1. Основные понятия: база данных; СУБД; принципы и этапы построения, основные модели данных, основные понятия теории реляционных баз данных.

База данных (БД) – это совокупность взаимосвязанных данных при такой минимальной избыточности, которая допускает их использование оптимальным образом в определенной предметной области.

СУБД – система управления базами данных.

Требования к базам данных:

1. Интеграция данных - все данные должны накапливаться и храниться централизованно.

2. Независимость прикладных программ от данных.

3. Безопасность данных - защита данных от случайного или преднамеренного доступа к ним лиц, не имеющих на это права.

4. Минимальная избыточность данных - требования новых приложений должны удовлетворяться за счет существующих данных, а не путем создания новых файлов.

5. Обработка запросов для поиска информации в базе данных должна быть обеспечена с помощью высокоуровневого языка запросов.

6. Для разработки и управления БД используются специальные прикладные программы – системы управления базами данных (СУБД)

2. Проектирование баз данных. Модели данных.

Проектирование Баз Данных:

1. Определение границ исследуемой предметной области.

2. Определение объектов и связей между ними.

3. Построение логической схемы базы данных в соответствии с определенными правилами (моделью данных).

4. Реализация базы данных.

Предметная область – это часть реального мира, подлежащая изучению, с целью описания и управления.

База данных создается для многих пользователей. Каждый пользователь имеет свое представление о базе данных. Совокупность всех представлений – это логическая схема данных.

Модели данных:

1. Иерархические (данные представляются в виде древовидной структуры.)

Иерархические БД обладают высокой скоростью доступа к требуемой информации в направлении от основания дерева к его вершинам, но затрудненным доступом к информации в обратном направлении - от периферии к основанию дерева. (IBM IMS)

2. Сетевые

В сетевых моделях снимается указанный выше недостаток иерархических БД, но язык манипулирования данными для таких моделей всегда является гораздо более сложным, так как содержит большое число разнотипных команд и слабо формализуется. (SOFTWARE AG ADABAS)

3. Реляционные

В реляционных БД (РБД) объекты (записи)разделяются на типы. Каждый тип имеет свои характеристики – атрибуты, значения которых содержатся в полях (столбцах) таблицы.

Один из атрибутов однозначно идентифицирует объект в таблице, т.е. каждому конкретному значению такого атрибута будет соответствовать одна и только одна запись (строка) в таблице.

Значения такого атрибута в столбце таблицы должны быть уникальны. Этот атрибут называется первичным ключом.

Если в таблице нет одного атрибута с неповторяющимися значениями, такой ключ может состоять из нескольких атрибутов, и тогда он называется составным ключом.

Такие атрибуты могут использоваться для организации связей между таблицами, благодаря которым обеспечивается целостность БД и взаимодействие данных.

Можно выделить следующие преимущества РБД:

- отношения (таблицы) не зависят от способа хранения данных в компьютере;

- теория отношений является областью математической логики (реляционной алгебры), и поэтому хорошо формализована;

- имеется простой и единообразный способ представления данных в виде таблиц;

- осуществляется надежное обеспечение целостности и защиты данных.

Недостатки: медленная работы для больших БД, жесткость структуры данных и некоторые другие.

Определения реляционной алгебры:

1. -домен – множество (например: домен, описывающий фамилии преподавателей – это множество строк длиной от 1 до 50);

2. - таблица – отношение;

3. - атрибут – имя столбца таблицы;

4. - заголовок таблицы – множество всех атрибутов;

5. - кортеж – элемент отношения или строка таблицы;

Операции реляционной алгебры (алгебры Кодда):

1. UNION (объединение),

2. INTERSECT (пересечение),

3. MINUS (разность),

4. TIMES (декартово произведение),

5. WHERE (ограничение),

6. PROJECT (проекция),

7. JOIN (соединение),

8. DIVIDE BY (реляционное деление),

9. RENAME (переименование),

10. := (присваивание).

Создать базу данных, а также объекты базы данных (таблицы, хранимые процедуры, индексы, запросы и т.п.) можно при помощи дизайнеров и диалоговых окон среды разработки, а также при помощи сценариев SQL.

Можно создать базу с помощью скрипта.

В верхней части большинства диалоговых окон для создания и изменения объектов, например, в верхней части диалогового окна для создания индексов New Index, находится меню генерации сценариев Script.

Среда SSMS предлагает большое число шаблонов, предназначенных для решения многих типовых задач. Шаблоны – это стандартные заготовки сценариев, которые м. использовать для создания собственных сценариев SQL.

Шаблоны содержат параметры, значения которых вводятся пользователем, такие, например, как имя таблицы. Используя эти параметры, м. ввести имя только один раз, а затем автоматически скопировать его во все требуемые позиции в скрипте. Также м. создавать собственные пользовательские шаблоны.

Доступ к шаблонам м. получить, открыв окно проводника шаблонов Template Explorer. По двойному щелчку любого узла с шаблоном открывается окно для редактирования запроса Query Edit с SQL-кодом соответствующего шаблона.

Пользователю база данных может быть представлена различными способами: в простейшем случае в виде подмножества таблиц или представлений (объекты БД, о которых будет рассказано позже), набора процедур и функций на языке T-SQL и пр.

В общем случае, пользователь БД видит то, что предоставляет инструментарий, используемый им для доступа к данным и применяемый интерфейс. Кроме того, представление пользователя зависит от настроек системы безопасности СУБД, регулирующей область видимости данных и порядок доступа к ресурсам БД.

С точки зрения проектировщика БД – это логическая схема данных, представляющая собой набор специальным образом структурированных данных с заданными ограничениями и связями.

С точки зрения системного администратора БД – это набор файлов операционной системы, имеющих определенную физическую структуру, формат и организацию.

Теория реляционных БД предполагает независимость трех уровней друг от друга.

Другими словами физическое размещение данных (физический уровень) никак не должно ограничивать возможности проектирования логической схемы данных, а представления пользователей о БД не зависят от структуры логической схемы. В реальности современные СУБД обеспечивают лишь частичную независимость уровней друг от друга.

3. Нормализация реляционной базы данных (1, 2, 3 формы).

Нормализация – это формализованная процедура, в процессе выполнения которой атрибуты данных группируются в таблицы, а таблицы – в БД.

При этом таблицы базы данных проверяются на наличие зависимостей между столбцами.

Если зависимости существуют, то исходная таблица разделяется на несколько таблиц, в которых нет зависимостей между столбцами.

Если одна из этих таблиц все еще содержит зависимости, то процесс нормализации повторяется, пока не будут устранены все зависимости.

Функциональная зависимость означает, что при использовании известного значения одного столбца соответствующее значение другого столбца всегда может быть однозначно определено.

Цели нормализации:

1. - исключить дублирование и избыточность информации;

2. - обеспечить возможность проведения непротиворечивых и корректных изменений данных в таблицах;

3. - упростить и ускорить поиск информации в БД.

В нормализованной базе данные не концентрируются в одной большой таблице, а распределяются по нескольким таблицам с меньшим количеством столбцов.

Есть 6 нормальных форм, но уже после 3 база имеет оптимальную структуру.

Первая нормальная форма (First Normal Form, 1NF) означает, что таблица не имеет атрибутов с несколькими значениями (многозначных атрибутов) или составных атрибутов. Каждая строка в единственном экземпляре.

2-й нормальной форме (2NF), необходимо, чтобы она уже находилась в 1-й нормальной форме и все не ключевые поля полностью зависели от ключевого.

Для перехода к 3-й нормальной форме (3NF), необходимо обеспечить, чтобы все таблицы БД находились во 2-й нормальной форме, все не ключевые поля в таблицах не зависели друг от друга.

отсутствуют транзитивные функциональные зависимости.

BCNF – Нормальная форма Бойса-Кодда.

Признак отсутствия BCNF при 3NF: наличие 2х возможных составных ключей, имеющих общий атрибут.

4. Основы SQL: стандарты, группы операторов и их назначение.

Язык SQL (Structured Query Language, язык структурированных запросов) – специализированный язык, предназначенный для написания запросов к реляционной БД.

Основной единицей этого языка является SQL-оператор. Запрос к БД представляет собой один или несколько операторов языка.

Прототипом SQL является язык SEQUEL, кот. был разработан в начале 1970-х годов компанией IBM для экспериментальной СУБД SystemR.

В 1986 году ANSI (American National Standards Institute, Американский национальный институт стандартизации) принял первый стандарт языка SQL. На сегодняшний день действует еще ряд стандартов: SQL-2003, SQL-2006, SQL-2008, которые в той или иной степени поддерживаются различными СУБД.

Чтобы выполнить часть сценария, надо эту часть выделить и нажать Execute или клавишу <F5>.

Операторы SQL делятся на несколько групп:

1. DDL (Data Definition Language, язык определения данных)

2. DML (Data Manipulation Language, язык манипулирования данными)

3. TCL (Transaction Control Language, язык управления транзакциями)

4. DCL (Data Control Language, язык управления данными)

5. Основные принципы работы SQL Server: клиент, сервер, системные базы данных.

Использование MSS подразумевает применение или разработку клиент-серверных приложений.

В качестве сервера выступает сам MSS, а в качестве клиента может быть либо утилита (специализированная программа), входящая в состав MSS, либо программа, разработанная на одном из языков программирования.

Для взаимодействия клиентского приложения с сервером MSS служит специальная DLL-библиотека, называемая SNAC (SQL Native Access Client) и включающая в себя программные интерфейсы: провайдер OLEDB (Object Linking and Embedding, Database) и драйвер ODBC (Open Data Connectivity).Для передачи данных по компьютерной сети SNAC использует сетевые библиотеки MSS, с пом. которых обеспечивается передача данных в соответствии с протоколом прикладного уровня TDS (Tabular Data Stream).

ADO.NET и интерфейс JDBC, интерфейсы для java и .Net.

Вне зависимости от используемого программного интерфейса клиентское приложение осуществляет соединение со специальным объектом MSS – конечной точкой. При инсталляции MSS автоматически создается несколько конечных точек, позволяющих установить соединение с сервером с пом. нескольких сетевых протоколов (TCP/IP, Named Pipes, Shared Memory, HTTP и пр.).

Программное обеспечение сервера условно м. разбить на три уровня:

1. реляционный уровень

(БД представляется пользователю (или программному интерфейса) в виде набора реляционных объектов (таблиц, представлений, ограничений целостности и пр.))

2. уровень хранилища данных

(Размещение, надежное хранение реляционных объектов и доступ (синхронизация, блокировка, кеширование и пр.))

3. уровень взаимодействия с операционной системой (SQLOS).

Слой SQLOS обеспечивает взаимодействие MSS с операционной системой Windows. В этом смысле SQLOS м. представить как специализированную виртуальную машину, на кот. работает MSS.

Служба Database Engine является ядром системы управления реляционной БД.

На одном компьютере может быть установлено несколько экземпляров службы Database Engine.

При этом только один экземпляр может быть службой (сервером) по умолчанию (с именем MS SQL SERVER), другие экземпляры должны быть именованными службами и иметь уникальные имена.

Каждый экземпляр службы Database Engine требует отдельной инсталляции, конфигурации и настройки безопасности.

При этом каждый сервер в своей работе использует пять системных БД.

Системная база данных, Назначение

master Хранит все системные данные Database Engine, а также информацию о других БД.

msdb хранит информацию о резервном копировании БД.

Tempdb Пространство для временных объектов Database Engine и пользовательских временных таблиц. База данных пересоздается при каждой перезагрузке

model Шаблон, используемый при создании всех БД, управляемых экземпляром Database Engine.

resource БД, используемая только для чтения. Содержит системные объекты экземпляра Database Engine. Файлы БД являются скрытыми.

На физическом уровне базы данных, управляемые сервером Microsoft SQL Server 2008, можно разбить на две группы: системные и пользовательские.

Системные БД (master, msdb, tempdb, model, resource) создаются при инсталляции сервера и используются сервером в процессе его работы.

6. Создание базы данных: оператор создания, файловые группы, журнал транзакций, основные опции.

CREATE DATABASE [NAME DATABASE];

Все файлы БД, кроме файлов журнала транзакций, распределены по файловым группам. Файловые группы – это поименованный набор файлов БД.

Файловые группы используются в основном для упрощения администрирования БД.

Существует операторы (например, копирования и восстановления БД), позволяющие рассматривать файловую группу как единое целое и выполнять операции не для каждого файла по отдельности, а сразу для файловой группы.

Кроме того, файловые группы применяются при секционировании таблиц – технологии, позволяющей разместить фрагменты таблицы в разных файловых группах, что при некоторых дополнительных условиях позволяет повысить производительность запросов к этой таблице.

Обязательной является файловая группа, называемая первичной и файл журнала транзакций.

Для обозначения первичной файловой группы используются ключевые слова ON PRIMARY.

Журнал транзакций описывается отдельно в секции, обозначенной ключевыми словами LOG ON.

USE master

go

CREATE database UNIVER

ON PRIMARY --первичная файловая группа

( name = N'BSTU\_mdf', --логическое имя файла

filename = N'D:\BD\UNIVER\_mdf.mdf', --физич. файл

size = 10240Kb, --первонач. размер файла

maxsize = UNLIMITED, --максим. размер файла

filegrowth = 1024Kb --приращение

)

LOG ON --журнал транзакций

( name = N' UNIVER\_log', --логическое имя файла

filename =N'D:\BD\UNIVER\_log.ldf', --физич.файл

size = 10240Kb, --первонач. размер файла

maxsize = 2048Gb, --максим. размер файла

filegrowth = 10% --приращение

)

go

Ключевое слово On используется для определения местонахождения файла.

Первичный файл содержит всю информацию, необходимую для нормального функционирования БД, а также сами данные (например, строки таблиц).

Запись модифицированных данных не осуществляется непосредственно в файл базы данных. Сначала информация записывается в журнал транзакций. Потом в определенный момент времени изменения переносятся в базу.

База имеет произвольный доступ, что ускоряет выборку данных, файл журнала транзакций – последовательный доступ, что обеспечивает отслеживание операций изменения в должном порядке.

Файл журнала транзакций используется для восстановления данных БД в случае аварийного завершения работы сервера.

Разработчик БД помимо первичной файловой группы может создавать дополнительные файловые группы, называемые вторичными.

Во вторичных файловых группах могут располагаться только вторичные файлы. В первичной файловой группе помимо обязательного первичного файла тоже могут быть расположены вторичные файлы.

USE master

go

CREATE database UNIVER

ON PRIMARY --первичная файловая группа

( name = N' UNIVER\_mdf', --первичный файл (mdf)

filename = N'D:\BD\ UNIVER\_mdf.mdf',

size = 10240Kb, maxsize = UNLIMITED,

filegrowth = 1024Kb

),

( name = N' UNIVER\_ndf', --вторичный файл (ndf)

filename = N'D:\BD\ UNIVER\_ndf.ndf',

size = 10240KB, maxsize = 1Gb,

filegrowth = 25%

),

FILEGROUP FG1 --вторичная файловая группа

( name = N' UNIVER\_fg1\_1', --вторичный файл (ndf)

filename = N'D:\BD\ UNIVER\_fgq-1.ndf',

size = 10240Kb, maxsize = 1Gb,

filegrowth = 25%

),

( name = N' UNIVER\_fg1\_2', --вторичный файл (ndf)

filename = N'D:\BD\ UNIVER\_fgq-2.ndf',

size = 10240Kb, maxsize = 1Gb,

filegrowth = 25%

)

LOG ON --журнал транзакций

( name = N' UNIVER\_log', --файл журнала транз.

filename = N'D:\BD\ UNIVER\_log.ldf',

size = 10240Kb, maxsize = 2048Gb,

filegrowth = 10%

)

Go

При создании таблиц и индексов дисковая память для них автоматически отводится в файловой группе по умолчанию. Для размещения этих объектов БД в другой файловой группе следует явно указывать ее имя в операторе CREATE, создающем таблицу или индекс.

ALTER DATABASE — изменение настроек БД;

Добавление нового вторичного файла в базу

alter database UNIVER add file

( name = N’ UNIVER1’,

filename = N’C:\ UNIVER1.ndf’,

size = 3072KB, maxsize = unlimited,

filegrowth = 1024KB

) to filegroup G1;

alter database UNIVER modify filegroup G1 default;

Изменяется свойство файловой группы.

Все параметры разбиты на несколько групп:

 параметры автоматических действий;

 параметры курсора;

 параметры внешних действий;

 параметры восстановления;

 параметры SQL;

 параметры моментальных снимков;

 параметры компонента Service Broker;

 параметры доступности БД.

Начальные значения (по умолчанию) параметров устанавливаются при создании БД.

Изменить их можно с помощью оператора ALTER DATABASE SET

ALTER database UNIVER set AUTO\_CLOSE off --автоматическое закрытие БД

ALTER database UNIVER set AUTO\_CREATE\_ STATISTICS on –автоматический сбор статистики

ALTER database UNIVER set AUTO\_SHRINK on --автоматическое сжатие файлов

С помощью встроенной функции DATABASEPROPERTYEX можно получить текущие установки параметров БД. Возвращает 1 либо 0;

Размещение таблицы в файловой группе

7. SQL DDL: операторы, назначение, применение.

Операторы DDL (Data Definition Language) предназначены для создания, удаления и изменения объектов БД или сервера СУБД.

CREATE, ALTER, DROP

8. SQL DML: операторы, назначение, применение.

Операторы DML (Data Manipulation Language) предназначены для работы с таблицами (поиск, вставка, удаление, изменение).

DML включает четыре оператора:

SELECT, INSERT, DELETE, UPDATE

Иногда к этой группе относят оператор TRUNCATE (быстрое удаление).

SELECT позволяет выбрать множество строк из одной или нескольких таблиц. При успешном выполнении оператора формируется результирующий набор, представляющий собой множество однотипных (с одинаковыми столбцами) строк.

INSERT INTO Spisok(Nomer, Name, B\_Date) VALUES (21, 'Ivanov', '30.01.2000');

DELETE FROM Spisok WHERE Name = 'Petrov';

UPDATE Spisok SET B\_Date date = '30.01.2001'

WHERE Name = 'Ivanov';

9. SQL TCL: операторы, назначение, применение.

Операторы TCL (Transaction Control Language) предназначены для создания транзакций.

Транзакция – это несколько DML-операторов, которые либо все успешно выполняются, либо все не выполняются.

BEGIN TRAN, SAVE TRAN, COMMIT TRAN, ROLLBACK TRAN

Оператор SAVE TRAN применяется для формирования контрольной точки – промежуточного состояния транзакции, к которому может быть осуществлен откат.

В тексте SQL-скрипта оператор SAVE TRAN должен находиться между операторами BEGIN TRAN и ROLLBACK TRAN.

10. SQL DCL: операторы, назначение, применение.

Операторы DCL (Data Control Language) предназначены для управления процессом авторизации.

