Лекция №1

###### Информационные системы

Термин «информационная система» соответствует системе, используемой для получения нужной в данный конкретный момент времени информации. В настоящее время информационные системы (ИС) функционируют на ЭВМ (компьютерах).

Информационные системы можно разделить на:

1. одиночные;
2. групповые;
3. корпоративные.

Одиночные системы реализуются на персональных компьютерах (ПК), содержат БД и несколько простых приложений, рассчитаны на работу одного пользователя, создаются с помощью “настольных” СУБД (Foxpro, Paradox, Access, Dbase, Clarion, Clipper).

Групповые системы ориентированы на коллективное использование информации, строятся как локальная вычислительная сеть ПК или централизованная вычислительная система (в основе большая ЭВМ и множество терминалов (ПК), обращающихся к этой ЭВМ). Совместное использование информации из БД организуется с помощью блокировок записей и таблиц (файлов). Для разработки ИП используются многопользовательские “настольные” СУБД и серверы БД для рабочих групп, а также инструментальные средства разработки.

Корпоративные системы поддерживают территориально разнесенные узлы сети, используют архитектуру клиент-сервер со специализацией серверов, а также независимые многоплатформенные инструментальные средства (PowerBuilder, Visual Basic, Visual C и др.), интерфейсы, драйверы и шлюзы для связи с разными СУБД.

Основные типы ИС:

1. Обработка транзакций (OLTP):

А) оперативная обработка транзакций;

Б) пакетная обработка транзакций (на больших ЭВМ).

1. Системы поддержки принятия решений (DSS) – технология OLAP:

А) Оперативная аналитическая обработка;

Б) Экспертные системы.

1. Информационно-справочные системы:

А) системы электронной документации;

Б) географические информационные системы;

В) гипертекстовые системы.

1. Офисные информационные системы:

А) документальные системы;

Б) автоматизация делопроизводства;

В) управление документооборотом.

В нашем курсе рассматриваются первые 2 типа ИС (обработка транзакций и оперативная аналитическая обработка), где фундаментом ИС является БД.

Информационное приложение (ИП) - прикладная программная подсистема, ориентированная на сбор, хранение, поиск и обработку текстовой и/или структурированной информации, работающая с пользователем в диалоговом режиме.

В ИП входят следующие программные компоненты:

1. Средства представления диалогового ввода/вывода – **PS** (текстовый терминал или ПК в режиме эмуляции);
2. Логика представления диалогового ввода/вывода – **PL** (взаимодействие пользователя с ЭВМ: выбор пункта меню, нажатие кнопки, выбор элемента из списка и т.д.);
3. прикладная логика обработки данных – **BL** (набор правил для принятия решений, вычислений и операций приложения);
4. логика управления данными – **DL** (операции с БД: SELECT, UPDATE, INSERT, DELETE, реализующие прикладную логику обработки данных);
5. операции с БД – **DS** ( определение данных, манипулирование данными, фиксация или откат транзакций, компиляция SQL-операторов и др.);
6. функции ОС – **FS** (дисковые операции чтения и записи данных для СУБД, обеспечение взаимодействия узлов сети и др.).

Значительная часть возможностей приложения закладывается в системном программном обеспечении, в библиотеках и конструкциях инструментальных средств разработки. Однако обязательна часть ИП, специфичная для конкретной предметной области.

Кроме программной составляющей в ИП есть информационная составляющая, задающая:

1. структуру данных;
2. атрибутику и типизацию данных;
3. ограничения целостности БД.

Информационная составляющая тесно связана с логикой управления данными, поэтому CASE-средства для приложений отдают приоритет информационной модели.

Существует 7 вариантов построения ИП:

1. многотерминальные централизованные вычислительные системы;
2. файл-серверные приложения;
3. системы с архитектурой клиент-сервер;
4. системы с распределенными вычислениями;
5. офисные системы;
6. системы на основе Internet/Intranet-технологий;
7. средства автоматизации проектирования приложений.

I Многотерминальные централизованные системы

В централизованной системе терминал реализует функции PS, а остальные функции (PL, BL, DL, DS, FS) – центр (большая ЭВМ). ИП разрабатываются с помощью традиционных систем программирования представлены 3GL (C, Pascal, Basic и др.). Для создания полноценных ИП необходимо расширение за счет библиотек диалога и доступа к БД, а также встроенного языка SQL.

В этих системах имеются две большие проблемы:

1. трудно обеспечить графический интерфейс;
2. каждый дополнительный пользователь и ИП вносят существенную нагрузку на сервер (теряется масштабируемость).

II Файл-серверные приложения

Объектами разработки в файл-серверном приложении являются компоненты PL, BL и DL. Каждый новый клиент добавляет вычислительную мощность к сети. Для разработки ИП используется инструментальное окружение «настольных» СУБД (FoxPro, Access, Paradox и др.). Эти средства предоставляют 3 уровня доступа:

1. к языку 3GL добавляются некоторые возможности языков 4GL;
2. создание и ведение структуры БД и индексов, а также интерактивная генерация ИП и его компонентов;

3) использование диалоговой среды и генераторов конечными пользователями для создания, ведения и просмотра БД, а также формирования несложных запросов и отчетов.

Однако “файл-сервер” имеет 3 недостатка:

1. некоторые запросы к БД могут перекачивать всю БД, что создает значительный сетевой трафик и время реакции;
2. проблема “толстого клиента” (Windows-интерфейс, СУБД и само ИП) - возможна перегрузка ПК;

3) “настольные” СУБД не поддерживают транзакции (не отвечают требованиям сохранности данных).

III Приложения клиент-сервер

Особенности “клиент-сервера”:

1. использование выделенных серверов БД, понимающих SQL-операторы;
2. выполнение поиска, сортировки и агрегирования информации серверами без перекачки данных на рабочие станции;
3. сервер БД содержит словарь-справочник данных, в котором записаны структура БД, ограничения целостности данных, форматы и серверные продукты обработки данных.

Для эффективной реализации двухзвенной модели «клиент-сервер» ИП располагается на клиенте, а СУБД - на сервере. Т.е. компоненты PS, PL (диалога) и BL, DL (логики) располагают на клиенте, а компоненты управления данными DS и FS - на сервере. Сложные ИП могут перегрузить клиента и сеть, поэтому компонент BL можно переместить на сервер, оформив эту логику в виде хранимых процедур. Хранимые процедуры улучшают целостность приложений и БД, а также безопасность доступа к данным.

Среды разработки ИП для серверов БД – это системы программирования 4GL или инструментальные средства быстрой разработки ИП - RAD.

Двухзвенные схемы «клиент-сервер» имеют следующие проблемы:

1. необходимость администрирования приложений для большого числа клиентов;
2. чрезмерное использование хранимых процедур снижает масштабируемость сервера и не способствует переносимости ИП;
3. возрастает время реакции для диалоговых пользователей во время работы пакетных заданий;
4. обеспечение целостности распределенной транзакции в неоднородной распределенной БД.

IV Распределенные вычисления

Распределенные вычисления выполняются с помощью трехзвенной архитектуры «клиент-сервер», где явно выделяется сервер приложений, на котором выполняются процедуры, запросы и задания клиента.

Клиент выделен для выполнения компонент PS и PL и имеет программный интерфейс для вызова ИП на сервере приложений. Сервер приложений выполняет BL и с которого DL вызывает операции с БД - DS. Сервер БД выделен для DS и FS.

Достоинства трехзвенной архитектуры:

1. централизация логики приложения упрощает администрирование и сопровождение, а также обеспечивает доступ к ресурсам по именам;
2. четко разделяются платформы и инструменты для реализации интерфейса и прикладной логики, что позволяет с наибольшей отдачей реализовать их специалистам узкого профиля;
3. изменения прикладной логики не затрагивают интерфейса, и наоборот;
4. границы между PL, BL и DL размыты, поэтому прикладная логика может появиться во всех трех звеньях;
5. сервер приложений с помощью монитора транзакций обеспечивает интерфейс с клиентами и другими серверами, может управлять транзакциями и гарантировать целостность распределенной БД путем двухфазной фиксации в неоднородной среде;
6. средства удаленного вызова процедур обеспечивают вызов из любого узла сети прикладной процедуры, расположенной на другом узле, передачу параметров, удаленную обработку и возврат результатов;
7. сервер приложений управляет асинхронными очередями сообщений, которые обеспечивают надежное завершение транзакций.

К средствам поддержки распределенных приложений относятся:

1. интерфейсы доступа к БД ODBC и IDAPI;
2. шлюзы для СУБД;
3. протоколы и команды мониторов обработки транзакций;
4. почтовые интерфейсы MAPI, VIM, MHS, X.400 и EDI;
5. средства обмена сообщениями MOM;
6. протоколы связывания и включения объектов OLE и динамического обмена данными DDE;
7. протоколы удаленного вызова процедур RPC и именованных конвейеров Named Pipes;
8. средства коммуникационного ввода-вывода BSD Sockets и WinSock.

Инструментальные наборы для разработки приложений клиент-сервер необходимо выбирать, исходя из следующих критериев:

1. объектно-ориентированная инфраструктура;
2. распределение приложений между клиентом и сервером;
3. поддержка мониторов транзакций;
4. CASE-репозитарий;

5) возможность переноса приложений и контроль версий.

V Офисные системы

В офисных системах для хранения информации, обрабатываемой с помощью текстовых процессоров и электронных таблиц, используется файловый или почтовый серверы. В системах управления электронными документами файл-сервер используется для хранения документов, а сервер БД служит для ведения библиотеки карточек документов. Существуют универсальные серверы БД, которые интегрируют структурированную и неструктурированную информацию.

VI Информационные системы на основе Internet/Intranet-технологии

В этой технологии клиент выполняет функции интерфейса пользователя и обеспечивает доступ к Internet. БД HTML-документов - это часть файловой системы, которая содержит текстовые файлы в формате гипертекста и связанные с ними графику и другие ресурсы.

При работе с приложениями БД адрес URL указывает не на Web-страницу, а на программу (сценарий), которая запускает запрос к БД и преобразует результаты (отчет по запросу) в формат HTML. Затем Web-сервер посылает полученную HTML-страницу Web-клиенту.

VII Средства автоматизации проектирования приложений

Для автоматизации проектирования ИП используются CASE-системы, которые поддерживают целостность системы в течение жизненного цикла, обеспечивают коллективную разработку, параллельное внесение изменений, откат к предыдущей версии и контроль доступа.

**Лекция № 2**

**Базы данных**

# История появления БД

1. ПП 2) ОП ПП 3) ПП 4) ПП

ОП Д

ВП Д ОП МД ОС ОП СУБД

Д МД ОС

Д

1. ППП

ОС СУБД БД АИС

Словарь БД

Существуют самостоятельно **наука о БД** и **технология БД**.

Разделы теории (науки) БД:

1. Теория моделей данных и методы их эквивалентных преобразований;
2. Теория функциональных зависимостей в БД;
3. Методы эквивалентных представлений БД и их схем;
4. Проблемы полноты и эквивалентности в реляционном исчислении;
5. Теория БД с неполной информацией;
6. Методы обеспечения целостности БД в условиях потока конкурентных запросов;
7. Методы структурирования памяти для хранения разнообразных данных.

Мы будем изучать технологию БД. Основное, что изучают в технологии БД – это проектирование БД, создание БД и эксплуатация БД. Чтобы хорошо знать и владеть БД, нужно хорошо знать ОС и СУБД, языки и технологии программирования, системы обработки данных (СОД), системы искусственного интеллекта.

**Основная концепция технологии БД:**

С одной стороны, данные должны быть структурированы так, чтобы наиболее адекватно отображать предметную область (ПО), с другой стороны, средства общения пользователей с данными должны быть такими, чтобы наиболее полно, своевременно, достоверно обеспечить пользователя информацией.

**Основные понятия курса (определения):**

1. БД – это большой набор именованных данных, расположенный во внешней памяти и отображающий предметную область.
2. ПО – это совокупность объектов, их свойств и связей между ними.
3. СУБД – это мощный комплекс технических, языковых и программных средств для создания, управления и использования БД многими пользователями.
4. СОД – это система, предназначенная для накопления, хранения, ведения, коллективного и многоаспектного использования данных в интересах конечного пользователя.

СОД включает в себя:

* 1. одна или несколько БД;
  2. СУБД;
  3. Вычислительные средства (компьютеры + общее программное обеспечение);
  4. Группа администратора БД;
  5. Словарь-справочник (метаданные).

**Пример ПО «Учащийся»:**

Свойства объекта «Учащийся»: Свойства объекта «Учебное заведение»:

1) Фамилия; 1) Название;

2) Имя; 2) Город, в котором находится;

3) Отчество; 3) Министерство, которому подчинено;

4) Дата рождения; 4) Страна, в которой находится.

5)Семейное положение;

6)Специальность;

7) Форма обучения;

8) Учебное заведение;

9) Дата приема;

10) Дата окончания.

Современные требования к СУБД:

1. поддержка широкого спектра представлений данных и операций над ними в БД (включая структурные, документальные, картинно-графические данные);
2. управление неоднородными мультибазами данных;
3. управление распределенными БД и интеграция неоднородных БД;
4. естественные и эффективные представления в моделях данных разнообразных отношений объектов ПО (например, пространственно-временных с обеспечением визуализации данных);
5. интеграция концепций БД и баз знаний и реализация дедуктивных БД;
6. обеспечение целостности БД в широком диапазоне разнообразных ПО и операционных обстановок.

Трудности развития СУБД:

1. фон-неймановская архитектура ЭВМ не соответствует требованиям СУБД, так как поиск, обновление, защита данных и обработка транзакций с помощью программ осуществляются неэффективно ( по производительности и по стоимости);
2. многоуровневое и сложное программное обеспечение СУБД снижает эффективность и надежность функционирования БД и СОД в целом;
3. ЭВМ оказываются существенно загруженными функциями управления БД;
4. Централизация и интеграция данных в сетях ЭВМ не реализуема с приемлемой стоимостью без специальных архитектурных и аппаратурных решений.

Примеры СУБД:

|  |  |  |
| --- | --- | --- |
| СУПЕРСУБД: | Средние СУБД: | Малые СУБД: |
| 1) INFORMIX; | 1) UNIFY; | 1)dBase; |
| 2) ORACLE; | 2) UNISQL; | 2) FoxPro; |
| 3) SYBASE; | 3) EMPRESS; | 3) Paradox; |
| 4) MS SQL SERVER; | 4) Btrieve. | 4) Access; |
| 5) DB2. |  | 5) Clarion; |
|  |  | 6) Clipper. |

## Характеристики СУБД

I Функциональная полнота – это набор средств, необходимый для успешной разработки и сопровождения СОД. Эти средства делятся на:

1. Язык описания данных (ЯОД), предоставляющий большое множество типов данных;
2. Язык манипулирования данными (ЯМД), примерами которого являются SQL, QBE, Transact SQL, языки программирования «настольных» СУБД и др.
3. CASE-средства (например: ERWIN) для автоматизации процессов описания модели ПО, ее проектирования и автоматическое отображение в описание модели ПО средствами СУБД.

II Разграничение прав доступа к данным для различных категорий пользователей (полномочия и роли), а также парольная защита, шифрование, создание представлений, сжатие и др.

III Различные технологии многопользовательского режима работы:

* + файл-сервер (запросы реализуются на клиенте);
  + клиент-сервер (запросы реализуются на сервере);
  + распределенные БД.

IV Поддержка средств multi-media (видео и звук).

V Разработка современного многооконного графического интерфейса для различных групп пользователей. Для этого нужно использовать современный язык объектно-ориентированного программирования, умеющего работать с БД (примеры: VB, VC, PB).

VI Экспорт-импорт данных для различных программных продуктов.

VII Интеграция с программными и информационными ресурсами сетей различных типов.

VIII Защита данных от потерь и разрушений.

Аварийные ситуации: 1) технические сбои; 2) неправильные действия пользователя.

Защита: 1) многооконный интерфейс; 2) блоки бесперебойного питания; 3) наличие архивных копий, журналов и др.

IX Задание компьютерной платформы.

**Основные свойства БД:**

I **Объемность**: БД по мере своего развития имеют тенденцию роста своего объема: малые БД (до 1 Гб), средние БД (1 : 10 ГБ) и большие БД ( > 10 Гб). ОП для малых БД – до 32 Мб, для средних – 32 : 64 Мб, для больших > 64 Мб.

II **Коллективное использование и производительность**:

Коллективное использование в трех режимах: а) пакетный; б) диалоговый; в) режим «запрос-ответ».

Все режимы работают: а) как локальный, так и удаленный варианты;

б) как в режиме реального времени, так и в режиме разделения вычислительных ресурсов.

2 Характеристики производительности:

1) Время ответа на запрос:  t = t 1символ ответа – t последний символ запроса < 2 сек. (при пакетном режиме это не существенно).

2) Пропускная способность – количество сообщений на единицу времени.

III **Интегрируемость**:

1. Исходные данные вводятся в БД 1 раз, затем они только корректируются;
2. Данные используются всеми пользователями;
3. Каждый пользователь имеет свою часть БД;
4. Часть БД используется несколькими пользователями одновременно (режим блокировки данных).

Интегрируемость дает min избыточность данных и устраняет противоречивость данных. Если в БД появляются избыточные данные, то требуется процедура реорганизации БД.

IV **Взаимосвязь данных или отношения в БД:**

1. между двумя объектами (отношения вхождения, порядка, следования, подчиненности и т.д.);
2. внутри одного объекта (между свойствами);
3. количественные отношения между объектами:

1:1 – однозначное;

1:n – иерархическое;

n:1

n:m - сетевое.

V Централизация обработки: группа АБД создает концептуальную модель ПО.

Традиционная обработка: Централизованная обработка:

П1Ф1 П2Ф2 П1 Ф1 П3

БД: Ф2

П3Ф3 П4Ф4 Ф3

П2 Ф4 П4

Централизация обработки дает:

1. min избыточность;
2. отсутствие противоречивости;
3. разработка новых программ для имеющихся данных;
4. защита от несанкционированного доступа;
5. обеспечение целостности данных;
6. возможность стандартизации процедур обработки данных, документации и всей технологии;
7. комплексная оптимизация ( - это удовлетворение некоторой целевой функции, которая формируется руководителем ПО).

VI Экономичность (БД позволяет упростить программирование приложений).

VII Форматность: в БД каждый объект однозначно определяется совокупностью его свойств.

VIII Унификация доступа к данным: каждый пользователь работает со своим представлением БД с помощью операторов ЯМД.

IX Ответы на непредусмотренные запросы (детерминированные и недетерминированные запросы по информации и по времени).

X Независимость программ от данных:

1. логическая (глобальный логический уровень данных);
2. физическая (физический уровень данных).

В СУБД используется динамическая привязка прикладной программы (ПП) к данным (непосредственно в процессе обращения к данным). Введение СУБД подняло на более высокий уровень логическое представление о данных.

**Лекция №3**

**Этапы проектирования БД**

**Обобщенная архитектура СУБД**

П1 П2 … Пi Пj … Пn

ВМ1БД … ВМkБД

О1

АБД К М Б Д

О2 УП СУБД

Вн М Б Д

Ф Б Д О С

Эта схема предложена группой ANSI/X3/SPARC института стандартов США и является общепризнанной при проектировании СУБД.

I Пользователи (П1, П2, Пi, Пj, Пn): 1) прикладные программисты; 2) АБД; 3)конечные пользователи - непрогаммисты. Каждый пользователь имеет один или несколько языков для общения с СУБД.

II Внутренняя модель ПО – это хранимый уровень представления данных. На этом уровне представлены структуры хранимых данных и способы доступа к ним. Внутренняя модель включает хранимые файлы, хранимые записи и определяет связи между записями. Известны методы доступа СУБД к этим записям (HSAM, HISAM, HDAM, HIDAM и др.). Описание этой модели осуществляется внутренней схемой. Связь ВнМБД с ФБД осуществляется методами доступа ОС.

III Концептуальная модель ПО – это глобально-логический уровень представления данных:

1. В ней отображены информационные потребности предметной области в целом и каждого пользователя в отдельности.
2. В ней отражены объекты, свойства, связи между ними. Эти элементы представлены в виде концептуальных файлов, записей и полей.
3. В КМ нет ничего от хранимого уровня представления данных.
4. В КМ отражены правила разграничения прав доступа и правила сохранения целостности данных.
5. Определение КМ осуществляется с помощью концептуальной схемы.

IV Внешняя модель ПО (логический уровень пользователя):

* + ВМ отражает информационное содержание БД в таком виде, в каком оно используется в конкретном приложении.
  + Допускается множество таких ВМ.
  + ВМ специфицируется внешней схемой, при этом описываются файлы, записи, поля ВМ.

V Отображения:

О1 – соответствие между ВМ и КМ;

О2 - соответствие между КМ и ВнМ.

Эти соответствия также имеют описания и хранятся в СУБД (как и схемы ВМ, КМ и ВнМБД). АБД, используя описание отображений и утилиты устанавливает соответствие между МД.

При изменениях на уровне КМ АБД меняет отображение О1 таким образом, чтобы не было изменений во ВМ. Таким образом обеспечивается логическая независимость ПП от данных.

При изменениях на уровне ВнМ АБД меняет отображение О2 таким образом, чтобы не было изменений на уровне КМ. Тем самым обеспечивается физическая независимость ПП от данных.

**Этапы проектирования БД**

IА)

МПО

ЛП1 … ЛПn

IБ)

КМ ПО

II

ВМ1БД … ВМnБД

IIIА)

КМ БД

IIIБ)

ВнМ БД

I Информационно-логическое проектирование:

А) Анализ предметной области;

Б) Концептуальное проектирование ПО.

II Выбор СУБД

III Даталогическое проектирование:

А) Логическое проектирование;

Б) Физическое проектирование.

IV Проектирование словаря-справочника.

I Информационно-логическое проектирование

Задача информационно-логического проектирования получить КМ ПО и локальные представления (ЛП) пользователей, не зависящих от конкретной СУБД.

А) Анализ ПО – анализируются информационные потоки: источники / потребители информации, документопотоки, технология хранения, обработки и использования документов. В результате анализа определяется:

1. Перечень сервисных и функциональных задач, возложенных на компьютер.
2. Кодификаторы и классификаторы информационных объектов.

Б) Концептуальное проектирование (моделирование) ПО.

Формируются ЛП для каждого конечного пользователя (КП). ЛП – это фрагмент ПО, удовлетворяющий запросы КП. Путем интеграции ЛП создается КМ ПО. Это процесс итерационный. КМ ПО удовлетворяет запросы всех конечных пользователей. КМ ПО не зависит от используемой СУБД.

Этапы КМ ПО:

1. Выделение объектов, их свойств и связей между ними. Связи устанавливаются с учетом задач (запросов), решаемых КП.
2. На основе выделенных задач строятся сценарии диалога КП с ЭВМ (в виде графа, дерева). КМ осуществляется с помощью CASE-средств (например: ERWIN).

II Выбор СУБД

* 1. Установка внешних ограничений:
  + ограничения вычислительного комплекса (модель ЭВМ, конфигурация, объем памяти, скорость обмена данных по шине);
  + ОС;
  + Ограничения сети.
  1. Выбор СУБД – претендента: на основе КМ ПО и внешних ограничений формулируются требования к СУБД и в соответствии с характеристиками СУБД выбирается из каталога.

III Даталогическое проектирование – это проектирование средствами СУБД.

1. Логическое проектирование:
   * конфигурирование системы КМ БД и обеспечение отображения КМ ПО в КМ БД;
   * конфигурирование ВМ БД и обеспечение отображения ЛПi в ВМi.
   * Согласование всех ВМi и КМ БД.
2. Физическое проектирование: определение параметров ВнМ БД и обеспечение отображения КМ БД во ВнМ БД, а также обеспечение отображения ВнМ БД на ФБД.

IV Параллельно с I, II, III осуществляется разработка словаря-справочника системы, комплекса программ решения сервисных и функциональных задач, обеспечение надежности хранения и обработки данных, обеспечение интерфейса.

**Лекция №4**

**Модели данных**

Модель данных (МД) – это средство абстракции данных, которое позволяет перейти от информационного содержания к конкретным значениям. Чем выше уровень абстракции, тем больше она ориентирована, как модель, на КП.

Элементарной единицей данных в МД являются следующие кортежи:

1. <O, P, t>
2. <(O1, O2, …, On), r, t>

где O – объект, P – свойства объекта, O1, O2, …, On – объекты, r – связь между объектами, t – время.

Эти элементарные единицы данных могут быть реализованы различными способами. Отсюда различные модели.

Схемой МД называется совокупность описанных типов структур данных на формализованном языке.

МД – это совокупность экземпляров типов структур данных, описанных схемой МД.

Модели данных делятся на семантические и структурные.

Семантические МД: ER-модель, семантическая сеть, расширенная реляционная модель, модель бинарных ассоциаций.

Структурные МД: иерархическая, сетевая, реляционная.

В иерархической МД объекты представлены в явном виде, т.е. в виде записей или сегментов, а отношения представлены неявно и отображаются с помощью взаимного расположения записей. В сетевой МД объекты и отношения представлены в явном виде: объекты – в виде записей, отношения - в виде указателя. Главное, что объединяет эти модели, это то, что объекты и отношения представлены различным способом, поэтому требуются различные операции ЯМД для обработки объектов и отношений. В ИМД и СМД пользователь с помощью ЯМД обязан динамически воспроизводить структуру модели. В РМД объект и отношение представлены одинаково с помощью понятий теории отношений (таблиц – отношений).

Пример:

Классы объектов: Поставщик (П), Детали (Д).

Свойства объектов: П (№П, НГ (название города)), Д (№Д, СТ (стоимость))

Значения свойства №П = {П1, П2}

Значения свойства НГ = {М, С\_П}

Значения свойства №Д = {Д1, Д2, Д3}

Значения свойства СТ = {СТ1, СТ2, СТ3}

П = {<П1, М>, <П2, С\_П>}

Д = {<Д1, СТ1>, <Д2, СТ2>, <Д3, СТ3>}

Устанавливаем связь: П Д (Поставщик детали): П1 поставляет детали Д1, Д2, Д3, а П2 – Д2, Д3.

Иерархическая МД:

П1 М1 П2 С\_П

Д1 СТ1 Д2 СТ2

Д2 СТ2 Д3 СТ3

Д3 СТ3

Сетевая МД: вводится связующая запись – стоимость между П и Д

П1 М П2 С\_П

СТ1 СТ2 СТ3 СТ2 СТ3

Д1 Д2 Д3

Реляционная МД:

|  |  |  |
| --- | --- | --- |
| П | №П | НГ |
|  | П1 | М |
|  | П2 | С\_П |

|  |  |  |
| --- | --- | --- |
| Д | №Д | СТ |
|  | Д1 | СТ1 |
|  | Д2 | СТ2 |
|  | Д3 | СТ3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| П\_Д | №П | НГ | №Д | СТ |
|  | П1 | М | Д1 | СТ1 |
|  | П1 | М | Д2 | СТ2 |
|  | П1 | М | Д3 | СТ3 |
|  | П2 | С\_П | Д2 | СТ2 |
|  | П2 | С\_П | Д3 | СТ3 |

Модели данных включают в себя следующие компоненты: МД = <S, R, I>

1. S – типы структур данных

Эта компонента устанавливает основные структуры данных: дерево, сеть, таблица. Они поддерживают в СУБД язык описания данных (ЯОД).

1. R – операторы (правила вывода)

Они применяются к любому экземпляру типа данных в S для получения новых данных, для модификации текущих и для восстановления исходных. Эта компонента поддерживает в СУБД язык манипулирования данными (ЯМД).

