Scan) вместо поиска (Seek).

### Соответствие типов параметров

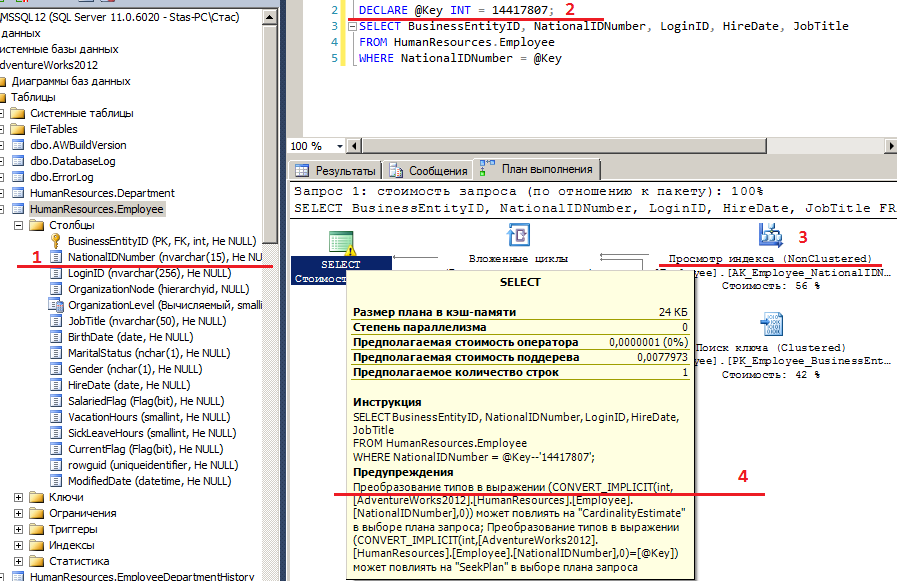
Если тип параметра хранимой процедуры не совпадает с типом колонки таблицы, SQL-сервер будет выполнять неявное преобразование типов данных. При этом дополнительная нагрузка на сервер по преобразованию типов незначительная, но она может значительно вырасти, потому, что если несоответствие типов произойдёт в индексе, то вместо быстрой и эффективной операции поиска (seek) может выполняться более «тяжелое» индексное сканирование (index scan) или сканирование по таблице (table scan).

На рисунке № 15 Показан пример запроса с несовпадающим типом параметров, где цифрами изображены:

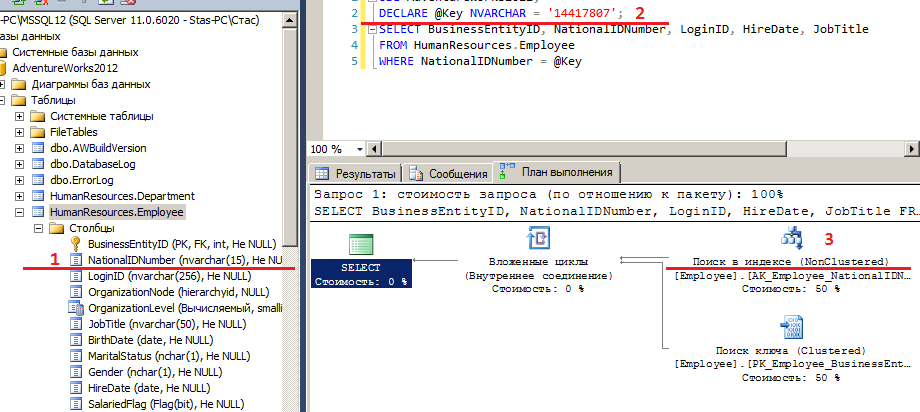
1. Тип колонки в базе данных
2. Тип параметра
3. Операция полного сканирования индекса
4. Предупреждение оптимизатора

А на рисунке № 16, пример запроса с совпадающими типами, видно, что во втором случае используется более производительный поиск в индексе вместо его сканирования.