GRANT, REVOKE, DENY

Авторизация – это процедура проверки разрешений на выполнение определенных операций.

Авторизация может выполняться на двух уровнях: на уровне сервера или на уровне БД. Каждому уровню соответствует свой перечень операций, которыми можно управлять с помощью DCL.

Разрешениями могут обладать специальные объекты сервера или БД, которые объединены под общим названием – принципалы.

Принципал – это объект сервера или БД, которому может быть выдано разрешение на выполнение операции, а также отобрано или запрещено разрешение.

Например, при подключении к серверу любой пользователь сервера обязан пройти процедуру аутентификации (проверку подлинности), которая в простейшем случае сводится к вводу имени и пароля.

После этого все действия пользователя на уровне сервера выполняются от имени объекта сервера, имеющего тип Login (учетная запись), а на уровне БД от имени объекта, имеющего тип User (пользователь БД).

Объекты типов Login и User – принципалы, которым могут быть выданы разрешения.

Оператор GRANT предназначен для выдачи разрешений на осуществление некоторых действий.

Отобрать разрешение можно с помощью оператора REVOKE,

Оператор DENY также позволяет запретить разрешения, выданные ранее

11. Типы ограничений целостности.

Ограничения, накладываемые на столбцы таблицы и предотвращающие появление данных, не соответствующих предварительно заданным свойствам таблицы, называются ограничениями целостности.

Условное обозначение ограничения целостности Действие ограничения

целостности

data type тип данных предотвращает появление в столбце значений, не соответствующих типу данных

not null запрет значений null предотвращает появление в столбце значений null

default значение по умолчанию устанавливает значение в столбце по умолчанию при выполнении операции INSERT

primary key первичный ключ предотвращает появление в столбце (группе столбцов) повторяющихся значений и пустого значения

Create table FACULTY --факультет

( FACULTY char(10) primary key, --идентификатор

FACULTY\_NAME varchar(50) --полное имя

);

foreign key внешний ключ устанавливает связь между таблицей со столбцом, имеющим свойство foreign key (FK внешний ключ) и таблицей, имеющей столбец со свойством primary key (PK – первичный ключ);

предотвращает несогласованные операции между PK и FK

unique уникальное значение аналогично primary key, но допускает пустые значения и не может быть использован для связи с foreign key

check проверка значений предотвращает появление в столбце значения, не удовлетворяющего логическому условию, записанному после check

Для ограничений целостности primary key, foreign key, unique и check может быть задано имя, которое при возникновении ошибки, связанной с этим ограничением, будет указано в сообщении сервера. Если имя не задано, то при создании таблицы сервер назначает ограничениям этих типов собственные имена.

Для того, чтобы задать собственное имя ограничению, следует использовать ключевое слово CONTSTRAINT:

Create table FACULTY--факультет

( FACULTY char(10)

constraint PK\_FACULTY primary key,

FACULTY\_NAME varchar(50)–полное имя

);

или вторая форма записи:

Create table FACULTY

( FACULTY char(10),

FACULTY\_NAME varchar(50),

constraint PK\_FACULTY

primary key (FACULTY));

Пример создания таблицы с составным первичным ключом (может использоваться только вторая форма записи):

Create table SHEDULE\_TEACHER –расписание преподавателей

( [DATENAME] smalldatetime, -- дата и время занятий

TEACHER char(10), -- преподаватель

[SUBJECT] char(10), -- дисциплина

AUDITORIUM char(10), -- аудитория

constraint PK\_ S\_TEACHER primary key

([DATENAME], TEACHER)

);

Not null

Ограничение PRIMARY KEY неявно подразумевает ограничение NOT NULL.

Create table FACULTY

( FACULTY char(10),

FACULTY\_NAME varchar(50) not null,

constraint PK\_FACULTY

primary key (FACULTY));

Иногда значения NULL целесообразно заменить некоторым другим преопределенным значением. В таких случаях используют свойство столбца (ограничение целостности) DEFAULT.

create table FACULTY

(FACULTY char(10),

FACULTY\_NAME varchar (50) default '&&&',

constraint FACULTY\_PK primary key (FACULTY));

Внешний ключ

Внешний ключ – ограничение целостности, основанное на связи, установленной между двумя таблицами БД.

Связь может быть установлена между строками таблицы, имеющей внешний ключ (FK-таблица – таблица с FOREIGN KEY – подчиненная таблица), и строками таблицы, имеющей первичный ключ (PK-таблица – таблица с PRIMARY KEY – главная таблица).

На одну строку PK-таблицы могут ссылаться несколько строк FK-таблицы (отношение «один ко многим»).

Ограничение целостности CHECK для контроля значений

При вводе или корректировке для проверки значений данных может применяться ограничение целостности CHECK.

После ключевого слова CHECK следует логическое выражение, проверяющее значение в столбце.

Оператор IN принимает значение «истина», если значение в столбце, указанном слева от IN, будет совпадать хотя бы с одним значением, указанным в списке справа от IN.

Ограничение целостности UNIQUE позволяет значения в столбце или комбинацию значений нескольких столбцов таблицы сделать уникальными в этой таблице.

Свойство IDENTITY столбца таблицы

Процесс нормализации таблиц приводит к делению одной таблицы на несколько. При этом для обеспечения нормальной формы новых таблиц иногда приходится прибегать к суррогатным ключам.

Суррогатный ключ – это дополнительный столбец, предназначенный служить первичным ключом.

СУБД Microsoft SQL Server 2008 позволяет создавать столбцы с автоматически формируемыми целочисленными уникальными значениями с помощью свойства IDENTITY.

Свойство IDENTITY имеет два параметра, которые указываются в скобках после ключевого слова IDENTITY.

Первый параметр задает стартовое значение идентификатора, второй – приращение.

Попытка явно ввести значение в столбец со свойством IDENTITY с помощью INSERT приводит к ошибке.

12. Оператор SELECT.

SELECT позволяет выбрать множество строк из одной или нескольких таблиц. При успешном выполнении оператора формируется результирующий набор, представляющий собой множество однотипных (с одинаковыми столбцами) строк.

Оператор SELECT состоит из нескольких секций:

SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, INTO

Основой для поиска является набор строк, описанный в секции FROM. В простейшем случае – это имя таблицы БД, а в общем случае, это выражение, содержащее имена нескольких таблиц, SELECT-запросов, представлений и табличных функций.

13. Операторы INSERT, DELETE, UPDATE.

INSERT INTO Spisok(Nomer, Name, B\_Date) VALUES (21, 'Ivanov', '30.01.2000');

Оператор INSERT относится к группе операторов DML SQL и предназначен для добавления строк в таблицу.

Оператор INSERT может добавить ноль, одну или более строк в заданную (таблица-приемник) таблицу БД. Источником строк для добавления служит тоже таблица (таблица-источник), содержащая ноль, одну или более строк. Причем таблица-приемник – это таблица БД (временная или постоянная) или таблица-переменная (этот тип таблиц рассматривается позже), а таблица-источник – это результирующий набор данных (строки, полученные в результате SELECT-запроса) или просто заданный список строк.

Значения столбцов со свойствами IDENTITY, TIMESTAMP и ROWVERSION формируются автоматически, поэтому при использовании оператора INSERT их писать не надо.

Если требуется ввести значение NULL, то это надо указывать явно.

Если надо ввести значение по умолчанию, то можно использовать ключевое слова DEFAULT.

Если требуется обозначить порядок значений в списке VALUES оператора INSERT, используется список столбцов, который в скобках после имени таблицы в секции.

Ключевое слово INTO при написании оператора INSERT не является обязательным и может быть опущено.

Если имени столбца нет в списке полей ΙNSERT, то введется значение NULL или значение по умолчанию.

INSERT-DELETE

С указанием порядка столбцов

При добавлении строк с помощью конструкции INSERT.SELECT можно ограничить количество добавляемых строк с помощью секции TOP, которая располагается сразу за ключевым словом INSERT.

Конструкция INSERT…EXECUTE

В общем случае, хранимая процедура может формировать несколько результирующих наборов. Однако при использовании такой процедуры в конструкции INSERT…EXECUTE выдвигается требования к однородности (одинаковое количество столбцов и совместимые типы соответ. столбцов) результирующих наборов, т. к. они исп. как один общий последовательно составленный из фрагментов результирующий набор.

Применение секции OUTPUT оператора INSERT

Секция OUTPUT может использоваться во всех трех DML-операторах, изменяющих БД: INSERT, DELETE, UPDATE.

Главной особенностью работы этой секции является то, что на ее работу не распространяется действие процедуры отката (ROLLBACK), устраняющей все изменения БД, выполненные DML-операторами в рамках транзакции в случае ее неуспешного завершения.

В рамках оператора INSERT секция OUTPUT позволяет получить и записать данные из INSERT-оператора в один из двух источников: в таблицу БД или в результирующий набор.

Использование выходных данных

Применение секции OUPUT не допускается в конструкции INSERT…EXECUTE, требует списка вводимых столбцов, а также накладывает ряд ограничений на таблицу, в которую секция OUTPUT записывает информацию.

DELETE FROM Spisok WHERE Name = 'Petrov';

Синтаксис DELETE предусматривает две секции: обязательную секцию FROM и необязательную секцию WHERE.

Оператор DELETE может удалить ноль, одну или более строк из таблицы, имя которой указывается в первой секции FROM (первое ключевое слово FROM может быть опущено).

Во второй секции FROM может быть указано JOIN-выражение для внутреннего соединения целевой таблицы с другими таблицами. Удалению подлежат только те строки целевой таблицы, для которых выполнилось соединение.

В секции WHERE указывается логическое выражение, фильтрующее строки, выбранные второй секцией FROM, подлежащие удалению.

Результат один, но выполняются по разному.

Применение опции TOP в общем случае не позволяет предугадать, какие строки именно будут удалены, так как никакой порядок выбора их в операторе DELETE не гарантируется, а сортировка удаляемых этим оператором строк не предусмотрена. Как правило, эту секцию используют при отладке приложений.

Output

UPDATE

UPDATE Spisok SET B\_Date date = '30.01.2001'

WHERE Name = 'Ivanov';

Синтаксис UPDATE предусматривает обязательную секцию SET и две необязательные секции FROM и WHERE. Кроме того, допускается применять секцию OUTPUT и опцию TOP.

Логика действия оператора UPDATE похожа логику оператора DELETE. Оператор UPDATE может изменить ноль, одну или более строк в одной таблице (в целевой таблице), имя кот. указывается сразу за ключевым словом UPDATE.

В секции FROM может быть указано JOIN-выражение для внутреннего соединения целевой таблицы с другими таблицами. Изменению подлежат только те строки целевой таблицы, для которых выполнилось соединение. Если секции FROM нет, то предполагается, что в этой секции выбираются все строки таблицы.

В секции WHERE указывается логическое выражение, фильтрующее строки, выбранные секцией FROM и подлежащие изменению.

Секция SET является обязательной. В ней записывается имя столбца, изменяемой таблицы, символ операции и выражение, которое может быть интерпретировано, как значение.

Операция Назначение

= присвоить значение

= default присвоить значение по умолчанию

+= присвоить сумму (конкатенация для символьных данных)

-= присвоить разницу

\*= присвоить произведение

/= присвоить частное

%= присвоить остаток от деления

&= присвоить результат побитового логического AND

^= присвоить результат побитового логического XOR

|= присвоить результат побитового логического OR

Принцип применения секции FROM оператора UPDATE аналогичен принципу применения второй секции FROM оператора DELETE: строки целевой таблицы, связанные JOIN-соединением с другими таблицами, являются исходными для выполнения изменений, определенных выражениями в секции SET.

Секция WHERE выполняется после секции FROM и содержит логическое выражение для фильтрации ее результирующего набора строк. Если секция FROM не используется, то логическое выражение применяется для фильтрации строк целевой таблицы.

Отобранные таким образом строки изменяются в соответствии с выражениями в секции SET.

Top

Output

Применение секции OUTPUT аналогично ее применению в других DML-операторах. Единственное отличие заключается в возможности использования префиксов INSERTED и DELETED в выражениях секции.

С пом. INSERTED можно вывести в результирующий набор новые значения (измененные) в столбцах таблицы, а с пом. DELETED – старые (до изменения).

14. Группировка в операторе SELECT, применение секции HAVING.

GROUP BY – секция, позволяющая сгруппировать данные по значениям.

Секция HAVING имеет такое же назначение, что и WHERE, но применяется после выполнения секции GROUP BY.

Использовать WHERE в запросах с агрегатными функциями нельзя, для этого и был введен HAVING.

Например, следующий запрос записан некорректно:

SELECT name, SUM(salary) FROM Employees

WHERE SUM(salary) > 1000

GROUP BY name

В данном случае больше подходит HAVING:

SELECT name, SUM(salary) FROM Employees

GROUP BY name HAVING SUM(salary) > 1000

Для сортировки результирующего набора применяется секция ORDER BY.

Опции ASC и DESC позволяют указать порядок сортировки строк (по возрастанию или по убыванию).

Секция INTO позволяет на основе результирующего набора создать новую таблицу.

В результате выполнения каждой секции формируются промежуточные результирующие наборы.

Опция DISTINCT позволяет избавиться от повторяющихся строк в результирующем наборе, а TOP – ограничить в нем количество строк в результате.

Если секция FROM отсутствует, то в результирующем наборе формируется только одна строка.

Заданные имена (их называют псевдонимами или алиасами) указываются через ключевое слово AS (оно может быть опущено).

Псевдоним можно указать также с применением квадратных скобок.

Символ \* (звездочка) используется для обозначения списка с именами всех столбцов FROM-набора.

Если не указаны псевдонимы, то имена столбцов результирующего набора совпадают с именами столбцов источника.

Допустимо применение TOP совместно с DISTINCT. В этом случае сначала устраняются повторяющиеся строки (выполняется DISTINCT), а затем выбирается заданное количество первых строк (TOP).

Можно указать количество строк в процентах:

15. Использование в запросах GROUP BY CUBE, GROUP BY ROLLUP.

Аналитическими запросами к базе данных принято называть запросы, сводные (агрегатные) результаты которых вычисляются над данными, хранящимися в таблицах базы данных.

SELECT sum(Количество) [Общее количество]

FROM Товары;

select Товар,

sum(Количество)[Количество по категориям]

from Товары

group by Товар;

select Товар,

Цвет,

sum(Количество) [Количество по цвету]

from Товары

group by Товар, Цвет

Эти же результаты можно получить при выполнении единственного запроса, если в его формулировке использовать специальный вид группировки ROLLUP.

Конструкция ROLLUP используется в секции GROUP BY и служит для вычисления агрегатных значений над подмножествами строк.

ROLLUP возвращает комби¬нацию групп и итоговых строк, которая опре¬делена в порядке, в котором заданы группируемые столбцы.

select Товар,

Цвет,

sum(Количество) [Количество по категориям и цвету]

from Товары

group by Rollup(Товар, Цвет);

Конструкция CUBE также используется в секции GROUP BY. Возвращает любую возможную комби-нацию групп и итоговых строк.

Правило формирования результирующего набора SELECT-запроса, применяющего в секции GROUP BY конструкцию ROLLUP.

- Результирующий набор содержит n + 1 групп строк, где n – количество выражений для группировки столбцов, указанных в скобках за ключевым словом ROLLUP.

- Первая группа строк является результатом выполнения группировки по всем n выражениям.

- Вторая группа строк является результатом группировки первой группы строк по n - 1 первым выражениям. Причем столбцы, по которым не выполнялась группировка, заполняются значениями NULL.

- Группа строк k является группировкой группы строк, полученной на предыдущем этапе по n – k + 1 первым выражениям.

- Последняя (n + 1)-я группа содержит одну строку.

select Товар,

Цвет,

sum(Количество) [Количество по категориям и цвету]

from Товары

group by Cube (Товар, Цвет);

Правило формирования результирующего набора SELECT-запроса, применяющего в секции GROUP BY конструкцию CUBE.

- Формируется множество всех подмножеств выражений, указанных в CUBE-списке.

- Для каждого непустого подмножества, сформированного в пункте выше, выполняется группировка.

- Если количество элементов подмножества меньше количества элементов CUBE-списка, то соответствующие значения в строках заполняются значением NULL (как это делалось конструкцией ROLLUP). Сформированные строки помещаются в результирующий набор секции GROUP BY.

- Для пустого подмножества выполняется группировка, аналогичная той, что выполнялась для ROLLUP.

16. Применение в запросах COMPUTE, COMPUTE BY.

Секции COMPUTE и COMPUTE BY предназначены для формирования дополнительных результирующих наборов с итоговыми строками.

Применение этих секций возможно только для БД, управляемых Microsoft SQL Server, так как они предусмотрены стандартами SQL.

Назначение секции COMPUTE BY – вычисление промежуточных итогов. Секция может применяться только совместно с секцией ORDER BY.

Эти секции не рекомендуется использовать, потому что они отмечены как недопустимые в следующих версиях Database Engine. Вместо этого надо применять GROUP BY и ROLLUP.

17. SQL-команды: UNION, UNION ALL, INTERSECT, EXCEPT.

Оператор UNION позволяет объединить результирующие наборы, сформированные двумя SELECT-запросами в общий результирующий набор.

Единственным требованием, предъявляемым к SELECT-запросам, является совместимость типов соответствующих столбцов сформированных результирующих наборов.

- Результирующий набор оператора UNION содержит объединение строк результирующих наборов первого и второго SELECT-запроса.

- Имена столбцов результирующего набора оператора UNION совпадают с именами столбцов результирующего набора, сформированного первым SELECT-запросом.

- Количество столбцов, результирующих наборов первого и второго SELECT-запросов должно быть одинаковое.

Оператор UNION является коммутативным: результат не зависит от порядка объединяемых наборов.

В операторе UNION типы могут преобразовываться автоматически, если это возможно.

Оператор UNION выполняет теоретико-множественную операцию объединения, т.е. результатом является множество, в котором строки не могут повторяться:

Если требуется механическое объединение строк, можно применить оператор UNION ALL.

Чаще всего операторы UNION и UNION ALL используются для объединения результатов SELECT-запросов к разным таблицам.

Операторы UNION и UNION ALL допускают применение секции ORDER BY.

После ключевого слова BY в секции ORDER можно указывать псевдонимы, назначенные в первом SELECT-списке или просто номер элемента списка:

Результатом оператора INTERSECT является набор строк, являющийся пересечением (в теоретико-мно-жественном смысле) двух исходных наборов строк.

Соответствующие столбцы должны быть совместимы по типам.

1. В исходных наборах строк должно быть одинаковое количество столбцов.

2. Результирующий набор оператора INTERSECT содержит строки, входящие одновременно и в первый и во второй исходные наборы данных.

3. Результатом является множество неповторяющихся строк.