1. I – основные правила обеспечения целостности данных. Они делятся на:

А) внутренние ограничения модели данных;

Б) явные ограничения, связанные с семантикой предметной области.

**Реляционная модель данных (РМД)**

Автор – Кодд (первая статья – 1970 год).

Исторически первой СУБД была иерархическая, затем сетевая, потом реляционная. Поэтому в реляционных СУБД есть свойства иерархических и сетевых СУБД, а основы теории реляционных СУБД распространяются на иерархические и сетевые.

Центральным понятием РМД является отношение. Теория РМД базируется на теории отношений, теории множеств и теории предикатов 1-го порядка. РМД весьма проста в структурном плане, поскольку единственным средством структуризации является отношение.

Язык описания данных (ЯОД) – основные определения

Атрибут – это такая информационная величина, которая представляет в РМД свойство объекта ПО. Атрибут имеет имя и значение. Например: «Цвет» - свойство объекта «Деталь» (имя атрибута); «Черный» - значение свойства объекта (значение атрибута).

Домен – это множество допустимых значений. Обозначения:

Ai – имя i-того атрибута;

Di – имя домена для i-того атрибута;

Di = dom (Ai)

Например: Dцвет = dom (цвет) = {красный, черный, синий}.

Схема отношения:

В РМД некоторому классу объектов R ставится в соответствие множество имен атрибутов: {A1, A2, …, An}, т.е. R  {A1, A2, …, An}. Это соответствие называется схемой отношения и обозначается R(A1, A2, …, An). Например, схема отношения для класса объектов Деталь: Деталь (ШД, Цвет, Стоимость, Количество)

Отношение:

Имеется схема отношения R(A1, A2, …An). Введем понятие домена схемы отношения: dom(R).

dom (R) = dom(A1) \* dom(A2) \* … \* dom(Ai) \* … \* dom(An) =

= {<a1k, a2k, …, aik, …, ank> ⏐k=};

кортеж tk

aik – это значение I-того атрибута в k-том кортеже.

tk – k-тый кортеж – это упорядоченное множество значений атрибутов.

P= pi, где pi – мощность dom(Ai), а P – мощность dom(R).

Таким образом, dom(R) – это универсальное множество, состоящее из кортежей tk, где k=.

Отношением r со схемой R(A1, A2, …, An) называется некоторое подмножество кортежей, входящих в dom(R):

R(A1, …, An) ⊆ dom(R) = {<a1kr, a2kr, …, ankr> ⏐ k=1, Pr}, Pr<p

Где n – степень отношения, P – мощность отношения (число кортежей, входящих в r).

Пример: ДЕТАЛЬ (ШД, Цвет) – схема отношения;

dom (ШД) = {Г, Б} P1=2

dom (Цвет) = {к, с, ч} P2=3

dom (Деталь) = dom (ШД) \* dom (Цвет) n=2; P=P1\*P2=2\*3=6

dom (Деталь) = {<Г, к>, <Г, с>, <Г, ч>, <Б, к>, <Б, с>, <Б, ч>}

деталь1 (ШД, Цвет) ⊂ dom (Деталь) и деталь2 (ШД, Цвет) ⊂ dom (Деталь)

деталь1 (ШД, Цвет) = {<Г, с>, <Г, ч>, <Б, ч>} n=2 P=3

деталь2 (ШД, Цвет) = {<Г, к>, <Г, с>} n=2 P=2

Представление отношений

Одним из естественных способов представления отношений является двумерная таблица, обладающая следующими свойствами:

* 1. Таблица имеет имя, совпадающее с именем отношения.
  2. Каждая строка таблицы является кортежем отношения.
  3. Каждый столбец таблицы имеет имя, совпадающее с именем атрибута
  4. Все строки различны.
  5. Порядок строк несущественен.
  6. Порядок столбцов существенен.

Пример: деталь1 (ШД, Цвет) = {<Г, с>, <Г, ч>, <Б, ч>}

|  |  |  |
| --- | --- | --- |
| деталь1 | ШД | Цвет |
|  | Г | С |
|  | Г | ч |
|  | Б | ч |

* 1. В теории отношений атрибуты отношений могут быть простыми и составными. Простой атрибут – это унарное отношение, т.е. n=1

Составной атрибут – это отношение с n>=2

Пример: Служ (#служ., фамилия, дети (имя, возраст))

Простые атрибуты составной атрибут

|  |  |  |  |
| --- | --- | --- | --- |
| #служ | Фамилия | Дети | |
| имя | возраст |
| 0405 | Иванов | Катя | 3 |
| Маша | 5 |
| 0406 | Сидоров | Витя | 7 |
| Коля | 9 |

Отношение, которое состоит из простых атрибутов, называется нормализованным и находится по меньшей мере в 1НФ. В РМД используются только нормализованные отношения, чтобы обеспечить единообразие структур данных и простоту доступа к ним.

* 1. В отношении не должно быть неопределенных значений атрибутов. Если значение атрибута неизвестно, то оно заменяется некоторым значением, которое означает, что это значение «неизвестно» (например, NULL).

**Лекция №5**

**РМД – продолжение**

Идентификация отношения и его элементов.

Любое отношение в РМД имеет уникальное имя. Любой атрибут внутри отношения также имеет уникальное имя. Для идентификации кортежей вводится понятие ключа отношения.

Определение ключа: дано отношение r(R), кортежи t(R), t(K)

Пусть t1(R) и t2(R) – два любых различных кортежа отношения и пусть имеется подмножество K имен атрибутов R; K ⊆ R. Набор атрибутов K является ключом отношения r(R) при выполнении двух условий:

1. условие однозначной идентификации кортежа в отношении: t1(K) ≠ t2(K).
2. условие отсутствия избыточности: ∀k’ ⊂ K: t1(k’) = t2(k’)

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| r | A | B | C |
|  | a1 | b1 | c1 |
|  | a1 | b2 | c2 |
|  | a1 | b3 | c3 |
|  | a2 | b1 | c1 |
|  | a2 | b2 | c2 |
|  | a1 | b1 | c2 |

Возможные ключи: k1={A, B}, k2={A, C} – удовлетворяют двум условиям

k3={A, B, C} – это не ключ, т.к. удовлетворяет только первому условию.

Атрибуты, входящие в возможные ключи, называются первичными атрибутами. Остальные – непервичные атрибуты.

Пример: студент (#зачетной\_книжки, ФИО, год\_рождения)

k1 k2 непервичный

первичные

Ключ, выбранный из возможных ключей для идентификации кортежей отношения, называется ключом отношения.

Ключ отношения выбирается из семантических соображений, удобства использования.

Ключ – это свойство схемы отношения.

Представление связей в РМД.

Связь между объектами представлена так же, как и объект, т.е. с помощью отношений.

Пример:

|  |  |  |
| --- | --- | --- |
| Пост. | ШП | НГ |
|  | П1 | М |
|  | П2 | К |

|  |  |  |
| --- | --- | --- |
| Деталь | ШД | НД |
|  | Д1 | Г |
|  | Д2 | Б |
|  | Д3 | Ш |

|  |  |  |  |
| --- | --- | --- | --- |
| Пост. Дет. | ШП | ШД | Стоим. |
|  | П1 | Д1 | 10 |
|  | П1 | Д2 | 20 |
|  | П1 | Д3 | 30 |
|  | П2 | Д1 | 40 |
|  | П2 | Д2 | 30 |

Таким образом, единообразие в представлении объектов и связей между ними способствует созданию единых процедур поиска и обработки данных (объектов и связей).

Таблица соответствия понятий ПО и РМД:

|  |  |
| --- | --- |
| ПО | РМД |
| Свойство объекта | Имя атрибута (столбца) |
| Значение свойства | Значение атрибута (столбца) |
| Конкретный объект (связь) | Кортеж (строка) |
| Значения класса объектов (связей) | Отношение (таблица) |
| Класс объектов | Схема отношения (заголовок таблицы) |
| Описание ПО | Множество схем отношений (множество заголовков таблиц) |
| ПО | Множество отношений (множество таблиц) |

Таким образом, с точки зрения структурной компоненты РМД – это множество отношений. Схема РМД – это множество схем отношений.

**РМД: языки манипулирования данными (ЯМД)**

ЯМД разбиваются на 3 больших класса:

1. Основанные на реляционной алгебре (РА) (процедурные языки);
2. Основанные на исчислении отношений (непроцедурные языки);
3. Основанные на операции отображения (SQL, QBE и др.)

ЯМД, основанный на РА

РА: UR=<R, ΦR>, где R – основное множество (отношения); ΦR  - сигнатура (множество операций).

РА является замкнутой алгеброй относительно R, т.е. операнды и результаты операций являются отношениями.

Теоретико-множественные операции РА

Они применяются к отношениям, имеющим одинаковую схему отношений:

R (A1, A2, …, An) и S (A1, A2, …, An)

1. одинаковое количество атрибутов;
2. одноименные атрибуты определены на одних и тех же доменах.

Такие отношения имеют одну и ту же область определения – U:

U = dom (A1) \* dom (A2) \* … \* dom (An)

РА определяется кортежом <P(n), ∨, ∧, ¬, ∅, U>

Пусть r и s – отношения со схемой R (A1, A2, …, An);

r(R) и s(R) – это исходные отношения, а q(R) – результирующее отношение.

|  |  |  |
| --- | --- | --- |
| r(R) | ШД | Цвет |
|  | Б | ч |
|  | Г | ч |
|  | Г | с |

|  |  |  |
| --- | --- | --- |
| s(R) | ШД | Цвет |
|  | Г | с |
|  | Г | к |

1. Операция объединения: q(R) = r(R) ∨ s(R) = {t(R) ⏐ t ∈ r ∨ t ∈ s}

Операция объединения используется для первоначальной загрузки таблиц БД, для добавления записей.

|  |  |  |
| --- | --- | --- |
| q(R) | ШД | Цвет |
|  | Б | ч |
|  | Г | ч |
|  | Г | с |
|  | Г | к |

1. Операция пересечения: q(R) = r(R) ∧ s(R) = {t(R) ⏐ t ∈ r ∧ t ∈ s}

Используется для выделения необходимых коррекций.

|  |  |  |
| --- | --- | --- |
| q(R) | ШД | Цвет |
|  | Г | с |

1. Операция разности: q(R) = r(R) \ s(R) = {t(R) ⏐ t ∈ r ∧ t ∉ s}

Используется для удаления кортежей из отношения.

|  |  |  |
| --- | --- | --- |
| q(R) | ШД | Цвет |
|  | Г | ч |
|  | Б | ч |

1. Операция дополнения (унарная операция): q(R) = r(R) = {t(R) ⏐ t ∉ r}

Или q(R) = dom (R) \ r

|  |  |  |
| --- | --- | --- |
| q(R) | ШД | Цвет |
|  | Б | к |
|  | Б | с |
|  | Г | к |

Специальные операции РА

1. Операция проекции отношения на один или несколько атрибутов (унарная операция)

Исходные данные: r(R) X ⊆ R

Результирующее отношение: q(X)

q(X) = { t(X) ⏐ t(R) ∈ r ∧ t(X) ∈ t(R)}

В q(X) входят такие подкортежи t(X), которые являются частью кортежей t(R), входящих в отношение r. Обозначение операции: πx(r) или r(X)

Пример:

|  |  |  |
| --- | --- | --- |
| R | ШД | Цвет |
|  | Б | ч |
|  | Г | с |
|  | Г | ч |

Найти имена всех деталей – q(ШД) = πШД(r)

|  |  |
| --- | --- |
| q | ШД |
|  | Б |
|  | Г |
|  | Г |

Дублирующие кортежи вычеркиваются (по определению)

1. θ - соединение, где θ - некоторая операция отношения.

Введем понятие θ - сравнимых атрибутов. Если операция θ определена на декартовом произведении доменов dom (A) \* dom (B), то A и B называются

θ-сравнимыми. Один атрибут может рассматриваться θ-сравнимым сам с собой.

Исходные данные: r(R) и s(S); R ∧ S = ∅; T = R ∨ S; A ∈ R; B ∈ S; A и B

θ-сравнимы. Обозначим: tr для t(R) ∈ r, ts для t(S) ∈ s. Результирующее отношение: q(T).

q(T) = {t(T) ⏐ t(R)= tr ∧ t(S)= ts ∧ tr(A) θ ts(B)}

В q(T) входят различные сцепления кортежей tr и ts, для которых предикат tr(A) θ ts(B) является истинным.

Короткая запись операции: q(T)=r[A θ B]s

Пример: A→B→C

|  |  |  |  |
| --- | --- | --- | --- |
| AB | № рейса | Вылет | Прилет |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| BC | № рейса1 | Вылет1 | Прилет1 |
|  |  |  |  |

AC = AB [Прилет < Вылет1] BC

Условие вхождения кортежей может быть логически сложным (логические операции: ∨, ∧, ¬).

На языке SQL: SELECT AB.\*, BC.\* FROM AB, BC WHERE AB.ПРИЛЕТ < BC.ВЫЛЕТ1

3) θ - ограничение (или выборка) – частный случай θ - соединения.

Исходные данные: r(R); A ∈ R; a ∈ dom (A);

Результирующее отношение: q(R).

q(R) = {t(R) ⏐ t(A) θ a} Короткая запись: q(R) = δAθa(R)

В q(R) входят такие кортежи t(R) из отношения r, для которых предикат t(A) θ a является истинным.

Пример:

|  |  |  |
| --- | --- | --- |
| R | ШД | Цвет |
|  | Б | ч |
|  | Г | с |
|  | Г | ч |

Найти кортежи, в которых черные детали: q(R) = δЦвет=’ч’(R)

|  |  |  |
| --- | --- | --- |
| Q | ШД | Цвет |
|  | Б | ч |
|  | Г | ч |

4) Операция декартова произведения кортежей отношения.

Исходные данные: r(R) и s(S); T = R ∨ S; R ∧ S = ∅.

Результирующее отношение: q(T).

q(T) = {t(T) ⏐ t(R)= tr ∨ t(S)= ts} Короткая запись: q(T)=r(R) \* s(S)

В q(T) входят всевозможные сцепления кортежей tr и ts. Это очень опасная операция, т.к. мощность результата P=P1\*P2.

1. Операция эквисоединения

Исходные данные: r(R) и s(S); F = R ∧ S ≠ ∅ (имеются одноименные атрибуты).

Результирующее отношение: q(T).

q(T) = {t(T) ⏐ t(R)= tr ∧ t(S)= ts ∧ tr(F) = ts(F)} Короткая запись: q(T) = r ∞ s

В q(T) входят различные сцепления tr и ts, для которых значения одноименных атрибутов равны.

Пример:

|  |  |  |
| --- | --- | --- |
| Пост. | ШП | ШД |
|  | П1 | Б |
|  | П2 | Г |

|  |  |  |
| --- | --- | --- |
| Деталь | ШД | Цвет |
|  | Б | ч |
|  | Г | с |
|  | Г | ч |

q(ШП, ШД, Цвет) = Пост. (ШП, ШД) ∞ Деталь(ШД, Цвет)

|  |  |  |  |
| --- | --- | --- | --- |
| q | ШП | ШД | Цвет |
|  | П1 | Б | ч |
|  | П2 | Г | с |
|  | П2 | Г | ч |

# Лекция №6

Определение ЯМД, основанного на ИО.

Исчисление отношений (ИО) – это совокупность обозначений и правил (формула) для описания результирующего отношения, выводимого из РМД. Эта формула называется α-выражением:

{ (t1, t2, …, tn): W}

целевой список предикат

Этот ЯМД в нашем курсе не рассматривается.

ЯМД, основанный на операции отображения.

Эти языки занимают промежуточное положение между РА и ИО. Под отображением в данном случае понимают операцию выбора данных, которая имитирует действия человека при поиске данных в таблице.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| r | ФАМ | Г\_Р | Группа | Пол |
|  |  |  |  |  |

Действия:

1. Пример столбца: группа
2. Выделение в нем некоторого элемента (аргумент отображения) А8-97
3. Просмотр строки, содержащей аргумент отображения
4. Выделение из строки значений, которые входят в столбцы ФАМ, Г\_Р

Таким образом, главная конструкция этого языка – это операция отображения, например:

Язык SQL

SELECT ФАМ, Г\_Р FROM r WHERE Группа = ‘А8-97’

аргумент отображения

Здесь нет ничего от РА; только исчисление отношений. В языке SQL есть блочная структура, т.е. возможно вложение одного элементарного блока в другой.

Пример: Сотрудники\_ЗИЛ (ФАМ, Г\_Р)

ВУЗ (ФАМ, Н\_ВУЗА)

Выдать года рождения сотрудников ЗИЛа, которые учатся в МЭИ.

SELECT Г\_Р FROM СОТР\_ЗИЛ WHERE ФАМ IN (SELECT ФАМ FROM ВУЗ WHERE

Н\_ВУЗА = ‘МЭИ’)

Запрос реализуется в 2 этапа: сначала – внутренний блок SELECT: из отношения ВУЗ извлекается множество ФАМ, которые учатся в МЭИ; затем полученный список ФАМ используется в условиях внешнего SELECT. Это вносит определенную процедурность в формулировку запросов и приближает рассмотренную группу языков к алгебраическим языкам.

Язык QBE (Query by Example)

Язык, сходный с SQL, однако QBE разработан специально для работы с терминала.

Всякая операция в QBE специфицируется с помощью одной или нескольких таблиц; каждая такая таблица строится на экране монитора частично системой, частично пользователем.

Первоначально система выдает на экран пустую таблицу. Пользователь, знающий, что ответ на запрос находится в таблице r, запишет r в качестве имени таблицы; тогда система ответит заполнением соответствующих имен столбцов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| R | A | B | C | D |
|  | P.ax | P.bx | C1 |  |

условие запроса

Пользователь формулирует запрос, заполняя три позиции в таблице. P. – означает печать. P указывает цель запроса (результат), которая должна быть выведена.

Q = πAB (δC=C1(r))

## Нормализация отношений

Нормализацией называется процесс приведения отношения к одной из нормальных форм (НФ). Отношения могут находиться в следующих состояниях: ненормализованное отношение, 1НФ, 2НФ, 3НФ, 4НФ, 5НФ. Однако 5НФ практического значения не нашла. Нормализация используется на этапе концептуального проектирования РМД.

Для чего вводится нормализация:

1. минимизация избыточности хранения данных;
2. обеспечение непротиворечивости данных;
3. расширение возможностей БД по добавлению, удалению и обновлению;
4. сделать БД более наглядной, семантически выразительной.

### 1НФ

Отношение, которое состоит из простых атрибутов, называется нормализованным и находится, по меньшей мере, в 1НФ.

Любое ненормализованное отношение можно привести к 1НФ.

Пример: Служ (#служ, ФИО, дети(имя, Г\_Р))

составной атрибут

служ

#служ ФИО дети

корневое отношение имя Г\_Р

подчиненное отношение

Существует алгоритм приведения к 1НФ:

1. Корневое отношение помещается в результирующее отношение.
2. Подчиненное отношение помещается в результирующее отношение вместе с атрибутами ключа корневого отношения.
3. Этот процесс продолжается до удаления всех деревьев из исходного отношения.

Таким образом, в нашем случае в результирующее отношение попадут 2 отношения:

Служ(#служ, ФИО)

Дети(#служ, имя, Г\_Р) 1НФ

При анализе нормализации важную роль играет понятие функциональной зависимости (ФЗ).

#### Определение ФЗ

Атрибут (группа атрибутов) Y отношения r(R) функционально зависит от атрибута (группы атрибутов) X этого же отношения, если в любой момент времени каждому значению атрибута X соответствует не более одного значения атрибута Y.

Изображение ФЗ: X→Y

Отсутствие ФЗ: X→Y

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| r | A | B | C |
|  | a1 | b2 | c3 |
|  | a2 | b3 | c3 |
|  | a1 | b4 | c3 |
|  | a2 | b5 | c3 |

Определить ключ отношения. Ключом k отношения r(R), k ∈ R является в том случае, если:

1. k → R
2. ∀k’ ⊂ k, k’ → R

A→A B→B C→C

A→B B→A C→A

A→C B→C C→B

AB→A AC→A BC→A

AB→B AC→B BC→B

AB→C AC→C BC→C

ABC→A ABC→B ABC→C

{B, AB, BC, ABC} – однозначность

{B} – отсутствие избыточности. Поэтому ключом отношения является атрибут B.

Итак, отношение находится в 1НФ, если оно состоит из простых атрибутов и в нем выделен ключ отношения.

##### **Лекция №7**

**Нормализация отношений (продолжение)**

Полная функциональная зависимость между X и Y: в отношении r(R), X и Y ⊂ R

1. X→Y
2. ∀X’ ⊂ X, X’ → Y

Атрибут (группа атрибутов) Y отношения r(R) функционально полно зависит от атрибута (группы атрибутов) X того же отношения, если Y функционально зависит от X, но не зависит от любого подмножества X (X’).

Определение 2НФ:

Отношение находится во 2НФ, если оно находится в 1НФ и каждый непервичный атрибут функционально полно зависит от любого возможного ключа.

Пример:

Деят.Пр. (про-ст, программа, #про-ста, #про-мы, кол\_час)

A1 A2 A3 A4 A5

A1, A2, A3, A4 – первичные атрибуты, входящие в возможные ключи.

A5 - непервичный атрибут.

|  |  |  |
| --- | --- | --- |
| k1 = {A1, A2} | k1→A5 | A1→A5 |
| k2 = {A1, A4} | k2→A5 | A2→A5 |
| k3 = {A3, A4} | k3→A5 | A3→A5 |
| k4 = {A3, A2} | k4→A5 | A4→A5 |
| возможные ключи | ∀Ai⊂kj | Ai→A5 |

Это отношение не соответствует 2НФ.

Пример2:

ПС (ШП, ШД, имя\_пост, свед\_о\_пост, цена)

A1 A2 A3 A4 A5

k={ A1, A2}

1) k→A3 2) A1 ⊂ k

k→A4 A1→A3

k→A5 A1→A4

нет полной функциональной зависимости

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ПС | ШП | ШД | Имя\_пост | Св\_о\_п | Цена |
|  | П1 | Д1 | Ф | П | 10 |
|  | П1 | Д2 | Ф | П | 15 |
|  | П1 | Д3 | Ф | П | 20 |
|  | П2 | Д1 | Щ | К | 25 |
|  | П2 | Д2 | Щ | К | 30 |

Приводим ко 2НФ:

ПОСТ (ШП, имя\_пост, св\_о\_п)

ПИ (ШП, ШД, цена)

|  |  |  |  |
| --- | --- | --- | --- |
| ПОСТ | ШП | Имя\_пост | Св\_о\_п |
|  | П1 | Ф | П |
|  | П2 | Щ | К |

|  |  |  |  |
| --- | --- | --- | --- |
| ПИ | ШП | ШД | Цена |
|  | П1 | Д1 | 10 |
|  | П1 | Д2 | 15 |
|  | П1 | Д3 | 20 |
|  | П2 | Д1 | 25 |
|  | П2 | Д2 | 30 |

Связь ПИ Связь ПИ Объект П Объект ПОСТ

ПС

Недостатки отношения, которое не находится во 2НФ:

1. Не можем иметь сведений о поставщике, если поставщик не начал еще поставлять деталей (аномалия добавления);
2. Если поставщик прекратил поставку деталей, то исчезнут данные и о нем (аномалия удаления);
3. Избыточность данных о поставщике, если он поставляет несколько деталей;
4. При коррекции данных об одном поставщике нужно изменять все записи с этим поставщиком, что может привести к появлению противоречивости данных (аномалия обновления).

Можно утверждать, что для любого отношения, которое не находится во 2НФ, существует 2НФ.

**Теорема Хита**: Отношение r, которое определено на атрибутах A, B, C и в котором имеется ФЗ: A→B, может быть представлено в виде двух отношений: r1(A, B)=πAB(r) и r2(A,C)=πAC(r). Причем, эта декомпозиция осуществляется без потерь, т.е.:

r(A, B, C)= r1(A, B) ∞ r2(A,C).

Теорема Хита доказывает существование 2НФ любого отношения и дает алгоритм приведения ко 2НФ.

Алгоритм приведения ко 2НФ:

Дано: r(R), k ⊂ R, где k – атрибуты возможного ключа.

Если отношение не находится во 2НФ, то это означает, что:

1. ∃ A ∈ k
2. ∃ B ⊂ R \ k
3. A → B

Применим теорему Хита:

r(R) = πAB(r) ∞ πR\B(r)

здесь нет k здесь нет B

A B C

Пример: ПС (ШП, ШД, имя\_пост, св\_о\_п, цена)

k A → B

r1(A, B) = πAB(ПС) = ПОСТ (ШП, имя\_пост, св\_о\_п)

r2(R\B)=πR\B(ПС) = ПИ (ШП, ШД, Цена)

k C

###### 3НФ

Введем понятие транзитивной зависимости в отношении. Пусть в отношении r(R) атрибуты A ⊂ R и C ⊂ R, а также ∃ такой B ⊂ R, для которого выполнены следующие условия:

1. B ⊆A
2. C ⊆ B
3. A → B
4. B → C
5. B → A

тогда ФЗ A → C называется транзитивной зависимостью.

Определение 3НФ: отношение r задано в 3НФ, если оно находится во 2НФ и каждый непервичный атрибут нетранзитивно зависит от каждого возможного ключа.

Пример: Служ (#служ, имя, ЗП, #проекта, дата\_оконч)

A B C

Отношение Служ. не находится в 3НФ.

Алгоритм приведения к 3НФ базируется на теореме Хита.

Дано: r(R), A, B, C ⊂ R. Нужно разорвать несмежные атрибуты A и C, т.е. выделить связь B → C.

В соответствии с теоремой Хита исходное отношение r(R) необходимо представить в виде двух проекций:

r1(B, C) = πBC(r), здесь нет A

r2(R\C)=πR\C(r), здесь нет C,

причем r = r1 ∞ r2 по атрибуту B. B C

Таким образом: r1 = ПРОЕКТ (#проекта, дата\_оконч)

r2= СЛУЖ (#служ, имя, ЗП, #проекта) – 3НФ

A B

Пример:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Служ | #служ | Имя | ЗП | #проекта | Дата\_оконч |
|  | 100 | A | 300 | 30 | 1.12.95 |
|  | 101 | B | 400 | 30 | 1.12.95 |
|  | 102 | C | 400 | 30 | 1.12.95 |
|  | 103 | D | 500 | 25 | 1.12.95 |
|  | 104 | E | 400 | 25 | 1.12.95 |

|  |  |  |
| --- | --- | --- |
| r1 | #проекта | Дата\_оконч |
|  | 30 | 1.12.95 |
|  | 25 | 1.12.95 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| r2 | #служ | Имя | ЗП | #проекта |
|  | 100 | A | 300 | 30 |
|  | 101 | B | 400 | 30 |
|  | 102 | C | 400 | 30 |
|  | 103 | D | 500 | 25 |
|  | 104 | E | 400 | 25 |

Недостатки отношения, которое не находится в 3НФ:

1. Аномалия добавления: нельзя добавить данные о проекте, если служащие еще не набраны для данного проекта.
2. Аномалия удаления: если служащие уволены с данного проекта, то исчезнут данные о проекте.
3. Аномалия обновления: данные об одном проекте обновляются столько раз, сколько служащих работает по этому проекту, что может вызвать противоречивость данных.

###### 4НФ

4НФ связана с понятием многозначной зависимости в отношениях.