1. Тип колонки в базе данных
2. Тип параметра
3. Операция поиска в индексе



*Рисунок 15. Сканирование при несовпадении типов параметров.*



*Рисунок 16. Поиск при совпадении типов параметров.*

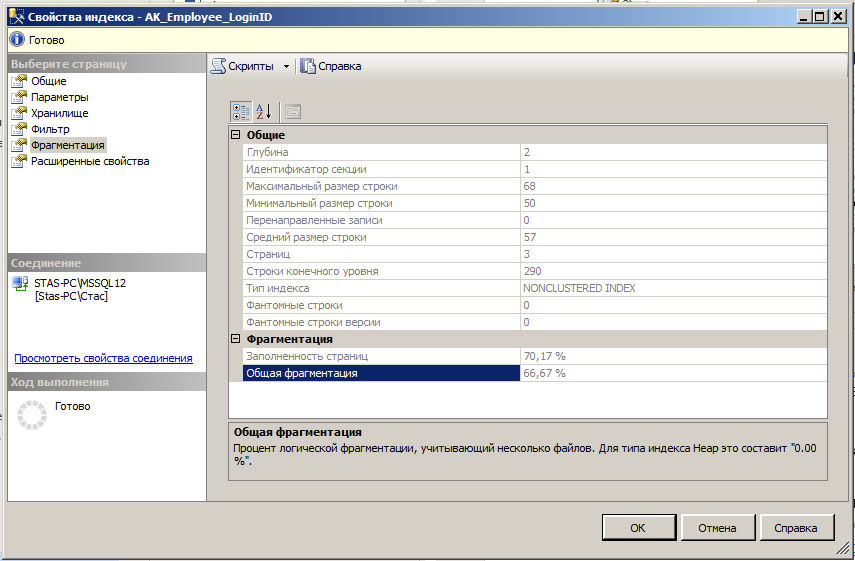
### Дефрагментация индексов

После создания индекса его данные последовательны. Со временем после большого количества операций вставки записей данные будут разбросаны по базе данных (фрагментированы). Значительно фрагментированные индексы могут серьезно снижать производительность запросов и служить причиной замедления откликов приложения. Для устранения фрагментации индексов их нужно реорганизовать.

#### Обнаружение дефрагментации

На рисунке № 17 показано окно SSMS в котором можно определить дефрагментацию индекса, для вызова данного окна необходимо последовательно выполнить команды:

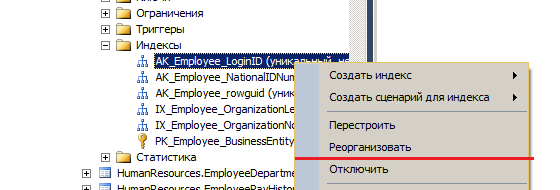
* В обозревателе объектов развернуть базу данных, содержащую таблицу, в которой необходимо проверить фрагментацию индекса.
* Разверните папку Таблицы.
* Разверните таблицу, в которой нужно проверить фрагментацию индекса.
* Разверните папку Индексы.
* Щелкните правой кнопкой мыши индекс, для которого нужно проверить фрагментацию, и выберите пункт Свойства.
* В разделе Выбор страницы выберите пункт Фрагментация.



*Рисунок 17. Определение фрагментации индекса.*

#### Устранение фрагментации индекса

Для устранения фрагментации индекса необходимо в Sql Server Management Studio выполнить команду «Реорганизовать» для нужного индекса, как показано на рисунке № 18:



*Рисунок 18. Команда «Реорганизовать» для индекса.*

**Примечание 1**: Если выполнить команду «Перестроить индекс», то старый индекс будет удалён и создан новый.

**Примечание 2**: Команды «Перестроить» и «Реорганизовать» очень затратные по ресурсам поэтому выполнять их в производственной среде нужно с осторожностью, по согласованию с администратором базы данных.

### Общие рекомендации по оптимизации запросов

#### Предпочтительный синтаксис запросов

В таблице № 3 приведены предпочтительные варианты запросов по сравнению с неоптимальными.

* Использование конкретных имён столбцов вместо \* в запросе SELECT

Запрос SQL, становится быстрее, если использовать имена столбцов в SELECT вместо \*. Это приводит к выигрышу общей производительности, уменьшению сетевого трафика.

* Использование альтернативных методов для возврата общего количества строк таблицы вместо COUNT (\*)

SELECT COUNT (\*) делает полное сканирование таблицы, это может занять много времени для больших таблиц. Если нужно узнать количество строк таблицы, можно использовать альтернативный способ – системную таблицу sysindexes. В ней присутствует столбец ROWS, содержащий общее количество строк для каждой таблицы в системе.

**Примечание:** Данный способ может вернуть не точное количество строк, если статистика на момент выполнения запроса неактуальна.

* Избегать предложения HAVING в операторе SELECT.

Использовать принцип: Сначала фильтрация, потом группировка.

* Свести к минимуму количество подзапросов.

Необходимо получать необходимые данные как можно меньшим количеством запросов.

* Использование операторов EXISTS и IN с осторожностью и правильно.

Эффективность использования EXISTS по сравнению с IN обычно выше, поскольку IN не индексируется. Но это зависит от реальной ситуации:

* + IN подходит для ситуаций, когда внешний набор больше внутреннего;
  + EXISTS подходит для ситуаций, когда внешний набор меньше внутреннего.
* Использование EXISTS вместо DISTINCT при запросе с объединением таблиц, которые имеют отношения один-ко-многим.