4. Имена и типы столбцов результирующего набора совпадают с именами и типами столбцов первого результирующего набора.

Оператор INTERSECT имеет более высокий приоритет, чем оператор UNION. Отличие предыдущего сценария от сценария ниже в отсутствии скобок, регламентирующих порядок выполнения операторов:

Оператор INTERSECT коммутативный

Оператор EXCEPT

Результатом оператора EXCEPT является набор строк, являющийся разностью (в теоретико-множественном смысле) двух исходных наборов строк.

Соответствующие столбцы должны быть совместимы по типам.

1. В исходных наборах строк должно быть одинаковое количество столбцов.

2. Результирующий набор оператора EXCEPT содержит строки, которые входят в первый набор строк, но не входят во второй.

3. Результатом является множество неповторяющихся строк.

4. Имена и типы столбцов результирующего набора оператора EXCEPT совпадают с именами и типами столбцов первого результирующего набора.

Оператор EXCEPT не коммутативный: результат зависит от порядка исходных наборов:

Наивысшим приоритетом обладает оператор INTERSECT, а операторы UNION и EXCEPT имеют одинаковый приоритет.

18. Внутреннее соединение таблиц: INNER JOIN. Многотабличные запросы.

SELECT \* FROM Person INNER JOIN City

ON Person.CityId = City.Id

Результирующий набор SELECT-запроса создается по следующим правилам:

 выполняется декартово произведение таблиц;

 из результирующего набора выбираются строки, удовлетворяющие условию логического выражения, указанного после ON;

 из всех столбцов результирующего набора выбираются столбцы, указанные в списке SELECT.

Операция INNER JOIN является коммутативной – формируемый результирующий набор не зависит от порядка, в котором указаны таблицы.

Логическое условие, указываемое после ключевого слова ON, не обязательно должно содержать условие, записываемое в виде равенства, – это может любое логическое выражение

С помощью INNER JOIN можно соединять таблицу саму с собой.

Обычно такие запросы используются для получения информации об иерархических связях между строками одной таблицы.

Ограничение FOREIGN KEY, указанное для таблицы [Руководители и Исполнители], ссылается на эту же таблицу.

19. Внешнее соединение таблиц: OUTER JOIN (LEFT, RIGHT, FULL). Соединение CROSS JOIN.

Соединение таблиц OUTER JOIN

Внешнее соединение двух таблиц формирует набор строк, состоящий из двух частей:

 результат внутреннего соединения двух таблиц;

 строки одной из двух таблиц, которые не смогли соединиться.

Значения в столбцах, соответствующие незаполненной (несоединенной) части строки, будут равны NULL.

Левое внешнее соединение LEFT OUTER JOIN включает в набор соединенные строки и несоединенные строки таблицы, имя которой записано слева от ключевых слов LEFT OUTER JOIN.

SELECT \* FROM

Person LEFT OUTER JOIN City

ON Person.CityId = City.Id

Три строки соединились и добавлена одна несоединенная строка из таблицы слева.

Вывести фамилии преподавателей, список кафедр, на которых они работают, и количество строк в результирующем наборе.

Количество несоединенных строк в таблице PULPIT и сами несоединенные строки таблицы (22 строки) можно определить с помощью запросов

Right как Left.

Полное внешнее соединение (FULL OUTER JOIN) определяется как объединение левого и правого внешних соединений.

При этом все строки из обеих таблиц будут представлены в результирующем наборе данных.

Если нет соответствующей строки для одной из таблиц, ее столбцы будут возвращены с пустыми значе¬ниями NULL.

SELECT \* FROM

Person FULL OUTER JOIN City

ON Person.CityId = City.Id

Три строки соединились и добавлены по одной несоединенной строке из таблицы справа и слева.

Операция коммутативна.

Допускается применение подзапросов:

SELECT (select COUNT(\*) from Заказы)

'Кол-во строк в табл. Заказы',

(select COUNT(\*) from Товары)

'Кол-во строк в табл. Товары',

(select COUNT(\*) from Товары at

FULL OUTER JOIN Заказы aa

on aa.Наименование\_товара = at.Наименование)

'Кол-во строк в соединении'

CROSS JOIN

Декартово произведение двух таблиц, где первая таблица содержит n строк, а вторая  m строк, создаст результат с количеством строк, равным n х m.

Используется редко.

20. Подзапросы: (NOT) IN, (NOT) EXISTS, ALL, ANY, SOME.

Для работы с результирующими наборами, представляющими собой множество однородных значений (значений одного типа), можно использовать операции IN, ANY и ALL.

Операция IN формирует логическое значение «истина» в том случае, когда значение, указанное слева от ключевого слова IN (значение в столбце NOTE таблицы PROGRESS), равно хотя бы одному из значений списка, указанного справа от IN (одно из значений 7, 9 или 10).

Выражение IN может быть заменено выражением, использующим операцию OR:

Операция ALL предназначена для сравнения со списком значений, который является результатом выполнения подзапроса.

Операция > ALL сформирует истинное значение в том случае, если значение стоящее слева (значение в столбце NOTE таблицы PROGRESS) больше каждого значения в списке, указанном справа (значения в столбце, сформированном подзапросом).

Операцию ALL можно использовать со всеми операциями сравнения и в сочетании с другими логическими операциями.

Операция ANY записывается и применяется аналогично операции ALL, но результат формируется иным образом.

Здесь стоящее слева значение с каждым значением списка, записанным справа (содержимое столбца, сформированного подзапросом). Если хотя бы для одного значения из списка операция сравнения принимает значение «истина», то вся операция (<=ANY) принимает значение «истина».

Операция EXISTS

Иногда в подзапросе, который используется в секции WHERE, бывает достаточно выяснить только сам факт наличия строк в результирующем наборе.

В таких случаях удобно использовать операцию EXISTS.

В скобках, следующих за ключевым словом EXISTS, записывается подзапрос. Результатом операции является «истина», если результирующий набор подзапроса содержит хотя бы одну строку, в противоположном случае операция EXISTS формирует значение «ложь».

Как и другие логические операции, EXISTS может использоваться в логических выражениях.

!!!!!!!Если результат выполнения подзапроса не содержит строк и определен параметр ALL, результат равен TRUE. Если же определен параметр SOME, результат равен FALSE.

ANY=SOME

21. Временные таблицы: локальные, глобальные, работа с временными таблицами. Примеры.

Временные таблицы

Отличие временных таблиц от постоянных в том, что они хранятся в системной БД TEMPDB и не могут иметь внешних ключей.

Как правило, временные таблицы создаются для временного хранения результатов SELECT-запросов.

База данных TEMPDB создается при каждом перезапуске сервера, поэтому сохранить или восстановить временную таблицу в случае сбоя невозможно.

Существует два вида временных таблиц: локальные и глобальные.

Локальные временные таблицы имеют имена, начинающиеся с символа #, доступны только пользователю, ее создавшему, и могут быть удалены с помощью оператора

DROP TABLE.

Если пользователь временную таблицу не удалил сам, то она удалится автоматически при отключении.

Глобальные временные таблицы имеют имена, начинающиеся с символов ##, доступны всем пользователям, подключенным к серверу, и могут быть удалены с помощью оператора DROP TABLE.

Обычно глобальные временные таблицы применяются для обмена данными между несколькими сеансами.

22. Представления: создание, применение, WITH CHECK OPTION, DML-операции, ORDER BY.

Представление (англ. View) – это объект БД, представляющий собой поименованный SELECT-запрос.

В БД хранится именно SELECT-запрос, а не результат его выполнения.

Представление создается с помощью оператора CREATE, удаляется с помощью оператора DROP и изменяется с помощью ALTER.

В операторе CREATE после ключевого слова VIEW следует имя представления и далее после ключевого слова AS текст SELECT-запроса, лежащего в основе представления.

При этом к SELECT-запросу представления предъявляется следующие требования:

• секцию ORDER BY можно использовать только совместно с опцией TOP;

• не допускается применение секции INTO, секций COMPUTE и COMPUTE BY;

• все столбцы результирующего набора, формируемого SELECT-запросом, должны быть поименованы.

Изменение представления с помощью ALTER бывает полезным в тех случаях, когда пользователю были назначены определенные разрешения относительно этого представления. В случае удаления старого и создания нового представления с тем же именем, все действующие разрешения будут утеряны.

При использовании ALTER разрешения будут сохранены.

Привязка к базовым таблицам

Удаление или изменение таблиц или представлений, используемых в SELECT-запросе, на котором основано представление, может повлечь за собой потерю работоспособности представления.

Для предотвращения проблемы применяется специальная опция SCHEMABINDING, с помощью которой устанавливается запрещение на операции с таблицами и представлениями.

При применении опции SCHEMABINDING требуется использовать в SELECT-запросе для имен таблиц и представлений двухкомпонентный формат.

Если первый компонент не известен, то используется стандартное имя схемы DBO.

Схема – это поименованный контейнер объектов БД, позволяющий разграничить объекты с одинаковыми именами. Схемы имеют такой же смысл, как пространства имен в языках программирования. Допускается использование двух таблиц с одинаковыми именами, но они должны находиться в разных схемах.

Удаление таблицы PROGRESS и представления Экзамен не выполнено из-за того, что эти объекты связаны с представлениями.

Для простых представлений допустимо выполнение DML-операций INSERT, DELETE и UPDATE. При создании представлений, позволяющих выполнять эти операции базовый SELECT-запрос должен удовлетворять правилам:

1. Запрос не должен содержать секцию группировки GROUP BY и агрегатные функции.

2. Запрос не должен использовать опции DISTINCT и TOP.

3. Запрос не должен использовать операторы UNION, INTERSECT и EXCEPT.

4. В SELECT-списке запроса не должно быть вычисляемых значений.

5. В секции FROM должна указываться одна таблица.

Существующие ограничения целостности базовой таблицы наследуются представлением.

Для того чтобы предотвратить несогласованность при создании или изменении представления с секцией WHERE указывается специальная опция CHECK OPTION.

При этом появляется еще одно ограничение, основанное на логическом выражении в секции WHERE базового SELECT-оператора.

23. Агрегатные функции: MIN, MAX, SUM, COUNT, AVG.

Наиболее часто используемые агрегатные функции:

AVG - вычисление среднего значения,

COUNT - вычисление количества строк,

MAX - вычисление максимального значения,

MIN - вычисление минимального значения,

SUM - вычисление суммы значений.

24. Transact-SQL: DECLARE, локальные переменные, инициализация, глобальные переменные, оператор PRINT.

TRANSACT-SQL – это процедурное расширение языка SQL (в SQL отсутствуют операторы перехода, цикла, ветвления и др.).

Применение языка T-SQL обусловлено современной тенденцией в программировании, которая заключается в переносе программного обеспечения на сторону сервера. Клиентское программное обеспечение на компьютере пользователя в основном решает задачи ввода и отображения данных.

T-SQL позволяет создавать программные объекты БД: хранимые процедуры, скалярные и табличные функции, триггеры.

Начиная с версии MSS 2005 года, разработчику предоставляется возможность разрабатывать программные объекты БД на языке C#.

Начиная с версии 2008 не различаются языки SQL и Т-SQL, все называется Transact-SQL.

Операторы языка T-SQL отделяются друг от друга точкой с запятой. Если операторы расположены на разных строках, то допускается не применять разделители.

Допускается любое сочетание букв верхнего и нижнего регистров.

Оператор DECLARE

Оператор DECLARE предназначен для объявления используемых в программе переменных. Для каждой переменной указывается имя и тип.

Имя должно начинаться с символа @ (но не c @@ , так как это сочетание символов зарезервировано для системных переменных и функций).

Переменная может быть проинициализирована в операторе DECLARE с помощью символа =.

В операторе DECLARE можно объявлять переменную с типом TABLE.

Этот тип позволяет создавать таблицы в памяти и использовать их для хранения промежуточных данных.

Переменной, объявленной в операторе DELCLARE, можно присвоить значение тремя способами:

- инициализировать в операторе DECLARE;

- присвоить значение с помощью оператора SET;

- присвоить значение с помощью SELECT.

Отличие операторов присваивания SET и SELECT заключается в возможности оператора SELECT присваивать значения одновременно нескольким переменным.

SQL Server позволяет использовать два вида переменных – локальные и глобальные. Имя локальной переменной начинается с одного символа глобальной – с двух "@@". Глобальные переменные позволяют только прочитать значение. Локальные переменные можно создавать, присваивать им значения и читать их.

Вывод данных в T-SQL возможен двумя способами:

- с помощью оператора SELECT можно сформировать выходной результирующий набор;

- с помощью оператора PRINT можно вывести строку в стандартный выходной поток.

Если одновременно выводятся данные, сформированные оператором SELECT и данные, сформированные оператором PRINT, то посмотреть последние можно на вкладке Messages.

25. Transact-SQL: оператор IF-ELSE, операторные скобки BEGIN/END, оператор RETURN, конкатенация строк, преобразование типов CAST.

Преобразование типов

Операции со строками

Немедленно завершить работу пакета можно с помощью оператора RETURN.

В первом пакете выполняется оператор RETURN, осуществляющий немедленное завершение этого пакета и переход на выполнение следующего пакета.

Кроме выхода из текущего пакета, оператор RETURN применяется для завершения работы процедур, функций и триггеров.

Операторные скобки BEGIN END

В некоторых случаях в сценарии T-SQL появляется необходимость указать, что группа следующих друг за другом операторов должна выполняться как единое целое. Это делается с помощью BEGIN END.

Объединение операторов T-SQL в блоки с помощью BEGIN END не влияет на действие оператора RETURN (в любом случае осуществляется выход из пакета) и на область видимости переменных (область видимости распространяется строго сверху вниз).

Оператор ветвления IF позволяет в зависимости от значения логического выражения выполнять или не выполнять оператор или блок операторов сценария T-SQL.

За логическим выражением оператора IF располагается оператор, выполняющийся в случае истинности значения логического выражения.

Если при этом требуется выполнить оператор при альтернативном значении логического выражения, следует применить конструкцию IF ELSE.

Сократить количество вычислений значений логических выражений позволяет конструкция IF ELSE IF, в которой проверка осуществляется только в ветке ELSE оператора IF.

Для выполнения нескольких операторов применяются операторные скобки:

26. Transact-SQL: конструкции CASE, WHILE, WAITFOR.

Оператор WHILE содержит две составляющие: логическое выражение и тело цикла. Логическое выражение определяет условие выполнения тела цикла. Тело цикла содержит один или более операторов, которые выполняются в том случае и до тех пор, пока логическое выражение принимает значение «истина».

Если тело цикла содержит более одного оператора, следует применять операторные скобки BEGIN END.

Инструкция CONTINUE прерывает выполнение блока операторов в теле WHILE и передает управление в начало цикла.

Если в теле цикла выполнилась инструкция BREAK, работа цикла прекращается и управление передается следующему за WHILE оператору.

Выражение CASE

Выражение CASE, как правило, применяется в операторе SELECT и служит для формирования одного из нескольких возможных значений.

Может применяться в операторах UPDATE (в секциях SET и WHERE), DELETE (в секции WHERE).

При записи сразу за CASE следует вычисляемое выражение, затем одно или более предложений WHEN, содержащих значения для сравнений.

Ключевое слово THEN указывает на один из возможных результатов, формируемых CASE.

Список WHEN-предложений завершается предложением ELSE, содержащим результат выражения CASE, формируемого в случае безуспешного сравнения значения исходного выражения со всеми предшествующими WHEN-значениями.

CASE имеет два формата:

первый сравнивает результат одного выражения со списком заданных значений,

второй – вычисляет набор логических выражений.

Применение CASE в GROUP BY позволяет формировать группы строк на основе заданного в этом выражении условия.

Выражение CASE в ORDER BY позволяет динамически формировать значения ключей сортировки.

Здесь каждое предложение WHEN содержит логическое выражение.

Логические выражения проверяются на истинность сверху вниз, и при первом успешном сравнении формируется результирующее значение, указанное за ключевым словом THEN. В том случае, если ни одно из логических WHEN-выражений не принимает истинного значения, в качестве результата CASE формируется значение, указанное в предложении ELSE.

Оператор WAITFOR

С помощью оператора WAITFOR можно приостановить выполнение пакета.

Оператор работает в двух режимах: DELAY и TIME.

В режиме DELAY оператор WAITFOR приостанавливает выполнение пакета на заданный интервал времени.

Здесь последовательно выполняются два оператора WAITFOR DELAY. Первый приостанавливает выполнение пакета на 2 секунды, второй на 3 секунды.

В режиме TIME можно приостановить выполнение пакета до наступления заданного значения системного времени на сервере.

Здесь последовательно выполняются два оператора WAITFOR TIME. Каждый из них приостанавливает выполнение пакета до момента времени указанного значением переменной.

Переменным при инициализации устанавливается значение с помощью встроенной функции SYSDATETIME, возвращающей текущее системное время сервера MSS.

Переменной @t1 устанавливается время со смещением в 2 секунды относительно текущего времени, переменной @t2 – со смещением в 3 секунды.

27. Transact-SQL: TRY/CATCH, процедура RAISEERROR.

Для обработки ошибок выполнения в сценарии T-SQL предусмотрена конструкция, состоящая из двух блоков: TRY и CATCH.

Блок TRY содержит код T-SQL, в котором могут возникнуть ошибки (охраняемый код), а блок CATCH – код, предназначенный для обработки ошибок.

Ошибка, возникающая в охраняемом коде, приводит к передаче управления в блок обработки ошибок.

В блоке CATCH можно использовать несколько системных функций, позволяющих диагностировать возникшую в охраняемом коде ошибку.

Наименование функции Функция возвращает

ERROR\_NUMBER код последней ошибки (int)

ERROR\_MESSAGE сообщение об ошибке (char)

ERROR\_SEVERITY уровень серьезности ошибки (int)

ERROR\_STATE метку ошибки (int)

ERROR\_LINE номер строки с ошибкой (int)

ERROR\_PROCEDURE имя процедуры или NULL (char)

Можно сгенерировать сообщение об ошибке с помощью специальной инструкции RAISERROR. При вызове этой инструкции надо передать три параметра: текстовое сообщение об ошибке, уровень серьезности ошибки и метку.

Здесь инструкция RAISERROR применяется дважды: в первом случае для предупреждения об ошибке при делении на нуль, во втором – об ошибке при нарушении ограничения PRIMARY KEY.

Инструкция ведет себя по-разному, в зависимости от заданного параметром уровня серьезности ошибки. Выполнение может продолжится, а может нет.

28. Индексы: определение, типы, назначение. Кластеризованные и некластеризованные индексы.

Обработка SQL-запроса, поступившего серверу MSS, проходит в несколько этапов: разбор, разрешение имен, оптимизация, компиляция и выполнение.

Обработка любого SQL-запроса начинается с синтаксического разбора текста запроса для проверки его на соответствие правилам языка.