Дано: r(R), X, Y ⊂ R, Z = R \ (X, Y). Обозначение: →→ - многозначная зависимость.

Определение:

X →→ Y, если при наличии кортежей t1 = <x1, y1, z1> и t2 = <x1, y2, z2> имеются кортежи t3 = <x1, y2, z1> и t4 = <x1, y1, z2> для ∀x ∈ dom(X), входящего в отношение.

Пример: Задано отношение в 3НФ:

|  |  |  |  |
| --- | --- | --- | --- |
| Курс\_лекций | Курс | Лектор | Учебник |
|  | Физика | Иванов | «Механика» |
|  | Физика | Иванов | «Оптика» |
|  | Физика | Петров | «Механика» |
|  | Физика | Петров | «Оптика» |

Необходимо добавить новый учебник:

|  |  |  |
| --- | --- | --- |
| Физика | Иванов | «Кв.механ.» |
| Физика | Петров | «Кв.механ.» |

В данном случае имеется многозначная зависимость:

Физика →→ Иванов, Петров и Физика →→ «Механика», «Оптика», «Квант.механика»

r = πXY (r) ∞ πXZ (r)

|  |  |  |
| --- | --- | --- |
| Лектор | Курс | Лектор |
|  | Физика | Иванов |
|  | Физика | Петров |

|  |  |  |
| --- | --- | --- |
| Учебник | Курс | Учебник |
|  | Физика | «Механика» |
|  | Физика | «Оптика» |
|  | Физика | «Кв.механ.» |

4НФ – это нормализованное отношение, находящееся в 3НФ и имеющее одну многозначная зависимость (МЗ) X →→ Y и функциональную зависимость всех непервичных атрибутов от атрибута X (обычно 4НФ – это бинарное отношение). 4НФ связана с устранением МЗ в отношении путем получения тривиальных многозначных зависимостей: πXY и πXZ.

5НФ – получается путем устранения зависимости по соединению (ФЗ – это частный случай МЗ, а МЗ – это частный случай зависимости по соединению).

Общая характеристика шагов нормализации:

Ненорм. Отношение

↓ 1 шаг: устранение составных атрибутов и выделение ключа

1НФ

↓ 2 шаг: устранение неполной ФЗ (Теорема Хита)

2НФ

↓ 3 шаг: устранение транзитивных зависимостей

3НФ

↓ 4 шаг: устранение МЗ

4НФ

↓ 5 шаг: устранение зависимостей по соединению

5НФ

Нормализация отношений используется в рамках реляционного подхода к построению концептуальной модели предметной области (КМ ПО).

**Лекция № 8**

# Платформа SQL Server

SQL Server – это комплексная платформа баз данных, обеспечивающая управление данными в масштабе предприятия и оснащенная интегрированными средствами бизнес-аналитики (Business Intelligence, BI). Ядро СУБД SQL Server обеспечивает безопасное и надежное хранение данных в реляционном формате, в формате XML а так же объектов среды CLR Microsoft .NET Framework. Благодаря поддержке широкого набора форматов, SQL Server обеспечивает гибкость, необходимую для создания современных комплексных решений. Эта платформа также предоставляет возможность создания и управления высокопроизводительными приложениями баз данных с высоким уровнем доступности.

В основе решения управления корпоративными данными лежит ядро базы данных SQL Server. Помимо поддержки реляционных баз данных и данных в формате XML, SQL Server также сочетает лучшее в таких областях, как анализ, подготовка отчетов, интеграция данных и рассылка уведомлений. Выгодным преимуществом SQL Server является тесная интеграция с Microsoft Visual Studio, Microsoft Office System и целым комплектом средств разработки, включая Business Intelligence Development Studio. Это позволяет создавать и развертывать экономически эффективные решения бизнес-аналитики, которые позволят сотрудникам доставлять аналитическую информацию в любое место предприятия при помощи карт ключевых показателей эффективности, панелей управления, веб-служб и мобильных устройств.

Сервер баз данных Microsoft SQL Server в качестве языка запросов использует версию языка SQL, получившую название Transact-SQL (сокращённо T-SQL). Язык T-SQL являетсяся реализацией SQL-92 (стандарт ISO для языка SQL) с множественными расширениями.

В SQL Server начиная с версии 2005 встроена поддержка .NET Framework. Благодаря этому хранимые процедуры БД могут быть написаны на любом языке платформы .NET, используя полный набор библиотек, доступных для .NET Framework, включая Common Type System (система обращения с типами данных в Microsoft .NET Framework). Однако, в отличие от других процессов, .NET Framework, будучи базисной системой для SQL Server 2005, выделяет дополнительную память и выстраивает средства управления SQL Server вместо того, чтобы использовать встроенные средства Windows. Это повышает производительность в сравнении с общими алгоритмами Windows, так как алгоритмы распределения ресурсов специально настроены для использования в структурах SQL Server.

Последняя версия SQL Server - SQL Server 2008 R2 направлена на то, чтобы сделать управление данными самонастраивающимся, самоорганизующимся и самообслуживающимся механизмом - для реализации этих возможностей были созданы технологии SQL Server Always On. Это позволяет уменьшить до нуля время нахождения сервера в нерабочем состоянии.

В SQL Server 2008 для поддержки структурированных и частичноструктурированных данных, включая цифровые форматы для изображений, звуков, видео и других типов мультимедиа был добавлен новый тип хранения больших двоичных объектов (BLOB) FILESTREAM, который объединяет компонент Database Engine с файловой системой NTFS, размещая данные больших двоичных объектов (BLOB) в файловой системе в виде файлов. Манипулирование данными, хранящимися в FILESTREAM осуществляется при помощи инструкций Transact-SQL, что позволяет использовать оптимизированное хранилище без изменения логики приложений его использующих. Интерфейсы файловой системы Windows также обеспечивают потоковый доступ к этим данным.

Кроме этого, были включены специализированные форматы даты и времени и пространственные (Spatial) типы для пространственно зависимых данных.

Для повышения эффективности администрирования в SQL Server были включены библиотеки Declarative Management Framework, позволяющие назначать в декларативной форме полномочия для баз данных или отдельных объектов БД. Были улучшены методы компрессии данных.

# Выпуски SQL Server

Различные выпуски SQL Server удовлетворяют индивидуальным требованиям каждой организации или отдельного лица к производительности, среде выполнения и цене. Набор устанавливаемых компонентов SQL Server зависит от потребностей конкретного пользователя.

В следующей таблице приведено описание серверных выпусков SQL Server.

|  |  |
| --- | --- |
| **Выпуск** | **Определение** |
| Datacenter  (x86, x64 и IA64) | Построенный на основе SQL Server 2008 R2 Enterprise, редакция SQL Server 2008 R2 Datacenter представляет собой высокопроизводительную платформу обработки данных, предоставляющую самый высокий уровень масштабируемости для крупных приложений, виртуализации и консолидации, а также управления инфраструктурой базы данных организации, что позволяет организациям эффективно определять ответственную среду. |
| Enterprise  (x86, x64 и IA64) | Выпуск SQL Server 2008 R2 Enterprise является многофункциональной платформой обработки данных, обеспечивающей встроенную безопасность, доступность и масштабируемость в сочетании с надежными возможностями бизнес-аналитики, что позволяет добиться высокого уровня обслуживания даже при критических рабочих нагрузках. |
| Standard  (x86 и x64) | Выпуск SQL Server 2008 R2 Standard является идеальной платформой для обработки данных и бизнес-аналитики, необходимой отделам и малым организациям для выполнения их приложений, что позволяет эффективно управлять базами данных при минимальном использовании ИТ-ресурсов.  Выпуск SQL Server Standard for Small Business включает все технические компоненты и возможности выпуска SQL Server Standard и лицензирован для использования в компаниях малого бизнеса, число компьютеров в которых не превышает 75. |

Специализированные выпуски SQL Server предназначены для особых пользовательских сред. В следующей таблице описаны эти выпуски SQL Server.

|  |  |
| --- | --- |
| **Условие** | **Определение** |
| SQL Server Developer  (x86, x64 и IA64) | Выпуск SQL Server Developer позволяет разработчикам строить приложения любого вида на базе SQL Server. Этот выпуск включает все функциональные возможности выпуска SQL ServerDatacenter, однако лицензируется для разработки и тестирования системы, а не для применения в качестве рабочего сервера. Выпуск SQL Server Developer является идеальным выбором для тех, кто создает и тестирует приложения. Выпуск SQL Server Developer можно обновить для производственного использования. |
| SQL Server Workgroup  (x86 и x64) | Выпуск SQL Server Workgroup является идеальным выбором для баз данных отделений - надежная платформа для управления данными и создания отчетов, включающая возможности обеспечения безопасности, удаленной синхронизации и управления. |
| SQL Server Web (x86 и x64) | Выпуск SQL Server Web - это вариант с низкой общей стоимостью владения, предназначенный для размещения веб-сайтов, который обеспечивает масштабируемость и функции управления для небольших и крупномасштабных веб-свойств. |
| SQL Server Express (x86 и x64)  SQL Server Express with Tools  (x86 и x64)  Выпуск SQL Server Express with Advanced Services (x86 и x64) | Платформа баз данных SQL Server Express основана на SQL Server. Она также является заменой для Microsoft Desktop Engine (выпуск MSDE). Благодаря интеграции со средой Visual Studio, выпуск SQL Server Express облегчает разработку управляемых данными приложений, обладающих богатыми возможностями, обеспечивающими безопасность хранения данных и не требующими много времени для развертывания.  SQL Server Express - бесплатный выпуск, который могут распространять независимые поставщики ПО (при соблюдении соответствующего соглашения). Выпуск SQL Server Express идеально подходит для обучения, а также для построения настольных или небольших серверных приложений. Этот выпуск - лучший выбор для независимых поставщиков программного обеспечения, непрофессиональных разработчиков и любителей, которые хотят создавать клиентские приложения. Если необходимы дополнительные функции базы данных, SQL Server Express можно с легкостью обновить до расширенных версий SQL Server. |
| Compact 3.1 (x86) | SQL Server Compact 3.5 - бесплатная внедренная база данных, которая идеально подходит для построения изолированных и иногда сетевых приложений для мобильных устройств, рабочих станций и веб-клиентов, работающих на всех платформах Windows. |

Более подробное сравнение функциональных возможностей различных редакций SQL Server 2008 R2 можно посмотреть по адресу:

**http://msdn.microsoft.com/ru-ru/library/cc645993.aspx**.

**Использование SQL Server 2008 с клиент-серверными приложениями**

На компьютер, где работают клиент-серверные приложения, которые подключаются непосредственно к экземпляру SQL Server, достаточно установить только клиентские компоненты SQL Server. Установка клиентских компонентов будет хорошим выбором также и в том случае, если администрируется экземпляр SQL Server на сервере базы данных или планируется разработка приложений SQL Server.

При выборе установки клиентских компонентов будут установлены следующие компоненты SQL Server: программы командной строки, средства служб Reporting Services, компоненты связи, модели программирования, средства управления и разработки и электронная документация по SQL Server.

# Компоненты SQL Server

При установке SQL Server мастер установки на странице «Выбор компонентов» позволяет выбрать компоненты SQL Server, которые должны быть установлены.

|  |  |
| --- | --- |
| **Компоненты сервера** | **Описание** |
| SQL Server Database Engine | В состав компонента SQL Server Database Engine входит компонент Database Engine, основная служба для хранения, обработки и обеспечения безопасности данных, репликации, полнотекстового поиска и средств управления реляционными и XML-данными. |
| Analysis Services | Службы Analysis Services содержат средства создания и управления приложениями аналитической обработки в сети (OLAP) и приложениями интеллектуального анализа данных. |
| Reporting Services | Службы Reporting Services включают в себя серверные и клиентские компоненты для создания, управления и развертывания табличных, матричных и графических отчетов, а также отчетов в свободной форме. Службы Reporting Services являются расширяемой платформой, которую можно использовать для разработки приложений отчетов. |
| Integration Services | Службы Integration Services представляют собой набор графических средств и программируемых объектов для перемещения, копирования и преобразования данных. |

|  |  |
| --- | --- |
| **Средства управления** | **Описание** |
| SQL Server Management Studio | Среда SQL Server Management Studio представляет собой интегрированную среду для доступа, настройки, управления, администрирования и разработки компонентов SQL Server. Среда Management Studio позволяет работать с SQL Server разработчикам и администраторам любого уровня подготовки. Обозреватель Internet Explorer 6 с пакетом обновления 1 или более поздней версии необходим для любой установки среды Management Studio. |
| Диспетчер конфигурации SQL Server | Диспетчер конфигурации SQL Server обеспечивает базовые возможности управления конфигурациями для служб, серверных протоколов, клиентских протоколов и псевдонимов клиентов SQL Server. |
| SQL Server Profiler | Приложение SQL Server Profiler предоставляет графический пользовательский интерфейс для наблюдения за экземпляром компонента Database Engine или служб Analysis Services. |
| Помощник по настройке ядра СУБД | Помощник по настройке ядра СУБД помогает создавать оптимальные наборы индексов, индексированных представлений и секций. |
| Business Intelligence Development Studio | Среда Business Intelligence Development Studio представляет собой интегрированную среду разработки для решений служб Analysis Services, Reporting Services и Integration Services. Обозреватель Internet Explorer 6 с пакетом обновления 1 или более поздней версии необходим для любой установки среды BI Development Studio. |
| Компоненты связи | Устанавливает компоненты для связи между клиентами и серверами и сетевые библиотеки для DB-библиотеки, ODBC и OLE DB. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Документация** | | | **Описание** |
| Электронная документация по SQL Server | | | Основная документация для SQL Server. |
|  |
|  |

# Экземпляры SQL Server

На одном компьютере одновременно можно запустить несколько экземпляров SQL Server, и каждый экземпляр выполняется независимо от других.

Каждый экземпляр SQL Server состоит из отдельного набора служб, которые имеют специальные параметры сортировки и другие настройки уровня сервера. Структура каталогов, структура реестра и имена служб отражают имя и идентификатор конкретного экземпляра, создаваемого в процессе установки SQL Server.

Каждый экземпляр является либо экземпляром по умолчанию, либо именованным. Имя экземпляра по умолчанию – «MSSQLSERVER». Имя экземпляра SQL ServerExpress по умолчанию – «SQLExpress». Клиенту при установлении соединения имя экземпляра указывать необязательно. Именованный экземпляр определяется пользователем при установке и может быть установлен без предварительной установки экземпляра по умолчанию. Экземпляром по умолчанию может быть SQL Server 2000, SQL Server 2005 или SQL Server 2008.

На одном сервере или процессоре поддерживается несколько экземпляров SQL Server, но только один из них независимо от версии может быть экземпляром по умолчанию. Все остальные экземпляры должны быть именованными.

# Системные базы данных

Когда Вы устанавливаете SQL Server, автоматически создается четыре системные базы данных, которые играют жизненно важную роль для нормального функционирования SQL Server - Master, Resource, Model, Msdb и Tempdb.

**База данных Master**

Эта база данных хранит всю критическую для сервера информацию о конфигурации, учётных записях пользователей, запускаемых процессах, сообщениях об ошибках, расположении главных файлов баз данных и т.д. При её повреждении, можно использовать утилиту rebuildm.exe, которая восстанавливает базу Master и затем восстановить пользовательские базы. Старайтесь всегда иметь актуальную копию системной базы данных Master.

**База данных Resource**

База данных Resource содержит все системные объекты, входящие в SQL Server, и доступна только для чтения. Физические файлы базы данных Resource имеют имена mssqlsystemresource.mdf и mssqlsystemresource.ldf. С каждым экземпляром SQL Server может быть связан один и только один файл mssqlsystemresource.mdf; кроме того, экземпляры не могут использовать этот файл совместно.

Системные объекты SQL Server, такие как sys.objects, физически хранятся в базе данных Resource, а логически отображаются в схеме sys для каждой базы данных. База данных Resource не содержит пользовательских данных или метаданных и скрыта от пользователя, она не отображается в Обозревателе объектов, информацию о ней нельзя получить вызовом системной хранимой процедуры sp\_helpdb или запросом к представлению sys.databases.

Резервное копирование базы данных Resource средствами SQL Server не предусмотрено. Пользователь может создать резервную копию файла mssqlsystemresource.mdf или диска с этим файлом сторонними средствами, но восстановить такие резервные копии с помощью SQL Server не удастся. Восстановить резервную копию файла mssqlsystemresource.mdf можно будет только вручную, при этом следует соблюдать осторожность, чтобы не перезаписать текущую базу данных Resource устаревшей или потенциально небезопасной версией.

**База данных Model**

Эта база данных используется, как шаблон для новых баз данных то есть, всякий раз, когда Вы создаете новую базу данных, объекты в этой базе данных копируется в новую базу данных. Например, когда создается новая база данных <db name> без указания её размера, SQL создаст её с размер Model, то есть 0.75 МБ и скопирует все объекты из образцовой базы в новую базу данных. Если Вы хотите исполнить некоторые схожие действия для всех вновь создаваемых пользовательских базах данных, Вы можете включать необходимые объекты, таблицы, процедуры и т. п. в Model, и тогда, каждая новая база данных также будет их содержать.

**База данных Msdb**

Это одна из важных системных баз данных SQL, обеспечивающая его управление и обслуживание. База данных Msdb содержит задания (task-scheduling), обработку исключений, аварийное управление и информацию об операторах системы, то есть содержит информацию для всех операторов об их адресах электронной почты или номерах пейджера, а также информацию об истории по всем сеансам резервного копирования или восстановления баз данных.

**База данных Tempdb**

Следуя своему названию, эта база данных хранит все временные таблицы и промежуточные результаты group by, order by, distinct и т.д. Временные данные хранятся в Tempdb, пока хоть один пользователь подключён к SQL Server. Существует два вида временных таблиц: локальные и глобальные. Локальные временные таблицы видны только их создателям до завершения сеанса соединения с экземпляром SQL Server, как только они впервые созданы или когда на них появляется ссылка. Локальные временные таблицы удаляются после отключения пользователя от экземпляра SQL Server. Глобальные временные таблицы видны всем пользователям в течение любых сеансов соединения после создания этих таблиц и удаляются, когда все пользователи, ссылающиеся на эти таблицы, отключаются от экземпляра SQL Server. Tempdb создаётся каждый раз заново при запуске SQL сервера, поэтому все временные данные будут потеряны после перезагрузки SQL сервера или при его крахе.

**Лекция № 9**

## Файлы и файловые группы

**База данных** - это некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия.

Базы данных MS SQL Server хранятся на жестком диске в виде файлов. Существуют три типа файлов баз данных MS SQL Server - это первичные файлы, вторичные файлы и журналы транзакций. База данных должна содержать первичный файл данных и, по крайней мере, один файл журнала транзакций. При необходимости можно создать один или несколько вторичных файлов данных и дополнительные файлы журналов транзакций.

* **Первичные файлы**

Первичный файл данных содержит сведения, необходимые для запуска базы данных, и ссылки на другие файлы в базе данных. Первичные файлы также могут содержать данные пользовательских таблиц и индексов. Данные и объекты пользователя могут храниться в данном файле или во вторичном файле данных. В каждой базе данных имеется один первичный файл данных. Для имени первичного файла данных рекомендуется расширение MDF.

* **Вторичные файлы**

Вторичные файлы данных не являются обязательными; это пользовательские файлы, в которых хранятся данные пользователя. Вторичные файлы могут быть использованы для распределения данных на несколько дисков, в этом случае каждый файл записывается на отдельный диск. Кроме того, если размер базы данных превышает максимальный размер для одного файла Windows, можно использовать вторичные файлы данных, таким образом база данных сможет расти дальше. Для имени вторичного файла данных рекомендуется расширение NDF.

* **Журналы транзакций**

В этих файлах содержатся данные журнала - информация об операциях, произведенных над базой данных. Протоколирование производимых операций необходимо для обеспечения возможности восстановления базы данных после сбоев или неверных изменений данных. В каждой базе данных должен быть, как минимум, один файл журнала транзакций, а может быть и несколько таких файлов. Для файлов журнала транзакций рекомендуется расширение LDF.

По умолчанию и данные, и журналы транзакций помещаются на один и тот же диск и имеют один и тот же путь. Это сделано для поддержки однодисковых систем, но для производственных сред это может быть неоптимальным решением.

Рекомендуется помещать данные и файлы журнала на разные физические диски. Таким образом, число перемещений и время на позиционирование пишущей головки жесткого диска будет минимизировано. Например, файл журнала транзакций на отдельном физическом диске может писаться параллельно с файлами данных не требуя дополнительного позиционирования пишущей головки, так как файл журнала транзакций пишется последовательно.

Файлы данных MS SQL Server объединены в файловые группы. Данные внутри файловой группы распределяются по файлам пропорционально свободному месту в файлах. Например, если в файле **f1** свободно 100 МБ, а в файле **f2** **-** 200 МБ, то в файл **f1** будет записана одна часть данных, а в файл **f2 -** две части, при этом оба файла будут заполнены примерно в одно и то же время.

Как только заполняются все файлы в группе, компонент Database Engine перебирает файлы файловой группы в поисках файла для которого разрешено автоматическое увеличение. Увеличение размера происходит циклически по одному файлу за раз. Например, файловая группа состоит из трех файлов, для всех разрешено автоматическое увеличение. Когда свободное пространство во всех файлах группы закончится, будет расширен только первый файл. Когда заполнится первый файл и в файловую группу снова нельзя будет записывать новые данные, будет расширен второй файл. Когда заполнится второй файл и в файловую группу опять нельзя будет записывать новые данные, будет расширен третий файл. Когда заполнится третий файл и в файловую группу нельзя будет записывать новые данные, будет снова расширен первый файл и т. д.

Используя этот механизм распределения данных по файлам, можно увеличить производительность доступа к данным, разместив файлы файловой группы на разных физических носителях (при таком размещении данные могут писаться параллельно, образуя подобие чередующегося дискового массива RAID). Например, три файла **f1**, **f2** и **f3**, могут быть созданы на трех дисках соответственно и отнесены к файловой группе **fgroup1**. Можно создать таблицу размещенную на файловой группе **fgroup1**. В этом случае, запросы данных из таблицы будут распределены по трем дискам, что несколько улучшит производительность. Подобного улучшения производительности можно достичь и с помощью одного файла, созданного на чередующемся наборе дискового массива RAID. Тем не менее, файлы и файловые группы позволяют без труда добавлять новые файлы на новые диски.

Все файлы данных хранятся в файловых группах, перечисленных в следующей таблице.

|  |  |
| --- | --- |
| **Файловая группа** | **Описание** |
| Первичная | Файловая группа, содержащая первичный файл. Все системные таблицы размещены в первичной файловой группе. |
| Определяемая пользователем | Любая файловая группа, созданная пользователем при создании или изменении базы данных. |

У каждой базы данных есть первичная файловая группа. Эта файловая группа содержит первичный файл данных и все вторичные файлы, не входящие в другие файловые группы. Пользовательские файловые группы могут создаваться для удобства администрирования, распределения и размещения данных.

Например, для больших баз данных возможно резервное копирование отдельных файловых групп, что занимает значительно меньше времени, чем полное резервное копирование базы данных. Также файловая группа может быть целиком помечена как доступная только для чтения.

### Файловая группа по умолчанию

Если в базе данных создаются объекты без указания файловой группы, к которой они относятся, они назначаются файловой группе по умолчанию. В любом случае только одна файловая группа создается как файловая группа по умолчанию. Файлы в файловой группе по умолчанию должны быть достаточно большими, чтобы вмещать новые объекты, не назначенные другим файловым группам.

Файловая группа **PRIMARY** является группой по умолчанию, если только она не была изменена инструкцией **ALTER DATABASE**.

Системные объекты и таблицы распределяются внутри первичной файловой группы, а не внутри файловой группы по умолчанию. В случае если произошло заполнение всех файлов внутри первичной файловой группы до максимального размера, создание новых объектов и таблиц будет невозможно. Поэтому, при проектировании базы, очень важно учитывать возможный размер первичной файловой группы.

## Физическая структура файлов данных

Основной единицей хранилища данных в SQL Server является страница. Место на диске, предоставляемое для размещения файла данных (MDF- или NDF-файл) в базе данных, логически разделяется на страницы с непрерывным перечислением от 0 до n. Дисковые операции ввода-вывода выполняются на уровне страницы - SQL Server считывает или записывает целые страницы данных, что при правильной организации структур хранения позволяет минимизировать количество операций ввода-вывода (данные объектов, которые часто запрашиваются вместе при правильной организации будут располагаться на одной странице).

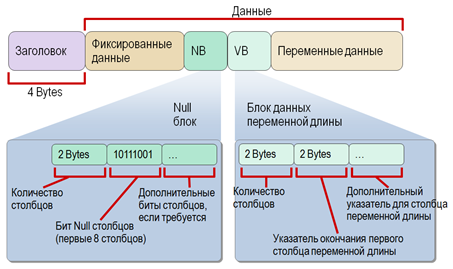
Файлы журнала не содержат страниц, в них содержится последовательность записей журнала.

### Страницы

В SQL Server размер страницы составляет 8 КБ. Это значит, что в одном мегабайте базы данных SQL Server содержится 128 страниц. Каждая страница начинается с 96-байтового заголовка, который используется для хранения системных данных о странице. Эти данные включают номер страницы, тип страницы, объем свободного места на странице и идентификатор единицы распределения объекта, которому принадлежит страница.

В следующей таблице представлены типы страниц, используемые в файлах данных базы данных SQL Server.

|  |  |
| --- | --- |
| **Тип страницы** | **Содержимое** |
| Данные | Строки данных со всеми данными, кроме данных типа **text**, **ntext**, **image**, **nvarchar(max)**, **varchar(max)** и **varbinary(max)**, а также данными типа **xml**, когда параметр **текст в строке** установлен в значение ON. |
| Индекс | Записи индекса. |
| Текст/изображение | Типы данных больших объектов: **text**, **ntext**, **image**, **nvarchar(max)**, **varchar(max)**, **varbinary(max)** и **xml**.  Столбцы переменной длины, когда строки данных превышают размер 8 КБ: **varchar**, **nvarchar**, **varbinary** и **sql\_variant**. |
| Глобальная карта распределения, общая глобальная карта распределения | Сведения о том, размещены ли экстенты. |
| Свободное место на страницах | Сведения о размещении страниц и доступном на них свободном месте. |
| Карта распределения индекса | Сведения об экстентах, используемых таблицей или индексом для единицы распределения. |
| Схема массовых изменений | Сведения об экстентах, измененных массовыми операциями со времени последнего выполнения инструкции BACKUP LOG для единицы распределения. |
| Схема разностных изменений | Сведения об экстентах, измененных с момента последнего выполнения инструкции BACKUP DATABASE для единицы распределения. |



### Поддержка больших строк

Максимальный объем данных и служебного кода, содержащихся в одной строке на странице, составляет 8 КБ. Это ограничение может быть нарушено только в двух случаях:

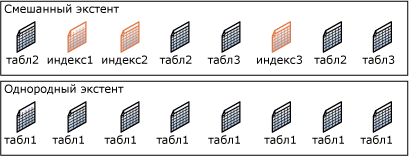
* если строка содержит данные типа **varchar(max)**, **nvarchar(max)** или **varbinary (max)** **-** размер этих данных не учитывается. Такие данные будут храниться в виде последовательности страниц типа «Текст/изображение», а в странице содержащей строку будет храниться только указатель на начало этой последовательности.
* если строка содержит данные переменной длины типа **varchar(не max)**, **nvarchar(не max)** **varbinary (не max)** или **sql\_variant**, то как только общий размер строки превысит предел в 8 060 байт, SQL Server динамически перемещает один или несколько столбцов переменной длины на отдельные страницы, начиная со столбца с наибольшим размером. Когда происходит перемещение столбца на отдельную страницу, на исходной странице сохраняется только указатель. Если при последующей операции размер строки уменьшается, SQL Server динамически перемещает столбцы обратно на исходную страницу данных. Важно отметить, что в данном случае предел в 8 060 байт может превышать только общий размер строки данных, при этом размер каждого поля строки не может быть больше размера страницы.