По возможности следует использовать EXISTS, а не DISTINCT, потому что DISTINCT сортирует извлеченные строки перед подавлением повторяющихся строк.

* Использование UNION ALL вместо UNION, если это возможно.

UNION ALL работает быстрее, чем UNION, так как UNION ALL не считает дубликаты и не сортирует для этого выходные данные. Инструкция UNION проверяет таблицу на наличие дубликатов строк даже если они отсутствуют.

* Сравнение с использованием BETWEEN.

BETWEEN является единственным предикатом вместо двух предикатов сравнения в сочетании с AND. Один предикат проще оптимизировать, чем два предиката.

* Применение модифицированного значения колонки в инструкции WHERE только при необходимости.

При таком подходе операция модификации будет вызвана для каждой записи из возвращаемого набора.

* Используйте IF EXISTS (SELECT 1) вместо IF EXISTS (SELECT \*)

Чтобы проверить наличие записи в другой таблице, используется выражение IF EXISTS. Данное выражение возвращает true если из внутреннего выражения возвращается хоть одно значение, не важно «1», все колонки или таблица. Возвращаемые данные, в принципе никак не используются. Таким образом для сжатия трафика во время передачи данных логичнее использовать «1».

|  |  |  |
| --- | --- | --- |
| **Рекомендация** | **Оценка** | **Пример** |
| Использование альтернативных методов для возврата общего количества строк таблицы вместо COUNT (\*) |  | SELECT rows FROM sysindexes  WHERE id = OBJECT\_ID ('table\_name') AND indid < 2. |
| Избегать предложения HAVING в операторе SELECT | плохо | SELECT Col\_1, count (Col\_1) FROM table\_name  GROUP BY Col\_1  HAVING Col\_1 != ‗testvalue1‘ AND Col\_1!= 'testvalue2'. |
| лучше | SELECT Col\_1, count (Col\_1) FROM table\_name  WHERE col\_1 != 'testvalue1' AND col\_1!= 'testvalue1'  GROUP BY col\_1; |
| Свести к минимуму количество подзапросов | плохо | SELECT col\_1 FROM table\_name1  WHERE col\_2 = (SELECT MAX (col\_2) FROM table\_name2)  AND col\_3 = (SELECT MAX (col\_3) FROM table\_name2)  AND col\_4 = 'testvalue1'. |
| лучше | SELECT col\_1 FROM table\_name1  WHERE (col\_2, col\_3) = (SELECT MAX (col\_2), MAX (col\_3) FROM table\_name2)  AND col\_4 = 'testvalue1'; |
| Использование операторов EXISTS и IN с осторожностью и правильно | плохо | SELECT \* FROM table1 t1  WHERE t1.col\_id IN (SELECT t2.col\_id FROM table2 t2). |
| лучше | SELECT \* FROM table1 t1  WHERE EXISTS (SELECT \* FROM table2 t2 WHERE t2.col\_id = t1.col\_id); |
| Использование EXISTS вместо DISTINCT | плохо | SELECT DISTINCT d.col\_id, d.col2 FROM table1 d, table2 e  WHERE e.col2 = e.col2. |
| лучше | SELECT d.col\_id, d.col2 FROM table1 d  WHERE EXISTS (SELECT 'X' FROM table2 e WHERE e.col2 = d.col2); |
| Сравнение с использованием BETWEEN | плохо | SELECT Col1, Col2 FROM table  WHERE Col3 >= MAX (Col3) and Col3 <= MIN (Col3). |
| лучше | SELECT Col1, Col2 FROM table  WHERE Col3 BETWEEN MAX (Col3) and MIN (Col3); |
| Применение модифицированного значения колонки в инструкции WHERE только при необходимости | плохо | SELECT id, Col1, Col2 FROM table  WHERE Col2 + 10000 < 35000. |
| лучше | SELECT id, Col1, Col2 FROM table  WHERE Col2 < 25000 |
| Используйте IF EXISTS (SELECT 1) вместо IF EXISTS (SELECT \*) | плохо | IF EXISTS (SELECT \* FROM sysobjects  WHERE name = 'MyTable' AND type = 'U') |
| лучше | IF EXISTS (SELECT 1 FROM sysobjects  WHERE name = 'MyTable' AND type = 'U') |

*Таблица 3. Предпочтительные варианты запросов.*

#### Практика оптимальных запросов

Для того что бы запросы к базе данных выполнялись эффективным способом объекты базы данных такие как таблицы, хранимые процедуры и прочие, а также сами запросы нужно настраивать в соответствии с правилами, неоднократно проверенными при эксплуатации базы данных в производственной среде. К таким правилам можно отнести следующие:

* Таблица должна иметь минимум один кластеризованный индекс и соответствующее число не кластеризованных индексов;
* Таблица должна иметь ключевое поле;
* Необходимо использовать табличные переменные вместо временных таблиц. Переменные занимают меньше системных ресурсов и ресурсов логов.
* Избегать использования триггеров, если это возможно. Включить логику триггера в хранимую процедуру.
* Избегать использования представлений (VIEW), постараться заменить их таблицами.
* Избегать инструкции DISTINCT, использовать ее только если это действительно необходимо.
* Использовать TOP в инструкции SELECT.
* Оформить повторяющийся код в хранимую процедуру. Это поможет улучшить производительность, и уменьшить сетевой трафик.
* Использование TRUNCATE вместо DELETE позволит ускорить удаление строк из таблицы, потому что такое удаление происходит без физического удаления данных.
* Избегать использования курсоров, если это возможно, они сильно замедляют производительность.
* Когда разрабатывается запрос с подзапросами:
  + Использовать коррелированный подзапрос только тогда, когда возвращаемый результат будет относительно небольшим и/или другие критерии быстродействия подзапроса будут эффективными.
  + Использовать не коррелированные подзапрос при работе с большими таблицами, из которых ожидается большой результат и/или подзапрос имеет низкие показатели эффективности.
  + Убедиться в том, что несколько подзапросов расположены в наиболее эффективном порядке. Сначала выполняются подзапросы, обладающие наибольшей селективностью (хотя Sql-сервер следит за порядком запросов, тем не менее стоит обращать на это внимание).
* Для хранения символьных и строковых данных используйте char/varchar вместо nchar/nvarchar, если нет необходимости в использовании UNICODE. В первом случае для хранения символов используется один байт, во втором – два;
* Очистить систему от неиспользуемых индексов, они занимают место на диске и замедляют операции DML (Data Manipulation Language (DML) – операции манипуляции данными);
* Если часто используется объединение одних и тех же таблиц, то стоит создать индекс для объединяемых столбцов.
* Включайте в хранимые процедуры строку — SET NOCOUNT ON.

Выполняя DML выражения в хранимой процедуре, SQL server возвращает сообщение, содержащее количество обработанных записей. Данная информация может быть полезна во время отладки кода, но после будет совершенно бесполезной. Прописывая SET NOCOUNT ON, мы отключаем эту функцию. Для хранимых процедур, содержащих несколько выражений или\и циклы данное действие может дать значительный прирост производительности, потому как количество трафика будет значительно снижено.

* В запросах используйте имя схемы с именем объекта.

Данная операция подсказывает серверу где искать объекты и вместо того чтобы беспорядочно искать по все рабочей области, он сразу будет знать куда ему нужно пойти и что взять. При большом количестве баз, таблиц и хранимых процедур может значительно сэкономить время.

* Не используйте префикс «sp\_» в имени хранимых процедур.

Если имя процедуры начинается с «sp\_», SQL Server в первую очередь будет искать её в своей главной базе данных. Дело в том, что данный префикс используется для личных внутренних хранимых процедур сервера.

### Хинты

Хинт (hint) – это указание оптимизатору запросов, которое переопределяет его поведение по умолчанию на время выполнения SQL инструкции.

Иными словами, с помощью хинта мы можем сказать оптимизатору запросов, как именно поступить в той или иной ситуации в процессе построения плана запроса.

Например, мы можем сказать, какой конкретно индекс использовать, какой конкретно алгоритм физического соединения таблиц применить, или, допустим, на время выполнения запроса разрешить «грязное чтение».

Обычно оптимизатор запросов SQL Server выбирает самый оптимальный план выполнения запроса и крайне редко ошибается, поэтому использовать подсказки, т.е. хинты, без острой на то необходимости не рекомендуется.

Лучше всего использовать хинты только на стадии разработки, или оптимизации запроса.

В Microsoft SQL Server хинты используются в разных секциях запроса, и на основе этого можно выделить следующие типы хинтов.

* Join Hints – хинты в соединениях
* Query Hints – хинты на уровне запроса
* Table Hints – хинты для таблиц

#### Join Hints

С помощью этого типа хинтов можно влиять на выбор алгоритма физического соединения таблиц.