На этапе разрешения имен осуществляется проверка наличия используемых в запросе объектов БД (таблиц, столбцов таблиц, пользовательских и встроенных функций и пр.).

Запрос, прошедший этапы разбора и разрешения имен, поступает на обработку специальной компоненте сервера, называемой оптимизатором. Основная задача оптимизатора – построение плана запроса. План запроса представляет собой алгоритм выполнения SQL-запроса.

План запроса можно посмотреть, выполнив в контекстном меню запроса команду

Display Estimated Execution Plan

(Показать предполагаемый план выполнения)

При построении плана запроса для каждого шага вычисляется числовая величина, пропорциональная продолжительности выполнения шага, называемая стоимостью.

Суммарная стоимость шагов плана составляет стоимость всего запроса и, соответственно, является величиной, пропорциональной продолжительности выполнения запроса.

Критерием оптимизации при построении плана является минимизация общей стоимости запроса.

Оптимизатор, формирующий план запроса, учитывает статистику, собираемую сервером СУБД, а также наличие специальных объектов БД, предназначенных для ускорения выполнения запросов и называемых индексами.

После построения плана запроса осуществляется его компиляция. Откомпилированный запрос (точнее план запроса), помещается в область памяти, называемую библиотечным кэшем.

Кэш используется для ускорения выполнения будущих аналогичных запросов.

Откомпилированный план выполняется сервером СУБД.

Индекс – это объект БД, позволяющий ускорить поиск в определенной таблице.

Для получения данных из таблицы наличие индексов не является обязательным.

Если нет подходящего индекса, оптимизатор запланирует последовательный перебор всех строк таблицы (просканирует таблицу) для поиска необходимых данных.

Если индекс не позволяет ускорить процесс поиск данных – он является бесполезным.

Как и любой другой объект базы данных, индекс может быть создан с помощью CREATE, модифицирован с помощью ALTER и удален с помощью оператора DROP.

Для одной таблицы может быть построено несколько индексов.

Структура, используемая MSS для хранения индексов в БД, представляет собой сбалансированное дерево, узлами которого являются страницы – стандартные блоки данных размером 8К.

У бинарных деревьев поиска значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева - больше значения в самой вершине.

Дерево является сбалансированным, если для каждого узла высота его двух поддеревьев различается не более чем на 1.

MSS поддерживает несколько типов индексов: кластеризованные и некластеризованные индексы, уникальные и неуникальные, XML-индексы и др.

Кластеризованные индексы

Основной особенностью кластеризованного индекса является то, что строки таблицы, имеющей кластеризованный индекс, физически упорядочены в соответствии со значениями индексируемого столбца (столбцов).

В таблице может быть только один кластеризованный индекс.

Наличие первичного ключа (ограничение PRIMARY KEY) в таблице БД влечет автоматическое создание кластеризованного индекса при создании таблицы.

Кластеризованный индекс является уни¬кальным по умолчанию, т. е. каждое значение данных может появиться толь¬ко однажды в столбце.

Если в свойствах столбца первичного ключа указано ключевое слово NONCLUSTERED, то кластеризованный индекс не создается.

С помощью системной процедуры SP\_HELPINDEX можно получить перечень индексов, связанных с заданной таблицей.

В столбце index\_description – описание индекса (кластеризованный, уникальный, созданный для первичного ключа, расположен в файловой группе FG1), в столбце index\_keys - имя столбца (TEACHER) таблицы, по значениям которого осуществляется построение индекса.

Пример. Создать временную таблицу #EXPLORE и заполнить 10000 строк.

В столбец TKEY с помощью встроенной функции RAND вводится случайное числовое значение (floor - нижнее целое, replicate - повторить строку).

Таблица #EXPLORE не имеет индексов. Поэтому SELECT-запрос, выполняющий поиск и сортировку строк, с заданным диапазоном значений столбца TKEY приводит к полному сканированию таблицы (компонент Table Scan плана запроса) #EXPLORE и последующей сортировке (компонент Sort).

Здесь оператор CHECKPOINT позволяет записать образы страниц из буферного кэша в файлы БД.

После выполнения этого оператора все образы страниц, расположенные в буферном кэше сервера, и соответствующие им страницы в файлах БД совпадают.

Оператор DBCC DROPCLEANBUFFER позволяет очистить буферный кэш.

После выполнения этого оператора буферный кэш не содержит никаких образов страниц, и поэтому следующая DML-операция приведет к физическому чтению страниц из файлов БД.

Совместное применение этих двух операторов позволяет более объективно сравнивать время выполнения нескольких запросов.

Общая стоимость запроса (Estimate Subtree Cost) составляет 0,174. На сканирование таблицы пришлось 58% стоимости, на сортировку – 42%.

По столбцу tkey можно создать индекс:

Создается кластеризованный индекс с именем #EXPLORE\_CLU по столбцу TKEY таблицы #EXPLORE.

Ключевое слова ASC указывает порядок, в котором строится индекс.

Сканирование осуществляется не по всей таблице, а по части кластеризованного индекса (компонент Clustered Index Seek). Отсутствует компонент сортировки. При этом стоимость запроса (Estimate Subtree Cost) составляет 0,035, что почти в 5 раз меньше, чем при запросе к таблице, не имеющей индекса.

Пятикратное повышение скорости выполнения запроса достигается за счет следующего.

- Выбор строк таблицы #EXPLORE по диапазону значению столбца TKEY не требует сканирования всей таблицы. Индекс по столбцу TKEY позволяет серверу сразу позиционироваться на первую строку выбираемой группы строк и сканировать только строки, соответствующие заданному диапазону.

- Строки таблицы #EXPLORE уже расположены в порядке возрастания значений столбца TKEY, так как она сама является частью индекса. Поэтому сортировка не требуется.

В индексируемом столбце TKEY таблицы #EXPLORE есть повторения. Такие индексы называются неуникальными.

Если удалить неуникальный кластеризованный индекс #EXPLORE\_CLU и создать уникальный индекс #EXPLORE\_UNIQ\_CLU по столбцу TKEY, то возникнет ошибка из-за того, что при создании уникального индекса в индексируемом столбце были обнаружены дубликаты значений:

Создание уникального индекса по другому столбцу CC, содержащему только уникальные значения (столбец обладает свойством IDENTITY), завершается успешно:

Но время выполнения запроса большое (0,1788), т. к. здесь в соответствии с планом запрос будет осуществляться в два этапа:

- полное сканирование кластеризованного индекса #EXPLORE\_UNIQ\_CLU для выбора значений, соответствующих WHERE-условию;

- сортировка выбранных строк.

В запросе ниже время выполнения запроса меньше (0,098).

Здесь сортировка выполняется по значениям индексируемого столбца CC. Операции сортировки не требуется, так как строки таблицы #EXPLORE являются частью кластеризованного индекса #EXPLORE\_UNIQ\_CLU и уже находятся в требуемом порядке.

Если изменить WHERE-условие таким образом, чтобы фильтрация строк осуществлялась по значениям индексируемого столбца, то стоимость запроса еще больше снизится (0,02):

Некластеризованные индексы

Некластеризованные индексы следует использовать только тогда, когда есть уверенность, что производительность повысится.

Некластеризованные индексы в отличие от кластеризованных никак не влияют на физический порядок строк в таблице.

Для одной таблицы БД допускается создавать до 1023 некластеризованных индексов.

Индексы, соответствующие ограничению PRIMARY KEY, не являются кластеризованными, если используется nonclustered.

Пример создания таблицы TEACHER, имеющей столбец со свойство PRIMARY KEY и соответствующим некластеризованным индексом.

Во всех приводимых до сих пор примерах, применялись индексы, созданные по одному столбцу таблицы. Такие индексы называются простыми.

MSS допускает создавать индексы по нескольким столбцам – такие индексы называются составными.

Составными могут быть кластеризованные и некластеризованные индексы. Например, составному первичному ключу по умолчанию соответствует составной кластеризованный индекс.

Пример создания составного неуникального, некластеризованного индекса по двум столбцам TKEY и CC таблицы #EXPLORE.

Некластеризованный составной индекс не применяется оптимизатором ни при фильтрации, ни при сортировке строк таблицы #EXPLORE. В обоих случаях осуществляется сканирование всей таблицы (на плане – Table Scan).

Не используется при фильтрации по индексируемым столбцам

Не используется при сортировке по индексируемым столбцам

Но, если хотя бы одно из индексируемых значений зафиксировать (задать одно значение), то оптимизатор применит индекс.

Наиболее часто составной индекс применяется при группировке и вычислениям по индексируемым столбцам.

При группировке с вычислениями по двум индексируемым столбцам хранилище строк не используется, выборка данных осуществляется через сканирование индекса.

Индекс применяется лишь в тех случаях, когда в SELECT-списке и в WHERE-выражении используется один тот же индексируемый столбец.

Иначе выполняется сканирование всей таблицы.

Совместное применение кластеризованного

и некластеризованного индексов

Пусть созданы некластеризованный и кластеризованный индексы в таблице #EXPLORE:

Кластеризованный индекс построен по столбцу CC, а некластеризованный по столбцу TKEY.

Первый запрос:

В первом случае выполнение запроса сводится к позиционированию в некластеризованном индексе #EXPLORE\_TKEY и считыванию строк в соответствии с диапазоном, заданным во WHERE-выражении. При этом значения столбца CC будут храниться в строках некластеризованного индекса #EXPLORE\_TKEY.

Во втором случае некластеризованный индекс не применяется. Для извлечения значений столбца TFIELD оптимизатор использовал кластеризованный индекс.

Таким образом, избежать полного сканирования таблицы можно с помощью кластеризованного, некластеризованного индексов или их совместного применения.

29. Индексы покрытия, уникальные некластеризованные индексы, фильтруемые некластеризованные индексы.

Обычно кластеризованные индексы создаются автоматически при создании таблицы если в ней присутствует первичный ключ (ограничение PRIMARY KEY).

Кластеризованные индексы физически упорядочены в соответствии со значениями индексируемых столбцов. В таблице может быть только один кластеризованный индекс

Некластеризованные индексы не влияют на физический порядок строк в таблице.

Некластеризованный индекс покрытия запроса позволяет включить в состав индексной строки значения одного или нескольких неиндексируемых столбцов. Например, индекс покрытия #EX\_TKEY\_X включает значения столбца CC (ключевое слово INCLUDE):

CREATE index #EX\_TKEY\_X on #EX(TKEY) INCLUDE (CC)

Если запросы основаны на WHERE-фильтрации строк, то может быть эффективным применение фильтруемых некластеризованных индексов.

Пусть имеется три запроса. Надо оценить планы их выполнения.

SELECT TKEY from #EX where TKEY between 5000 and 19999;

SELECT TKEY from #EX where TKEY>15000 and TKEY < 20000

SELECT TKEY from #EX where TKEY=17000

Затем можно создать фильтрующий индекс с именем, например #EX\_WHERE:

CREATE index #EX\_WHERE on #EX(TKEY) where (TKEY>=15000 and

TKEY < 20000);

Здесь фильтруемый индекс создается только для строк таблицы #EX, которые удовлетворяют логическому условию. Стоимость запросов уменьшится.

Операции добавления и изменения строк базы данных могут повлечь образование неиспользуемых фрагментов в области памяти индекса. Процесс образования неиспользуемых фрагментов памяти называется фрагментацией.

30. Перестройка и реорганизация индексов.

Реорганизация (REORGANIZE) выполняется быстро, но после нее фрагментация будет убрана только на самом нижнем уровне.

Пусть выполнена реорганизация с помощью оператора ALTER для индекса #EX\_TKEY.

ALTER index #EX\_TKEY on #EX reorganize;

Тогда выполнение соответствующего запроса покажет, что уровень фрагментации значительно снизился, но не до конца.

Операция перестройки (REBUILD) затрагивает все узлы дерева, поэтому после ее выполнения степень фрагментации равна нулю.

Пусть выполнена перестройка с помощью оператора ALTER для индекса #EX\_TKEY в режиме OFFLINE.

ALTER index #EX\_TKEY on #EX rebuild with (online = off);

Уровнем фрагментации можно в некоторой степени управлять, если при создании или изменении индекса использовать параметры FILLFACTOR и PAD\_INDEX.

Параметр FILLFACTOR указывает процент заполнения индексных страниц нижнего уровня.

31. Курсоры: определение, общая схема работы, локальные/глобальные, динамические/статические, только для чтения.

Использование курсоров

Курсор является программной конструкцией, которая служит для обработки строк результирующего набора запись за записью.

Кроме того, курсор  это механизм для хранения результата запро¬са

Работа с любым курсором осуществляется в следующей последовательности.

1. Курсор объявляется в операторе DECLARE.

2. Открывается с помощью оператора OPEN.

3. С помощью оператора FETCH считывается одна или несколько строк результирующего набора, связанного с курсором SELECT-оператора, и обрабатывается нужным образом. Результат каждого считывания проверяется с помощью системной функции @@FETCH\_STATUS.

4. Курсор закрывается оператором CLOSE.

5. Если курсор глобальный, то он должен быть освобожден с использованием оператора DEALLOCATE.

При открытии курсора с помощью оператора OPEN становится доступным для чтения результирующий набор строк, и указатель текущей строки курсора устанавливается на первую строку.

Оператор FETCH считывает одну строку из результирующего набора и продвигает указатель на следующую.

После ключевого слова INTO следует список переменных. Количество переменных в этом списке должно быть равно количеству столбцов результирующего набора строк, а порядок их должен соответствовать порядку перечисления столбцов в SELECT-списке, связанного запроса.

После выполнения FETCH следует проверить значение системной функции @@FETCH\_STATUS, которая возвращает одно из значений:

0 - оператор FETCH выполнен успешно;

-1 - достигнут конец результирующего набора, строка не считывается;

-2 - выбранная строка отсутствует в БД.

После завершения работы с курсором, он должен быть закрыт с помощью оператора CLOSE.

В тех случаях, когда курсор является глобальным (при объявлении указан атрибут GLOBAL) и не предполагается его дальнейшее применение, следует записать оператор DEALLOCATE, выполняющий освобождение всех ресурсов, выделенных курсору при его объявлении в операторе DECLARE.

Локальные и глобальные курсоры

Локальный курсор может применяться в рамках одного пакета и ресурсы, выделенные ему при объявлении, освобождаются сразу после завершения работы пакета.

Признаком того, что курсор является локальным, является атрибут LOCAL, указанный при объявлении курсора.

Глобальный курсор может быть объявлен, открыт и использован в разных пакетах. Выделенные ему при объявлении ресурсы освобождаются только после выполнения оператора DEALLOCATE или при завершении сеанса пользователя.

Для того, чтобы объявить глобальный курсор следует применить атрибут GLOBAL:

Обычно по умолчанию курсор является глобальным.

Статические и динамические курсоры

При объявлении статического курсора должен указываться атрибут STATIC.

Открытие статического курсора приводит к выгрузке результирующего набора строк во временную таблицу системной БД TEMPDB, и все дальнейшие курсорные операции осуществляются с этой таблицей.

После открытия курсора изменения в исходных таблицах, которые осуществляются в рамках этого или других сеансов, не будут отражаться в курсорном результирующем наборе.

Здесь с помощью системной функции @@CURSOR\_ROWS определяется количество строк в результирующем наборе статического курсора.

Все изменения (операторы UPDATE, DELETE и INSERT) выполненные в исходной таблице TEACHER никак не отразились на результате выборки строк курсора с помощью оператора FETCH.

Курсоры с атрибутами FOR UPDATE

и READ\_ONLY

Для курсоров с установленным свойством FOR UPDATE помимо чтения данных из строк с помощью оператора FETCH, можно эти строки изменять или удалять с помощью специального формата операторов UPDATE и DELETE.

При объявлении курсора C\_TEACHER используется атрибут FOR UPDATE:

32. SCROLL-курсоры: дополнительная навигация со SCROLL-курсорами, конструкция WHERE CURRENT OF.

По умолчанию для курсора установлен атрибут SCROLL, позволяющий применять оператор FETCH с дополнительными опциями позиционирования.

Здесь функция ROW\_NUMBER предназначена для нумерации строк результирующего набора и относится к классу оконных встроенных функций T-SQL.

Признаком оконной функции является ключевое слово OVER.

, за которым в скобках следуют опции, разбивающие результирующий набор на подмножества (окна) строк, а также сортирующие эти строки внутри подмножества.

В приведенном примере разбиение не осущ., все строки результирующего набора только сортируются. Результатом вып. функции является номер строки в результирующем наборе.

Курсор, демонстрирующий опции позиционирования оператора FETCH, доступные в SCROLL-курсорах:

Курсоры с атрибутом FORWARD\_ONLY допускают чтение строк результирующего набора только в одном порядке – сверху вниз (FETCH NEXT).

Здесь в секции WHERE операторы DELETE и UPDATE используется операция CURRENT OF, для которой указывается имя курсора. Такой формат операторов позволяет удалять или изменять строки в таблице, соответствующие текущей позиции курсора в результирующем наборе.

Курсору может быть установлено свойство READ\_ONLY, запрещающее применение операции CURRENT OF в секции WHERE операторов DELETE и UPDATE.

Однако свойство READ\_ONLY не мешает применять эти операторы без использования CURRENT OF:

Здесь для удаления и изменения строк таблицы TEACHER используются операторы DELETE и UPDATE не применяющие операцию CURRENT OF в секции WHERE.

Атрибут READ\_ONLY имеет смысл только для динамических курсоров, статические курсоры по определению применяются только для чтения.

(Использовать локальный статический курсор; в SELECT-запросе курсора применить внешнее соединение таблиц, использовать встроенные функции RTRIM, SUBSTRING, LEN).

use BSTU

go

declare report cursor local for

select f.FACULTY, p.PULPIT,

isnull(s.SUBJECT, 'нет') SUBJECT

from FACULTY f join PULPIT p

on f.FACULTY = p.FACULTY

left outer join SUBJECT s

on p.PULPIT = s.PULPIT

order by f.FACULTY, p.PULPIT, SUBJECT;

declare @f varchar(50), @p varchar(50),

@s varchar(100);

declare @fx varchar(50) = '###',

@px varchar(50) = '###', @kx int = 0,

@sx varchar(200)= '';

open report

fetch report into @f, @p, @s;

while @@FETCH\_STATUS = 0

begin

if(@fx != '###' and

(@fx != @f or @px != @p))

begin

set @sx= SUBSTRING(@sx,1,LEN(@sx)-1)+ '.';

print ' Дисциплины: ' + @sx;

set @sx = '';

end;

if(@fx != @f)

begin

select @fx = @f;

print 'Факультет: '+ @fx

end;

if(@px != @p)

begin

set @px = @p;

print ' Кафедра:'+ @px;

set @kx = (select count(\*) from TEACHER where PULPIT =@px);

print ' Количество преподавателей: ' + cast(@kx as varchar(10));

end;

set @sx = @sx+ rtrim(@s)+ ', ';

fetch report into @f, @p, @s;

end;

close report;

go

33. Транзакции: определение, назначение, свойства ACID.