### Экстенты

Экстенты являются основными единицами организации пространства и используются для эффективного управления страницами. Экстент - это коллекция, состоящая из восьми физически непрерывных страниц.

SQL Server поддерживает два типа экстентов.

* Однородные экстенты принадлежат одному объекту; все восемь страниц в кластере могут быть использованы только этим владеющим объектом.
* Смешанные экстенты могут находиться в общем пользовании у не более восьми объектов. Каждая из восьми страниц в экстенте может находиться во владении разных объектов.



Использование смешанных экстентов позволяет сделать распределение места более эффективным, поскольку SQL Server не размещает целые экстенты для таблиц с небольшим объемом данных.

Новая таблица или индекс - это обычно страницы, выделенные из смешанных экстентов. При увеличении размера таблицы или индекса до восьми страниц эти таблица или индекс переходят на использование однородных экстентов для последовательных единиц распределения. При создании индекса для существующей таблицы, в которой содержится достаточно строк, чтобы сформировать восемь страниц в индексе, все единицы распределения для индекса находятся в однородных экстентах.

## FILESTREAM

На практике при создании приложений часто приходится хранить большие объемы неструктурированных данных, например, текстовые документы, изображения и видеоролики. Хранение множества таких объектов в реляционной базе данных может привести к падению производительности SQL Server Database Engine, поскольку для их кэширования будет использоваться буферный пул SQL Server. Поэтому обычно такие неструктурированные данные хранятся за пределами базы данных отдельно от структурированных данных. Подобное разделение как правило приводит к усложнению логики приложения. Либо, если данные связаны со структурированным хранилищем, могут быть ограничены возможности файловых потоков и производительность.

В SQL Server 2008 был добавлен новый тип хранения больших двоичных объектов (BLOB) FILESTREAM, который объединяет компонент Database Engine с файловой системой NTFS, размещая данные больших двоичных объектов (BLOB) типа **varbinary(max)** в файловой системе в виде файлов. Манипулирование данными, хранящимися в FILESTREAM осуществляется при помощи инструкций Transact-SQL, что позволяет использовать оптимизированное хранилище без изменения логики приложений его использующих. Интерфейсы файловой системы Windows также обеспечивают потоковый доступ к этим данным.

Для кэширования данных файлов в хранилище FILESTREAM используется системный кэш NT, что позволяет снизить возможное влияние данных FILESTREAM на производительность компонента Database Engine, поскольку буферный пул SQL Server не используется и память доступна для обработки запросов.

В SQL Server 2008 большие двоичные объекты (BLOB) могут храниться двумя способами:

* данные стандартного типа **varbinary(max)**, данные которых хранятся в таблице
* объекты FILESTREAM типа **varbinary(max)**, данные которых хранятся в файловой системе

Выбор в качестве хранилища базы данных или файловой системы определяется размером и назначением данных. Объекты FILESTREAM следует использовать в следующих случаях:

* средний размер сохраняемых объектов превышает 1 МБ, поскольку при работе с объектами меньшего размера сохранение больших двоичных объектов (BLOB) типа **varbinary(max)** в базе данных часто позволяет добиться лучшей производительности потоков;
* важен быстрый доступ для чтения;

Хранилище FILESTREAM реализовано в виде столбца типа **varbinary(max)**, данные которого хранятся в файловой системе как большие двоичные объекты (BLOB). Стандартное ограничение типа **varbinary(max)**, согласно которому размер файла не должен превышать 2 ГБ, не применяется к объектам BLOB, сохраняемым в файловой системе - размеры объектов ограничены только размером тома файловой системы.

Данные FILESTREAM могут сохраняться только в файловых группах FILESTREAM. Файловая группа FILESTREAM представляет собой особый тип файловой группы, в которой вместо файлов базы данных содержатся системные каталоги файлов. Эти системные каталоги файлов называются **контейнерами данных**, и являются интерфейсом между хранилищем компонента Database Engine и хранилищем файловой системы.

## Журнал транзакций и восстановление

Ведение журнала и процедура восстановления присущи не только SQL Server. Операции, выполняемые в большинстве систем управления реляционными базами данных, регистрируются (или записываются) на физическом и логическом уровне в виде событий, происходящих в структурах хранилища базы данных. Для каждого изменения в структурах хранилища создается отдельная запись журнала, описывающая изменяемую структуру и непосредственное изменение. Ведение журнала осуществляется способом, позволяющим повторить изменение или, при необходимости, отменить изменение и вернуть базу данных в исходное состояние. Записи журнала хранятся в специальном файле, который называется журналом транзакций.

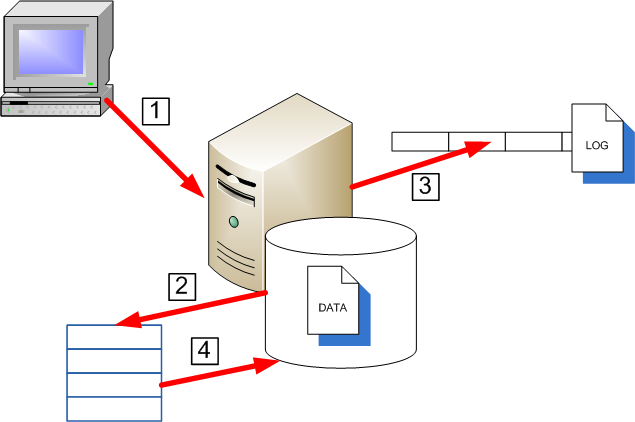
Набор из одного или нескольких изменений можно сгруппировать (и, фактически, так всегда и делается) в транзакцию, обеспечивающую базовую единицу процесса внесения изменений в базу данных. Транзакция завершается успешно (**фиксируется**) или завершается аварийно или отменяется (**откатывается**). В первом случае гарантируется, что изменения, входящие в транзакцию, отражены в базе данных, а во втором случае, - что изменения не будут отражены в базе данных.

MS SQL Server использует журнал с упреждающей записью, который гарантирует, что никакие изменения данных не будут записаны до сохранения на диске записи журнала. Работа механизма упреждающей записи основана на особенностях записи измененных страниц базы данных на диск. MS SQL Server поддерживает буферный кэш, из которого система считывает страницы данных при извлечении необходимых данных. Изменения данных не заносятся непосредственно на диск, а записываются на копии страницы в буферном кэше. Запись измененной страницы данных из буферного кэша на диск называется сбросом страницы на диск. Страница, измененная в кэше, но еще не записанная на диск, называется **грязной** страницей, такие страницы не записываются на диск, пока в базе данных не возникает контрольная точка.

Во время изменения страницы в буфере в кэше журнала строится запись журнала, соответствующая изменению данных. Запись журнала должна быть перенесена на диск до того, как соответствующая «грязная» страница будет записана из буферного кэша на диск. SQL Server реализует алгоритм, который защищает «грязную» страницу от записи на диск до переноса на него соответствующей записи журнала, а запись журнала будет сохранена на диск только после того, как соответствующая транзакция будет зафиксирована.

В качестве примера рассмотрим, что происходит, если обновляется одна строка таблицы. Допустим, имеется некоторая таблица **SimpleTable** содержащая столбец **c1** с целочисленными данными и столбец **c2** с символьными данными. В таблице имеется 10 000 строк, и пользователь отправляет запрос на обновление следующим образом:

UPDATE SimpleTable SET c1 = 10 WHERE c2 LIKE '%Paul%';



При получении запроса на обновление данных выполняются следующие операции:

* Страницы данных из **SimpleTable** считываются с диска в память (буферный кэш), после чего можно выполнить поиск соответствующих запросу строк. Оказывается, что на трех страницах данных имеется пять строк, соответствующих предикату предложения WHERE.
* Эти три страницы и пять строк данных блокируются для выполнения обновлений.
* Изменения вносятся в пять записей данных на трех страницах данных, находящихся в памяти.
* Изменения записываются также в записи журнала транзакций на диске.
* При возникновении контрольной точки данные три страницы будут сохранены на диске.

Контрольные точки позволяют достичь повышения производительности за счет группирования операций ввода/вывода, а также сократить время, необходимое для восстановления после сбоя:

* В терминах производительности, если бы страница данных вытеснялась на диск при каждом ее обновлении, число операций ввода/вывода в активно используемой системе могло бы превысить возможности подсистемы ввода/вывода. Сбрасывание на диск «грязных» страниц (те, которые были изменены с момента их считывания с диска) с некоторой периодичностью, требует гораздо меньше ресурсов, чем запись на диск страниц незамедлительно после внесения в них изменений.
* До тех пор, пока записи журнала, описывающие изменения, находятся на диске, изменения им соответствующие легко можно откатить, а в случае сбоя можно быстро восстановить данные, и результаты транзакции не будут утрачены.

Восстановление представляет собой процесс воспроизведения в базе данных изменений, описанных в записях журнала, или возврат базы данных к состоянию до этих изменений. Воспроизведение изменений данных по записям журнала называется фазой **REDO** (или наката) восстановления. Обращение изменений записей журнала называется фазой **UNDO** (или отката) восстановления. Другими словами, процедура восстановления обеспечиваете для транзакции и всех соответствующих записей журнала либо полное воспроизведение, либо полную отмену.

Восстановление принимает простую форму в случае отмены отдельной транзакции, когда она откатывается, и база данных не испытывает никаких последствий. Более сложную форму имеет восстановление в случае сбоя, когда выходит из строя SQL Server (по какой бы то ни было причине), и данные необходимо восстановить с целью возврата базы данных в состояние, согласованное с точки зрения транзакций. Это означает, что для всех транзакций, зафиксированных на момент сбоя, необходимо выполнить накат, чтобы результаты этих транзакций были отражены в базе данных. А для всех незавершенных на момент сбоя транзакций необходимо выполнить откат, чтобы результаты этих транзакций не были записаны в базу данных, поскольку в SQL Server не существует средств продолжения транзакции после сбоя. Если бы для результатов частично завершенной транзакции не был выполнен откат, база данных оказалась бы в несогласованном состоянии (возможно, даже с повреждениями структуры, в зависимости от операции, выполнявшейся транзакцией в момент сбоя).

В простом случае отката одной транзакции механизм восстановления может быстро и без труда проследовать по цепочке записанных в журнал операций, от самой последней до первой операции, и обратить результаты всех операций в обратном их выполнению порядке. Страницы базы данных, подвергшиеся воздействию транзакции, находятся либо все еще в буферном пуле, либо уже на диске. В любом случае гарантируется, что доступное изображение страницы несет на себе последствия выполнения транзакции, и их необходимо отменить.

Во время восстановления после сбоя механизм более сложен. Поскольку страницы базы данных не записываются на диск при фиксации транзакции, нет гарантии того, что набор страниц базы данных на диске точно отражает набор изменений, описанных в журнале транзакций - как для зафиксированных, так и для незафиксированных транзакций. Для решения этой проблемы в заголовках всех страниц (96-байтная часть 8КБ страницы, содержащая метаданные с информацией о странице) базы данных имеется поле, отражающее номер последней записи журнала, оказавшей влияние на страницу. Это дает возможность системе восстановления принять решение относительно конкретной записи журнала, которую необходимо восстановить.

* Для записи журнала из зафиксированной транзакции, в которой страница базы данных имеет номер транзакции не меньший, чем номер транзакции записи журнала, не требуется никаких действий. Результаты воздействия записи журнала уже записаны в страницу на диске.
* Для записи журнала из зафиксированной транзакции, в которой страница базы данных имеет номер транзакции, меньший, чем номер транзакции записи журнала, необходимо выполнить накат записи журнала, чтобы обеспечить сохранение результатов транзакции.
* Для записи журнала из незафиксированной транзакции, в которой страница базы данных имеет номер транзакции, не меньший, чем номер транзакции записи журнала, необходимо выполнить откат записи журнала, чтобы результаты транзакции не были сохранены.
* Для записи журнала из незафиксированной транзакции, в которой страница базы данных имеет номер транзакции, меньший, чем номер транзакции записи журнала, не требуется никаких действий. Результаты воздействия записи журнала не были сохранены на странице на диске, и в таком случае ничего делать не требуется.

Механизм восстановления после сбоя прочитывает журнал транзакций и обеспечивает сохранение всех результатов зафиксированных транзакций в базе данных, а результаты всех незафиксированных транзакций не сохраняются в базе данных - то есть, выполняются фазы **REDO** и **UNDO**, соответственно. По завершении восстановления после сбоя база данных становится согласованной с точки зрения транзакций и доступной для использования.

Ранее упоминалось, что одним из достоинств применения контрольных точкек является сокращение времени, занимаемого процедурой восстановления после сбоя. Периодическое сбрасывание на диск всех «грязных» страниц сокращает число страниц, измененных зафиксированными транзакциями, но изображений которых еще нет на диске. Это, в свою очередь, сокращает число страниц, к которым требуется применять операцию REDO во время восстановления после сбоя.

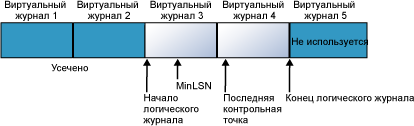
## Физическая архитектура журнала транзакций

Компонент SQL Server Database Engine делит каждый физический файл журнала на несколько виртуальных файлов журнала. Виртуальные файлы журнала не имеют фиксированных размеров, и компонент Database Engine динамически определяет размер виртуальных файлов журнала при создании или расширении файлов журнала. Также не существует определенного числа виртуальных файлов журнала, приходящихся на один физический файл журнала. Компонент Database Engine стремится обслуживать небольшое число виртуальных файлов. Администраторы не могут настраивать или устанавливать размеры и число виртуальных файлов журнала.

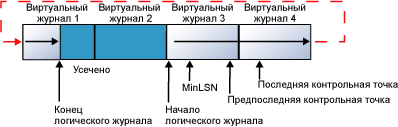
Виртуальные файлы журнала влияют на производительность системы лишь в том случае, если эти файлы журнала определяются малыми значениями **size** и **growth\_increment**. Если эти файлы журнала в результате большого числа малых приращений будут разрастаться до крупных размеров, они будут иметь множество виртуальных файлов журнала. В итоге может увеличиться время запуска базы данных, а также снизиться скорость операций резервного копирования и восстановления. Рекомендуется выбирать для файлов журнала значение параметра **size**, близкое к окончательному требуемому размеру, а также задавать относительно большое значение **growth\_increment**.

Журнал транзакций состоит из отдельных записей, каждая из которых идентифицируется регистрационным номером транзакции в журнале (LSN). Регистрационный номер транзакции в журнале является возрастающим номером из трех частей, однозначно определяющим положение записи журнала в журнале транзакций. Все записи журнала в транзакции хранятся в последовательном порядке в журнале транзакций и содержат код транзакции и LSN предыдущей записи транзакции.

Журнал транзакций пишется циклично. Для иллюстрации рассмотрим следующий пример. Пусть база данных имеет один физический файл журнала, разделенный на четыре виртуальных файла журнала. При создании базы данных логический файл журнала начинается в начале физического файла журнала. Новые записи журнала добавляются в конце логического журнала и приближаются к концу физического файла журнала. Усечение журнала освобождает любые виртуальные журналы, все записи которых находятся перед минимальным регистрационным номером восстановления в журнале транзакций (**MinLSN**). **MinLSN** является в журнале транзакций регистрационным номером самой старой записи, которая необходима для успешного отката на уровне всей базы данных.



Когда конец логического журнала достигнет конца физического файла журнала, новые записи журнала будут размещаться в начале физического файла журнала.



Этот цикл повторяется бесконечно, пока конец логического журнала не совмещается с началом этого логического журнала. Если старые записи журнала усекаются достаточно часто, так что при этом всегда остается место для новых записей журнала, созданных с новой контрольной точки, журнал постоянно остается незаполненным. Однако, если конец логического журнала совмещается с началом этого логического журнала, происходит одно из двух событий, перечисленных ниже.

* Если для данного журнала применена установка **FILEGROWTH** и на диске имеется свободное место, файл расширяется на величину, указанную в **growth\_increment**.
* Если установка **FILEGROWTH** не применяется или диск, на котором размещается файл журнала, имеет меньше свободного места, чем это указано в **growth\_increment**, формируется ошибка.

Если в журнале содержится несколько физических файлов журнала, ведение журнала также осуществляется циклически, то есть логический журнал будет продвигаться по всем физическим файлам журнала до тех пор, пока он не вернется на начало первого физического файла журнала.

**Лекция № 10**

## Опции базы данных

Все опции базы данных можно разделить на следующие категории:

### Параметры сортировки (collation)

Параметр сортировки символьных строк для базы данных выбирается из списка. Параметр определяет страницу кодировки и опции сравнения - чувствительность к регистру, учитывать или нет диакритические знаки (например, различать ли при сортировке буквы е и ё).

### Модель восстановления (RECOVERY)

Существуют три модели восстановления (**FULL**, **BULK\_LOGGED**, **SIMPLE**), и все они оказывают влияние на поведение журнала транзакций и способ регистрации операций, либо и на то, и на другое.

**Модель восстановления FULL** подразумевает, что регистрируется каждая часть каждой операции, и это называется полной регистрацией. После выполнения полного резервного копирования базы данных в модели восстановления FULL в журнале транзакций не будет проводиться автоматическое усечение до тех пор, пока не будет выполнено резервное копирование журнала. Если вы не намерены использовать резервные копии журнала и возможность восстановления состояния базы данных на конкретный момент времени, не следует использовать модель восстановления FULL.

**Модель восстановления BULK\_LOGGED** обладает такой же семантикой усечения журнала транзакций, как и модель восстановления FULL, но допускает только частичную регистрацию некоторых операций, что называется минимальной регистрацией. Примерами являются повторное создание индекса и операции массовой загрузки - в модели восстановления FULL регистрируется вся операция, а модели восстановления BULK\_LOGGED регистрируются только изменения распределения. Это радикально сокращает число создаваемых записей журнала и, в свою очередь, уменьшает разрастания журнала транзакций.

**Модель восстановления SIMPLE**, фактически ведет себя с точки зрения ведения журнала так же, как и модель восстановления BULK\_LOGGED, но имеет совершенно другую семантику усечения журнала транзакций. В модели восстановления SIMPLE невозможны резервные копии журнала, что означает, что журнал может быть усечен (если ничто не удерживает записи журнала в активном состоянии) при возникновении контрольной точки. Таким образом, если вы намерены использовать резервные копии журнала и возможность восстановления состояния базы данных на конкретный момент времени, не следует использовать данную модель восстановления.

### Уровень совместимости (COMPATIBILITY\_LEVEL)

Последняя версия SQL Server, которую поддерживает база данных. Возможными значениями являются **SQL Server 2008**, **SQL Server 2005** и **SQL Server 2000**.

### Автоматические

**Автоматическое закрытие (AUTO\_CLOSE)**

Будет ли база данных полностью закрываться и освобождать ресурсы после выхода последнего пользователя. Допустимые значения **-** **ON** и **OFF**. Если значение равно **ON**, база данных закрывается полностью и освобождает ресурсы после того, как из системы выходит последний пользователь. Параметр AUTO\_CLOSE полезен для настольных баз данных, поскольку он позволяет управлять файлами базы данных так же, как обычными файлами. Они могут быть перемещены, скопированы для создания резервной копии или даже отосланы по электронной почте другим пользователям.

**Автоматическое сжатие** **(AUTO\_SHRINK)**

И файлы данных, и файлы журналов могут быть автоматически сжаты, то есть может быть уменьшено количество неиспользуемого пространства внутри файлов. При включенном параметре AUTO\_SHRINK файлы будут сжаты, если более 25 процентов файла содержит неиспользуемое пространство. Файл будет сжат до размера, в котором 25 процентов файла — неиспользуемое пространство, или до того размера, который был у файла при создании, каким бы большим он ни был. AUTO\_SHRINK уменьшает размер журнала транзакций только в том случае, если выбрана простая модель восстановления базы данных или была создана резервная копия журнала. Если этот параметр установлен в состояние **OFF**, файлы базы данных не будут автоматически сжиматься при периодической проверке на неиспользуемое пространство.

**Автоматическое создание статистики (AUTO\_CREATE\_STATISTICS)**

Исполнению запроса компонентом Database Engine предшествует его анализ оптимизатором запросов, с целью определения наиболее эффективного для конкретного условия фильтра способа извлечения запрошенных данных. Помимо самого запроса оптимизатором запросов используется схема базы данных (определения таблиц и индексов) и статистики базы данных.

Для оценки количества строк в результатах запроса и как следствие нагрузки на вычислительные ресурсы оптимизатор запросов полагается на статистику распределения. Статистика - это объект, содержащие статистические сведения о распределении значений в одном или нескольких столбцах таблицы. Например, оптимизатор запросов может использовать оценочное количество элементов, чтобы выбрать метод для обращения к данным в таблице.

Каждый объект статистики создается для списка из одного или нескольких столбцов таблицы и содержит гистограмму, в которой отображается распределение значений в первом столбце. Объекты статистики для нескольких столбцов также хранят статистические сведения о корреляции значений между столбцами. Эти статистические данные корреляции называются значениями плотности и получаются из числа уникальных строк значений столбцов.

Данный параметр определяет, будет ли база данных автоматически создавать отсутствующие статистические данные оптимизации. Допустимые значения - **ON** и **OFF**. Оптимизатор запросов в случае необходимости создает статистику по отдельным столбцам в предикатах запросов, чтобы улучшить планы запросов и повысить производительность запросов. Значение по умолчанию - ON. Для большинства баз данных рекомендуется использовать значение по умолчанию. Если значение равно **ON**, во время оптимизации автоматически формируются все отсутствующие статистические данные, необходимые запросу для оптимизации.

**Автоматическое обновление статистики** **(AUTO\_UPDATE\_STATISTICS)**

Указывает, что оптимизатор запросов обновляет статистику, если она используется в запросе и может оказаться устаревшей. Статистика становится устаревшей, после того как операции вставки, обновления, удаления или слияния изменяют распределение данных в таблице. Оптимизатор запросов определяет, когда статистика может оказаться устаревшей, подсчитывая операции изменения данных с момента последнего обновления статистики и сравнивая количество изменений с пороговым значением. Пороговое значение основано на количестве строк в таблице или индексированном представлении. Допустимые значения **ON** и **OFF**. Если значение равно **ON**, во время оптимизации автоматически формируются все устаревшие статистические данные, необходимые запросу для оптимизации.

**Асинхронное автоматическое обновление статистики** **(AUTO\_UPDATE\_STATISTICS\_ASYNC)**

При значении **ON** запросы, которые запускают автоматическое обновление устаревшей статистики, не будут ожидать обновления статистики перед компиляцией. Последующие запросы будут использовать обновленную статистику, когда она будет доступна.

При значении **OFF** запросы, которые запускают автоматическое обновление устаревшей статистики, будут ожидать, пока обновленная статистика не будет использоваться в плане оптимизации запроса.

Установка данного параметра в значение **ON** имеет смысл только в том случае, если параметр **Автоматическое обновление статистики** также имеет значение **ON**.

### Курсор

**Закрывать курсор при разрешении фиксации** **(CURSOR\_CLOSE\_ON\_COMMIT)**

Будет ли курсор закрываться после фиксации транзакции, открывшей этот курсор. Допустимые значения — **True** и **False**. Если значение равно **True**, закрываются все курсоры, открытые при фиксации или откате транзакции. Если значение равно **False**, при фиксации транзакции такие курсоры остаются открытыми. Если значение равно **False**, откат транзакции закрывает все курсоры, за исключением определенных как INSENSITIVE или STATIC.

**Курсор по умолчанию** **(CURSOR\_DEFAULT)**

Поведение курсора по умолчанию. Если значение равно **LOCAL**, курсор объявляется по умолчанию как **LOCAL** и область видимости курсора будет локальна по отношению к пакету, хранимой процедуре или триггеру, в которых он был создан. Имя курсора действительно только внутри этой области. Если значение равно **GLOBAL**, курсоры языка Transact-SQL по умолчанию объявляются как глобальные.

### Разное

**ANSI NULL по умолчанию (ANSI\_NULL\_DEFAULT)**

Поведение по умолчанию операторов сравнения «равно» (=) и «не равно» (<>) при использовании со значениями NULL. Допустимые значения — **True** (вкл.) и **False** (выкл.).

**Включены ANSI NULL (ANSI\_NULLS)**

Поведение операторов сравнения «равно» (=) и «не равно» (<>) при использовании со значениями NULL. Допустимые значения - **True** (вкл.) и **False** (выкл.). Если значение равно **True**, всем сравнениям со значениями NULL присваивается значение UNKNOWN. Если значение равно **False**, сравнения значений, отличных от Юникода, со значениями NULL получают значение **True**, если оба они равны NULL.

**Включено заполнение ANSI** **(ANSI\_PADDING)**

Включено ли заполнение ANSI. Допустимые значения — **True** (вкл.) и **False** (выкл.).

**Включены предупреждения ANSI (ANSI\_WARNINGS)**

Поведение по стандарту ISO для некоторых условий возникновения ошибок. Если значение равно **True**, формируется предупреждающее сообщение, если в статистических функциях (таких как SUM, AVG, MAX, MIN, STDEV, STDEVP, VAR, VARP или COUNT) появляются значения NULL. Если значение равно **False**, предупреждающее сообщение не выдается.

**Включено прерывание при делении на ноль (ARITHABORT)**

Включен ли параметр, разрешающий аварийное прерывание арифметических действий. Допустимые значения **-** **True** и **False**. Если значение равно **True**, ошибка переполнения или деления на ноль приводит к прерыванию выполнения запроса или пакета. Если произошла ошибка в транзакции, для этой транзакции выполняется откат. Если значение равно **False**, выводится предупреждающее сообщение, но запрос, пакет или транзакция продолжают выполняться, как если бы ошибки не произошло.

**Объединение со значением NULL дает NULL** **(CONCAT\_NULL\_YIELDS\_NULL)**

Способ конкатенации значений NULL. Если свойство имеет значение **True**, то **string** + значение NULL возвращает NULL. При значении **False** результатом будет **string**.

**Включена оптимизация корреляции дат** **(DATE\_CORRELATION\_OPTIMIZATION)**

При значении **True** SQL Server поддерживает статистику корреляции между любыми двумя таблицами в базе данных, которые связаны ограничением FOREIGN KEY и имеют столбцы **datetime**.

При значении **False** статистика корреляции не поддерживается.

**Автоокругление чисел** **(NUMERIC\_ROUNDABORT)**

Способ обработки ошибок округления базой данных. Допустимые значения — **True** и **False**. Если значение равно **True**, формируется ошибка, когда в выражении происходит потеря точности. Если значение равно **False**, потери точности не приводят к формированию сообщений об ошибках, а результат округляется до степени точности столбца или переменной, в которых сохраняется результат.

**Определение параметров** **(PARAMETERIZATION)**

Если выбрано значение **SIMPLE**, параметризация запросов основывается на поведении базы данных по умолчанию. Если выбрано значение **FORCED**, SQL Server выполняет параметризацию всех запросов в базе данных.