Существуют следующие джоин-хинты:

* LOOP – указывает, что соединение будет выполнено с помощью алгоритма Nested Loops (вложенные циклы).
* HASH – указывает, что соединение будет выполнено с помощью алгоритма Hash Match (хэш-соединение).
* MERGE – указывает, что соединение будет выполнено с помощью алгоритма Merge (соединение слиянием).
* REMOTE – указывает, что соединение будет выполнено на стороне правой таблицы, если она расположена удаленно. Обычно это используется, когда таблица слева является локальной, а таблица справа удаленной и в ней гораздо больше строк, чем в левой таблице.
  + Если правая таблица является локальной, соединение выполняется локально.
  + Если обе таблицы являются удаленными, но из разных источников данных, REMOTE вызывает выполнение соединения на стороне правой таблицы.
  + Если обе таблицы являются удаленными таблицами из одного источника данных, REMOTE не требуется.
  + Аргумент REMOTE может быть использован только при операциях INNER JOIN.

На рисунке № 19 приведён пример использования хинта Hash, что привело к принудительной установке типа физического соединения таблиц – Hash Match.

#### Query Hints

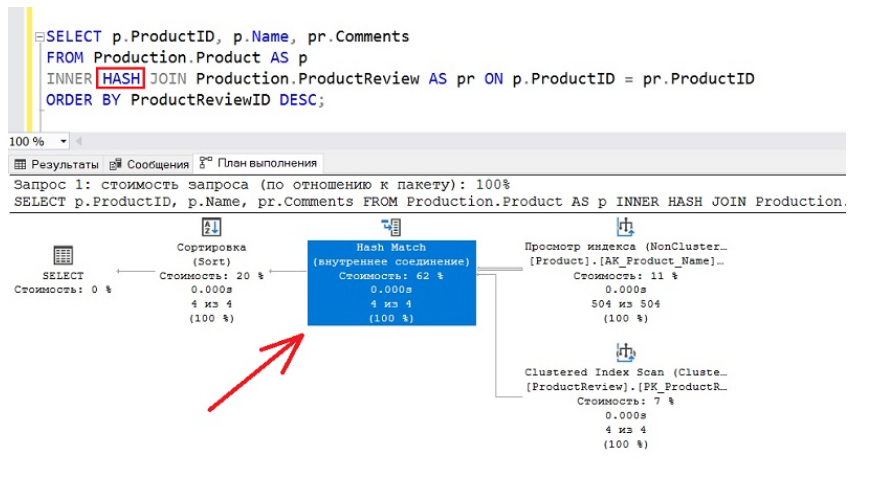
Хинты на уровне запросов распространяются на весь запрос. Иными словами, они влияют на все операторы в SQL инструкции.

Определяются такие хинты с помощью ключевого слова OPTION. В SQL запросе может быть определено только одно предложение OPTION, при этом в нем можно указать несколько хинтов.

Ниже приведены некоторые хинты OPTION:

FORCE ORDER – параметр указывает, что при оптимизации запроса сохраняется порядок соединения, заданный синтаксисом запроса.

OPTIMIZE FOR – указывает оптимизатору запросов, что при компиляции и оптимизации запросов нужно использовать конкретное значение для локальной переменной. Значение используется только в процессе оптимизации запроса, но не в процессе выполнения.



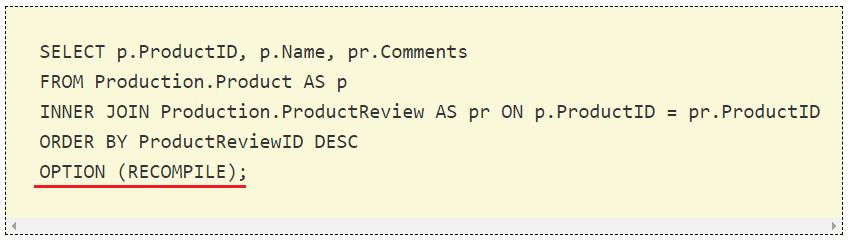
*Рисунок 19. Использование хинта Hash.*

RECOMPILE – параметр указывает SQL Server, что необходимо создать временный план для запроса, который будет удален сразу же после выполнения этого запроса. Как уже указывалось выше без указания подсказки RECOMPILE SQL Server кэширует планы запросов и использует их повторно, иными словами, с параметром RECOMPILE план выполнения запроса будет строиться каждый раз при запуске запроса.

QUERYTRACEON – данный параметр позволяет включить флаг трассировки, только для того запроса в котором он указан. Флаги трассировки используются для временной установки определенных характеристик сервера или отключения определенного режима. Флаги трассировки часто используются для диагностики проблем, связанных с производительностью, отладки хранимых процедур или сложных компьютерных систем

USE HINT – указывает дополнительные подсказки оптимизатору запросов.

На рисунке № 20 показано использование хинта, RECOMPILE.