Состояние БД называется согласованным, если данные удовлетворяют всем ограничениям целостности и не противоречат друг другу

Рассогласование будет происходить в том случае, когда при необходимости одновременного выполнения нескольких изменений в БД выполнится только их часть, а другая часть не выполнится вообще.

Нужен механизм, позволяющий объединять несколько операторов изменения БД в одну атомарную операцию, которая либо полностью выполнялась, либо полностью не выполнялась.

Транзакция - это механизм БД, позволяющий таким образом объединять несколько операторов, изменя-ющих БД, чтобы при выполнении этой совокупности операторов они или все выполнились или все не выполнились.

аббревиатура ACID:

Atomicity – атомарность,

Consistency – согласованность,

Isolation – изолированность,

Durability – долговечность

Атомарность транзакции заключается в гарантии того, что никакая транзакция не будет зафиксирована в БД частично: операторы изменения БД, включен-ные в транзакцию, либо выполнятся все, либо не выполнится ни один. В БД это свойство реализуется с помощью механизма отката, позволяющего отме-нить выполненные, но незафиксированные изменения.

Согласованность гарантирует, что изменение данных в рамках одной транзакции переводит базу дан-ных из одного согласованного состояния в другое согласованное состояние.

Современные СУБД допускают одновременную работу нескольких транзакций. При этом параллельные транзакции могут изменять одни и те же данные в БД. Свойство изолированности гарантирует отсутствие взаимного влияния параллельных транзакций на результаты их выполнения с точностью до определенного уровня изолированности. Современные СУБД могут обеспечивать несколько уровней изолированности транзакций.

Долговечность. Любая транзакция завершается либо фиксацией (оператор COMMIT) либо откатом (оператор ROLLBACK). Откат – это отмена всех из-менений в БД, выполненных в рамках транзакции. Фиксация – подтверждение успешного завершения транзакции. Свойство долговечности заключается в том, что изменения в БД, выполненные и зафиксиро-ванные транзакцией, не могут быть отменены, кроме как с помощью новой транзакции.

34. Уровни изоляции транзакций, операторы TCL и их применение.

Одним из основных свойств транзакции является изолированность. Под изолированностью подразумевается ограничение на одновременное чтение или изменение данных в параллельно работающих транзакциях.

Для установки уровня изолированности транзакций в текущем подключении служит инструкция

SET TRANSACTION ISOLATION LEVEL

Обычно при параллельном доступе рассматриваются три проблемы: неподтвержденное чтение, неповторяющееся чтение, фантомное чтение.

Неподтвержденное чтение. До момента t1 транзакцией B выполняются два оператора: INSERT и UPDATE. Эти операторы изменяют таблицы БД, но до момента времени t2 не фиксируют и не откатывают эти изменения. После момента t1 транзакция A считывает содержимое таблиц, измененных транзакцией B и «видит» измененные или добавленные строки. При этом изменения остаются до момента t2 в неподтвержденном состоянии, т. е. могут быть как зафиксированными, так и отмененными.

Неповторяющееся чтение. Одна транзакция читает данные несколько раз, а другая изменяет те же данные между двумя операциями чтения в первом процессе. По этой причине данные, прочитанные в различных операциях, будут разными.

Фантомное чтение. Две последовательные операции чтения могут получать различные значения, т. к. дополнительные строки, называемые фантом¬ными, могут добавляться другими транзакциями.

выбрав уровень SERIALIZABLE, программист решает все проблемы изолированности транзакции. Это действительно так, но решив проблему изолированности, разработчик столкнется с проблемой производительности, поскольку изолированность достигается с помощью механизма блокировок.

Уровень изолированности SNAPSHOT (снимок) обеспечивает полную изолированность и при этом не блокирует строки таблицы.

Транзакция с этим уровнем изолированности «видит» строки таблиц в версии, в которой она застала их при первом чтении. При последующих чтениях транзакция будет «видеть» только те изменения БД, которые произвела сама.

Применение уровня изолированности SNAPSHOT требует установки опции

ALLOW\_SNAPSHOT\_ISOLATION

для БД на уровне сервера.

Операторы TCL (Transaction Control Language) предназначены для создания транзакций.

TCL SQL включает четыре оператора:

BEGIN TRAN, SAVE TRAN, COMMIT TRAN, ROLLBACK TRAN

Оператор BEGIN TRAN указывает на начало транзакции.

Оператор COMMIT TRAN фиксирует транзакцию – подтверждает ее успешное завершение.

Оператор ROLLBACK TRAN откатывает транзакцию – отменяет все изменения, произведенные DML-операторами в таблицах БД в рамках данной транзакции.

Оператор SAVE TRAN применяется для формирования контрольной точки – промежуточного состояния транзак-ции, к которому может быть осуществлен откат.

В тексте SQL-скрипта оператор SAVE TRAN должен находиться между операторами BEGIN TRAN и ROLLBACK TRAN.

35. Хранимые процедуры T-SQL: параметры, создание, вызов, возврат значения, системные процедуры.

Хранимая процедура – это поименованный код на языке Transact-SQL. Хранимая процедура может быть создана с помощью CREATE, изменена с помощью ALTER и удалена с помощью оператора DROP. Процедура может принимать входные и формировать выходные параметры.

Вызов процедуры осуществляется оператором EXECUTE (EXEC).

В хранимых процедурах допускается применение основных DDL, DML и TCL-операторов, конструкций TRY/CATCH, курсоров, временных таблиц.

Результатом выполнения процедуры может быть:

- целочисленное значение, возвращаемое к точке вызова с помощью оператора RETURN;

- один или более результирующих наборов, сформированных операторами SELECT;

- содержимое стандартного выходного потока, полученного при выполнении операторов PRINT.

Кроме того, результирующий набор хранимой процедуры может быть использован в качестве исходно-го источника строк для оператора INSERT.

Модифицированная процедура имеет два параметра: входной параметр @f и выходной (указано ключевое слово OUTPUT) параметр @c.

Конструкция INSERT…EXECUTE

Процедуры, формирующие результирующий набор и не имеющие выходных параметров, могут быть применены в операторе INSERT в качестве источника строк для добавления в целевую таблицу.

36. Функции: типы, параметры, создание, вызов, возврат значения, принципы применения.

Функция – это объект БД, представляющий собой поименованный код T-SQL.

Для создания, удаления и изменения функций применяются CREATE, DROP и ALTER соответственно.

Отличие функций от хранимых процедур в огра-ничениях, накладываемых на код функции, в форме представления результата работы, а также в спосо-бе вызова.

В функции не допускается применение DDL- операторов, DML-операторов изменяющих БД (INSERT, DELETE, UPDATE), конструкций TRY/CATCH, а также использование транзакций.

Результатом выполнения функции является возвращаемое к точке вызова единственное значение.

Если функция возвращает значение в виде числа, строки, даты, времени и пр., то она называется скалярной.

Функция, возвращающая таблицу, называется табличной.

Скалярная функция возвращает единственное значение и принимает любое количество параметров. Допускается возвращать и принимать все типы дан-ных, за исключением TIMESTAMP и устаревших IMAGE, TEXT и NTEXT.

При создании или изменении функции можно ука-зать для параметров значения по умолчанию. При использовании значения параметра по умолча-нию надо в списке параметров указывать ключевое слово default в соответствующей этому параметру позиции.

Встроенные табличные функции возвращают к точке вызова результирующий набор SELECT-запроса и по своим возможностям напоминают представления.

Встроенные функции можно использовать в операциях внутреннего (INNER JOIN) и внешнего (OUTER JOIN) соединения таблиц.

Встроенные функции не дают выигрыша в произво-дительности ни в сравнении с явными SELECT-запросами, ни в сравнении с представлениями.

Главная ценность встроенных функций в их по-вторной применимости, а также в возможности су-щественно сократить размер SELECT-запроса.

В зависимости от структуры кода, различают встроенные функции и многооператорные табличные функции.

Многооператорные табличные функции возвраща-ют к точке вызова результирующий набор, и при этом допускают более сложную логику работы.

Вызов многооператорной функции осуществ-ляется аналогично вызову встроенных табличных функций

При вызове функции надо указывать ее имя с точностью до схемы БД. Если при создании функции имя схемы не указано, то она размещается по умолчанию в схеме DBO.

Различные варианты обращения к функции позволяют проанализировать ее работу:

select \* from dbo.FTovCena(NULL, NULL);

select \* from dbo.FTovCena('Стул', NULL);

select \* from dbo.FTovCena(NULL, 400);

select \* from dbo.FTovCena('Шкаф', 340);

select Наименование\_фирмы,

dbo.COUNT\_Zakazy(Наименование\_фирмы)

from Заказчики;

37. Триггеры: типы триггеров, создание, назначение, применение, вложенные и рекурсивные триггеры.

Триггер – это особый вид хранимой процедуры, предназначенной для обработки событий в БД. Поддерживается два типа триггеров: DDL-триггеры и DML-триггеры. Для каждого типа определено свое семейство событий, обработку которых триггер этого типа может выполнять.

DML-триггеры бывают двух типов: AFTER-триггеры и INSTEAD OF-триггеры. Триггеры типа AFTER исполняются после выполнения оператора, вызвавшего соответствующее событие. При этом создаются автоматически две псевдотаблицы INSERTED и DELETED.

Триггер типа INSTEAD OF выполняется вместо оператора, вызвавшего соответствующее событие. Выполнение INSTEAD OF триггера предшествует проверке установленных для таблицы ограничений целостности.

DML-триггеры являются триггерами операторного типа. Такие триггеры вызываются один раз для каждого оператора, независимо от количества строк таблицы, обработанных этим оператором.

Один AFTER-триггер может реагировать на несколько событий. При этом после ключевого слова AFTER должны быть перечислены эти события.

Для таблицы или представления допускается создание только по одному INSTEAD OF-триггеру, реагирующему на каждое событие.

Выполнение INSTEAD OF-триггера предшествует проверке установленных для таблицы ограничений целостности.

Различают два вида DDL-триггеров:

триггеры уровня сервера (ALLSERVER),

триггеры уровня базы данных (DATABASE).

Триггеры уровня сервера могут обрабатывать события сервера СУБД (создание, изменение, удаление объектов сервера, а также подключение к серверу)

Триггеры уровня базы данных предназначены для обработки событий, происходящих в рамках базы данных.

События DDL-триггеров являются иерархическими. Иерархия событий имеет пять уровней.

Триггеры уровня сервера предназначаются для отслеживания настроек сервера, а также для обеспечения его безопасности.

Условно триггеры уровня сервера можно разбить на две группы:

- триггеры для обработки событий, связанных с созданием, изменением или удалением объектов сервера;

- триггеры, обрабатывающие событие подключения к серверу.

На уровне сервера (ALLSERVER) может быть создан LOGON-триггер, предназначенный для обработки подключения к серверу

Удалить DDL-триггеры можно с помощью опера-тора DROP TRIGGER.

При этом следует указывать уровень действия удаляемого триггера (ON ALL SERVSER или ON DATABASE).

create trigger TRIG\_Tov on Товары after INSERT, DELETE, UPDATE

38. Использование таблиц INSERTED, DELETED, последовательность выполнения триггеров.

При выполнении триггера разработчику доступны две псевдотаблицы с именами INSERTED и DELETED. В зависимости от типа события, активизи-ровавшего триггер, содержимое таблиц разное.

Событие INSERT приводит к тому, что в таблицу INSERTED помещаются строки, добавленные опера-тором INSERT, вызвавшим это событие. При этом таблица DELETED остается пустой.

При событии DELETE в таблицу DELETED копиру-ются удаленные строки, а таблица INSERTED остается пустой.

При изменении строк таблицы с помощью оператора UPDATE заполняются обе псевдотаблицы. При этом INSERTED содержит обновленные версии строк, а таблица DELETED версию строк до их изменения.

Как любой объект БД, триггер может быть изменен с помощью DDL-оператора ALTER

После изменения каждый триггер будет формировать результирующий набор, содержащий объединение строк таблиц INSERTED и DELETRED

Порядок выполнения DML-триггеров

Если для таблицы или представления созданы INSTEAD OF и AFTER-триггеры, реагирующие на одно и то же событие, то выполняется только INSTEAD OF- триггер.

Для нескольких AFTER-триггеров, реагирующих на одно и то же событие, имеется возможность частично упорядочить их выполнение с помощью системной процедуры SP\_SETTRIGGERORDER.

Процедура SP\_SETTRIGGERORDER принимает три параметра:

имя триггера (triggername), позиция в последовательности вызова (order) и тип события (stmttype).

Параметр order может принимать только два значения: First (первый) и Last (последний).

Параметр stmttype может принимать одно из трех значений: INSERT, UPDATE и DELETE.

Для заданного параметром stmttype события с помощью процедуры SP\_SETTRIGGERORDER могут быть указаны только первый и последний триггеры таблицы. Порядок выполнения других триггеров, реагирующих на это же событие той же таблицы, не гарантируется.

С помощью системных представлений SYS.TRIGGERS и SYS.TRIGGER\_EVENTS можно получить перечень триггеров БД

39. Применение XML: конструкции FOR XML RAW, FOR XML AUTO, FOR XML PATH.

XML (Extensible Markup Language) – расширенный язык разметки, разработка которого началась в 1996 году консорциумом W3C.

Язык XML задает общие правила, по которым создаются тэги (элементы языка разметки, предназначенные для разделения данных в текстовых файлах) и оформляются XML-документы. XML-документ представляет собой текстовый файл, содержащий данные, разделенные тэгами, составленными по правилам XML.

Совокупность XML-тэгов, предназначенных для разметки документов определенного типа, называют словарями XML или XML-реализациями.

Использование XML-формата влечет к значительному увеличению избыточности данных, но при этом XML-формат позволяет создавать самодокументированные данные.

XML-формат часто используется для обмена данными между компонентами одной информационной системы или на межсистемном уровне.

При этом проявляются главные преимущества XML:

- большинство программных платформ «понимают» XML одинаково;

- программные системы имеют встроенные меха-низмы, позволяющие эффективно обрабатывать XML-данные.

Для разработчиков программного обеспечения важ-ными являются две задачи:

- преобразование табличных данных в XML-структуры;

- преобразование XML-структур в строки реляци-онной таблицы.

Преобразование табличных данных в XML-структуры

Для преобразования результата SELECT-запроса в формат XML в операторе SELECT применяется секция FOR XML.

При этом можно использовать один из четырех режимов: RAW, AUTO, PATH и EXPLICIT.

Режим RAW

По умолчанию в режиме RAW в результате SELECT-запроса создается XML-фрагмент, состоящий из последовательности элементов с именем row.

Каждый элемент row соответствует строке результирующего набора, имена его атрибутов совпадают с именами столбцов результирующего набора, а зна-чения атрибутов равны значениям полей.

Используется два варианта: атрибутная форма и элементная.

Для того, чтобы изменить стандартное имя элемента row на заданное, достаточно его указать в скобках по-сле ключевого слова RAW

Построить корневой элемент для XML-фрагмента можно с помощью ключевого слова ROOT в инструкции FOR XML

Если в сформированных XML-элементах требуется, чтобы имена атрибутов отличались от имен столбцов исходных таблиц, следует в SELECT-списке для столбцов указать псевдонимы, которые будет исполь-зованы в качестве имен атрибутов

Для того чтобы сформировать результирующий набор в элементном виде, надо в инструкции FOR XML использовать ключевое слово ELEMENTS.

Тогда имена столбцов (или их псевдонимы) являются именами вложенных элементов; данные представле-ны, как содержимое элементов

Режим AUTO

Результат, полученный в режиме AUTO для простых SELECT-запросов похож на результат, полученный в режиме RAW.

Основное отличие проявляется в многотабличных запросах.

Режим AUTO позволяет построить XML-фрагмент с применением вложенных элементов. Порядок вложенности основывается на порядке столбцов, указан-ных в SELECT-списке.

Допускается применение ключевого слова ROOT (имя корневого элемента) и псевдонимов

Формирование элементного вида XML-фрагмента осуществляется с помощью ключевого слова ELEMENTS

Режим PATH

Режим PATH позволяет разработчику наиболее полно управлять процессом формирования XML-структуры.

Если псевдонимы задавать традиционным способом (что соответствует первому шаблону), то сформирует-ся XML-фрагмент, состоящий из элементов с именем row, соответствующих строкам результирующего набора. Каждый row-элемент включает в себя все значения столбцов одной строки в элементной форме. При этом элементы будут иметь имена, совпадающие с заданными в SELECT-списке псевдонимами.

В режиме PATH при формировании результата у разработчика появляется возможность сочетать атри-бутную и элементную форму

Применение TYPE во внутреннем подзапросе позволяет указать на то, что формируемый XML-фрагмент следует рассматривать, как вложенный.

40. Функция OPENXML, тип XML, коллекции XML-схем и их применение, типизированные XML-столбцы.

Для преобразования XML-данных в строки таблицы предназначена функция OPENXML.

Она принимает три входных параметра:

- дескриптор, сформированный системной процедурой SP\_XML\_PREPAREDOCUMENT

- выражение XPATH (состоит из последовательности тэгов);

- целое положительное число, определяющее режим работы функции.

Процедура SP\_XML\_PREPAREDOCUMENT должна быть выполнена до SELECT-запроса, использующего OPENXML. Она принимает в качестве входного параметра XML-документ (в формате строки) и возвращает дескриптор, который впоследствии применя-ется функцией OPENXML.

Второй параметр - выражение XPATH - предназначено для выбора требуемых данных из исходного XML-документа.

Третий параметр указывает на тип преобразования (режим).

с помощью WITH указывается структура формируемого результирующего набора (имена столбцов совпадают с именами атрибутов в XML-структуре).

Из XML-фрагмента с помощью XPATH-выражения выделяется множество тэгов с именем аудитория.

Применяется режим работы 0, подразумевающий атрибутивное сопоставление.

После использования функции OPENXML должна быть выполнена системная процедура SP\_XML\_REMOVEDOCUMENT, освобождающая использованные OPENXML ресурсы.

Чтобы явно указать сопоставление XML-структуры со структурой строки результирующего набора, можно в выражении WITH каждого столбца результирующего набора указать XPATH-выражение, определяющее со-держимое этого столбца.

Тип данных XML

В большинстве своем данные в таблицах БД имеют фиксированный формат.

В тех случаях, когда данные являются составными и при этом не имеют фиксированный формат, удобно использовать тип XML для их записи.

Например, если в таблице хранится информация о гражданах разных стран, то структура их почтового адреса может значительно отличаться в зависимости от страны проживания.