**Включены заключенные в кавычки идентификаторы** **(QUOTED\_IDENTIFIER)**

Можно ли использовать ключевые слова SQL Server как идентификаторы (имена объектов или переменных), если они заключены в кавычки. Допустимые значения - **True** и **False**.

**Включены рекурсивные триггеры (RECURSIVE\_TRIGGERS)**

Могут ли триггеры запускаться другими триггерами. Допустимые значения - **True** и **False**. Если значение равно **True**, рекурсивный запуск триггеров разрешен. Если значение равно **False**, запрещается только прямая рекурсия. Чтобы отключить косвенную рекурсию, необходимо присвоить параметру сервера вложенные триггеры значение 0, используя процедуру **sp\_configure**.

**Доверенная БД** **(TRUSTWORTHY)**

При значении **True** этот параметр указывает, что SQL Server разрешает доступ к ресурсам вне базы данных в рамках контекста олицетворения, установленного для базы данных. Контекст олицетворения для базы данных можно установить при помощи пользовательской инструкции EXECUTE AS или предложения EXECUTE AS в модулях базы данных.

Это свойство также разрешает создание и выполнение небезопасных и внешних сборок в базе данных. Владелец базы данных должен задать для этого свойства значение **True**. В дополнение к этому, он должен обладать разрешением EXTERNAL ACCESS ASSEMBLY или UNSAFE ASSEMBLY на уровне сервера.

По умолчанию для всех пользовательских и системных баз данных (за исключением **MSDB**) это свойство установлено в значение **False**. Оно не изменяется для баз данных **model** и **tempdb**.

### Восстановление

**Проверка страниц (PAGE\_VERIFY)**

Обнаруживает поврежденные страницы базы данных, вызванные ошибками дискового ввода-вывода. Ошибки дискового ввода-вывода могут быть причиной повреждения базы данных и обычно происходят при сбоях питания или сбоях оборудования диска, происходящих во время записи страницы на диск.

* **CHECKSUM**

Вычисляет контрольную сумму по содержимому целой страницы и сохраняет полученное значение в ее заголовке при записи страницы на диск. При чтении страницы с диска контрольная сумма вычисляется повторно и сравнивается с сохраненным в заголовке страницы значением. Если значения не соответствуют, будет выведено сообщение об ошибке 824 (ошибка контрольной суммы) как в журнал ошибок SQL Server, так и в журнал событий Windows. Ошибка контрольной суммы указывает на проблему ввода-вывода. Чтобы определить первопричину, необходимо тщательно проверить оборудование, драйверы, BIOS, фильтрующее программное обеспечение (например, антивирусное) и другие компоненты ввода-вывода.

* **TORN\_PAGE\_DETECTION**

Сохраняет определенный двухбитовый шаблон для каждого 512-байтового сектора в 8-килобайтной (КБ) странице базы данных и сохраняет в базе данных заголовок страницы при записи страницы на диск. При чтении страницы с диска биты разрыва, хранимые в заголовке страницы, сравниваются с действительными сведениями о секторах страницы. Несовпадающие значения указывают, что только часть страницы была записана на диск. В этой ситуации сообщение об ошибке 824 (ошибка разрыва страницы) будет выведено как в журнал ошибок SQL Server, так и в журнал событий Windows. Разорванные страницы обычно обнаруживаются при восстановлении базы данных, если они действительно не полностью записаны.

* **NONE**

Страница базы данных при записи не будет формировать значение CHECKSUM или TORN\_PAGE\_DETECTION. SQL Server не будет проверять контрольную сумму и разрывы страниц при считывании, даже если значение CHECKSUM или TORN\_PAGE\_DETECTION будет присутствовать в заголовке страницы.

### Состояние

**База данных только для чтения**

Определяет будет ли база данных доступна только для чтения. Допустимые значения **-** **True** и **False**. Если значение равно **True**, пользователи могут только считывать данные в базе данных. Им не разрешается изменять данные или объекты базы данных. Тем не менее саму базу данных можно удалить, используя инструкцию DROP DATABASE. Базу данных нельзя использовать, когда задается новое значение параметра **База данных только для чтения**. Исключением является база данных **master**, и только системный администратор может использовать базу данных **master** во время задания параметра.

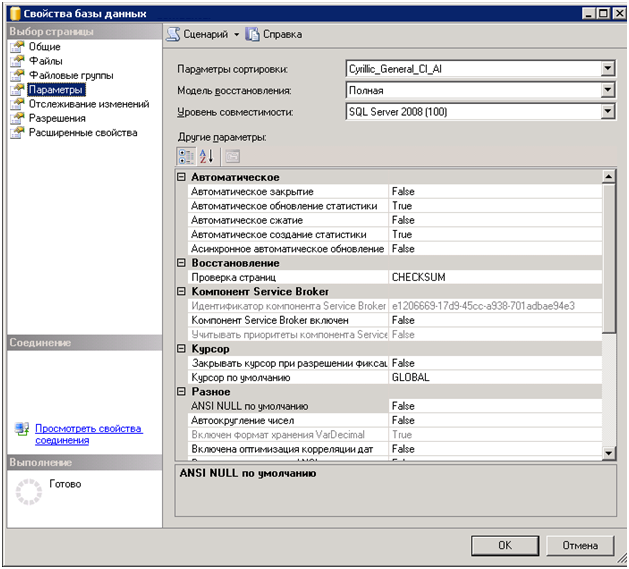
**Ограничение доступа**

Какие пользователи имеют доступ к базе данных. Допустимые значения:

* **Несколько** (MULTI\_USER)  
  Нормальное состояние производственной базы данных, позволяет нескольким пользователям получать одновременный доступ к базе данных.
* **Один** (SINGLE\_USER)  
  Используется для операций обслуживания, одновременный доступ к базе данных получает только один пользователь.
* **Ограничен** (RESTRICTED\_USER)  
  Базу данных могут использовать только члены ролей **db\_owner**, **dbcreator** или **sysadmin**.

## Источники информации о базе данных

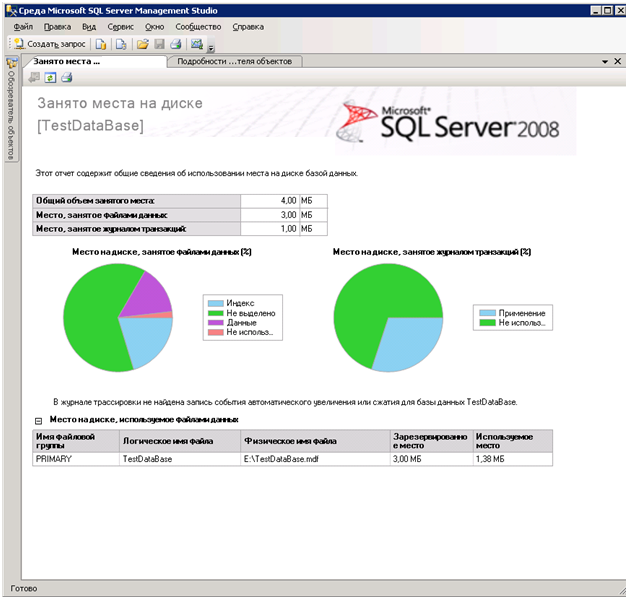
### Окно свойств SQL Server Management Studio



### Представления каталога

|  |  |
| --- | --- |
| Представление каталога | Отображение сведений о |
| **sys.databases** | Информация о всех базах данных и опциях для них установленных. |
| **sys.filegroups** | Информация обо всех файловых группах. |
| **sys.database\_files** | Информация о файлах баз данных. |

### Отчеты SQL Server Management Studio



**Лекция № 11**

## Создание базы данных

В одном экземпляре SQL Server может быть создано до 32 767 баз данных. В каждой базе данных должно быть по крайней мере 2 файла (первичный файл и файл журнала транзакций) и по крайней мере одна файловая группа. Для каждой базы данных может указываться не более 32 767 файлов и 32 767 файловых групп.

При создании новой базы данных в нее копируются все пользовательские объекты из базы данных **model**. Таким образом, можно добавлять любые объекты, например, таблицы, представления, хранимые процедуры и типы данных, в базу данных **model**, чтобы включать их во все вновь создаваемые базы данных. Файлы данных следует делать как можно большего размера, в соответствии с максимальным предполагаемым объемом данных в базе данных.

Создание базы данных осуществляется выполнением команды **CREATE DATABASE**:

CREATE DATABASE database\_name

[ ON

{ [ PRIMARY ] [ <filespec> [ ,...n ]

[ , <filegroup> [ ,...n ] ]

[ LOG ON { <filespec> [ ,...n ] } ] }

]

[ COLLATE collation\_name ]

]

<filespec> ::=

{

(

NAME = logical\_file\_name ,

FILENAME = { 'os\_file\_name' }

[ , SIZE = size [ KB | MB | GB | TB ] ]

[ , MAXSIZE = { max\_size [ KB | MB | GB | TB ] | UNLIMITED } ]

[ , FILEGROWTH = growth\_increment [ KB | MB | GB | TB | % ] ]

) [ ,...n ]

}

<filegroup> ::=

{

FILEGROUP filegroup\_name [ CONTAINS FILESTREAM ] [ DEFAULT ]

<filespec> [ ,...n ]

}

**database\_name**

Имя новой базы данных. Имена баз данных должны быть уникальны внутри экземпляра SQL Server и должны соответствовать правилам для идентификаторов.

Аргумент database\_name может иметь максимальную длину 128 символов, если для файла журнала не указано логическое имя. Если логическое имя файла не указано, то SQL Server формирует для журнала имена logical\_file\_name и os\_file\_name путем добавления суффикса к database\_name. Это ограничивает длину аргумента database\_name 123 символами, чтобы формируемое логическое имя файла было не длиннее 128 символов.

Если имя файла данных не указано, то SQL Server использует аргумент database\_name в качестве имен logical\_file\_name и os\_file\_name. Путь по умолчанию берется из реестра. Его можно изменить в «Свойствах сервера» (страница «Параметры базы данных») в Management Studio. Изменение пути по умолчанию требует перезапуска SQL Server.

**ON**

Указывает, что дисковые файлы, используемые для хранения разделов данных в базе данных, определяются явно. Параметр ON необходимо применять, если за ним следует список элементов <filespec> с разделителями-запятыми, которые определяют файлы данных первичной файловой группы. За списком файлов в первичной файловой группе может следовать необязательный список элементов <filegroup> с разделителями-запятыми, которые определяют файловые группы пользователей и принадлежащие им файлы.

**PRIMARY**

Указывает, что связанный список <filespec> определяет первичный файл. Первый файл, указанный в элементе <filespec> в первичной файловой группе, становится первичным файлом. В базе данных может быть только один первичный файл.

Если параметр PRIMARY не указан, то первый файл списка в инструкции CREATE DATABASE становится первичным файлом.

**LOG ON**

Указывает, что дисковые файлы, используемые для хранения журнала базы данных, то есть файлы журналов, определяются явно. За параметром LOG ON следует список элементов <filespec> с разделителями-запятыми, которые определяют файлы журналов. Если параметр LOG ON не указан, автоматически создается один файл журнала, размер которого определяется большей из следующих двух величин: 512 КБ или 25 процентов от суммы размеров всех файлов с данными в базе данных. Этот файл помещается в местоположение для журнала по умолчанию.

**<filespec>**

Управляет свойствами файла.

**NAME logical\_file\_name**

Логическое имя, используемое в SQL Server при указании ссылки на файл. Аргумент **logical\_file\_name** должен быть уникальным в базе данных и должен соответствовать правилам для идентификаторов. Имя может быть символом или константой Юникода, а также обычным идентификатором или идентификатором с разделителями.

**FILENAME { 'os\_file\_name' }**

Задает физическое имя **-** путь и имя файла, используемые операционной системой при создании файла. Файл должен находиться на одном из следующих устройств: на локальном сервере, где установлен SQL Server, в сети хранения данных SAN или в сети на основе iSCSI. Указанный путь должен существовать до выполнения инструкции CREATE DATABASE.

Для файловой группы FILESTREAM параметр FILENAME указывает путь, где будут храниться данные FILESTREAM. Должен существовать путь вплоть до последнего каталога, но последний каталог существовать не должен. Например, если указать путь "D:\DATA\ FilestreamData ", папка " D:\DATA" должна существовать до запуска инструкции CREATE DATABASE, а папка "FilestreamData" - не должна.

Для файловой группы FILESTREAM может быть только один файл, причем файловую группу и файл необходимо создавать в одной инструкции.

Параметры **SIZE, MAXSIZE и FILEGROWTH** не применяется к файловой группе FILESTREAM и недоступны, если путь к файлу указан в формате UNC.

Если файл находится в необработанной секции (**raw partition**), аргумент os\_file\_name должен указывать только букву диска существующей необработанной секции. В каждой необработанной секции может быть создан только один файл.

Файлы данных не следует размещать в файловых системах со сжатием, за исключением случаев, когда файлы являются вторичными и доступны только для чтения или вся база данных доступна только для чтения. Файлы журналов ни в коем случае не должны размещаться в сжатых файловых системах.

**SIZE size**

Задает начальный размер файла. Параметр SIZE не может указываться, если аргумент os\_file\_name задан как путь в формате UNC.

Если аргумент size не задан для первичного файла, то компонент Database Engine использует размер первичного файла, указанный в базе данных model. Когда указан вторичный файл данных или журнала, но параметр size для файла не указан, компонент Database Engine задает размер файла равным 1 МБ. Размер, указанный для первичного файла, не должен быть менее размера первичного файла базы данных model.

Можно использовать суффиксы килобайт (KB), мегабайт (MB), гигабайт (GB) и терабайт (TB). По умолчанию — MБ. Аргумент Size имеет тип integer. Для значений, превышающих 2 147 483 647, используются более крупные единицы измерения.

**MAXSIZE max\_size**

Задает максимальный размер, до которого может расти файл. Параметр **MAXSIZE** нельзя указывать, если аргумент os\_file\_name задан как путь в формате UNC. Можно использовать суффиксы KB, MB, GB и TB. По умолчанию — MБ. Укажите целое число (без дробной части). Если аргумент max\_size не указан, размер файла будет увеличиваться до заполнения диска. Аргумент Max\_size имеет тип integer. Для значений, превышающих 2 147 483 647, используются более крупные единицы измерения. **UNLIMITED у**казывает, что файл может расти вплоть до заполнения диска. В SQL Server файл журнала, для которого задано неограниченное увеличение размера, имеет максимальный размер 2 ТБ, а файл данных — 16 ТБ.

**FILEGROWTH growth\_increment**

Задает автоматическое приращение размера файла. Значение параметра FILEGROWTH для файла не может превосходить значение параметра MAXSIZE. Параметр FILEGROWTH нельзя указывать, если аргумент os\_file\_name задан как путь в формате UNC.

Значение может быть указано в килобайтах, мегабайтах, гигабайтах, терабайтах или процентах (%). Если указано число без суффикса MB, KB или %, то по умолчанию используется MB. Если размер указан в процентах (%), то шаг роста это заданная часть в процентах от размера файла во время этого файла. Указанный размер округляется до ближайших 64 КБ.

Значение 0 указывает, что автоматическое приращение отключено и добавление пространства запрещено.

Если параметр FILEGROWTH не задан, значением по умолчанию является 1 МБ для файлов данных и 10% для файлов журналов, минимальное значение — 64 КБ.

**<filegroup>**

Управляет свойствами файловой группы. Файловая группа не может указываться для моментального снимка базы данных.

**FILEGROUP filegroup\_name**

Логическое имя файловой группы. Аргумент filegroup\_name должен быть уникальным в базе данных и не может быть именем PRIMARY или PRIMARY\_LOG, предоставленным системой. Имя может быть символом или константой Юникода, а также обычным идентификатором или идентификатором с разделителями. Имя должно соответствовать правилам для идентификаторов.

**CONTAINS FILESTREAM**

Указывает, что файловая группа хранит большие двоичные объекты (BLOB) FILESTREAM в файловой системе.

**DEFAULT**

Задает указанную файловую группу как файловую группу по умолчанию в базе данных.

**Пример:**

В следующем примере создается база данных с именем TestDatabase. В ней создаются 3 файла - первичный файл данных (логическое имя "TestDatabaseData" в первичной файловой группе), вторичный файл данных (логическое имя "TestDatabaseUserFile" в файловой группе USERFG), FILESTREAM (логическое имя " Stream1" в файловой группе FileStreamG1) и файл журнала транзакций "TestDatabase\_log". Для каждого файла задается физическое имя файла (путь в файловой системе), начальный размер и автоматическое приращение.

CREATE DATABASE [TestDatabase] ON PRIMARY

( NAME = 'TestDatabaseData', FILENAME = 'C:\MSSQL\DATA\TestDatabaseData.mdf' , SIZE = 3072KB , FILEGROWTH = 1024KB ),

FILEGROUP [USERFG]

( NAME = 'TestDatabaseUserFile', FILENAME = 'D:\MSSQL\DATA\TestDatabaseUserFile.ndf' , SIZE = 30MB , FILEGROWTH = 2MB )

FILEGROUP [FileStreamG1] CONTAINS FILESTREAM( NAME = Stream1,

FILENAME = ' D:\MSSQL\DATA\FilestreamData ')

LOG ON

( NAME = 'TestDatabase\_log', FILENAME = N'C:\MSSQL\DATA\TestDatabase\_log.ldf' , SIZE = 1024KB , FILEGROWTH = 10%)

## Изменение настроек базы данных

Изменение настроек базы данных производится с помощью команды **ALTER DATABASE**.

Данная команда позволяет добавлять или удалять файлы и файловые группы из базы данных, а также изменять параметры базы данных или ее файлов и файловых групп.

ALTER DATABASE database\_name

{

   MODIFY NAME = new\_database\_name

  | COLLATE collation\_name

  | <add\_or\_modify\_files>

| <add\_or\_modify\_filegroups>

}

<add\_or\_modify\_files>::=

{

ADD FILE <filespec> [ ,...n ]

[ TO FILEGROUP { filegroup\_name } ]

| ADD LOG FILE <filespec> [ ,...n ]

| REMOVE FILE logical\_file\_name

| MODIFY FILE <filespec>

}

<filespec>::=

(

NAME = logical\_file\_name

[ , NEWNAME = new\_logical\_name ]

[ , FILENAME = {'os\_file\_name' | 'filestream\_path' } ]

[ , SIZE = size [ KB | MB | GB | TB ] ]

[ , MAXSIZE = { max\_size [ KB | MB | GB | TB ] | UNLIMITED } ]

[ , FILEGROWTH = growth\_increment [ KB | MB | GB | TB| % ] ]

)

<add\_or\_modify\_filegroups>::=

{

ADD FILEGROUP filegroup\_name [ CONTAINS FILESTREAM ]

| REMOVE FILEGROUP filegroup\_name

| MODIFY FILEGROUP filegroup\_name

{

<filegroup\_updatability\_option>

|DEFAULT

| NAME = new\_filegroup\_name

}

}

<filegroup\_updatability\_option>::=

{

{ READONLY | READWRITE }

| { READ\_ONLY | READ\_WRITE }

}

**database\_name** - имя изменяемой базы данных.

**MODIFY NAME = new\_database\_name** - присваивает базе данных новое имя, указанное в аргументе **new\_database\_name**.

**COLLATE collation\_name** - определяет параметры сортировки для базы данных. Аргумент collation\_name может быть либо именем параметров сортировки Windows, либо именем параметров сортировки SQL. Если аргумент не указан, базе данных будут назначены параметры сортировки экземпляра SQL Server.

**<** **add\_or\_modify\_files >::=**

**ADD FILE**

Добавляет файл к базе данных.

**TO FILEGROUP { filegroup\_name }**

Указывает файловую группу, к которой необходимо добавить указанный файл. Чтобы отобразить текущую файловую группу и узнать, какая файловая группа в данный момент установлена по умолчанию, используйте представление каталога **sys.filegroups**.

**ADD LOG FILE**

Добавляет файл журнала в указанную базу данных.

**REMOVE FILE logical\_file\_name**

Удаляет логическое описание файла из экземпляра SQL Server и физический файл. Файл не может быть удален, если он не пуст. **logical\_file\_name** **-** логическое имя, используемое в SQL Server при обращении к файлу.

**MODIFY FILE**

Указывает файл, который должен быть изменен. Одновременно может быть изменено только одно свойство файла. Предложение **NAME** всегда должно быть указано, чтобы определить, какой файл будет изменен. Если указано предложение **SIZE**, новый размер файла должен быть больше, чем текущий.

Чтобы изменить логическое имя файла данных или файла журнала, указывается логическое имя файла, который будет переименован, в предложении **NAME** и новое логическое имя для файла в предложении **NEWNAME**.

Например:

MODIFY FILE ( NAME = logical\_file\_name, NEWNAME = new\_logical\_name )

Чтобы переместить файл данных или файл журнала в новое расположение, указывается текущее логическое имя файла в предложении **NAME** и новый путь и имя файла в операционной системе в предложении **FILENAME**.

Например:

MODIFY FILE

( NAME = logical\_file\_name, FILENAME = ' new\_path/os\_file\_name ' )

**<add\_or\_modify\_filegroups>::=**

**ADD FILEGROUP filegroup\_name**

Добавляет файловую группу в базу данных.

**REMOVE FILEGROUP filegroup\_name**

Удаляет файловую группу из базы данных. Файловая группа не может быть удалена, пока она не пустая. Вначале удалите из файловой группы все файлы.

**MODIFY FILEGROUP filegroup\_name{ <filegroup\_updatability\_option> | DEFAULT | NAME = new\_filegroup\_name }**

Изменяет файловую группу, меняя ее состояние на READ\_ONLY или READ\_WRITE, делая ее файловой группой по умолчанию для базы данных или изменяя имя файловой группы.

**DEFAULT**

Изменяет файловую группу по умолчанию базы данных на аргумент **filegroup\_name**.

**NAME = new\_filegroup\_name**

Изменяет имя файловой группы на аргумент **new\_filegroup\_name**.

**<filegroup\_updatability\_option>** **-** устанавливает свойство «только для чтения» или «чтение и запись» для файловой группы.

* + **READ\_ONLY | READONLY**

Определяет, что файловая группа находится в состоянии только для чтения. Изменение ее объектов запрещено. Первичную файловую группу перевести в состояние только для чтения нельзя. Чтобы изменить это состояние, необходимо обладать монопольным доступом к базе данных.

* + **READ\_WRITE | READWRITE**

Определяет, что файловая группа находится в состоянии **READ\_WRITE**. Разрешено изменять объекты в файловой группе. Чтобы изменить это состояние, необходимо обладать монопольным доступом к базе данных. Ключевое слово READWRITE будет удалено в будущей версии Microsoft SQL Server. Вместо него следует используовать READ\_WRITE.

**<set\_database\_options >::=**

Одна из опций базы данных, приведенных выше, или их комбинация.

**Примеры изменения настроек базы данных:**

**Пример 1:**

В следующем примере к базе данных TestDatabase добавляется файл данных размером 5 МБ.

ALTER DATABASE TestDatabase

ADD FILE

(

NAME = NewDataFile,

FILENAME = 'C:\MSSQL\DATA\NewDataFile.ndf',

SIZE = 5MB,

MAXSIZE = 100MB,

FILEGROWTH = 5MB

)

**Пример 2:**

В следующем примере в базе данных TestDatabase создается файловая группа NewUserFG и добавляется два файла по 5 МБ в эту файловую группу.

ALTER DATABASE TestDatabase

ADD FILEGROUP NewUserFG;

GO

ALTER DATABASE TestDatabase

ADD FILE

(

NAME = NewDataFile2,

FILENAME = 'C:\MSSQL\DATA\NewDataFile2.ndf',

SIZE = 5MB,

MAXSIZE = 100MB,

FILEGROWTH = 5MB

),

(

NAME = NewDataFile3,

FILENAME = 'C:\MSSQL\DATA\NewDataFile3.ndf',

SIZE = 5MB,

MAXSIZE = 100MB,

FILEGROWTH = 5MB

)

TO FILEGROUP newUserFG;

**Пример 3:**

В следующем примере к базе данных TestDatabase добавляется файл журнала NewLogFile размером 5 МБ.

ALTER DATABASE TestDatabase

ADD LOG FILE

(

NAME = NewLogFile,

FILENAME = 'C:\MSSQL\DATA\NewLogFile.ldf',

SIZE = 5MB,

MAXSIZE = 100MB,

FILEGROWTH = 5MB

);

**Пример 4:**

В следующем примере из базы данных TestDatabase удаляется один из ранее добавленных файлов с логическим именем NewDataFile3. Прежде чем файл может быть удален, он должен быть совершенно пуст. Для переноса данных из одного файла данных в другие файлы той же файловой группы используется инструкция **DBCC SHRINKFILE** с указанием предложения **EMPTYFILE**.

-- перемещает данные из файла NewDataFile3

-- в другие файлы файловой группы newUserFG

DBCC SHRINKFILE (NewDataFile3, EMPTYFILE);

GO

-- удаляет пустой файл

ALTER DATABASE TestDatabase

REMOVE FILE NewDataFile3;

## Удаление базы данных

Удаление базы данных осуществляется командой **DROP DATABASE**.

DROP DATABASE database\_name

При удалении базы данных она удаляется из экземпляра SQL Server, а ее физические файлы удаляются с диска. Если база данных или один из ее файлов во время удаления находится в автономном режиме, файлы с диска не удаляются, эти файлы можно удалить вручную при помощи обозревателя Windows.

Удалить базу данных, которая используется в текущий момент времени, невозможно. Для отключения пользователей от базы данных можно использовать инструкцию ALTER DATABASE для перевода базы данных в режим SINGLE\_USER.

База данных может быть удалена независимо от ее состояния: в автономном режиме, доступна только для чтения, подозрительная и т. д. Для просмотра текущего состояния базы данных можно воспользоваться представлением каталога **sys.databases**.

Удаленная база данных может быть повторно создана только с помощью восстановления из резервной копии.

**Пример:**

В следующем примере удаляется ранее созданная база данных TestDatabase.

DROP DATABASE TestDatabase ;

**Лекция № 12**

## Типы данных Microsoft SQL Server

В Microsoft SQL Server у каждого столбца таблицы, локальной переменной, выражения и параметра есть определенный тип данных. Тип данных - это атрибут, определяющий, какого рода данные могут храниться в объекте: целые числа, символы, данные денежного типа, метки даты и времени, двоичные строки и так далее.

SQL Server предоставляет набор системных типов данных, определяющих все типы данных, которые могут использоваться в нем. Можно также определять собственные типы данных в Transact-SQL или Microsoft .NET Framework.

### Точные числа

**Типы точных числовых данных, использующие целые значения.**

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Диапазон** | **Хранилище** |
| **bigint** | от -2^63 (-9 223 372 036 854 775 808) до 2^63-1 (9 223 372 036 854 775 807) | 8 байт |
| **int** | от -2^31 (-2 147 483 648) до 2^31-1 (2 147 483 647) | 4 байта |
| **smallint** | от -2^15 (-32 768) до 2^15-1 (32 767) | 2 байта |
| **tinyint** | от 0 до 255 | 1 байт |

**Логические значения**

bit - целочисленный тип данных, который может принимать значения 1, 0 или NULL. Компонент SQL Server Database Engine оптимизирует хранение столбцов типа bit, объединяя значения отдельных битовых полей в байты, например, если в таблице имеется 8 или меньше столбцов типа bit, они хранятся как 1 байт. Если имеется от 9 до 16 столбцов типа bit, они хранятся как 2 байта и т.д.