*Рисунок 20. Использование хинта RECOMPILE.*

#### Table Hints

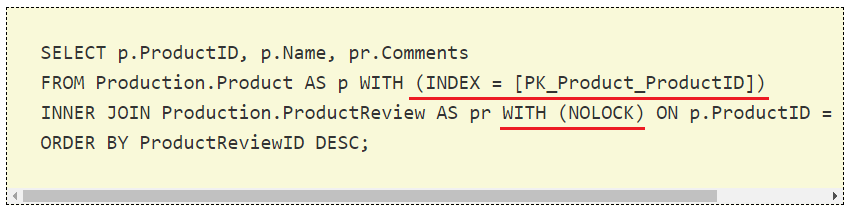
Табличные хинты относятся только к таблицам и представлениям, для которых они указаны. Такие хинты предназначены для определения способа блокировки, указания конкретного индекса, способа обработки данных в таблицах, например, сканирование или поиск.

Ниже приведены несколько популярных табличных хинтов:

* INDEX – указывает индекс, который будет принудительно использован оптимизатором запросов при обработке инструкции.
* FORCESEEK – указывает, что в качестве пути доступа к данным таблиц или представлений оптимизатор запросов использует только операцию поиска в индексе.
* FORCESCAN – указывает, что в качестве пути доступа к данным или представлений оптимизатор запросов использует только операцию сканирования в индексе.
* ROWLOCK – указывает, что вместо блокировки страниц или таблиц применяются блокировки строк.
* HOLDLOCK – накладывает дополнительные ограничения на совмещаемую блокировку, в частности удерживает ее до завершения транзакции вместо снятия блокировки сразу после того, как таблица или страница данных больше не требуется. Иными словами, поведение становится как при уровне изоляции SERIALIZABLE. Хинт HOLDLOCK эквивалентен хинту SERIALIZABLE.
* NOLOCK – разрешает «грязное чтение». Поведение становится похоже на уровень изоляции READ UNCOMMITTED. Хинт NOLOCK равнозначен хинту READUNCOMMITTED.
* NOEXPAND – указывает, что при обработке запроса индексированное представление не расширяется для доступа к базовым таблицам. Оптимизатор запросов обрабатывает представление так же, как и таблицу с кластеризованным индексом. Данный хинт применяется только для индексированных представлений.

Табличные хинты задаются в предложении FROM с помощью ключевого слова WITH, которое указывается после названия таблицы или представления.

На рисунке № 21 показаны установка хинта INDEX для таблицы Production.Product, с помощью которого задаётся конкретный индекс, а для таблицы Production.ProductReview указали хинт NOLOCK, который позволяет обрабатывать данную таблицу в режиме READ UNCOMMITTED, т.е. читаются неподтверждённые данные (грязное чтение).



*Рисунок 21. Использование хинтов INDEX и NOLOCK.*

#### Хинты FORCE\_LEGACY и DEFAULT\_CARDINALITY\_ESTIMATION

Cardinality Estimation, СЕ (оценка кардинальности) – это оценка предполагаемого числа строк, которое будет обработано тем или иным оператором запроса. Оценка – один из ключевых факторов при построении плана запроса. Оценку числа строк осуществляет компонент Cardinality Estimator.

До 2014 сервера, была всего одна версия этого компонента, разработанная для SQL Server 7.0

Начиная с SQL Server 2014 у сервера появилась новая модель оценки строк. Эта модель оценки имеет новую архитектуру, расширяема и дополняема, версия этой модели получила номер 120 (по аналогии с уровнем совместимости БД, соответствующим серверу 2014 – 120). В 2016 сервере модель была расширена и получила номер версии 130, при этом версия 120 сохранилась. В 2017 SQL-Server появилась модель версии 140.

Таким образом на данный момент есть два поколения моделей оценки:

* Устаревшее поколение (legacy) – версия 70 – все сервера младше 2014
* Современное поколение (modern) – версии 120, 130, 140, 150 – сервера 2014, 2016, 2017, 2019

Поскольку механизм оценки представляет собой математическую модель, то, она не учитывает все условия реальности. В каких-то случаях лучше работает устаревшая модель оценки, в каких-то случаях современная, поэтому, важно иметь возможность, в случае возникновения проблем переключаться между ними.

В таблице № 4 показаны версии моделей Cardinality Estimator (оценщика кардинальности).

|  |  |  |  |
| --- | --- | --- | --- |
| SQL Server Version | Compatibility Level | Cardinality Estimation  Model Version | Поколение модели |
| <= 2012 | <= 110 | 70 | ligacy |
| 2014 | 120 | 120 | modern |
| 2016 | 130 | 130 | modern |
| 2017 | 140 | 140 | modern |
| 2019 | 150 | 150 | modern |

*Таблица 4. Версии моделей Cardinality Estimator.*

Для переключения между моделями MS предлагает несколько вариантов.