Информация об образовании людей (учебные заведения, даты начала и завершения обучения, специальности, квалифика-ции и пр.) будут различными у разных людей.

XML-данные могут быть добавлены в таблицу с помощью оператора INSERT.

В простейшем случае сами XML-данные задаются в виде строкового литерала, который автоматически будет преобразован в XML-тип

С помощью оператора SELECT данные XML-типа могут быть извлечены из БД. В простейшем случае извлекается полностью XML-структура

Часто требуется извлечь не полностью хранящуюся в БД XML-структуру, а лишь ее подмножество.

Для этого можно воспользоваться методами XML-типа, приведенными в таблице

XQUERY является новым языком запросов для XML. Фактически XQUERY содержит XPATH в качестве своего подъязыка.

- XML-тип является объектным типом (имеет методы и свойства);

- в первых двух случаях применения метода value извлекаются значения атрибутов серия и номер;

- в третьем случае применения value извлекается текстовое значение элемента телефон;

- во всех трех случаях применения метода value в конце заданного выражения стоит число в квадрат-ных скобках, обозначающее номер выбираемого экземпляра (единица позволяет получить только первый экземпляр);

- второй параметр метода value указывает на тип данных, к которому должно быть преобразовано выбранное значение;

- метод query, применяемый в четвертом элементе SELECT-списка, позволяет выбрать XML-фрагмент.

С помощью операторов SELECT и SET этим пе-ременным присваиваются значения.

С помощью оператора PRINT предварительно преобразованные значения XML-переменныx в строки выводятся в выходной поток.

В логическом выражении оператора IF применя-ется метод exists, позволяющий определить наличие заданного XML-узла в значении XML-переменной.

Сложность XML-данных требует иного подхода к механизму ограничения целостности для этого типа данных.

В семействе XML-технологий существует техноло-гия, основанная на языке XML-Schema.

XML-Schema – это одна из реализаций языка XML, поддерживаемая консорциумом W3C и предназначен-ная для описания структуры XML-документа.

С помощью языка XML-Schema можно описать правила, которым должен подчиняться XML-документ.

Файл, содержащий XML-Schema, обычно имеет расширение XSD (XML Schema definition).

Для хранения документов XML-Schema в БД MSS предусмотрен специальный объект XML SCHEMA COLLECTION

Каждый такой объект может содержать один или более XML-SCHEMA-документов.

При создании таблицы для XML-столбца можно указать имя коллекции схем. Такой столбец называется типизированным столбцом. Типизированный XML-столбец должен удовлетворять хотя бы одной схеме из связанной с ним коллекции.

Тип данных, размещаемых в атрибутах или теле элементов данных, определяется значением атрибута type.

41. Настройка производительности базы данных

Улучшение производительности базы данных требует принятия множества решений. Основные из них: где хранить данные и как осуществлять доступ к данным.

Производительность Database Engine (и любой другой реляционной СУБД) определяется по двум критериям:

- время реакции;

- пропускная способность.

Время реакции определяет производительность отдельной транзакции или программы.

Время реакции трактуется как длительность во времени от момента, когда пользователь вводит команду или оператор, до момента, когда система отмечает, что выполнение команды (оператора) завершено.

Для достижения оптимального времени реакции всей системы почти все существующие команды и операторы не должны превышать указанное ограничение времени реакции.

Пропускная способность измеряет общую работу системы, вычисляя количество транзакций, которые могут быть обработаны в заданный период времени.

Обычно пропускная способность измеряется в количестве транзакций в секунду.

Факторы, влияющие на производительность

Производительность зависит от параметров:

- приложений базы данных;

- системы базы данных;

- системных ресурсов компьютеров.

1. На производительность приложений базы данных влияют следующие факторы:

- эффективность кода приложения;

- физическое проектирование.

Эффективность кода приложения. Приложения содержат собственный код, используемый системным программным обеспечением и Database Engine. По этой причине они могут создавать проблемы производительности, если задано неправильное использование системных ресурсов.

Большинство проблем производительности в программах приложений появляется в результате неправильного выбора операторов Transact-SQL и их последовательности в программах приложений.

Физическое проектирование. В процессе физического проектирования базы данных выбираются конкретные структуры хранения и пути доступа к файлам базы данных. На этом шаге проектирования иногда рекомендуется выполнять денормализацию некоторых таблиц базы данных с целью достижения хорошей производительности для различных приложений базы данных.

Денормализация таблиц означает, что две или более нормализованных таблицы объединяются вместе, что приводит к некоторой избыточности данных.

Преимущества денормализации:

- если есть столбец, который зависит от другого столбца таблицы для данных, часто используемых в запросах, то можно исключить операцию соединения таблиц, которая оказывает влияние на производительность приложений.

- денормализованные данные требуют меньшего количества таблиц, чем нормализованные данные.

Недостатки:

- денормализованные таблицы требуют дополнительного объема дисковой памяти;

- а изменение данных выполнять сложнее по причине избыточности данных.

Другим вариантом повышения производительности, является создание индексов.

2. Database Engine и производительность

Database Engine может значительно влиять на производительность всей системы. Двумя наиболее важными компонентами (механизмами) Database Engine, которые влияют на производительность, являются оптимизатор и блокировки.

Оптимизатор создает несколько планов выполнения запросов для чтения строк, а затем принимает решение, какой план будет использоваться. Это решение касается выбора наиболее подходящего плана выполнения, включая то, какие индексы должны быть использованы, как осуществлять доступ к таблицам и в каком порядке должны соеди-няться таблицы. Все эти решения могут значительно повлиять на производительность приложений

Блокировки. Система базы данных использует блокировки в качестве механизма защиты одного пользователя от другого. Поэтому блокировки служат для управления доступом к данным для всех пользователей в одно и то же время и для предотвращения вероятных ошибок, которые могут возникнуть в результате одновременного доступа к одним и тем же данным.

Блокировка на уровне строки таблицы обеспечивает наилучшую производительность системы, потому что она оставляет незаблокированными все строки на странице, кроме одной, и, следовательно, позволяет выполнять больше параллельных действий, чем блокировки на уровне страницы или на уровне всей таблицы.

Уровни изоляции влияют на длительность блокировки операторов SELECT. При использовании низких уровней изоляции, таких как READ UNCOMMITTED и READ COMMITTED, может быть улучшена доступность данных и, следовательно, количество одновременных обращений к данным.

3. Системные ресурсы и производительность

Database Engine выполняется под управлением операционной системы Windows, которая сама использует системные ресурсы. Производительность любой системы базы данных зависит от четырех основных системных ресурсов:

- центральный процессор;

- память;

- дисковые операции ввода/вывода;

- сетевое окружение.

Центральный процессор вместе с оперативной памятью является ключевым компонентом, определяющим

быстродействие компьютера. Чем мощнее процессор, тем лучше работает вся система.

Database Engine динамически запрашивает и освобождает память по мере необходимости. Чем больше памяти доступно на компьютере, тем лучше работает вся система.

Существуют два момента, связанных с дисковым вводом/выводом: дисковая скорость и пропускная способность диска.

Дисковая скорость определяет, как быстро выполняются операции чтения и записи для диска.

Пропускная способность диска задает, какой объем данных может быть записан на диск в единицу времени (время обычно измеряется в секундах).

Чем быстрее диск, тем больший объем данных может быть обработан. Так же, большее количество дисков лучше, чем один диск в случае, когда много пользователей одновременно обращается к системе базы данных.

В конфигурации "клиент/сервер" система базы данных иногда работает хуже, если существует множество клиентских соединений. В этом случае объем данных, которые нужно пересылать по сети, может превышать возможности сетевого окружения. Чтобы устранить это узкое место в производительности, следует принять во внимание следующие общие рекомендации:

- сервер базы данных должен отправлять приложению лишь те строки, которые действительно требуются;

- если выполняющееся в течение долгого времени пользовательское приложение выполняется на клиентской стороне, то лучше переместить его на сторону сервера (например, выполняя его как хранимую процедуру).

Все системные ресурсы взаимно зависят друг от друга. Это означает, что проблемы производительности с одним ресурсом могут привести к проблемам производительности, связанным с другими ресурсами.

Аналогично, улучшения, связанные с одним ресурсом, могут значительно повысить производительность некоторого другого ресурса (или даже всех).

42. Основы безопасности: аутентификация; шифрование; авторизация; отслеживание изменений.

Наиболее важные концепции безопасности:

- аутентификация;

- шифрование;

- авторизация;

- отслеживание изменений.

Аутентификация требует ответа на вопрос: "Имеет ли пользователь легитимные права на доступ в систему?" Эта концепция безопасности осуществляет проверку полномочий пользователей: есть ли у пользователя пароль или магнитная карта или жетон, цифровая подпись, отпечатки пальцев и т.п.

Шифрование данных - это процесс кодирования информации таким образом, что она становится непонятной, пока не будет дешифрована для конкретного пользователя. Для шифрования данных могут быть использованы различ-ные методы.

Авторизация - процесс, который осуществляется после аутентификации. Система определяет, какие ресурсы может использовать данный пользователь.

Отслеживание изменений означает, что действия неавторизованных пользователей отслеживаются и документируются на компьютере.

Аутентификация

Система безопасности Database Engine включает две различные подсистемы безопасности:

- безопасность Windows;

- безопасность SQL Server.

Безопасность Windows задает безопасность на уровне операционной системы, т. е. задает метод, при помощи ко-торого пользователи соединяются с операционной системой, используя учетную запись пользователя Windows.

Безопасность SQL Server задает дополнительную безопасность на уровне системы после того, как пользователь вошел в операционную систему, и может соединиться с сервером базы данных. Служба безопасности SQL Server определяет учетную запись SQL Server (называемую логином), которая создается системой и связана с паролем. Некоторые учетные записи SQL Server идентичны учетным записям пользователя Windows.

Основываясь на этих двух системах безопасности, Database Engine может работать в одном из следующих режимов аутентификации:

- режим Windows;

- смешанный режим.

Режим Windows требует учетную запись пользователя Windows исключительно для соединения с системой. Си-стема принимает учетную запись пользователя, предполагая, что она уже проверена на уровне операционной системы. Этот вид соединения с системой базы данных называ-ется доверительным соединением, потому что система доверяет тому, что операционная система уже проверила эту учетную запись и соответствующий ей пароль.

Смешанный режим дает пользователям возможность соединяться с Database Engine, используя аутентификацию Windows или аутентификацию SQL Server. Это означает, что некоторые учетные записи пользователя могут быть установлены для использования и в подсистеме безопасности SQL Server, и в подсистеме безопасности Windows.

Для выбора одного из существующих режимов аутентификации используется SQL Server Management Studio.

Чтобы установить режим Windows, надо щелкнуть правой кнопкой мыши по серверу и выбрать пункт Properties. В диалоговом окне Server Properties выбрать страницу Security и щелкнуть по Windows Authentication Mode.

Для выбора смешанного режима нужно щелкнуть по Server and Windows Authentication Mode в диалоговом окне Server Properties.

После того как пользователь будет успешно соединен с Database Engine, доступ пользователя к объектам базы дан-ных становится независимым от того, какой режим аутентификации используется — Windows Authentication или SQL Server Authentication.

Шифрование данных

Database Engine защищает данные при помощи иерар-хических уровней шифрования и инфраструктуры управления ключами. Каждый уровень защищает следующий за ним уровень шифрования, используя комбинацию сертификатов, асимметричных и симметричных ключей.

Сервисный главный ключ создается автоматически при инсталляции Database Engine. Важным свойством сервисного главного ключа является то, что он управляется системой.

Каждая база данных имеет один главный ключ базы данных, который создается при использовании оператора CREATE MASTER KEY. Поскольку главный ключ базы данных защищен главным сервисным ключом, для системы есть возможность автоматически дешифровать главный ключ базы данных.

Так как существует главный ключ базы данных, пользователи могут его использовать для создания пользовательских ключей, которые создаются и поддерживаются пользователями.

Существуют три формы пользовательских ключей: симметричные ключи, асимметричные ключи и сертификаты.

Симметричные ключи

Система шифрования, которая использует симметричные ключи, - это та, в которой отправитель и получатель со-общения используют общий ключ. Следовательно, этот ключ применяется как для шифрования, так и для дешифрования данных.

Одним из преимуществ использования симметричных ключей является то, что они могут защищать значительно больший объем данных и обрабатываются быстрее, чем ключи других типов.

С другой стороны, в распределенных системах использование этого типа ключей может сделать почти что невозможным сохранение безопасности шифрования, т. к. один и тот же ключ служит для дешифрования и шифрования данных в обоих концах системы.

Поэтому симметричные ключи должны использоваться в приложениях, где данные сохраняются в виде зашифрованного текста в одном месте.

Язык Transact-SQL поддерживает некоторые операторы и системные функции для работы с симметричными ключами.

Асимметричные ключи

Асимметричный ключ состоит из двух частей: личного ключа и соответствующего общего ключа. Каждый из ключей может дешифровать данные, зашифрованные другим ключом. По причине существования личного ключа асимметричное шифрование обеспечивает более высокий уровень безопасности, чем это делает симметричное шифрование.

Язык Transact-SQL поддерживает некоторое количество операторов и системных функций, связанных с асимметричными ключами.

Оператор create asymmetric key создает новый асимметричный ключ, а оператор alter asymmetric key изменяет свойства асимметричного ключа. Оператор drop asymmetric key удаляет существующий асимметричный ключ.

После того как создан асимметричный ключ, можно использовать системную функцию EncryptByAsymKey для шифрования данных. Эта функция имеет два входных параметра: идентификатор ключа и текст, который должен быть зашифрован. Для дешифрования используется системная функция DecryptByAsymKey.

Сертификаты

Общий ключ сертификата, обычно называемый просто сертификатом, является двоичной подписью, связывающей значение общего ключа с идентификацией человека, устройства или сервиса, который содержит соответствующий личный ключ.

Сертификаты вызываются и обозначаются при помощи авторизации сертификации (Certification Authority, СА). Сущность, которая получает сертификат от СА, является субъектом этой сертификации. Сертификаты содержат следующую информацию:

- значение общего ключа субъекта;

- информацию, идентифицирующую субъекта;

- издателя идентифицирующей информации;

- двоичную подпись издателя.

Основное преимущество сертификатов в том, что они освобождают хосты от необходимости поддерживать наборы паролей для индивидуальных субъектов.

Когда хост, например Web-сервер безопасности, определяет запрашивающую сторону, как доверенного пользователя, хост неявно доверяет тому, что эта запрашивающая сторона проверила идентичность сертифицированного субъекта.

Сертификаты предоставляют наивысший уровень шифрования в модели безопасности Database Engine. Алгоритмы шифрования для сертификатов дают большую нагрузку на процессоры. По этой причине их надо исполь-зовать только при реальной необходимости.

Авторизация - это процесс, который применяется после проверки того, что пользователь соответствует аутенти-фикации.

В течение этого процесса система определяет, какие ресурсы может использовать данный пользователь. Иными словами, информация из структурного и системного каталога о конкретной сущности теперь доступна лишь доверенным лицам – тем людям, которые имеют доступ к данной сущности.

Только авторизованные пользователи имеют возможность выполнять операторы и осуществлять операции над сущностями. Если неавторизованный пользователь попытается выполнить оператор Transact-SQL или операцию с объектом базы данных, такое действие будет отменено системой.

Существуют три оператора Transact-SQL, связанные с авторизацией: GRANT (предоставляет полномочия к объектам безопасности), DENY (удаляет существующие полномочия у учетных записей пользователей), REVOKE (удаляет одно или более из ранее предоставленных или отмененных полномочий).

Отслеживание изменений означает документирование всех действий по добавлению, изменению и удалению данных применительно к таблицам базы данных. Задокументированные изменения затем могут быть просмотрены для выяснения того, кто обращался к данным и когда были выполнены эти обращения.

Существуют два способа, при помощи которых можно выполнять отслеживание изменений:

- использование триггеров;

- применение средств перехвата, отслеживающих изменения данных.

43. Копирование и восстановление базы данных: причины потерь данных, типы резервного копирования.

Предотвращение потери данных – одна из самых важных задач, стоящих перед администратором базы данных.

Потери данных возможны по следующим причинам:

- неисправность аппаратного обеспечения;

- ошибки в системном программном обеспечении;

- ошибки в пользовательском программном обеспече-нии;

- ошибки системного администратора;

- некорректная работа пользователей (DROP, ALTER, UPDATE, DELETE);

- аварийные ситуации (пожар, наводнение и пр.);

- вандализм, хакерские атаки, вирусы.

Администратор должен разработать стратегию восстановления данных. Стратегия должна быть спланирована и протестирована.

В основе любой стратегии восстановления данных лежит создание резервных копий базы данных.

Резервное копирование означает процесс создания ко-пии базы данных и/или протоколов транзакций для разделения устройств, которые в дальнейшем при необходимости могут быть использованы для восстановления данных.

Восстановление - это процесс использования устройства резервной копии для замены неподтвержденных, несогласованных или потерянных данных.

Методы резервного копирования. Резервное копирование базы данных является процессом выгрузки данных (из базы данных, протокола транзакций или из файла) на устройства резервной копии, которые создаются и поддер-живаются системой.

Устройство резервной копии может быть дисковым файлом или магнитной лентой.

Database Engine обеспечивает статические и динамические резервные копии.

Статическая резервная копия означает, что в процессе копирования только одна активная сессия, поддерживаемая системой, является той сес

копию. Иными словами, недопустимы пользовательские процессы во время выполнения копирования.

Динамическое резервное копирование означает, что копирование базы данных может выполняться без останова сервера базы данных, удаления пользователей или даже закрытия файлов.

Database Engine предоставляет четыре различных метода резервного копирования:

- полное копирование базы данных;

- дифференцированное резервное копирование;

- резервное копирование протокола транзакций;

- резервное копирование файла или файловой группы

Все типы операций резервного копирования могут быть выполнены при использовании двух операторов Transact-SQL: BACKUP DATABASE, BACKUP.LOG.

Резервное копирование можно также осуществлять с помощью Management Studio.

Копирование системных баз данных

В процессе инсталляции Database Engine генерируются следующие системные базы данных:

- master;

- model;

- tempdb;

- msdb.

База данных master является одной из наиболее важных баз данных Database Engine. Она содержит все системные таблицы, необходимые для работы, содержит информацию обо всех других базах данных, управляемых Database Engine, о клиентских соединениях с системой и об авторизациях пользователей.

База данных master изменяется каждый раз, когда выполняются различные системные операции. Поэтому следует осуществлять ее резервное копирование после выполнения каждой такой операции.

База данных model применяется в качестве шаблона при создании баз данных, определяемых пользователем. Она содержит подмножество всех системных таблиц базы данных master, которые нужны каждой базе данных, создаваемой пользователем. Системный администратор может изменять свойства базы данных model для адаптации ее к специфическим требованиям системы.