Строковые значения TRUE и FALSE можно преобразовать в значения типа bit: TRUE преобразуется в 1, а FALSE - в 0.

**Десятичные и числовые**

Типы числовых данных с фиксированными точностью и масштабом.

**decimal**[ **(**p[ **,**s] **)**] и **numeric**[ **(**p[ **,**s] **)**]

Числа с фиксированной точностью и масштабом. При использовании максимальной точности числа могут принимать значения в диапазоне от -10^38+1 до 10^38-1. В стандарте ISO синонимом типа **decimal** являются типы **dec** и **dec(**p, s**)**. Тип **numeric** функционально равнозначен типу **decimal**.

**p (точность)**

Максимальное количество десятичных разрядов числа (как слева, так и справа от десятичной запятой). Точность должна принимать значение от 1 до 38. По умолчанию для точности принимается значение 18.

**s(масштаб)**

Максимальное количество десятичных разрядов числа справа от десятичной запятой. Масштаб может принимать значение от 0 до *p*. Масштаб может быть указан только совместно с точностью. По умолчанию масштаб принимает значение 0; поэтому 0 <= **s** <= **p**.

Максимальный размер хранилища зависит от точности.

|  |  |
| --- | --- |
| **Точность** | **Байты хранилища** |
| 1 - 9 | 5 |
| 10-19 | 9 |
| 20-28 | 13 |
| 29-38 | 17 |

**Типы данных, представляющие денежные (валютные) значения**

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Диапазон** | **Хранение** |
| **money** | От -922 337 203 685 477,5808 до 922 337 203 685 477,5807 | 8 байт |
| **smallmoney** | От -214 748,3648 до 214 748,3647 | 4 байта |

Типы данных **money** и **smallmoney** имеют точность до одной десятитысячной денежной единицы, которую они представляют.

### Приблизительные числа

Типы приблизительных числовых данных, используемые для хранения числовых данных с плавающей запятой. Данные с плавающей запятой являются приблизительными, поэтому не все значения из диапазона могут быть отображены точно.

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Диапазон** | **Объем памяти** |
| **float** | - 1,79E+308 — -2,23E-308, 0 и 2,23E-308 — 1,79E+308 | Зависит от значения n |
| **real** | - 3,40E + 38 — -1,18E - 38, 0 и 1,18E - 38 — 3,40E + 38 | 4 байта |

**float** [ **(**n**)** ]

Где n – это количество битов, используемых для хранения мантиссы числа в формате **float** при экспоненциальном представлении. Определяет точность данных и размер для хранения. Значение параметра n должно лежать в пределах от **1** до **53**. Значением по умолчанию для параметра n является **53**.

|  |  |  |
| --- | --- | --- |
| n | **Точность** | **Объем памяти** |
| **1–24** | 7 знаков | 4 байта |
| **25–53** | 15 знаков | 8 байт |

|  |
| --- |
|  |

### Дата и время

### В предыдущих версия Microsoft SQL Server существовали только два типа, обеспечивавших хранение дат и времени: **datetime и** smalldatetime.

### При разработке реальных приложений часто приходилось сталкиваться с практическими ограничениями этих типов. Поскольку оба типа предназначены для хранения даты и времени, при необходимости хранить только дату или только время приходилось оговаривать фиксированную часть. Например, если необходимо хранить только время, для даты можно выбрать фиксированное значение 01-01-2010. Недостатком такого подхода является усложнение логики приложения, поскольку при изменении значений каждый раз необходимо дату на соответствие 01-01-2010. В противном случае возможны некорректные сравнения, если не совпадает дата.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип данных | Диапазон дат | Диапазон времени | Объем памяти | Максимальная Точность |
| **datetime** | От 01.01.1753 до 31.12.9999 | От 00:00:00 до 23:59:590,997 | 8 байт | Округлено до приращения 0,000, 0,003 или 0,007 секунд |
| **smalldatetime** | От 01.01.1900 до 06.06.2079 | От 00:00:00 до 23:59:59 | 4 байта, фиксированный. | Одна минута |
| **date** | От 01.01.0001 до 31.12.9999 |  | 3 байта, фиксированный | Один день |
| **time** [(0-7)] |  | От 00:00:00 до 23:59:59.9999999 | 3-5 байт, зависит от точности | 100 наносекунд |
| **datetime2** [(0-7)] | От 01.01.0001 до 31.12.9999 | От 00:00:00 до 23:59:59.9999999 | 6 байт для 2 и менее цифр, 7 байт для 3 и 4 цифр. 8 байт. | 100 наносекунд |
| **Datetimeoffset** [(0-7)]  Учитывает часовой пояс | От 01.01.0001 до 31.12.9999 | От 00:00:00 до 23:59:59.9999999 | 10 байт, зависит от точности | 100 наносекунд |

### Типы данных **datetime и** smalldatetime, существовавшие в предыдущих версиях SQL Server, имели более низкую точность по сравнению с новыми типами, и не имели возможности учитывать часовой пояс.

### Символьные строки

**char** [ ( n ) ]

Символьные данные фиксированной длины, не в Юникоде, с длиной *n* байт. Значение *n* должно находиться в интервале от 1 до 8000. Размер хранения данных этого типа равен *n* байт. Синонимом по стандарту ISO для типа **char** является **character**.

**varchar** [ ( n | **max** ) ]

Символьные данные переменной длины, не в Юникоде. *n* может иметь значение от 1 до 8 000. **max** означает, что максимальный размер хранения равен 2^31-1 байт. Размер хранения равен фактической длине данных плюс два байта. Введенные данные могут иметь длину 0 символов. Синонимами по стандарту ISO для типа **varchar** являются типы **charvarying** или **charactervarying**.

**nchar** [ ( n ) ]

Символьные данные в Юникоде длиной в *n* символов. Аргумент *n* должен иметь значение от 1 до 4000. Размер хранилища вдвое больше *n* байт. Синонимами по стандарту ISO для типа **nchar** являются типы **national char** и **national character**.

**nvarchar** [ ( n | **max** ) ]

Символьные данные в Юникоде переменной длины. Аргумент *n* может принимать значение от 1 до 4 000. Аргумент **max** указывает, что максимальный размер хранилища равен 2^31-1 байт. Размер хранилища в байтах вдвое больше числа введенных символов + 2 байта. Введенные данные могут иметь длину в 0 символов. Синонимами по стандарту ISO для типа **nvarchar** являются типы **national char varying** и **national character varying**.

### Двоичные данные

**binary** [ ( n ) ]

Двоичные данные фиксированной длины размером в *n* байт, где *n* — значение от 1 до 8000. Размер хранения составляет *n* байт.

**varbinary** [ ( n | **max**) ]

Двоичные данные переменной длины. *n* могут иметь значение от 1 до 8000; **max** означает максимальную длину хранения, которая составляет 2^31-1 байт. Размер хранения — это фактическая длина введенных данных плюс 2 байта. Введенные данные могут иметь размер 0 символов. В ANSI SQL синонимом для **varbinary** является **binary varying**.

### Пространственные типы данных

Пространственные типы данных появились только в Microsoft SQL Server 2008. Эти типы данных представляют сведения о физическом расположении и форме геометрических объектов. Такими объектами могут быть точки или более сложные объекты, например страны, дороги, озера.

SQL Server поддерживает два пространственных типа данных:

* **geometry** (плоский) - представляет данные в эвклидовом пространстве (плоской системе координат).
* **geography** (геодезический) - позволяет сохранять эллипсоидальные (сферические) данные, такие как координаты широты и долготы GPS.

### Специальные типы данных

Типы специальных данных не подходят ни к одной из других категорий типов данных. В SQL Server присутствуют следующие особые типы данных: **hierarchyid**, **sql\_variant**, **sysname**, **table**, **rowversion, xml, uniqueidentifier, cursor**.

**Тип данных hierarchyid**

В Microsoft SQL Server 2008 был введен новый тип данных **hierarchyid,** который используется для представления положения в древовидной иерархии. Столбец типа **hierarchyid** не принимает древовидную структуру автоматически. Приложение должно создать и назначить значения **hierarchyid** таким образом, чтобы они отражали требуемые связи между строками.

Значения **hierarchyid** обладают следующими свойствами:

* Исключительная компактность  
  Среднее число бит, необходимое для представления узла в древовидной структуре с n узлами, зависит от среднего количества потомков у узла. Для представления узла в иерархии организации, насчитывающей 100 000 человек со средним уровнем ветвления 6, необходимо около 38 бит. Эта величина округляется до 40 бит (5 байт), которые необходимы для хранения.
* Поддержка произвольных вставок и удалений  
  В любой момент можно создать одноуровневый элемент, расположенный справа от заданного узла, слева от заданного узла или между любыми двумя другими одноуровневыми элементами. Свойство сравнения сохраняется, если произвольное число узлов вставляется в иерархию или удаляется из нее.
* Кодировка в типе данных **hierarchyid** ограничена 892 байтами. Следовательно, узлы, имеющие слишком много уровней, чтобы уместиться в 892 байта, не могут быть представлены типом данных **hierarchyid**.

**Тип данных sql\_variant**

Тип данных sql\_variant позволяет сохранять данные различных типов в одном столбце, параметре или переменной. В каждом экземпляре столбца типа **sql\_variant** хранятся значения и метаданные, описывающие эти значения (Базовый тип данных, Максимальный размер, Масштаб, Точность, Параметры сортировки).

**Тип данных sysname**

Тип данных **sysname** используется в столбцах таблицы, переменных и параметрах хранимых процедур, содержащих названия объектов. Тип данных **sysname** является псевдонимом типа **nvarchar(128)**, по умолчанию **sysname** имеет значение NOT NULL. В более ранних версиях SQL Server тип **sysname** фунционально эквивалентен **varchar(30)**.

**Тип данных table**

Тип данных **table** используется для определения временных таблиц. В переменных этого типа сохраняются результирующие наборы для последующей обработки. Этот тип данных может использоваться только для определения локальных переменных типа **table** и значений, возвращаемых пользовательской функцией.

Определение табличной переменной или возвращаемого значения включает определения столбцов, типа данных, точности и размера каждого столбца. Определенная пользователем таблица не может быть использована в качестве пользовательского типа данных.

**Тип данных rowversion**

Значения **rowversion** являются двоичными числами, указывающими относительную последовательность, в которой происходило изменение данных в базе данных **-** значение **rowversion** изменяется всякий раз при изменении строки. Размер при хранении составляет 8 байт. Тип данных **rowversion** не связан со временем или с датой.

**Тип данных xml**

Тип данных **xml** позволяет хранить XML-документы и их фрагменты в базе данных SQL Server. Database Engine позволяет создавать столбцы и переменные типа xml, в которых можно хранить экземпляры XML. При этом размер хранимого экземпляра типа данных **xml** не должен превышать 2 Гб. SQL Server поддерживает валидацию значений столбцов и переменных типа **xml** при помощи XML-схем, а также преобразование реляционных данных в **xml** и обратно.

**Тип данных uniqueidentifier**

Данный тип представляет собой 16-байтный GUID - уникальный идентификатор, генерируемый на основе MAC-адреса сетевой карты и текущего времени. Его главная особенность - уникальность, и хотя уникальность каждого отдельного GUID не гарантируется, общее количество уникальных ключей настолько велико, что вероятность того, что в будут независимо сгенерированы два совпадающих ключа, достаточно мала.

**Тип данных cursor**

Тип данных **cursor** применяется для итеративной обработки строк, когда необходимо обрабатывать строки из некоторого набора по одной. **cursor** создается на основе запросы получающего набор строк и может использоваться только для объявления переменных или выходных параметров хранимых процедур. Необходимо учитывать, что курсор является самым медленным способом обработки данных, и в должен использоваться только в тех случаях когда действительно требуется обработка каждой строки по отдельности. На практике наиболее часто **cursor** применяется для вызова хранимой процедуры для каждой строки и определенного набора.

### Устаревшие типы данных

Типы данных **ntext**, **text**, **image** и **timestamp** в будущей версии Microsoft SQL Server будут удалены. Следует избегать их использования при разработке новых приложений и запланировать изменение приложений, в которых эти типы в данное время используются. Вместо этих типов данных следует использовать типы **nvarchar(max)**, **varchar(max)**, **varbinary(max)** и **rowversion** соответственно.

**ntext**

Этот тип данных представляет символьные данные в Юникоде переменной длины, включающие до 2^30 – 1 (1 073 741 823) символов. Объем занимаемого этим типом пространства (в байтах) в два раза превышает число символов.

**text**

Этот тип данных представляет данные, отличные от данных Юникод, представленные с использованием кодовой страницы сервера. Максимальная длина данных — 2^31 – 1 (2 147 483 647) символов. Если в кодовой странице сервера используются двухбайтовые символы, объем занимаемого типом пространства все равно не превышает 2 147 483 647 байт.

**image**

Этот тип представляет двоичные данные переменной длины, включающие от 0 до 2^31 – 1 (2 147 483 647) байт.

**timestamp**

Тип данных **timestamp** является синонимом типа данных **rowversion**.

## Пользовательские типы данных

Пользовательские типы данных, также именуемые как псевдонимы типов данных, позволяют расширить базовые типы данных SQL Server (например, **varcha**r) содержательным именем и форматом, который можно приспособить для конкретного использования.

Создание пользовательского типа данных осуществляется вызовом команды **CREATE TYPE**:

CREATE TYPE type\_name

{

FROM base\_type

[ ( precision [ , scale ] ) ]

[ NULL | NOT NULL ]

}

type\_name

Имя псевдонима типа данных или определяемого пользователем типа данных.

base\_type

Предоставляемый SQL Server тип данных, на основе которого формируется псевдоним.

precision

Для типа **decimal** или **numeric** является неотрицательным целым числом, которое указывает на максимальное общее число подлежащих сохранению десятичных знаков как слева, так и справа от десятичного разделителя, отделяющего десятичную дробь от целого числа.

scale

Для типа **decimal** или **numeric** является неотрицательным целым числом, которое указывает на максимальное общее число подлежащих сохранению десятичных знаков справа от разделителя, отделяющего десятичную дробь от целого числа. Значение должно быть меньше или равно заданной степени точности.

**NULL | NOT NULL**

Указывает, может ли данный тип иметь значение **NULL**. Если не указано иное, по умолчанию принимается значение **NULL**.

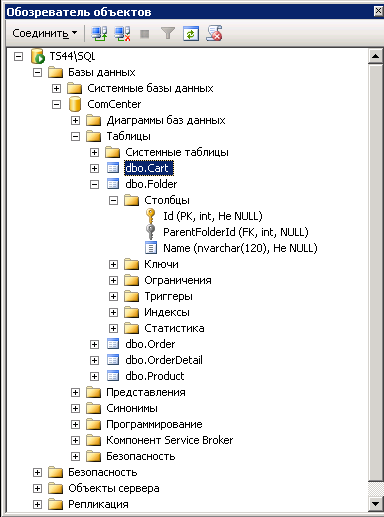
**Пример:**

Следующая инструкция создает определяемый пользователем тип данных **birthday**, основанный на типе данных **datetime**, допускающий значения **NULL**.

CREATE TYPE birthday FROM datetime NULL ;

Источники информации о таблицах

### Обозреватель объектов SQL Server Management Studio



### Представления каталога

|  |  |
| --- | --- |
| **Представление каталога** | **Отображение сведений о** |
| **sys.tables** | Информация обо всех таблицах в базе данных |
| **sys.columns** | Информация о столбцах таблиц, типе данных столбцов |
| **sys.types** | Информация о системных и пользовательских типах данных |

## Создание таблиц

SQL Server 2008 поддерживает в базе данных до двух миллиардов таблиц.

Таблица состоит из набора столбцов, с каждым из которых соотнесен тип данных, допустимый в указанном столбце и дополнительные ограничения с ним связанные. Таблицы могут содержать не более 1024 столбцов. Число строк и общий размер таблицы ограничиваются только доступным пространством для хранения.

CREATE TABLE

    [ database\_name . [ schema\_name ] . | schema\_name . ] table\_name

        ( { <column\_definition>

| <computed\_column\_definition>

        } )

[ ON { filegroup | "default" } ]

<column\_definition> ::=

column\_name <data\_type>

    [ COLLATE collation\_name ]

    [ NULL | NOT NULL ]

    [ IDENTITY ]

<data type> ::=

[ type\_schema\_name . ] type\_name

[ ( precision [ , scale ] | max ]

<computed\_column\_definition> ::=

column\_name AS computed\_column\_expression

**database\_name**

Имя базы данных, в которой создается таблица. В качестве аргумента **database\_name** должно быть указано имя существующей базы данных. Если аргумент **database\_name** не указан, таблица создается в текущей базе данных.

**schema\_name**

Имя схемы, которой принадлежит новая таблица.

**table\_name**

Имя новой таблицы. Аргумент **table\_name** может состоять не более чем из 128 символов, за исключением имен локальных временных таблиц (имена с префиксом номера #), длина которых не должна превышать 116 символов.

**column\_name**

Имя столбца в таблице. Имена столбцов должны соответствовать правилам для идентификаторов и быть уникальными в данной таблице. Аргумент **column\_name** может содержать от 1 до 128 символов.

**computed\_column\_expression**

Выражение, определяющее значение вычисляемого столбца. Вычисляемый столбец представляет собой виртуальный столбец, физически не хранящийся в таблице, если для него не установлен признак **PERSISTED**. Значение столбца вычисляется на основе выражения, использующего другие столбцы той же таблицы. Например, определение вычисляемого столбца может быть следующим: **cost** AS **price** \* **qty**. Выражение может быть именем невычисляемого столбца, константой, функцией, переменной или любой их комбинацией, соединенной одним или несколькими операторами. Выражение не может быть вложенным запросом или содержать псевдонимы типов данных.

Компонент Database Engine автоматически определяет для вычисляемых столбцов допустимость значений NULL на основе использованных выражений. Считается, что результат большинства выражений допускает значение NULL, даже если используются только столбцы, для которых значение NULL запрещено, так как при возможном переполнении или потере точности может формироваться значение NULL.

**ON { filegroup | "default" }**

Указывает файловую группу, в которой хранится таблица. Если указан аргумент **filegroup**, таблица сохраняется в файловой группе с таким именем. Это должна быть существующая файловая группа в базе данных. Если указано значение **"default"** или параметр ON не определен вообще, таблица сохраняется в файловой группе по умолчанию. Механизм хранения таблицы, указанный в инструкции CREATE TABLE, изменить в дальнейшем невозможно.

**[ type\_schema\_name. ] type\_name**

Указывает тип данных столбца и схему, к которой он принадлежит. Тип данных может быть одним из следующих.

* Системный тип данных.
* Пользовательский тип данных на основе системного типа данных SQL Server. Состояние признака NULL или NOT NULL для псевдонима типа данных может быть переопределено с помощью инструкции CREATE TABLE. Однако его длину изменить нельзя; длина типа данных-псевдонима не определяется инструкцией CREATE TABLE.

**precision**

Точность указанного типа данных.

**scale**

Масштаб указанного типа данных.

**max**

Применяется только к типам данных **varchar**, **nvarchar** и **varbinary** для хранения 2^31 байт символьных и двоичных данных или 2^30 байт данных в Юникоде.

**IDENTITY**

Указывает, что новый столбец является столбцом идентификаторов. При добавлении в таблицу новой строки компонент Database Engine формирует для этого столбца уникальное последовательное значение. Свойство IDENTITY может быть установлено для столбцов типа **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)** или **numeric(p,0)**. Для каждой таблицы можно создать только один столбец идентификаторов. Необходимо указать оба параметра - начальное значение и приращение, или же не указывать ничего. Если ничего не указано, применяется значение по умолчанию (1,1).

* **seed -** значение, используемое для самой первой строки, загружаемой в таблицу.
* **increment -** значение приращения, добавляемое к значению идентификатора предыдущей загруженной строки.

**COLLATE collation\_name**

Задает параметры сортировки для столбца. Имя параметров сортировки может быть либо именем параметров сортировки Windows, либо именем параметров сортировки SQL. Аргумент **collation\_name** применим только к столбцам типов данных **char**, **varchar**, **text**, **nchar**, **nvarchar** и **ntext**. Если этот аргумент не указан, столбцу назначаются либо параметры сортировки определяемого пользователем типа, если столбец принадлежит к определяемому пользователем типу данных, либо установленные по умолчанию параметры сортировки для базы данных.

**NULL | NOT NULL**

Определяет, допустимы ли для столбца значения NULL. Параметр NULL не является ограничением в строгом смысле слова, но может быть указан так же, как и NOT NULL. Ограничение NOT NULL может быть указано для вычисляемых столбцов только в случае, если одновременно указан параметр PERSISTED.

**Пример:**

Следующий пример демонстрирует создание таблицы, содержащей пять столбцов, три из которых не допускают значения NULL и физически хранятся в базе данных.

Столбец **MiddleName** допускает пустые значения и физически хранится в базе данных, а столбец **FullName** является вычисляемым и не хранится в базе данных.

Столбец **Id** является столбцом идентификаторов. При добавлении в таблицу новой строки компонент Database Engine формирует для этого столбца уникальное последовательное значение, начиная с 1 и последовательно увеличивая значение на 1 для каждой новой строки.

CREATE TABLE Human

(

Id int IDENTITY(1,1) NOT NULL,

FirstName nvarchar(50) NOT NULL,

LastName nvarchar(50) NOT NULL,

MiddleName nvarchar(50) NULL ,

FullName AS FirstName + ' ' + LastName

) ON [PRIMARY];

## Изменение таблиц

Изменение определения таблицы путем изменения, добавления или удаления столбцов и осуществляется при помощи команды ALTER TABLE.

ALTER TABLE [ database\_name . [ schema\_name ] . | schema\_name . ] table\_name

{

ALTER COLUMN column\_name

{

[ type\_schema\_name. ] type\_name [ ( { precision [ , scale ] | max } ) ]

[ COLLATE collation\_name ]

[ NULL | NOT NULL ]

}

    | ADD

   {

<column\_definition>

   | <computed\_column\_definition>

   } [ ,...n ]

    | DROP

   {

COLUMN column\_name

}

}

database\_name

Имя базы данных, в которой создана таблица.

schema\_name

Имя схемы, которой принадлежит таблица.

table\_name

Имя таблицы, подлежащей изменению. Если таблицы нет в текущей базе данных или она не содержится в схеме, которой владеет текущий пользователь, то и база данных, и схема должны быть явно указаны.

**ALTER COLUMN**

Указывает, что именованный столбец подлежит изменению.

column\_name

Имя столбца, подлежащего изменению, добавлению или удалению. Аргумент column\_name может содержать максимум 128 символов. Для новых столбцов аргумент column\_name может быть опущен, если столбцы создаются с типом данных **timestamp**. Имя **timestamp** используется, если column\_name не указано для столбца типа данных **timestamp**.

**[**type\_schema\_name**.]**type\_name

Новый тип данных для измененного столбца или тип данных для добавленного столбца. Тип данных для существующих столбцов секционированных таблиц изменить невозможно. Аргумент type\_name может быть одним из следующих.

* Системным типом данных SQL Server.
* Псевдонимом типа данных, основанным на системном типе данных SQL Server. Прежде чем псевдонимы типов данных можно будет использовать в определении таблицы, их нужно создать с помощью инструкции CREATE TYPE.
* Определяемым пользователем типом .NET Framework и схемой, к которой он принадлежит. Перед использованием в определении таблицы определяемые пользователем типы .NET Framework создаются при помощи инструкции CREATE TYPE.

type\_name изменяемого столбца должен отвечать следующим условиям:

* Предыдущие типы данных должны быть неявно преобразуемыми в новый тип данных.
* type\_nameне может иметь тип **timestamp**.
* По умолчанию для аргумента ANSI\_NULL инструкции ALTER COLUMN всегда установлено значение ON; если не указано иное, столбец может содержать значения NULL.
* Аргумент заполнения ANSI\_PADDING для инструкции ALTER COLUMN всегда принимает значение ON.
* Если изменяемый столбец является столбцом идентификаторов, то аргумент new\_data\_type должен иметь тип данных, который поддерживает свойство идентификатора.
* Текущая установка для аргумента SET ARITHABORT пропускается. Инструкция ALTER TABLE функционирует аналогично случаю, когда для аргумента ARITHABORT установлено значение ON.

precision

Точность указанного типа данных.

scale

Масштаб указанного типа данных.

**max**

Применим только к типам данных **varchar**, **nvarchar** и **varbinary** для хранения 2^31-1 байт символов, двоичных данных и данных Юникода.

Если в столбце имеются данные, новый размер не может быть меньше максимального размера данных.

**COLLATE <** collation\_name **>**

Задает новые параметры сортировки для изменяемого столбца. Если не указано, столбцу назначаются параметры сортировки, принятые в базе данных по умолчанию. Именем параметров сортировки может быть либо имя параметров сортировки Windows, либо имя параметров сортировки SQL.

Предложение COLLATE может быть использовано для изменения параметров сортировки только для столбцов типов **char**, **varchar**, **nchar** и **nvarchar**. Чтобы изменить параметры сортировки столбца пользовательского псевдонима типа данных, необходимо выполнить отдельные инструкции ALTER TABLE, чтобы изменить столбец на системный тип данных SQL Server, изменить параметры сортировки, а затем снова перевести столбец в псевдоним типа данных.

Инструкция ALTER COLUMN не может изменить параметры сортировки, если выполняется одно или несколько из следующих условий:

* Если на изменяемый столбец ссылается ограничение CHECK, ограничение FOREIGN KEY или вычисляемые столбцы.
* Если на базе столбца создан какой-нибудь индекс, статистика или полнотекстовый индекс. Статистика, автоматически созданная на базе изменяемого столбца, удаляется, если изменяются параметры сортировки столбца.
* Если связанное со схемой представление или функция ссылаются на столбец.

**NULL | NOT NULL**

Указывает, может ли столбец принимать значение NULL.

Столбцы, не допускающие значения NULL, могут быть добавлены инструкцией ALTER TABLE только в том случае, если для них указаны значения по умолчанию или если таблица пуста. Ограничение NOT NULL может быть указано для вычисляемых столбцов только в случае, если одновременно указан параметр PERSISTED. Если новый столбец допускает значения NULL, а значение по умолчанию не задано, то новый столбец содержит значение NULL для каждой строки в таблице. Если новый столбец допускает значение NULL, а определение по умолчанию добавляется с новым столбцом, то инструкция WITH VALUES может использоваться для хранения значений по умолчанию в новом столбце для каждой существующей строки в таблице.

Если новый столбец не допускает значение NULL и таблица не пуста, то определение DEFAULT должно быть добавлено с новым столбцом. Новый столбец автоматически загружается со значениями по умолчанию в каждой существующей строке нового столбца.

Значение NULL может указываться в инструкции ALTER COLUMN, чтобы принудить столбец NOT NULL допускать значения NULL. Значение NOT NULL может быть указано в инструкции ALTER COLUMN, только если столбец не содержит значения NULL.

Значения NULL следует обновить, присвоив некоторые значения, прежде чем разрешить инструкцию ALTER COLUMN NOT NULL, например:

UPDATE MyTable SET NullCol = N'some\_value' WHERE NullCol IS NULL

ALTER TABLE MyTable ALTER COLUMN NullCOl NVARCHAR(20) NOT NULL

При создании или изменении таблицы инструкцией CREATE TABLE или ALTER TABLE установки базы данных и сеанса изменяются и, возможно, переопределяют разрешение содержать значение NULL для типа данных, используемого в определении столбца. Рекомендуется всегда явно определять невычисляемые столбцы как NULL или NOT NULL.