Самый общий уровень управления – это уровень совместимости БД (compatibility level). По умолчанию, сервер использует ту версию модели оценки (Cardinality Estimation Model Version), которая соответствует уровню совместимости БД, в контексте которой выполняется запрос.

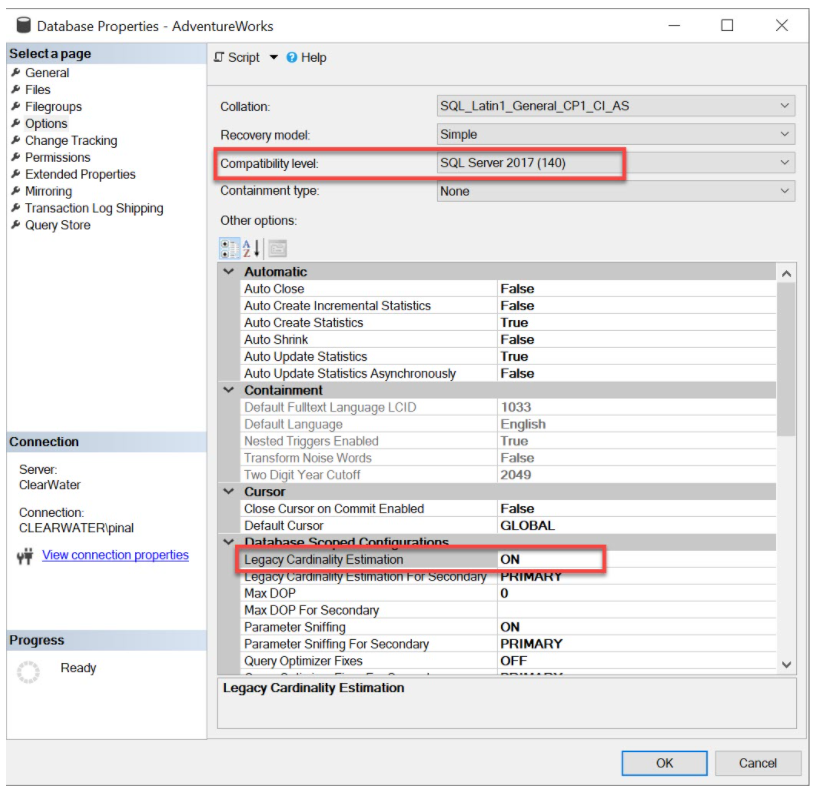
Уровень совместимости можно изменить из SSMS (меню Options в настройках базы данных) и с помощью команды T-SQL: ALTER DATABASE SET COMPATIBILITY\_LEVEL.

На рисунке № 22 показано окно настроек базы данных, в котором можно изменить уровень совместимости и версию модели Cardinality Estimator.

Однако, использование изменения уровня совместимости БД, как средства управления моделью Cardinality Estimator (оценщика кардинальности) имеет один большой недостаток. Понижая уровень совместимости БД, и отказываясь от новой модели, происходит отказ от многих других функций новой версии, не связанных с оценкой.

Для решения этой проблемы MS в 2016-ой версии сервера добавил параметр уровня БД LEGACY\_CARDINALITY\_ESTIMATION, управляемый при помощи команды T-SQL ALTER DATABASE SCOPED CONFIGURATION. Таким образом, выполнив команду ALTER DATABASE SCOPED CONFIGURATION SET LEGACY\_CARDINALITY\_ESTIMATION=ON, можно переключиться на устаревшую, legacy модель оценки.

Важно понимать, что, например, находясь на уровне совместимости 130 и выполнив данную команду, сервер будет использовать версию модели 70 (legacy), а не 120 (modern).

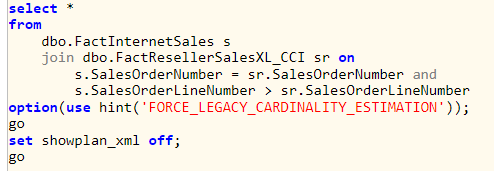


*Рисунок 22. Установка уровня совместимости и версии модели Cardinality Estimator.*

Если нужно управлять моделью оценки не глобально (на уровне сервера) а на уровне запроса, то это можно делать с помощью хинтов:

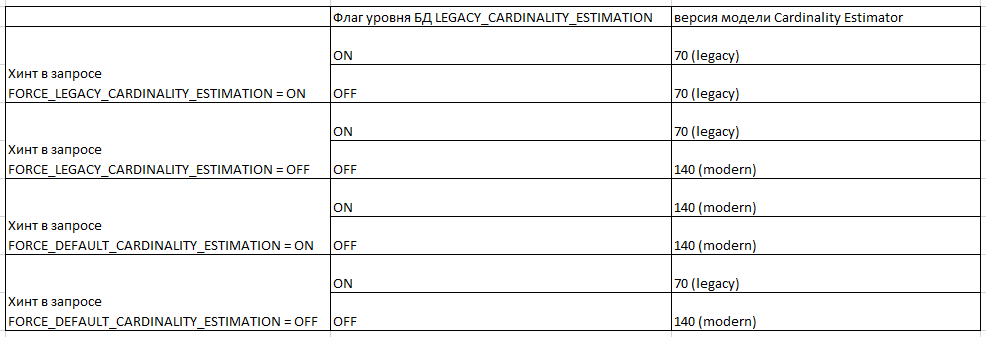
* Хинт FORCE\_LEGACY\_CARDINALITY\_ESTIMATION – форсирует legacy модель оценки версии 70.
* Хинт FORCE\_DEFAULT\_CARDINALITY\_ESTIMATION – форсирует модель оценки, соответствующую уровню совместимости БД, т.е. переопределяет действие настройки LEGACY\_CARDINALITY\_ESTIMATION.

На рисунке № 23 приведён пример запроса, устанавливающий версию модели Cardinality Estimator (оценщика кардинальности) – legacy.



*Рисунок № 23. Запрос, устанавливающий версию модели Cardinality Estimator – legacy.*

В таблице № 5 приведены комбинации флагов и устанавливаемые версии модели Cardinality Estimator (оценщика кардинальности), уровень совмести для всех вариантов соответствует SQL-Server 2017.



*Таблица 5. Комбинация хинтов и флагов LEGACY\_CARDINALITY\_ESTIMATION.*

Из таблицы № 3 видно, что, комбинируя флаги уровня БД LEGACY\_CARDINALITY\_ESTIMATION и табличные хинты FORCE\_LEGACY\_CARDINALITY\_ESTIMATION и FORCE\_DEFAULT\_CARDINALITY\_ESTIMATION можно устанавливать разные версии модели Cardinality Estimator для всей базы данных и для конкретных таблиц.

# Заключение

В данной работе были рассмотрены способы и этапы настройки производительности Microsoft SQL Server. Представлен обзор основных встроенных в Microsoft SQL-Server инструментов мониторинга процессов. Представлены наиболее универсальные приемы оптимизации SQL-кода таким образом, чтобы можно было достичь максимальной производительности при разработке программного обеспечения, использующего в своём составе работу с базой данных Microsoft SQL-Server.

# Вопросы

1. Для чего служат индексы (1)
2. Для упорядоченного доступа к данным
3. Для адресной доставки сообщений
4. Для выбора элементов массива
5. На основании чего SQL-Server строит «Предполагаемый план» (3)
6. На основании исторических данных
7. На основании настроек пользователя
8. На основе оценок
9. Как оператор Clustered Index Scan получает данные (1)
10. Путём просмотра всего индекса
11. Путём поиска в индексе
12. Путём деления на 2
13. Как оператор Clustered Index Seek получает данные (2)
14. Путём просмотра всего индекса
15. Путём поиска в индексе
16. Путём деления на 3
17. Как читать графический план запросов (4)
18. Как удобней
19. Слева направо с низу вверх
20. Слева на право с верху вниз
21. Справа на лево с верху вниз

# Список литературы

1. «SQL Server 2012, руководство для начинающих», Душан Петкович, Санкт-Петербург, 2013.
2. «Введение в системы баз данных», К. Дж. Дейт, Издательский дом Вильяме, 2005.
3. «Настройка SQL для профессионалов», Дэн Тоу, Спб. «Питер», 2004.
4. Интернет-блог «Заметки Дмитрия Пилюгина о Microsoft SQL Server» <http://www.queryprocessor.ru/cardinality-estimation-framework-2014-first-look/>
5. Интернет-блог «Заметки IT специалиста», <https://info-comp.ru/what-are-hints-in-t-sql>
6. Интернет-статья «Методы оптимизации запросов к SQL Server - Советы для написания эффективных и быстрых запросов» <https://masters.donntu.org/2016/fknt/prischepa/library/article10.htm>
7. Интернет-блог «Програмысли», <https://www.flenov.info/story/show/Optimizaciya-SQL-zaprosov-dlya-MS-SQL-Server-s-pomoschyyu-indeksov>
8. Интернет-статья «Проектирование хранилищ данных для приложений систем деловой осведомленности (Business Intelligence Systems)»
9. <https://intuit.ru/studies/courses/599/455/lecture/10187?page=2>

1. Подключение DI контейнера Autofac в Net.Core:

<https://habr.com/ru/post/437002/>