База данных tempdb предоставляет место для хранения временных таблиц и других необходимых временных объектов. Система, например, сохраняет промежуточные результаты вычислений сложных выражений в базе данных tempdb. База данных tempdb используется всеми базами данных, принадлежащими системе. Ее содержимое уничтожается каждый раз при рестарте системы.

Поскольку база данных tempdb используется системой довольно активно, то надо убедиться, что объем памяти для tempdb достаточно большой.

База данных msdb используется для протоколирования сообщений и заданий. Эта системная база данных содержит сведения о планировании задач, обработку исключений, управление сообщениями и информацию системных операторов. Содержит адреса электронной почты, номера страниц, историю обо всех операциях резервного копирования и восстановления баз данных.

44. Индексы хелп

Основы индексов в SQL Server

Одним из важнейших путей достижения высокой производительности SQL Server является использование индексов. Индекс ускоряет процесс запроса, предоставляя быстрый доступ к строкам данных в таблице, аналогично тому, как указатель в книге помогает вам быстро найти необходимую информацию. В этой статье я приведу краткий обзор индексов в SQL Server и объясню как они организованы в базе данных и как они помогают ускорению выполнения запросов к базе данных.

Структура индекса

Индексы создаются для столбцов таблиц и представлений. Индексы предоставляют путь для быстрого поиска данных на основе значений в этих столбцах. Например, если вы создадите индекс по первичному ключу, а затем будете искать строку с данными, используя значения первичного ключа, то SQL Server сначала найдет значение индекса, а затем использует индекс для быстрого нахождения всей строки с данными. Без индекса будет выполнен полный просмотр (сканирование) всех строк таблицы, что может оказать значительное влияние на производительность.

Вы можете создать индекс на большинстве столбцов таблицы или представления. Исключением, преимущественно, являются столбцы с типами данных для хранения больших объектов (LOB), таких как image, text или varchar(max). Вы также можете создать индексы на столбцах, предназначенных для хранения данных в формате XML, но эти индексы устроены немного иначе, чем стандартные и их рассмотрение выходит за рамки данной статьи. Также в статье не рассматриваются columnstore индексы. Вместо этого я фокусируюсь на тех индексах, которые наиболее часто применяются в базах данных SQL Server.

Индекс состоит из набора страниц, узлов индекса, которые организованы в виде древовидной структуры — сбалансированного дерева. Эта структура является иерархической по своей природе и начинается с корневого узла на вершине иерархии и конечных узлов, листьев, в нижней части, как показано на рисунке:

Когда вы формируете запрос на индексированный столбец, подсистема запросов начинает идти сверху от корневого узла и постепенно двигается вниз через промежуточные узлы, при этом каждый слой промежуточного уровня содержит более детальную информацию о данных. Подсистема запросов продолжает двигаться по узлам индекса до тех пор, пока не достигнет нижнего уровня с листьями индекса. К примеру, если вы ищете значение 123 в индексированном столбе, то подсистема запросов сначала на корневом уровне определит страницу на первом промежуточном (intermediate) уровне. В данном случае первой страница указывает на значение от 1 до 100, а вторая от 101 до 200, таким образом подсистема запросов обратится ко второй странице этого промежуточного уровня. Далее будет выяснено, что следует обратиться к третьей странице следующего промежуточного уровня. Отсюда подсистема запросов прочитает на нижнем уровне значение самого индекса. Листья индекса могут содержать как сами данные таблицы, так и просто указатель на строки с данными в таблице, в зависимости от типа индекса: кластеризованный индекс или некластеризованный.

Кластеризованный индекс

Кластеризованный индекс хранит реальные строки данных в листьях индекса. Возвращаясь к предыдущему примеру, это означает что строка данных, связанная со значение ключа, равного 123 будет храниться в самом индексе. Важной характеристикой кластеризованного индекса является то, что все значения отсортированы в определенном порядке либо возрастания, либо убывания. Таким образом, таблица или представление может иметь только один кластеризованный индекс. В дополнение следует отметить, что данные в таблице хранятся в отсортированном виде только в случае если создан кластеризованный индекс у этой таблицы.

Таблица не имеющая кластеризованного индекса называется кучей.

Некластеризованный индекс

В отличие от кластеризованного индекса, листья некластеризованного индекса содержат только те столбцы (ключевые), по которым определен данный индекс, а также содержит указатель на строки с реальными данными в таблице. Это означает, что системе подзапросов необходима дополнительная операция для обнаружения и получения требуемых данных. Содержание указателя на данные зависит от способа хранения данных: кластеризованная таблица или куча. Если указатель ссылается на кластеризованную таблицу, то он ведет к кластеризованному индексу, используя который можно найти реальные данные. Если указатель ссылается на кучу, то он ведет к конкретному идентификатору строки с данными. Некластеризованные индексы не могут быть отсортированы в отличие от кластеризованных, однако вы можете создать более одного некластеризованного индекса на таблице или представлении, вплоть до 999. Это не означает, что вы должны создавать как можно больше индексов. Индексы могут как улучшить, так и ухудшить производительность системы. В дополнение к возможности создать несколько некластеризованных индексов, вы можете также включить дополнительные столбцы (included column) в свой индекс: на листьях индекса будет храниться не только значение самих индексированных столбцов, но и значения этих не индексированных дополнительных столбцов. Этот подход позволит вам обойти некоторые ограничения, наложенные на индекс. К примеру, вы можете включить неидексируемый столбец или обойти ограничение на длину индекса (900 байт в большинстве случаев).

Типы индексов

В дополнение к тому, что индекс может быть либо кластеризованным, либо некластеризованным, возможно его дополнительно сконфигурировать как составной индекс, уникальный индекс или покрывающий индекс.

Составной индекс

Такой индекс может содержать более одного столбца. Вы можете включить до 16 столбцов в индекс, но их общая длина ограничена 900 байтами. Как кластеризованный, так и некластеризованный индексы могут быть составными.

Уникальный индекс

Такой индекс обеспечивает уникальность каждого значения в индексируемом столбце. Если индекс составной, то уникальность распространяется на все столбцы индекса, но не на каждый отдельный столбец. К примеру, если вы создадите уникальных индекс на столбцах ИМЯ и ФАМИЛИЯ, то полное имя должно быть уникально, но отдельно возможны дубли в имени или фамилии.

Уникальный индекс автоматически создается когда вы определяете ограничения столбца: первичный ключ или ограничение на уникальность значений:

Первичный ключ

Когда вы определяете ограничение первичного ключа на один или несколько столбцов, тогда SQL Server автоматически создаёт уникальный кластеризованный индекс, если кластеризованный индекс не был создан ранее (в этом случае создается уникальный некластеризованный индекс по первичному ключу)

Уникальность значений

Когда вы определяете ограничение на уникальность значений, тогда SQL Server автоматически создает уникальный некластеризованный индекс. Вы можете указать, чтобы был создан уникальный кластеризованный индекс, если кластеризованного индекса до сих пор не было создано на таблице

Покрывающий индекс

Такой индекс позволяет конкретному запросу сразу получить все необходимые данные с листьев индекса без дополнительных обращений к записям самой таблицы.

Проектирование индексов

Насколько полезны индексы могут быть, настолько аккуратно они должны быть спроектированы. Поскольку индексы могут занимать значительное дисковое пространство, вы не захотите создавать индексов больше, чем необходимо. В дополнение, индексы автоматически обновляются когда сама строка с данными обновляется, что может привести к дополнительным накладным расходам ресурсов и падению производительности. При проектирование индексов должно приниматься во внимание несколько соображений относительно базы данных и запросов к ней.

База данных

Как было отмечено ранее индексы могут улучить производительность системы, т.к. они обеспечивают подсистему запросов быстрым путем для нахождения данных. Однако, вы должны также принять во внимание то, как часто вы собираетесь вставлять, обновлять или удалять данные. Когда вы изменяете данные, то индексы должны также быть изменены, чтобы отразить соответствующие действия над данными, что может значительно снизить производительность системы. Рассмотрим следующие рекомендации при планировании стратегии индексирования:

Для таблиц которые часто обновляются используйте как можно меньше индексов.

Если таблица содержит большое количество данных, но их изменения незначительны, тогда используйте столько индексов, сколько необходимо для улучшение производительности ваших запросов. Однако хорошо подумайте перед использованием индексов на небольших таблицах, т.к. возможно использование поиска по индексу может занять больше времени, нежели простое сканирование всех строк.

Для кластеризованных индексов старайтесь использовать настолько короткие поля насколько это возможно. Наилучшим образом будет применение кластеризованного индекса на столбцах с уникальными значениями и не позволяющими использовать NULL. Вот почему первичный ключ часто используется как кластеризованный индекс.

Уникальность значений в столбце влияет на производительность индекса. В общем случае, чем больше у вас дубликатов в столбце, тем хуже работает индекс. С другой стороны, чем больше уникальных значения, тем выше работоспособность индекса. Когда возможно используйте уникальный индекс.

Для составного индекса возьмите во внимание порядок столбцов в индексе. Столбцы, которые используются в выражениях WHERE (к примеру, WHERE FirstName = 'Charlie') должны быть в индексе первыми. Последующие столбцы должны быть перечислены с учетом уникальности их значений (столбцы с самым высоким количеством уникальных значений идут первыми).

Также можно указать индекс на вычисляемых столбцах, если они соответствуют некоторым требованиям. К примеру, выражение которые используются для получения значения столбца, должны быть детерминистическими (всегда возвращать один и тот же результат для заданного набора входных параметров).

Запросы к базе данных

Другое соображение которое следует учитывать при проектировании индексов это какие запросы выполняются к базе данных. Как было указано ранее, вы должны учитывать как часто изменяются данные. Дополнительно следует использовать следующие принципы:

Старайтесь вставлять или модифицировать в одном запросе как можно больше строк, а не делать это в несколько одиночных запросов.

Создайте некластеризованный индекс на столбцах которые часто используются в ваших запросах в качестве условий поиска в WHERE и соединения в JOIN.

Рассмотрите возможность индексирования столбцов, использующихся в запросах поиска строк на точное соответствие значений.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

А теперь, собственно:

14 вопросов об индексах в SQL Server, которые вы стеснялись задать

Почему таблица не может иметь два кластеризованных индекса?

Если кластеризованная таблица даёт множество преимуществ, то зачем использовать кучу?

Как изменить установленное по умолчанию значение коэффициента заполнения индекса?

Можно ли создать кластеризованный индекс на столбце, содержащем дубликаты?

Как хранится таблица, если не был создан кластеризованный индекс?

Какая взаимосвязь между ограничениями на уникальность значения и первичным ключом с индексами таблицы?

Почему в SQL Server кластеризованные и некластеризованные индексы называются сбалансированным деревом?

Как вообще индекс может улучшить производительность запросов, если приходится переходить по всем этим индексным узлам?

Если индексы настолько замечательны, то почему бы просто не создать их на каждый столбец?

Обязательно ли создавать кластеризованный индекс на столбце с первичным ключом?

А что если проиндексировать представление, то это по-прежнему будет представление?

Зачем использовать покрывающий индекс взамен составного индекса?

Имеет ли значение количество дубликатов в ключевом столбце?

Можно ли создать некластеризованный индекс только для определенного подмножества данных ключевого столбца?

Почему таблица не может иметь два кластеризованных индекса?

Хотите короткий ответ? Кластеризованный индекс – это и есть таблица. Когда вы создаете кластеризованный индекс у таблицы, подсистема хранения данных сортирует все строки в таблице в порядке возрастания или убывания, согласно определению индекса. Кластеризованный индекс это не отдельная сущность как другие индексы, а механизм сортировки данных в таблице и облегчения быстрого доступа к строкам с данными.

Представим, что у вас есть таблица, содержащая историю операций по продажам. Таблица Sales включает в себя такую информация как идентификатор заказа, позицию товара в заказе, номер товара, количество товара, номер и дату заказа и т.д. Вы создаёте кластеризованный индекс по столбцам OrderID и LineID, с сортировкой в порядке возрастания, как показано в следующем T-SQL коде:

CREATE UNIQUE CLUSTERED INDEX ix\_oriderid\_lineid

ON dbo.Sales(OrderID, LineID);

Когда вы запустите этот скрипт все строки в таблице будут физически отсортированы сначала по столбцу OrderID, а затем по LineID, но сами данные останутся в единственном логическом блоке, в таблице. По этой причине вы не можете создать два кластеризованных индекса. Может быть только одна таблица с одними данными и эта таблица может быть отсортирована только один раз в определенном порядке.

Если кластеризованная таблица даёт множество преимуществ, то зачем использовать кучу?

Вы правы. Кластеризованые таблицы отличны и большинство ваших запросов будут лучше выполнятся к таблицам, имеющим кластеризованный индекс. Но в некоторых случаях вы возможно захотите оставить таблицы в их естественном первозданном состоянии, т.е. в виде кучи, и создать лишь некластеризованные индексы для поддержания работоспособности ваших запросов.

Куча, как вы помните, хранит данные в случайном порядке. Обычно подсистема хранения данных добавляет в таблицу данные в той последовательности в которой они вставляются, однако подсистема также любит перемещать строки с целью более эффективного хранения. В результате у вас нет ни единого шанса предсказать в каком порядке будут храниться данные.

Если подсистема запросов должна найти данные без преимуществ некластеризованного индекса, то она сделает полное сканирование таблицы для нахождения нужных ей строк. На очень маленьких таблицах это обычно не проблема, но как только куча растет в своих размерах производительность быстро падает. Конечно, некластеризованный индекс может помочь, используя указатель на файл, страницу и строку где хранятся необходимые данные – обычно это намного лучшая альтернатива сканированию таблицы. Но даже в этом случае трудно сравнивать с преимуществами кластеризованного индекса при рассмотрении производительности запросов.

Однако куча может помочь улучшить производительность в определенных ситуациях. Рассмотрим таблицу с большим количеством вставок, но редкими обновлениями или удалением данных. К примеру, таблица, хранящая лог, преимущественно используется для вставки значений до тех пор пока не будет архивирована. В куче вы не увидите разбиением страниц и фрагментацию данных, как это случается с кластеризованным индексом, потому что строки просто добавляются в конец кучи. Слишком большое разделение страниц может иметь значительное влияние на производительность и в не самом хорошем смысле. В общем, куча позволяет производить вставку данных относительно безболезненно и вам не надо будет бороться с накладными расходами на хранение и обслуживание, как это бывает в случае кластеризованного индекса.

Но отсутствие обновления и удаления данных не должны рассматриваться как единственная причина. Способ выборки данных также является важным фактором. К примеру, вы не должны использовать кучу, если часто выполняете запросы диапазонов данных или запрашиваемые данные часто должны быть сортированы или сгруппированы.

Всё это означает, что вы должны рассматривать возможность использования кучи только когда работаете с особо-маленькими таблицами или всё ваше взаимодействие с таблицей ограничено вставкой данных и ваши запросы чрезвычайно просты (и вы все-равно используете некластеризованные индексы). В противном случае держитесь хорошо спроектированного кластеризованного индекса, к примеру определенного на простом возрастающем ключевом поле, как широко применяемый столбец с IDENTITY.

Как изменить установленное по умолчанию значение коэффициента заполнения индекса?

Изменение установленного по умолчанию коэффициента заполнения индекса это одно дело. Понимание того как установленный по умолчанию коэффициент работает это другое. Но сначала пару шагов назад. Коэффициент заполнения индекса определяет количество пространства на странице для хранения индекса на нижнем уровне (уровень листьев) перед тем как начать заполнять новую страницу. К примеру, если коэффициент выставлен в значение 90, то при росте индекс займет на странице 90%, а затем перейдет на следующую страницу.

По умолчанию, значение коэффициента заполнения индекса в SQL Server равно 0, что равнозначно значению 100. В результате все новые индексы автоматически наследуют эту настройки, если вы специально в коде не укажете отличное от стандартного для системы значения или измените поведение по умолчанию. Вы можете воспользоваться SQL Server Management Studio для корректировки установленного по умолчанию значения или запустить системную сохраненную процедуру sp\_configure. К примеру, следующий набор T-SQL команд устанавливает значение коэффициента равное 90 (предварительно необходимо переключится в режим продвинутых настроек):

EXEC sp\_configure 'show advanced options', 1;

GO

RECONFIGURE;

GO

EXEC sp\_configure 'fill factor', 90;

GO

RECONFIGURE;

GO

После изменения значения коэффициента заполнения индекса необходимо перезагрузить сервис SQL Server. Теперь вы можете проверить установленное значение, запустив процедуру sp\_configure без указанного второго аргумента:

EXEC sp\_configure 'fill factor'

GO

Данная команда должна вернуть значение равное 90. В результате все вновь создаваемые индексы будут использовать это значение. Вы можете проверить это, создав индекс и запросить значение коэффициента заполнения:

USE AdventureWorks2012; -- ваша база данных

GO

CREATE NONCLUSTERED INDEX ix\_people\_lastname

ON Person.Person(LastName);

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

AND name='ix\_people\_lastname';

В данном примере мы создали некластеризованный индекс в таблице Person в базе данных AdventureWorks2012. После создания индекса мы можем получить значение коэффициента заполнения из системной таблиц sys.indexes. Запрос должен вернуть 90.

Однако, представим, что мы удалили индекс и снова создали его, но теперь указали конкретное значение коэффициента заполнения:

CREATE NONCLUSTERED INDEX ix\_people\_lastname

ON Person.Person(LastName)

WITH (fillfactor=80);

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

AND name='ix\_people\_lastname';

В этот раз мы добавили инструкцию WITH и опцию fillfactor для нашей операции создания индекса CREATE INDEX и указали значение 80. Оператор SELECT теперь возвращает соответствующее значение.

До сих пор всё было довольно-таки прямолинейно. Где вы реально можете погореть во всём этом процессе, так это когда вы создаёте индекс, использующий значение коэффициента по умолчанию, подразумевая, что вы знаете это значение. К примеру, кто-то неумело ковыряется в настройках сервера и он настолько упорот, что ставит значение коэффициента заполнения индекса равное 20. Тем временем вы продолжаете создавать индексы, предполагая значение по умолчанию равное 0. К сожалению, у вас нет способа узнать значение коэффициента до тех пор как вы не создадите индекс, а затем проверите значение, как мы делали в наших примерах. В противном случае, вам придётся ждать момента когда производительность запросов настолько упадёт, что вы начнёте что-то подозревать.