Если добавляется столбец определяемого пользователем типа данных, то рекомендуется определить для этого столбца то же состояние допустимости значений NULL, что и в определяемом пользователем типе данных и задать для него значение по умолчанию.

**ADD**

Указывает, что добавляется одно или несколько определений столбца или определений вычисляемого столбца.

**DROP { COLUMN** column\_name **}**

Указывает, что из таблицы должен быть удален столбец column\_name. Может быть перечислено несколько столбцов.

**Пример 1:**

Следующий пример показывает добавление столбца, который допускает значения NULL. Если в таблицу уже были добавлены данные, то в новом столбце в каждой строке будет установлено значение **NULL**.

ALTER TABLE simpleTable ADD column\_b VARCHAR(20) NULL ;

**Пример 2:**

В следующем примере таблица изменяется путем удаления столбца.

ALTER TABLE simpleTable DROP COLUMN column\_b ;

**Пример 3:**

Следующий пример меняет столбец таблицы с **INT** на **DECIMAL**.

CREATE TABLE simpleTable (column\_a INT ) ;

GO

INSERT INTO simpleTable (column\_a) VALUES (10) ;

GO

ALTER TABLE simpleTable ALTER COLUMN column\_a DECIMAL (5, 2) ;

## Удаление таблиц

Удаление таблиц производится командой DROP TABLE. Любое представление или хранимая процедура, ссылающаяся на удаленную таблицу, должна быть явно удалена с помощью инструкции DROP. Получить отчет о зависимостях в таблице можно с помощью **sys.dm\_sql\_referencing\_entities**.

DROP TABLE [ database\_name . [ schema\_name ] . | schema\_name . ]

table\_name [ ,...n ]

database\_name

Имя базы данных, в которой создана таблица.

schema\_name

Имя схемы, которой принадлежит таблица.

table\_name

Имя таблицы, предназначенной для удаления.

Инструкцию DROP TABLE нельзя использовать для удаления таблицы, на которую ссылается ограничение FOREIGN KEY. Сначала следует удалить ссылающееся ограничение FOREIGN KEY или ссылающуюся таблицу. Если и ссылающаяся таблица, и таблица, содержащая первичный ключ, удаляются с помощью одной инструкции DROP TABLE, ссылающаяся таблица должна быть первой в списке.

Если удаляемая таблица ссылается на первичный ключ другой таблицы, которая также удаляется, ссылающаяся таблица с внешним ключом должна стоять в списке перед таблицей, содержащей указанный первичный ключ.

При удалении таблицы относящиеся к ней правила и значения по умолчанию теряют привязку, а любые связанные с таблицей ограничения или триггеры автоматически удаляются. Если таблица будет создана заново, нужно будет заново привязать все правила и значения по умолчанию, заново создать триггеры и добавить необходимые ограничения.

Большие таблицы и индексы из более чем 128 экстентов удаляются в два этапа: логический и физический. На логическом этапе существующие единицы распределения, используемые в таблице, отмечаются для освобождения и остаются заблокированными до фиксации транзакции. На физическом этапе страницы IAM, отмеченные для освобождения, физически удаляются пакетами.

**Пример:**

Следующий пример удаляет таблицу ProductVendor из текущей базы данных.

DROP TABLE simpleTable;

**Лекция № 13**

**Операторы манипулирования данными**

**Выборка данных**

**Общее описание оператора SELECT**

Выборка данных осуществляется с помощью оператора SELECT:

select [all | distinct] <список полей SELECT>

[into [[<имя БД>.]<владелец>.]<имя таблицы>

[from [[<имя БД>.]<владелец>.]{<имя таблицы> | <имя представления>}

[holdlock | noholdlock][shared]

[,[[<имя БД>.]<владелец>.]{<имя таблицы | <имя представления>}

[holdlock | noholdlock][shared]]...]

[where <условие поиска>]

[group by [all]<aggregate\_free\_expression>

[,<aggregate\_free\_expression]...]

[having <условие поиска>]

[order by {[[[<имя БД>.]<владелец>.]{<имя табл.>. | <имя представления>.}]

<имя колонки> | <номер в списке SELECT> | <выражение>}

[asc | desc]

[,{[[[<имя БД>.<владелец>.]{<имя таблицы>. | <имя представления>.}]

<имя колонки> | <номер в списке SELECT> | <выражение>}

[asc|desc]]...]

[compute row\_aggregate (<имя колонки>)

[,row\_aggregate (<имя колонки>)]...

[by <имя колонки> [,<имя колонки>]...]]

[for browse]

**Список выборки** - это перечень колонок, определенных в главной фразе предложения SELECT. Список выборки представляет собой один или более разделенных запятыми элементов данных. Элемент данных может быть именем колонки, константой или комбинацией имен колонок и констант, связанных арифметическими операторами: + (сложить), - (вычесть), \* (умножить), / (разделить), % (остаток от деления). Эти арифметические операторы используются для связки данных только числового типа. Кроме этого, элементу в списке выборки может предшествовать одна из агрегатных функций, описанных ниже.

Если вместо списка выборки ставится символ \*, то это означает, что данные выбираются по всем колонкам таблицы.

При выдаче отчета по запросу, по умолчанию, заголовком колонки является имя колонки. Но у пользователя есть возможность изменить заголовок колонки, задав в качестве элемента данных следующее:

<заголовок колонки> = <имя колонки>

или

<имя колонки> <заголовок колонки>

Если в <заголовок колонки> входят специальные символы, то он заключается в кавычки.

**Примеры:**

1. select Publisher = pub\_name, pub\_id from publishers
2. select pub\_name Publisher, pub\_id from publishers
3. select “Publisher’s Name” = pub\_name from publishers
4. select title\_id, total\_sales \* price - advance / 2 from titles

Перед списком выборки могут задаваться ключевые слова all или distinct. All определяет, что должны быть выбраны все повторяющиеся значения (all предполагается по умолчанию). Distinct определяет, что повторяющиеся значения не должны выбираться, т. е. исключаются дубликаты.

**Структура into** используется для создания новых таблиц БД путем выполнения запроса в уже существующих таблицах БД. Например:

select pub\_id, pub\_name into newtable from publishers

Предложение select…into не только создает новую таблицу, но и загружает ее данными.

**Структура from** должна включать в себя список всех таблиц и представлений, содержащих колонки, которые перечислены в списке выборки и структуре where. Максимальное количество таблиц и представлений в структуре from равно 16. Ключевые слова holdlock, noholdlock и shared используются системным администратором и здесь не рассматриваются.

**Условие выборки указывается во фразах where и having.**

В условие выборки включается следующее:

1) Операции сравнения: =, <, >, ! =, < >, >=, <=, !>, !<.

2) Диапазоны (between и not between).

3) Списки (in, not in).

4) Символы сравнения (like, not like).

5) Неопределенные значения (is null, is not null).

6) Комбинации из логических операторов: and, or, not.

7) Условия соединения таблиц.

8) Подзапросы.

**Операции сравнения** используются следующим образом:

where <выражение1> <операция сравнения> <выражение2>

где <выражение> может быть константой, именем колонки, функцией, подзапросом и любой комбинацией из них, соединенных арифметическими и поразрядными операциями. Например:

select \* from titleauthor where royaltyper < 50

**Функция between** дает возможность определить условие, если искомое значение находится в заданных границах (диапазоне). Функция between имеет следующий формат:

<выражение1> [not] between <выражение2> and <выражение3>

Условие считается выполненным, если <выражение1> равно или больше, чем <выражение2> и равно или меньше, чем <выражение3>. Если используется опция not, то условие считается выполненным, если <выражение1> меньше <выражения2> или больше <выражения3>. Например:

select title\_id, total\_sales from titles where total\_sales between 4095 and 12000

**Функция in** предоставляет возможность эффективного сравнения значения выражения со списком значений данных. Функция in имеет следующий формат:

<выражение> [not] in (<список значений>)

Условие удовлетворяется, если <выражение> равно одному из значений, указанных в <списке значений>. Если используется опция not, то не равно ни одному из значений из <списка>. Например:

select au\_name, state from authors where state in (‘CA’, ‘IN’, ‘MD’)

Список значений может быть сформирован посредством обращения к подзапросу, рассматриваемому ниже.

**Функция like** предоставляет возможность поиска имеющих вид символьной строки значений данных, которые включают заданную строку в качестве подстроки. Функция like имеет следующий формат:

<имя колонки> [not] like <символьная строка>

Колонка должна иметь тип символьной строки, а <символьная строка> представляет собой любую символьную комбинацию. Кроме того, <символьная строка> может включать специальные символы:

1. символ “%” - представляет любую строку, состоящую из 0 или более символов;
2. символ “\_” (подчеркивания) - представляет любой одиночный символ;
3. [<описатель>] - определяет диапазон или множество символов, в которые должен входить одиночный символ;
4. [^<описатель>] - определяет диапазон или множество символов, в которые не должен входить одиночный символ.

То есть <описатель> задается двумя способами:

а) в виде диапазона: r1-r2, например: [a-f]

б) в виде множества: r1r2…, например: [abcdef]

Условие not like является истинным, если ложно соответствующее условие like.

Примеры:

1. like ‘M%’ - строка начинается на М;
2. like ‘%er’ - строка заканчивается на er;
3. like ‘%en%’ - строка имеет подстроку en в любом месте;
4. like ‘\_\_\_ryl’ - строка имеет в длину 6 символов и заканчивается на ryl;
5. like ‘[CK]ars[eo]n’ - строка имеет в длину 6 символов, начинается либо с С, либо с K, а пятый символ - либо e, либо o;
6. like ‘[M-Z]ing’ - 4хсимвольная строка, заканчивается на ing, а начинается с буквы, принадлежащей диапазону от M до Z;
7. like ‘M[^C]%’ - строка начинается на M, а в качестве второй буквы (цифры) берется любой символ, отличный от C.

**Функция null** обеспечивает способ просмотра в таблице неопределенных значений или, напротив, исключения null значений из результата запроса. Функция null имеет следующий формат:

<имя колонки> is [not] null

**В структуре where** отдельные условия могут соединяться **логическими операторами and, or и not.** Операция and(и) используется для определения двух и более условий, которые одновременно удовлетворяются в отбираемых строках, например:

select \* from quotations where qonorder > 0 and suppno = 54

Оператор or(или) используется для определения двух и более условий, которое обеспечивает отбор строк, удовлетворяющих по крайней мере одному из этих условий, например:

select \* from authors where au\_fname = ‘Anne’ or au\_fname = ‘Ann’

Оператор not используется для определения отрицания условия, например:

select \* from authors where not state = ‘CA’

Для группирования условий могут быть использованы скобки, например:

select \* from quotations where suppno = 61 and (purtno = 221 or purtno = 222)

**Структура group by** определяет группы совпадающих значений в указанных колонках. Максимальное количество колонок (выражений) равно 16. В ответе на запрос возвращается только одна результирующая строка для каждой группы. Почти всегда в структуре group by используются агрегатные функции. Например, **функция count(\*)** оценивает количество строк в группе. Если какие-либо строки в группируемой колонке содержат null значения, то каждая такая строка рассматривается как принадлежащая отдельной группе, содержащей одну строку.

Пример:

select nazvanie\_goroda from vuz\_gorod where cod\_stran = ‘01’ group by nazvanie\_goroda

**Структура having** определяет одно или более условий, накладываемых на группы, т. е. заставляет СУБД возвращать результат только для тех групп, которые удовлетворяют заданному <условию поиска>. Having - это то же самое для структуры group by, что и where для структуры select.

Пример:

select nomer from poss group by nomer having count(\*) > 1

**Сортировка значений полей** по возрастанию или убыванию осуществляется с помощью **структуры order by**, в которой перечисляются через запятую либо названия колонок, либо номера этих колонок в списке полей SELECT. Опция asc определяет сортировку по возрастанию, а опция desc - по убыванию. Если опция не определена, то предполагается упорядочение по возрастанию.

Пример:

select cod\_vuza,nazvanie\_vuza from vuz\_gorod where cod\_stran = ‘01’ order by 1

**Структура compute** используется с агрегатными функциями для получения в отчетах дополнительных записей, содержащих итоговые значения по каждой группе значений полей. С помощью структур compute можно подсчитать итоговые значения и для подгрупп, а также можно задавать несколько структур compute для одной группы. В структуре compute используются следующие агрегатные функции: sum, avg, min, max, count (sum и avg используются только для числовых колонок). Элемент данных, следующий за одной из этих функций, должен быть заключен в круглые скобки.

**Агрегатные функции**

**Функция avg** вычисляет среднее среди отобранных значений элемента. Эта функция предназначена только для числовых колонок и может быть использована с ключевым словом distinct. При вычислении среднего null значения игнорируются.

**Функция sum** вычисляет сумму отобранных значений элемента. Эта функция также предназначена только для числовых колонок, может быть использована с ключевым словом distinct и null значения игнорируются.

**Функция max** находит наибольшее среди отобранных значений элемента. Эта функция может быть применена к колонке любого типа и null значения игнорируются.

**Функция min** находит наименьшее среди отобранных значений элемента и также может быть применена к колонке любого типа, null значения игнорируются.

**Функция count** используется одним из двух способов:

* count(distinct <имя колонки>) возвращает число, равное количеству отличных друг от друга строк, удовлетворяющих условию поиска;
* count(\*) возвращает число, равное количеству строк, удовлетворяющих условию поиска.

**Правила использования структуры compute:**

1. в агрегатных функциях нельзя использовать distinct;
2. поля в структуре compute должны обязательно присутствовать в списке выборки;
3. в операторе select, использующего compute, нельзя применять структуру into;
4. при использовании compute by обязательно присутствие структуры order by, при этом список полей в compute by либо идентичен списку полей в order by, либо является его подмножеством с сохранением следования полей слева направо, начинающегося с одного и того же выражения и без пропусков каких-либо выражений.

Пример: если указывается order by a, b, c , то можно использовать:

compute <агр. функция> (<имя колонки>) by a, b, c

compute <агр. функция> (<имя колонки>) by a, b

compute <агр. функция> (<имя колонки>) by a

и нельзя использовать:

compute <агр. функция> (<имя колонки>) by b,c

compute <агр. функция> (<имя колонки>) by a, c

compute <агр. функция> (<имя колонки>) by c

1. для подсчета общих итогов используется compute без by.

Примеры:

1. если в compute после by указывается больше одного поля, то группа значений разбивается на подгруппы и агрегатная функция подсчитывается на нижнем уровне группирования:

select type, pub\_id, price from titles

order by type, pub\_id, price

compute sum(price) by type, pub\_id

1. если агрегатную функцию нужно подсчитать на каждом уровне группирования, то необходимо использовать compute больше одного раза:

select type, pub\_id, price from titles

order by type, pub\_id, price

compute sum(price) by type, pub\_id

compute sum(price) by type

Кроме того, в структуре compute можно указывать одну и ту же агрегатную функцию для нескольких колонок, а также разные агрегатные функции для различных колонок.

1. если необходимо подсчитать только общие итоги, которые печатаются в конце отчета, то надо использовать compute без by:

select type, price,advance from titles

where price > $20

compute sum(price), sum(advance)

**Оператор union**

**Оператор union** служит для объединения двух и более запросов в один, для которого

можно использовать структуры order by и compute. **Синтаксис оператора:**

<запрос1>

[union [all] <запрос N>]…

[<структура order by>]

[<структура compute>]

где <запрос1>: select <список выборки>

[<структура into>]

[<структура from]

[<структура where>]

[<структура group by>]

[<структура having>]

а <запрос N>: select <список выборки>

[<структура from]

[<структура where>]

[<структура group by>]

[<структура having>]

Пример:

select \* from t1 union select \* from t2

Оператор union, по умолчанию, уничтожает дублируемые записи из результата запросов. Если используется опция all, то все записи включаются в отчет.

**Правила использования оператора union:**

1. все списки выборки в операторе union должны иметь одинаковую структуру (одинаковое количество однотипных выражений);
2. заголовки колонок в отчете по объединенному запросу берутся из первого запроса;
3. структуру into можно использовать только в первом запросе;
4. структуры order by и compute можно указывать только после последнего запроса для сортировки и подсчета итогов в объединенном запросе;
5. структуры group by и having можно использовать только в индивидуальных запросах;
6. оператор union можно использовать в операторе insert, например:

insert into tour

select city, state from stores

union

select city, state from authors

1. нельзя использовать union в операторе create view;
2. нельзя использовать опцию browse в предложениях select оператора union.

**Встроенные функции**

**Встроенные функции** используются в списке выборки, в структуре where и в различных выражениях. Встроенные функции можно разделить на классы:

**- системные функции**, которые применяются к системным таблицам БД;

**- строковые функции**, которые применяются к значениям следующих типов: char, nchar, varchar, nvarchar, binary, varbinary;

**- текстовые функции**, которые применяются к значениям типа text и image;

**- математические функции** (тригонометрические, геометрические и др.);

**- функции даты**, которые применяются к значениям типа datetime и smalldatetime;

**- функции преобразования одних типов данных в другие и форматирования дат.**

**Системные функции**

Системные функции обеспечивают наиболее быстрый способ обращения к системным таблицам. Общий синтаксис вызова системной функции:

select <имя функции> (<аргумент[ы]>)

Таблица 1

|  |  |  |
| --- | --- | --- |
| Функция | Аргумент(ы) | Результат |
| 1) col\_name | ( <ид. объекта>,  <ид. колонки>) | имя колонки |
| 2) col\_length | (“<имя объекта>”,  “<имя колонки>”) | длина колонки в таблице |
| 3) data\_pgs | (<ид. объекта>,  {doampg | ioampg}) | количество страниц, занимаемое таблицей или индексом (не включаются страницы, используемые внутренними структурами) |
| 4) datalength | (<выражение>) | длина выражения в байтах |
| 5) db\_id | (“<имя БД>”) | номер идентификатора БД |
| 6) db\_name | (<номер ид. БД>) | имя БД |
| 7) host\_id | () | номер идентификатора главного процесса |
| 8) host\_name | () | имя текущей главной ЭВМ |
| 9) index\_col | (“<имя объекта>”,  <ид. индекса>, <ключ>) | имя колонки индекса |
| 10) isnull | (<выражение>, <значение>) | замена заданного значения null значением |
| 11) object\_id | (“<имя объекта БД>”) | номер идентификатора объекта БД |
| 12) object\_name | (<ид. объекта БД>) | имя объекта БД |
| 13) reserved\_pgs | (<ид. объекта>, {doampg | ioampg}) | количество страниц, занимаемое таблицей или индексом (включаются страницы, используемые внутренними структурами |
| 14) rowcnt | (doampg) | количество записей в таблице |
| 15) sused\_id | ([“<имя клиента>”]) | номер идентификатора клиента |
| 16) sused\_name | ([<ид. клиента>]) | имя клиента |
| 17) tsequal | (<вр. метка>, <вр. метка2>) | сравнивает значения временных меток измененной записи; <вр. метка> - временная метка после выборки записи для просмотра; <вр. метка2> - временная метка сохраненной записи после обновления |
| 18) used\_pgs | (<ид. объекта>, doampg, ioampg) | общее количество страниц, занимаемое таблицей и ее индексом (включаются страницы, используемые внутренними структурами) |
| 19) user\_id | ([“<имя пользов.>”]) | номер идентификатора пользователя |
| 20) user\_name | ([<ид.пользователя>]) | имя пользователя |
| 21) valid\_name | (“<строка>”) | возвращает 0, если <строка> содержит недопустимые символы или длиной больше 30 байт, и не 0 - в противном случае |

Примеры:

1. select x = col\_length (“titles”, “title”)
2. select length = datalength (pub\_name), pub\_name from publishers
3. select name from sysusers where name = user\_name(1)

**Строковые функции**

Строковые функции позволяют работать со строковыми и двоичными данными.

Синтаксис вызова строковых функций:

select <имя функции> (<аргументы>)

Кроме этого, можно соединять операцией конкатенации двоичные и символьные выражения следующим образом:

select (<выражение> + <выражение> [ + <выражение>]…)

Обозначение типов аргументов в таблице2:

1. char\_expr - типы char, varchar, nchar и nvarchar;
2. expression - типы char\_expr и типы binary и varbinary;
3. pattern - типы данных char\_expr, которые можно включать в сопоставление с образцами;
4. approx\_numeric - типы float, real и double precition;
5. integer\_expr - типы tinyint, smallint и int;
6. start - тип integer\_expr;
7. length - тип integer\_expr.

Таблица 2

|  |  |  |
| --- | --- | --- |
| Функция | Аргумент(ы) | Результат |
| 1) ascii | (char\_expr) | код ASCII для первого символа в выражении |
| 2) char | (integer\_expr) | преобразует цифру, занимающую 1 байт в символ длиной 1 байт |
| 3) charindex | (expression1, expression2) | ищет первое вхождение выражения1 в выражение2 и возвращает номер позиции; если не находит вообще, то возвращает 0 |
| 4) char\_length | (char\_expr) | количество символов в строке или тексте |
| 5) difference | (char\_expr1, char\_expr2) | разность между двумя значениями типа soundex (см. ниже) |
| 6) lower | (char\_expr) | преобразует большие буквы в маленькие |
| 7) ltrim | (char\_expr) | убирает начальные пробелы |
| 8) patindex | (“%pattern%”, char\_expr [using {bytes | chars | characters}]) | возвращает номер первой позиции вхождения pattern в char\_expr или 0, если вхождения нет |
| 9) replicate | (char\_expr, integer\_expr) | возвращает строку типа char\_expr, содержащуюся в аргументе1 и повторяемую <аргумент2> раз (max длина строки 255 байт) |
| 10) reverce | (char\_expr) | реверс строки, например: “abcd” - “dcba” |
| Функция | Аргумент(ы) | Результат |
| 11) right | (char\_expr, integer\_expr) | возвращает часть строки, состоящей из <аргумента2> символов, считая справа |
| 12) rtrim | (char\_expr) | убирает замыкающие пробелы |
| 13) soundex | (char\_expr) | возвращает четырехсимвольный код символьной строки, состоящий из римских букв |
| 14) space | (integer\_expr) | возвращает строку из указанного количества пробелов |
| 15) str | (approx\_numeric [, length [, decimal]]) | символьное представление числа с плавающей точкой; length устанавливает общее число знаков, а decimal - число знаков после десятичной точки; если length и decimal не указываются, то по умолчанию length=10, а decimal=0 |
| 16) stuff | (char\_expr1, start, length, char\_expr2) | удаляет length символов из expr1, начиная с start, а затем вставляет expr2 в expr1, начиная с start; если expr2=null, то только удаляет |
| 17) substring | (expression, start, length) | выделение подстроки в expression длиной length, начиная с start |
| 18) upper | (char\_expr) | преобразование маленьких букв в большие |
| 19) + | expression + expression | конкатенация двух и более символьных или бинарных выражений |

Примеры:

1. select au\_lname, substring (au\_fname, 1, 1) from authors
2. select charindex(“wonderful”, notes), patindex(“wonderful”, notes) from titles

where title\_id = “TC3218”

1. select stuff(“abc”, 2, 3, “xyz”)
2. select (“abc” + “def”)

Строковые функции могут вкладываться друг в друга.

1. select substring(pub\_id + title\_id, 1, 6) from titles where price > $20

**Текстовые функции**

Текстовые функции используются для работы с данными типа text и image.

Таблица 3

|  |  |  |
| --- | --- | --- |
| Функция | Аргумент(ы) | Результат |
| 1) patindex | (“%pattern%”,char\_expr [using {bytes | chars | characters}]) | возвращает числовое представление значения первой позиции первого вхождения pattern в символьную строку или 0 - если pattern не найден |
| 2) textptr | (<имя текстовой колонки>) | возвращает указатель на текст (16байтное двоичное число) |
| 3) textvalid | (“<имя таблицы>..<имя колонки>”, <указатель на текст>) | возвращает 1, если указатель допустимый и 0 - в противном случае |
| 4) set textsize | {n | 0} | задает max длину в байтах для колонки типа text/image в select-предложении; если 0, то max длина равна 32 K |

Пример:

declare @val varbinary(16)

select @val = textptr(blurb) from texttest

**Математические функции**

Общий синтаксис вызова математической функции:

<имя функции> (<аргументы>)

Типы аргументов:

1. approx\_numeric - это типы float, real и double precition;
2. integer - типы tinyint, smallint и int;
3. numeric - это типы approx\_numeric, numeric, dec, decimal, все integer и money;
4. power - это типы numeric, approx\_numeric и money.

Таблица 4

|  |  |  |
| --- | --- | --- |
| Функция | Аргумент(ы) | Результат |
| 1) abs | (numeric) | абсолютное значение аргумента |
| 2) acos | (approx\_numeric) | арккосинус (в радианах) |
| 3) asin | (approx\_numeric) | арксинус (в радианах) |
| 4) atan | (approx\_numeric) | арктангенс (в радианах) |
| 5) atn2 | (approx\_numeric1, approx\_numeric2) | арктангенс деления аргумента1 на аргумент2 |
| 6) ceiling | (numeric) | округление до ближайшего целого, большего или равного аргументу |
| 7) cos | (approx\_numeric) | косинус (в радианах) |
| 8) cot | (approx\_numeric) | котангенс (в радианах) |
| 9) degrees | (numeric) | преобразование радианов в градусы |
| 10) exp | (approx\_numeric) | число e в степени аргумент |
| 11) floor | (numeric) | округление до ближайшего целого, меньшего или равного аргументу |
| 12) log | (approx\_numeric) | натуральный логарифм |
| 13) log10 | (approx\_numeric) | десятичный логарифм |
| 14) pi | () | число пи |
| 15) power | (numeric, power) | преобразование числа типа numeric в число типа power |
| 16) radians | (numeric) | преобразование градусов в радианы |
| 17) rand | ([integer]) | функция random на отрезке [0;1] или для числа типа integer |
| 18) round | (numeric, integer) | округление числа типа numeric до числа из integer знаков |
| 19) sign | (numeric) | знак числа |
| 20) sin | (approx\_numeric) | синус (в радианах) |
| 21) sqrt | (approx\_numeric) | квадратный корень |
| 22) tan | (approx\_numeric) | тангенс (в радианах) |

Примеры:

1. select ceiling(123.45) => 124.
2. select round(123.4545,2) => 123.4500

**Функции даты (времени)**

Таблица 5

|  |  |  |
| --- | --- | --- |
| Часть даты | Аббревиатура | Значения |
| 1) год | 1) yy | 1) 1753÷9999 |
| 2) квартал | 2) qq | 2) 1÷4 |
| 3) месяц | 3) mm | 3) 1÷12 |
| 4) день года | 4) dy | 4) 1÷366 |
| 5) день | 5) dd | 5) 1÷31 |
| 6) неделя | 6) wk | 6) 1÷54 |
| 7) день недели | 7) dw | 7) 1÷7 (1-Sunday) |
| 8) час | 8) hh | 8) 0÷23 |
| 9) минута | 9) mi | 9) 0÷59 |
| 10) секунда | 10) ss | 10) 0÷59 |
| 11) миллисекунда | 11) ms | 11) 0÷999 |

Таблица 6

|  |  |  |
| --- | --- | --- |
| Функция | Аргумент(ы) | Результат |
| 1) getdate | ( ) | Текущая системная дата и время |
| 2) datename | (datepart, date) | часть даты как строка ASCII |
| 3) datepart | (datepart, date) | часть даты как целое число |
| 4) datediff | (datepart, date, date) | количество времени между 2-мя датами в частях даты |
| 5) dateadd | (datepart, number, date) | дата, получаемая прибавлением частей даты к другой дате |

Примеры:

1. select getdate();
2. select datediff(month, pubdate, ”Nov 30 1985”) from titles;
3. select dateadd(day, 3, pubdate) from titles

**Функции преобразования одних типов данных в другие и форматирование дат**

SQL-сервер поддерживает три функции преобразования типов:

1. convert;
2. inttohex;
3. hextoint.