Другая проблема о которой вам стоит помнить это перестроение индексов. Как и при создании индекса вы можете конкретизировать значение коэффициента заполнения индекса, когда его перестраиваете. Однако, в отличие от команды создания индекса, перестройка не использует серверные настройки по умолчанию, несмотря на то что так может показаться. Даже больше, если вы конкретно не укажете значение коэффициента заполнения индекса, то SQL Server будет использовать то значение коэффициента, с которым этот индекс существовал до его перестройки. К примеру, следующая операция ALTER INDEX перестраивает только что созданный нами индекс:

ALTER INDEX ix\_people\_lastname

ON Person.Person REBUILD;

GO

SELECT fill\_factor FROM sys.indexes

WHERE object\_id = object\_id('Person.Person')

AND name='ix\_people\_lastname';

Когда мы проверим значение коэффициента заполнения мы получим значение равное 80, потому что именно его мы указали при последнем создании индекса. Значение по умолчанию не учитывается.

Как вы видите изменить значение коэффициента заполнения индекса не такое уж сложно дело. Намного сложнее знать текущее значение и понимать когда оно применяется. Если вы всегда конкретно указывается коэффициент при создании и перестройки индексов, то вы всегда знаете конкретный результат. Разве что вам приходится заботиться о том, чтобы кто-то другой снова не напортачил в настройках сервера, вызвав перестройку всех индексов со смехотворно низким значением коэффициента заполнения индекса.

Можно ли создать кластеризованный индекс на столбце, содержащем дубликаты?

И да, и нет. Да вы можете создать кластеризованный индекс на ключевом столбце, содержащем дубликаты значений. Нет, значение ключевого столбца не смогут остаться в состоянии не уникальности. Позвольте объяснить. Если вы создаёте неуникальный кластерный индекс (non-unique clustered index) на столбце, то подсистема хранения данных добавляет к дублирующему значению целочисленное значение (uniquifier), чтобы удостовериться в уникальности и, соответственно, обеспечить возможность идентифицировать каждую строку в кластеризованной таблице.

К примеру, вы можете решить создать в таблице с данными о клиентах кластеризованный индекс по столбцу LastName, хранящим фамилию. Столбец содержит такие значения как Franklin, Hancock, Washington и Smith. Затем вы вставляете значения Adams, Hancock, Smith и снова Smith. Но значение ключевого столбца обязательно должны быть уникальны, поэтому подсистема хранения данных изменит значение дубликатов таким образом, что они будут выглядеть примерно так: Adams, Franklin, Hancock, Hancock1234, Washington, Smith, Smith4567 и Smith5678.

На первый взгляд такой подход кажется нормальным, но целочисленное значение увеличивает размер ключа, что может стать проблемой при большом количестве дубликатов, а эти значения станут основой некластеризованного индекса или ссылкой внешнего ключа. По этим причинам вы всегда должны стараться создавать уникальный кластеризованный (unique clustered indexes) при любой возможности. Если это невозможно, то по крайней мере постарайтесь использовать столбцы с очень высоким содержание уникальных значений.

Как хранится таблица, если не был создан кластеризованный индекс?

SQL Server поддерживает два типа таблиц: кластеризованные таблицы, имеющие кластеризованный индекс и таблицы-кучи или просто кучи. В отличие от кластеризованных таблиц данные в куче не сортированы никоим образом. По сути это и есть нагромождение (куча) данных. Если вы добавите строку к такой таблице, то подсистема хранения данных просто добавит её к концу страницы. Когда страница заполнится данными, то они будут добавлены на новую страницу. В большинстве случаев, вы захотите создать кластеризованный индекс на таблице, чтобы получить преимущества от возможности сортировки и ускорения запросов (попробуйте представить себе найти телефонный номер в адресной книге, не отсортированной по какому-либо принципу). Однако, если вы решите не создавать кластеризованный индекс, то вы по-прежнему можете создать у кучи некластеризованный индекс. В этом случае каждая строка индекса будет иметь указатель на строку кучи. Указатель включает в себя идентификатор файла, номер страницы и номер строки с данными.

Какая взаимосвязь между ограничениями на уникальность значения и первичным ключом с индексами таблицы?

Первичный ключ и и ограничение уникальности обеспечивают, что значения в столбце будут уникальны. Вы можете создать только один первичный ключ у таблицы и он не может содержать значения NULL. Вы можете создать у таблицы несколько ограничений на уникальность значения и каждый из них может иметь единственную запись с NULL.

Когда вы создаете первичный ключ, подсистема хранения данных так же создает уникальный кластеризованный индекс, в случае если уже кластеризованный индекс не был создан. Однако, вы можете переопределить установленное по умолчанию поведение и тогда будет создан некластеризованный индекс. Если кластеризованный индекс существует когда вы создаёте первичный ключ, то будет создан уникальный некластеризованный индекс.

Когда вы создаете ограничение на уникальность, подсистема хранения данных создает уникальный некластеризованный индекс. Но вы можете указать создание уникального кластеризованного индекса, если он не был создан ранее.

В общем случае, ограничение на уникальность значение и уникальный индекс это одно и то же.

Почему в SQL Server кластеризованные и некластеризованные индексы называются сбалансированным деревом?

Базовые индексы в SQL Server, кластеризованные или некластеризованные, распространяются по наборам страниц – узлам индекса. Эти страницы организованы в виде определенной иерархии с древовидной структурой, называемой сбалансированным деревом. На верхнем уровне находится корневой узел, на нижнем, конечные узлы листьев, с промежуточными узлами между верхним и нижним уровнями, как показано на рисунке:

Корневой узел предоставляет главную точку входа для запросов, пытающихся получить данные через индекс. Начиная с этого узла, подсистема запросов инициирует переход по иерархической структуре вниз к подходящему конечному узлу, содержащему данные.

К примеру, представим, что поступил запрос на выборку строк, содержащих значение ключа равное 82. Подсистема запросов начинает работу с корневого узла, который отсылает к подходящему промежуточному узлу, в нашем случае 1-100. От промежуточного узла 1-100 происходит переход к узлу 51-100, а оттуда к конечному узлу 76-100. Если это кластеризованный индекс, то на листе узла содержится данные строки, ассоциированной с ключом равным 82. Если же это некластеризованный индекс, то лист индекса содержит указатель на кластеризованную таблицу или конкретную строку в куче.

Как вообще индекс может улучшить производительность запросов, если приходится переходить по всем этим индексным узлам?

Во-первых, индексы не всегда улучшают производительность. Слишком много неверно созданных индексов превращают систему в болото и понижают производительность запросов. Правильнее сказать, что если индексы были аккуратно применены, то они могут обеспечить значительный прирост в производительности.

Подумайте об огромной книге, посвященной настройке производительности SQL Server (бумажной, не об электронном варианте). Представьте, что вы хотите найти информацию о конфигурировании Регулятора ресурсов. Вы можете водить пальцем постранично через всю книгу или открыть содержание и узнать точный номер страницы с искомой информацией (при условии, что книга правильно проиндексирована и в содержании верные указатели). Безусловно, это сэкономит вам значительное время, не смотря на то, что вам надо сначала обратиться к совершенно другой структуре (индексу), чтобы получить необходимую вам информацию из первичной структуры (книги).

Как и книжный указатель, указатель в SQL Server позволяет вам выполнять точные запросы к нужным данным вместо полного сканирования всех данных, содержащихся в таблице. Для маленьких таблиц полное сканирование обычно не проблема, но большие таблицы занимают много страниц с данными, что в результате может привезти с значительному времени выполнения запроса, если не существует индекса, позволяющего подсистеме запросов сразу получить правильное месторасположение данных. Представьте, что вы заблудились на многоуровневой дорожной развязке перед крупным мегаполисом без карты и вы поймёте идею.

Если индексы настолько замечательны, то почему бы просто не создать их на каждый столбец?

Ни одно доброе дело не должно оставаться безнаказанным. По крайней мере, именно так и обстоит дело с индексами. Разумеется, индексы отлично себя показывают, пока вы выполняете запросы на выборку данных оператором SELECT, но как только начинается частый вызов операторов INSERT, UPDATE и DELETE, так пейзаж очень быстро меняется.

Когда вы инициируется запрос данных оператором SELECT, подсистема запросов находит индекс, продвигается по его древовидной структуре и обнаруживает искомые данные. Что может быть проще? Но все меняется, если вы инициируете оператор изменения, такой как UPDATE. Да, для первой части оператора подсистема запросов может снова использовать индекс для обнаружения модифицируемой строки – это хорошие новости. И если происходит простое изменение данных в строке, не затрагивающее изменение ключевых столбцов, то процесс изменения пройдет вполне безболезненно. Но что, если изменение приведет к разделению страниц, содержащих данные, или будет изменено значение ключевого столбца, приводящее к переносу его в другой индексный узел – это приведёт к тому, что индексу может потребоваться реорганизация, затрагивающая все связанные индексы и операции, в результате будет повсеместное падение производительности.

Аналогичные процессы происходят при вызове оператора DELETE. Индекс может помочь найти месторасположение удаляемых данных, но само по себе удаление данных может привести к перестановке страниц. Касаемо оператора INSERT, главного врага всех индексов: вы начинаете добавлять большое количество данных, что приводит к изменению индексов и их реорганизации и все страдают.

Так что учитывайте виды запросов к вашей базе данных при размышлениях какой тип индексов и в каком количестве стоит создавать. Больше не значит лучше. Перед тем как добавить новый индекс на таблицу просчитайте стоимость не только базовых запросов, но и объем занимаемого дискового пространства, стоимость поддержания работоспособности и индексов, что может привести к эффекту домино для других операций. Ваша стратегия проектирования индексов один из важнейших аспектов внедрения и должна включать в рассмотрение множество соображений: от размера индекса, количества уникальных значений, до типа поддерживаемых индексом запросов.

Обязательно ли создавать кластеризованный индекс на столбце с первичным ключом?

Вы можете создать кластеризованный индекс на любой столбце, соответствующем необходимым условиям. Это верно, что кластеризованный индекс и ограничение первичного ключа созданы друг для друга и их брак заключен на небесах, так что усвойте факт, что когда вы создаете первичный ключ, тогда же будет автоматически создан кластеризованный индекс, если он не был создан ранее. Тем не менее, вы можете решить, что кластеризованный индекс будет лучше работать в другом месте, и часто ваше решение будет вполне оправданным.

Главная цель кластеризованного индекса это сортировка всех строк к вашей таблице на основе ключевого столбца, указанного при определении индекса. Это обеспечивает быстрый поиск и легкий доступ к данным таблицы.

Первичный ключ таблицы может быть хорошим выбором, потому что он однозначно идентифицирует каждую строку в таблицы без необходимости добавлять дополнительные данные. В некоторых случаях лучшим выбором будет суррогатный первичный ключ, обладающий не только признаком уникальности, но и малым размером, а значения которого увеличиваются последовательно, что делает некластеризованные индексы, основанные на этом значении более эффективными. Оптимизатор запросов также любит такое сочетание кластеризованого индекса и первичного ключа, потому что соединение таблиц происходит быстрее, чем при соединении другим способом, не использующим первичный ключ и ассоциированный с ним кластеризованный индекс. Как я и говорил это брак, заключенный на небесах.

В конце стоит, однако, отметить, что при создании кластеризованного индекса необходимо принять во внимание несколько аспектов: как много некластеризованных индексов будет основываться на нём, как часто будут изменяться значение ключевого столбца индекса и на сколько ни большие. Когда значение в столбцах кластеризованого индекса изменятся или индекс не будет обеспечивать должной производительности, тогда все другие индексы таблицы могут быть задеты. Кластеризованный индекс должен быть основан на наиболее устойчивом столбце, значения которого увеличиваются в определенном порядке, но не изменяются в случайном. Индекс должен поддерживать запросы к наиболее часто используемым данным таблицы, таким образом запросы получают все преимущества того, что данные сортированы и доступны на корневых узлах, листьях индекса. Если первичный ключ соответствует этому сценарию, то используйте его. Если же нет, то выберите другой набор столбцов.

А что если проиндексировать представление, то это по-прежнему будет представление?

Представление – это виртуальная таблица, формирующая данные из одной или нескольких таблиц. По сути, это именованный запрос, который получает данные из нижележащих таблиц, когда вы вызываете запрос к этому представлению. Вы можете улучшить производительность запросов, создав кластеризованных индекс и некластеризованные индексы у этого представления, аналогично как вы создаете индексы у таблицы, но основной нюанс состоит в том, что первоначально создается кластеризованный индекс, а затем вы можете создать некластеризованный.

Когда создается индексированное представление (материализованное представление), тогда само определение представления остается отдельной сущностью. Это, в конце концов, всего лишь жестко прописанный оператор SELECT, хранящийся в базе данных. А вот индекс совсем другая история. Когда вы создаете кластеризованный или некластеризованный индекс у предастваления, то данные физически сохраняются на диск, аналогично обычному индексу. В дополнение, когда в нижележащих таблицах изменяются данные, то индекс представления автоматически изменяется (это означает, что вы можете захотеть избежать индексирования представлений тех таблиц, в которых происходят частые изменения). В любом случае, представление остается представлением — взглядом на таблицы, но именно выполненном в данный момент, с индексами ему соответствующими.

Перед тем как вы сможете создать индекс у представления, оно должно соответствовать нескольким ограничениям. К примеру, представление может ссылаться только на базовые таблицы, но не другие представления и эти таблицы должны находиться в той же самой базе данных. На самом деле там множество других ограничений, так что не забудьте обратиться к документации по SQL Server за всеми грязными подробностями.

Зачем использовать покрывающий индекс взамен составного индекса?

Во-первых, давайте убедимся, что мы понимаем различие между ними. Составной индекс это просто обычный индекс, в который включено больше одного столбца. Несколько ключевых столбцов может использоваться для обеспечения уникальности каждой строки таблицы, также возможен вариант, когда первичный ключ состоит из нескольких столбцов, обеспечивающих его уникальность, или вы пытаетесь оптимизировать выполнение часто вызываемых запросов к нескольким столбцам. В общем, однако, чем больше ключевых столбцов содержит индекс, тем менее эффективна работа этого индекса, а значит составные индексы стоит использовать разумно.

Как было сказано, запрос может извлечь огромную выгоду, если все необходимые данные сразу расположены на листьях индекса, как и сам индекс. Это не проблема для кластеризованного индекса, т.к. все данные уже там (вот почему так важно хорошенько подумать когда вы создаете кластеризованный индекс). Но некластеризованный индекс на листьях содержит только ключевые столбцы. Для доступа ко всем остальным данным оптимизатору запросов необходимы дополнительные шаги, что может вызвать значительные дополнительные накладные расходы для выполнения ваших запросов.

Вот где покрывающий индекс спешит на помощь. Когда вы определяете некластеризованный индекс, то можете указать дополнительные столбцы к вашим ключевым. К примеру, представим, что ваше приложение часто запрашивает данные столбцов OrderID и OrderDate в таблице Sales:

SELECT OrderID, OrderDate

FROM Sales

WHERE OrderID = 12345;

Вы можете создать составной некластеризованный индекс на обоих столбцах, но столбец OrderDate только добавит накладных расходов на обслуживание индекса, но так и не сможет служить особо полезным ключевым столбцом. Лучшее решение будет это создание покрывающего индекса с ключевым столбцом OrderID и дополнительно включенным столбцом OrderDate:

CREATE NONCLUSTERED INDEX ix\_orderid

ON dbo.Sales(OrderID)

INCLUDE (OrderDate);

При этом вы избегаете недостатков, возникающих при индексации излишних столбцов, в то же время сохраняете преимущества хранения данных на листьях при выполнении запросов. Включенный столбец не является частью ключа, но данные хранятся именно на конечном узле, листе индекса. Это может улучшить производительность выполнения запроса без каких либо дополнительных расходов. К тому же, на столбцы, включенные в покрывающий индекс, накладывается меньше ограничений, нежели на ключевые столбцы индекса.

Имеет ли значение количество дубликатов в ключевом столбце?

Когда вы создаете индекс, вы обязаны постараться уменьшить количество дубликатов в ваших ключевых столбцах. Или более точно: стараться держать коэффициент повторяющихся значений настолько низким, насколько это возможно.

Если вы работаете с составным индексом, то дублирование относится ко всем ключевым столбцам в целом. Отдельный столбец может содержать множество повторяющихся значений, но повторения среди всех столбцов индекса должно быть минимальным. К примеру, вы создаете составной некластеризованный индекс на столбцах FirstName и LastName, вы можете иметь множество значений равных John и множество Doe, но вы хотите иметь как можно меньше значений John Doe, или лучше только одно значение John Doe.

Коэффициент уникальности значений ключевого столбца называется избирательностью индекса. Чем больше уникальных значений, тем выше избирательность: уникальный индекс обладает наибольшей возможной избирательностью. Подсистема запросов очень любит столбцы с высоким значением избирательности, особенно если эти столбцы участвуют в условиях выборки WHERE ваших наиболее часто выполняемых запросов. Чем выше избирательность индекса, тем быстрее подсистема запросов может уменьшить размер результирующего набора данных. Обратной стороной, разумеется, является то, что столбцы с относительно небольшим количеством уникальных значений редко будут хорошими кандидатами на индексирование.

Можно ли создать некластеризованный индекс только для определенного подмножества данных ключевого столбца?

По умолчанию, некластеризованный индекс содержит по одной строке для каждой строки таблицы. Конечно, вы можете сказать то же самое относительно кластеризованного индекса, принимая в расчет, что такой индекс это и есть таблица. Но что касается некластеризованного индекса, то отношение «один к одному» важный концепт, потому что, начиная с версии SQL Server 2008, у вас есть возможность создать фильтруемый индекс, который ограничивает включенные в него строки. Фильтруемый индекс может улучшить производительность выполнения запросов, т.к. он меньше по размеру и содержит отфильтрованную, более аккуратную, статистику, чем вся табличная — это приводит к созданию улучшенных планов выполнения. Фильтруемый индекс также требует меньше места для хранения и меньших затрат на обслуживание. Индекс обновляется только когда изменяются подходящие под фильтр данные.

В дополнение, фильтруемый индекс легко создать. В операторе CREATE INDEX просто необходимо указать в WHERE условие фильтрации. К примеру, вы можете отфильтровать из индекса все строки, содержащие NULL, как показано в коде:

CREATE NONCLUSTERED INDEX ix\_trackingnumber

ON Sales.SalesOrderDetail(CarrierTrackingNumber)

WHERE CarrierTrackingNumber IS NOT NULL;

Мы можем, фактически, отфильтровать любые данные, которые не важны в критических запросах. Но будьте внимательны, т.к. SQL Server накладывает несколько ограничений на фильтруемые индексы, такие, как невозможность создать фильтруемый индекс у представления, так что внимательно читайте документацию.

Также, может случиться, что вы можно достичь подобных результатов созданием индексированного представления. Однако, фильтруемый индекс имеет несколько преимуществ, таких как возможность уменьшить стоимость обслуживания и улучшить качество ваших планов выполнения. Фильтруемые индексы также допускают перестройку в онлайн-режиме. Попробуйте это сделать с индексируемым представлением.