**Синтаксис функции convert:**

convert (<тип данных>, <выражение> [, <стиль>])

Пример:

select title, convert(char(5), total\_sales) from titles

**Правила преобразования:**

1) преобразование данных типов character в данные типов money, date/time, всех типов numeric и approx\_numeric осуществляется только в том случае, если все символы в строке допустимы в новом типе;

2) при преобразовании целых типов данных в тип character необходимо указывать соответствующую длину строки символов;

1. функцию convert можно использовать при преобразовании типов money, date/time, всех типов numeric, integer, binary и image.

**Функции inttohex и hextoint** служат для преобразования целых чисел в шестнадцатиричные и наоборот.

**Операция Join (соединения таблиц).**

Для выполнения операции соединения нескольких таблиц в предложении SELECT необходимо соблюдать три условия:

1) В списке выборки имена колонок указываются с именами таблиц, в которые входят эти колонки.

2) В предложении from указываются через запятую имена всех таблиц, участвующих в соединении, причем на первом месте указывается таблица, в которой осуществляется выборка данных.

3) В предложении where указываются все связи таблиц, где каждая таблица соединяется с другой по полям, определенным на одинаковых доменах, с помощью операций сравнения: =, >, >=, <, <=, !=, !>, !<.

При выполнении операции join в отчет по запросу входят только те записи, которые удовлетворяют условию соединения таблиц. Иногда желательно посмотреть данные, которые не удовлетворили этому условию. В таких случаях используют **операцию Outer join,** в которой применяются только два оператора сравнения:

1) \*= - включаются все записи из первой названной таблицы;

1. =\* - включаются все записи из второй названной таблицы.

При этом, если указывается операция “\*=”, то в отчет включаются все записи из первой таблицы, а в колонках, принадлежащих второй таблице, в записях, не удовлетворяющих условию соединения, ставятся null значения. Аналогично, если указывается операция “=\*”, то в отчет включаются все записи из второй таблицы, а в колонках, принадлежащих первой таблице, в записях, не удовлетворяющих условию соединения, ставятся null значения.

Пример:

select au\_fname, au\_lname, pub\_name from authors, publishers

where authors.city \*= publishers.city

**Подзапросы**

Подзапрос - это select - предложение, вложенное в другое select-, insert-, update- или delete-предложение или в другой подзапрос.

Предложения, которые включают подзапрос, обычно имеют один из следующих форматов:

1) where <выражение> [not] in (<подзапрос>)

2) where <выражение> <операция сравнения> [any|all] (<подзапрос>)

1. where [not] exists (<подзапрос>)

**Подмножества**

**Подмножество** - это порождение (представление) данных, строящееся на основе одной или нескольких таблиц БД, называемых базовыми. Кроме этого, представление может быть построено на основе другого представления.

Представления используются для:

1. предоставления пользователю только интересующих его данных;
2. упрощения манипулирования данными (представления определяются сложными операциями выборки, проекции и соединения);
3. предоставления различным пользователям видеть одни и те же данные по-разному;
4. обеспечения механизма секретности данных;
5. поддержки логической независимости данных.

**Механизм секретности данных** осуществляется **командами grant и revoke,** примененным к представлениям. При этом пользователи получают доступ к различным подмножествам данных:

1. доступ к подмножеству записей базовой таблицы;
2. доступ к подмножеству колонок базовой таблицы;
3. доступ к подмножеству записей и колонок базовой таблицы;
4. доступ к записям, получаемых путем соединения двух и более базовых таблиц;
5. доступ к статистическим итогам данных в базовой таблице;
6. доступ к подмножеству другого представления или некоторой комбинации представлений и базовых таблиц.

В основе команды создания представления лежит оператор select. **Синтаксис команды:**

create view [[<имя БД>.] <владелец>.] <имя представления>

[(<имя колонки> [, <имя колонки>]…)]

as <select-предложение>

[with check option]

Пример:

create view pub\_view (publisher, city, state)

as select pub\_name, city, state from publishers

Существует несколько ограничений на использование select-предложения в создании представления:

1. нельзя использовать структуры order by и compute;
2. нельзя использовать слово into;
3. нельзя ссылаться на временную таблицу.

Структура with check option используется в тех случаях, когда данное представление будет использоваться командами insert и update. При этом будут контролироваться вводимые и обновляемые записи на соответствие select-предложению в представлении.

Существуют ограничения применения операторов изменения данных (update, insert и delete рассматриваются в следующем разделе) к представлениям:

1. нельзя применять эти операторы к колонкам представления, значения которых подсчитываются с помощью агрегатных или встроенных функций;
2. нельзя применять эти операторы к представлениям, использующим агрегатные функции и структуру group by;
3. нельзя применять эти операторы к представлениям, использующим distinct;
4. нельзя использовать insert для представлений, построенных на таблицах с колонками not null, когда в эти колонки попадают null значения;
5. нельзя использовать delete для представлений, построенных на нескольких таблицах;
6. нельзя использовать insert для представлений, построенных на нескольких таблицах с опцией with check option;
7. нельзя применять insert и update в представлении, построенном на нескольких таблицах с опцией distinct;
8. нельзя использовать update для колонки identity.

**Удаление представлений** осуществляется командой **drop view:**

drop view [[<имя БД>.] <владелец>.] <имя представления>

[, [[<имя БД>.] <владелец>.] <имя представления>]…

Пример:

drop view pub\_view

**Обновление данных**

**Понятие транзакции**

Транзакция - это механизм, объединяющий множество действий в БД в логическую единицу работы. Другими словами, транзакция позволяет пользователю сгруппировать любое количество действий с БД в единое целое, которое может быть выполнено или отменено. Использование транзакций необходимо при выполнении любых изменений в БД.

Транзакции обеспечивают:

1. целостность данных при работе операторов манипулирования данными (insert, update и delete);
2. восстановление данных в случае сбоев.

По умолчанию, каждая из команд insert, update и delete рассматривается как одна транзакция.

Каждая определяемая пользователем транзакция, состоящая из операторов языка SQL, начинается командой begin transaction (начать транзакцию), а заканчивается либо командой commit transaction (выполнить транзакцию), либо командой rollback transaction (отменить транзакцию).

**Понятие пакета**

Пакет - это набор транзакций (одна или более), запускающий процесс работы SQL - сервера и завершающийся по признаку конца пакета. По окончании работы пакета автоматически обеспечивается работа инструментальных средств создания отчетов. Признаком конца пакета является слово "gо". Кроме SQL-предложений в пакетах можно использовать операторы программного языка (Transact-SQL). В свою очередь, набор пакетов можно в интерактивном режиме сохранять в файле операционной системы и затем загружать из файла операционной системы.

**Оператор INSERT** рассматривался выше при описании загрузки таблиц БД.

Пример:

insert poss values

(534305, 'ЖУРАВЛЕВ АВЕРЬЯН АЛЕКСЕЕВИЧ', '000073', 'M', '0', '001001', '05', 0129000, '08', '90', '03','95')

**Оператор DELETE** служит для удаления записей из таблицы по заданному условию поиска.

delete [from][[<имя БД>.]<владелец>.]{<имя таблицы>|<имя представления>}

[from][[<имя БД>.]<владелец>.]{<имя таблицы>|<имя представления>}

[,[[<имя БД>.<владелец>.]<имя таблицы>|<имя представления>]]...]

[where <условие поиска>]

Пример:

delete poss where nomer = 534305

удаление из таблицы poss записи с данными о Журавлеве Аверьяне Алексеевиче (см. предыдущий пример).

**Оператор TRUNCATE** позволяет быстро удалить все записи из таблицы.

truncate table [[<имя БД>.] <владелец>.] <имя таблицы>

Пример: truncate table poss

**Оператор UPDATE** служит для внесения изменений в записи таблиц БД.

update [[<имя БД>.]<владелец>.] {<имя таблицы>|<имя представления>}

set [[[<имя БД>.]<владелец>.]{<имя таблицы>.|<имя представления>.}]

<имя кол.1>={<выр.1> | null | (<предложение select>)}

[,<имя кол.2>={<выр.2> | null | (<предложение select>)}]...

[from [[<имя БД>.] <владелец>.]{<имя таблицы>|<имя представления>}

[,[[<имя БД>.]<владелец>.]{<имя таблицы>|<имя представления>}]]...

[where <условие поиска>]

Пример: Студент Иванов Иван Петрович был переведен из МЭИ в МГУ:

update poss set vuz\_k=2066426

where fio='Иванов Иван Петрович' and vuz\_k=2066414

где 2066414 - код МЭИ,

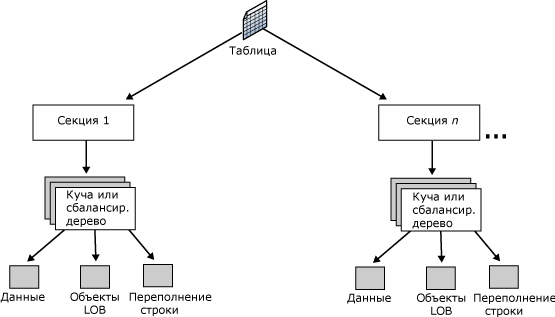
2066426 - код МГУ.

**Лекция № 14**

**Создание индексов и управление индексами**

## Структура хранения данных

Таблицы хранятся в виде коллекции страниц размером 8 КБ. Следующая иллюстрация показывает организацию таблицы. Таблица содержится в одной или нескольких секциях, а каждая секция содержит строки данных либо в куче, либо в структуре кластеризованного индекса.



Таблицы SQL Server используют один из двух методов организации страниц данных внутри секции:

* Кластеризованные таблицы - это таблицы, имеющие кластеризованный индекс.   
  Строки данных хранятся по порядку ключа кластеризованного индекса. Кластеризованный индекс реализуется в виде структуры индекса сбалансированного дерева, которая поддерживает быстрый поиск строк по их ключевым значениям в кластеризованном индексе. Страницы в каждом уровне индекса, включая страницы данных на конечном уровне, связаны в двунаправленный список. Однако перемещение из одного уровня на другой выполняется при помощи ключевых значений.
* Кучи - это таблицы, которые не имеют кластеризованного индекса.   
  Строки данных хранятся без определенного порядка, и какой-либо порядок в последовательности страниц данных отсутствует. Страницы данных не связаны в связный список. Поиск строк требует полного последовательного просмотра всех строк таблицы.

В соответствие с организацией страниц данных внутри секции будет происходить чтение страниц для выборки или изменения данных.

Во многих случаях полный последовательный просмотр всех строк таблицы является не оптимальной стратегией поиска записи. Например, чтобы обновить одну строку в таблице без кластеризованного индекса может потребоваться последовательно считать с диска все страницы этой таблицы.

Необходимо определить, что важнее - повышение производительности запросов или производительность при изменении данных и дополнительные требования к месту на диске.

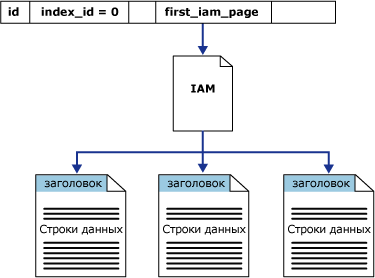
## Кучи

Кучей является таблица без кластеризованного индекса. По умолчанию у кучи есть одна секция. Если куча имеет несколько секций, каждая из них имеет структуру кучи, содержащую данные для этой определенной секции. Например, если у кучи четыре секции, имеются четыре структуры кучи, по одной на каждую секцию.

Внутренние структуры SQL Server хранят указатель на первую IAM-страницу в цепи IAM-страниц, SQL Server использует IAM-страницы для перемещения по куче. Страницы данных и строки в этих страницах не расположены в каком-либо порядке и не связаны. Единственным логическим соединением страниц данных являются данные, записанные в IAM-страницы.

Просмотр таблиц или последовательное считывание в куче может выполняться просмотром IAM-страниц для нахождения экстентов, хранящих страницы кучи. Так как карта IAM представляет экстенты в том же порядке, в котором они существуют в файлах данных, это означает, что последовательный просмотр кучи выполняется последовательно в каждом файле. Использование IAM-страниц для определения последовательности просмотра означает также, что строки из кучи обычно возвращаются не в том порядке, в котором они вставлялись.

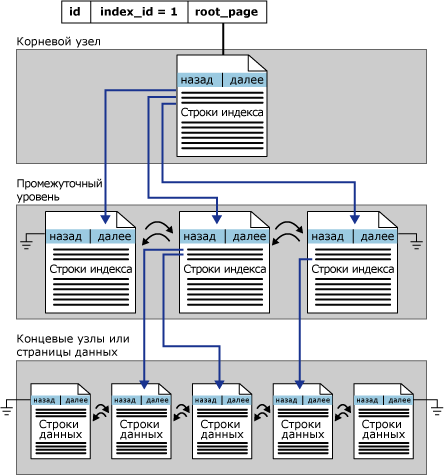
На следующей иллюстрации демонстрируется, как компонент SQL Server Database Engine использует IAM-страницы для получения строк данных из кучи с одной секцией.



## Структуры кластеризованного индекса

В SQL Server индексы организованы в виде сбалансированных деревьев (B-деревья, B-trees). Каждая страница в сбалансированном дереве индекса называется узлом индекса. Верхний узел сбалансированного дерева называется корневым. Узлы нижнего уровня индекса называются конечными. Все уровни индекса между корневыми и конечными узлами называются промежуточными. В кластеризованном индексе конечные узлы содержат страницы данных базовой таблицы. На страницах индекса корневого и промежуточного узлов находятся строки индекса. Каждая строка индекса содержит ключевое значение и указатель либо на страницу промежуточного уровня сбалансированного дерева, либо на строку данных на конечном уровне индекса. Страницы на каждом уровне связаны в двунаправленный список.

На следующем рисунке изображена структура кластеризованного индекса.



По умолчанию кластеризованный индекс занимает одну секцию. Если кластеризованный индекс занимает несколько секций, каждая секция содержит сбалансированное дерево, содержащее данные этой секции. Например, если кластеризованный индекс занимает четыре секции, существует четыре сбалансированных дерева: по одному в каждой секции.

Страницы и строки, которые они содержат, упорядочены по значению ключа кластеризованного индекса. Все строки вставляются так, чтобы значение ключа составляло вместе с существующими строками упорядоченную последовательность.

Внутренние структуры SQL Server хранят указатели на корневые узлы кластеризованного индекса для каждой секции. SQL Server движется вниз по индексу, чтобы найти строку, соответствующую ключу кластеризованного индекса. Чтобы найти диапазон ключей, SQL Server сначала находит начальное значение ключа в диапазоне, а затем сканирует страницы данных, используя указатели на следующую и предыдущую страницу. Чтобы найти первую страницу в цепочке страниц данных, SQL Server движется по самым левым указателям от корня индекса.

## Структуры некластеризованного индекса

Некластеризованные индексы имеют ту же структуру сбалансированного дерева, что и кластеризованные индексы; существуют только следующие различия:

* строки данных в базовой таблице не сортируются и хранятся в порядке, который основан на их некластеризованных ключах;
* конечный уровень некластеризованного индекса состоит из страниц индекса вместо страниц данных.

Некластеризованные индексы могут определяться на таблице или представлении с кластеризованным индексом, или на куче. Для каждой таблицы можно создать до 999 некластеризованных индексов.

Каждая строка некластеризованного индекса содержит некластеризованное ключевое значение и указатель на строку. Этот указатель определяет строку данных кластеризованного индекса или кучи, содержащую ключевое значение.

Указатели строк на строках некластеризованных индексов являются либо указателем на строку, либо ключом кластеризованного индекса для строки, как описано ниже.

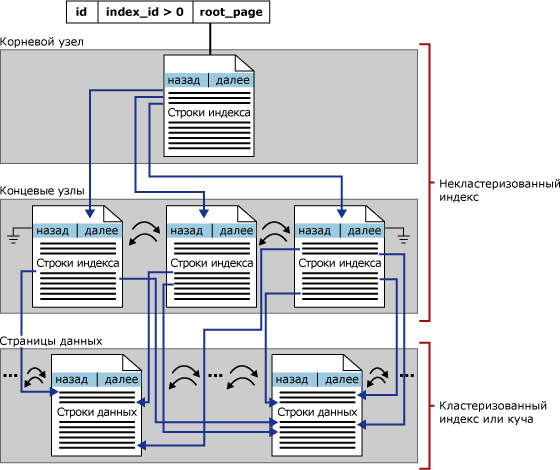
* Если таблица является кучей, что означает, что она не содержит кластеризованный индекс, то указатель строки является указателем на строку. Указатель строится на основе идентификатора файла (ID), номера страницы и номера строки на странице. Весь указатель целиком называется идентификатором строки (RID).
* Если для таблицы имеется кластеризованный индекс, то указатель строки - это ключ кластеризованного индекса для строки. Поскольку значение ключа кластеризованного индекса будет копироваться по все строки всех некластеризованных индексов созданных на данной таблице, необходимо очень тщательно выбирать столбцы для создания кластеризованного индекса. Использование в качестве ключа столбцов хранящих данные большой размерности может существенно увеличить место, необходимо для хранения таблицы и всех созданных на ней индексов.

Если кластеризованный индекс не является уникальным индексом, то SQL Server делает все имеющиеся повторяющиеся ключи уникальными путем добавления внутри созданного значения, называемого **uniqueifier**. Это четырехбайтовое значение невидимо для пользователей. Оно используется тогда, когда необходимо сделать кластеризованный ключ уникальным, чтобы использовать в некластеризованных индексах.

SQL Server получает строку данных путем поиска по кластеризованному индексу, используя ключ кластеризованного индекса, который хранится в конечной строке некластеризованного индекса.

По умолчанию некластеризованный индекс включает одну секцию. Если некластеризованный индекс состоит из нескольких секций, то каждая секция имеет структуру сбалансированного дерева, в которой содержатся индексные строки для данной конкретной секции.

На следующей иллюстрации показана структура некластеризованного индекса, состоящего из одной секции.



## Оптимизация индексов

**Использование неключевых столбцов**

Функциональность некластеризованных индексов может быть расширена путем добавления включенных столбцов, также называемых неключевыми столбцами, на конечный уровень индекса. Ключевые столбцы хранятся на всех уровнях некластеризованного индекса, тогда как неключевые столбцы хранятся только на конечном уровне.

Добавление неключевых столбцов позволяет создавать некластеризованные индексы, покрывающие больше запросов. Это обусловлено следующими преимуществами неключевых столбцов.

* Они могут содержать типы данных, не разрешенные для ключевых столбцов индекса.
* Они не учитываются компонентом Database Engine при расчете числа ключевых столбцов индекса и размера ключа индекса.

Должен быть определен как минимум один ключевой столбец. Максимальное количество неключевых столбцов равно 1023. Это на 1 меньше, чем максимальное количество столбцов таблицы.

Индекс со включенными неключевыми столбцами может значительно повысить производительность запроса, когда все столбцы запроса включены в индекс как ключевые или неключевые. В то время как ключевые столбцы сохраняются на всех уровнях индекса, неключевые столбцы сохраняются только на конечном уровне. Производительность повышается благодаря тому, что оптимизатор запросов может найти все значения столбцов в этом индексе; при этом нет обращения к данным таблиц или кластеризованных индексов, что приводит к меньшему количеству дисковых операций ввода-вывода.

Кластеризованные и некластеризованные индексы могут содержать до 16 ключевых столбцов и размер ключа индекса (суммарная длина всех ключевых столбцов) не должен превышать 900 байт. Индексы на столбцах типа **varchar**, размер которых превышает 900 байт, могут быть созданы, если в момент создания индекса объем существующих данных в столбцах не превышает 900 байт; но последующие операции вставки или обновления, вызывающие превышение общего размера 900 байт, будут заканчиваться ошибкой.

Компонент Database Engine не учитывает неключевые столбцы при расчете количества ключевых столбцов индекса и размера ключа индекса. Таким образом можно включать неключевые столбцы в некластеризованный индекс, чтобы избежать превышения текущих ограничений на размер индекса и размер ключа индекса.

Общий размер всех неключевых столбцов ограничен только размером столбцов, например столбцы varchar(max) могут иметь размер до 2 ГБ.

Следует избегать добавления неиспользуемых столбцов. Добавление слишком большого количества столбцов, может оказать следующее влияние на производительность:

* На странице будет помещаться меньше строк индекса. Это может привести к увеличению количества операций ввода-вывода и снизить эффективность кэша.
* Для хранения индекса потребуется больше места на диске. В частности, добавление типов данных **varchar(max)**, **nvarchar(max)**, **varbinary(max)** и **xml** в качестве неключевых индексных столбцов может значительно повысить требования к месту на диске. Это обусловлено тем, что значения столбцов копируются на конечный уровень индекса. Поэтому они находятся и в индексе, и в базовой таблице.
* Обслуживание индекса может увеличить время выполнения операций изменения, вставки, обновления и удаления в базовой таблице или индексированном представлении.

**Статистики и информация об отсутствующих индексах и**

Исполнению запроса компонентом Database Engine предшествует его анализ оптимизатором запросов, с целью определения наиболее эффективного для конкретного условия фильтра способа извлечения запрошенных данных. Помимо самого запроса оптимизатором запросов используется схема базы данных (определения таблиц и индексов) и статистики базы данных.

План выполнения запроса задает:

* Последовательность, в которой происходит обращение к исходным таблицам.

Как правило, существует много последовательностей, в которых сервер базы данных может обращаться к базовым таблицам для построения результирующего набора, и в общем случае последовательность обращения к таблицам при непосредственном выполнении запроса может не совпадать с последовательностью в которой эти таблицы указаны в запросе.

* Методы, используемые для извлечения данных из каждой таблицы.

Есть различные методы для обращения к данным в каждой таблице. Например, если необходимы только несколько строк с определенными ключевыми значениями, то Database Engine может использовать индекс. Если необходимы все строки в таблице, то может быть выполнен полный просмотр таблицы, игнорируя индексы. Если необходимы все строки в таблице, но есть индекс, ключевые столбцы которого находятся в ORDER BY, то выполнение просмотра индекса вместо просмотра таблицы позволит избежать отдельный сортировки результирующего набора.

Оптимизатор запросов SQL Server основан на оценке стоимости. Каждому возможному плану выполнения соответствует некоторая стоимость, определенная в терминах объема использованных вычислительных ресурсов. Оптимизатор запросов SQL Server выбирает план выполнения, основываясь не только на самой низкой стоимости ресурсов - он выбирает такой план, который возвращает результаты пользователю за минимальное время при разумной стоимости ресурсов. Например, параллельная обработка запроса обычно использует больше ресурсов, чем его последовательная обработка, но завершает выполнение запроса быстрее. Оптимизатор SQL Server будет использовать параллельный план выполнения для возврата результатов, если это не окажет негативного влияния на загрузку сервера.

Для оценки количества строк в результатах запроса и как следствие нагрузки на вычислительные ресурсы оптимизатор запросов полагается на статистику распределения. Статистика - это объект, содержащие статистические сведения о распределении значений в одном или нескольких столбцах таблицы. Например, оптимизатор запросов может использовать оценочное количество элементов, чтобы выбрать метод для обращения к данным в таблице.

Каждый объект статистики создается для списка из одного или нескольких столбцов таблицы и содержит гистограмму, в которой отображается распределение значений в первом столбце. Объекты статистики для нескольких столбцов также хранят статистические сведения о корреляции значений между столбцами. Эти статистические данные корреляции называются значениями плотности и получаются из числа уникальных строк значений столбцов.

Если статистика индекса не является актуальной, оптимизатор запросов, возможно, не сделает лучший выбор для текущего состояния таблицы.

Использование статистики на уровне базы данных контролируется следующими параметрами, рассмотренными в модуле 2:

* **AUTO\_CREATE\_STATISTICS**

Оптимизатор запросов в случае необходимости создает статистику по отдельным столбцам в предикатах запросов, чтобы улучшить планы запросов и повысить производительность запросов.

* **AUTO\_UPDATE\_STATISTICS**

Указывает, что оптимизатор запросов обновляет статистику, если она используется в запросе и может оказаться устаревшей. Статистика становится устаревшей, после того как операции вставки, обновления, удаления или слияния изменяют распределение данных в таблице.

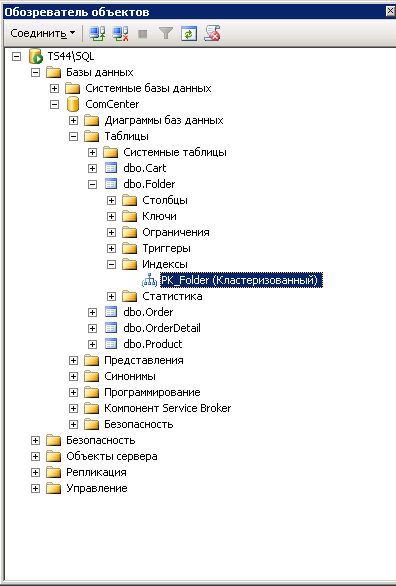
* **AUTO\_UPDATE\_STATISTICS\_ASYNC**

Указывает, что запросы, которые запускают автоматическое обновление устаревшей статистики, не будут ожидать обновления статистики перед компиляцией. Последующие запросы будут использовать обновленную статистику, когда она будет доступна.

Помимо статистик для оптимизации запросов используется информация об индексах. При анализе списка запрашиваемых столбцов и порядка сортировки оптимизатор запросов ищет наиболее подходящие индексы для ускорения обработки запроса и снижения нагрузки на вычислительные мощности сервера. Если наиболее подходящие индексы не существуют, оптимизатор запросов формирует близкий к оптимальному план запроса, при этом он сохраняет сведения об отсутствующих индексах. Функция отсутствующих индексов позволяет получить сведения об этих индексах, и решить, стоит ли их создавать в базе данных или нет.

## Источники информации об индексах

**Обозреватель объектов SQL Server Management Studio**



Представления каталога

|  |  |
| --- | --- |
| **Представление каталога** | **Отображение сведений о** |
| **sys.indexes** | Тип индекса, идентификатор файловой группы или схемы секционирования, а также текущие параметры индекса, хранящиеся в метаданных. |
| **sys.index\_columns** | Идентификатор столбца, положение в индексе, тип (ключевой или неключевой) и порядок сортировки (ASC или DESC). |
| **sys.stats** | Статистика, связанная с индексом, включая имя статистики, а также то, была ли она создана автоматически или пользователем. |
| **sys.stats\_columns** | Идентификатор столбца, связанного со статистикой. |
| **sys.xml\_indexes** | Тип XML-индекса (первичный или вторичный), тип и описание вторичного индекса. |

Функции работы с индексами

|  |  |
| --- | --- |
| **Функция** | **Отображение сведений о** |
| **sys.dm\_db\_index\_physical\_stats** | Размер и статистика фрагментации индекса. |
| **sys.dm\_db\_index\_operational\_stats** | Статистика операций ввода-вывода текущего индекса и таблицы. |
| **sys.dm\_db\_index\_usage\_stats** | Статистика использования индекса запросами определенного типа. |
| **sys.dm\_db\_missing\_index\_columns** | Сведения о столбцах таблиц базы данных, для которых отсутствуют индексы. |
| **sys.dm\_db\_missing\_index\_details** | Подробные сведения об отсутствующих индексах. |
| **sys.dm\_db\_missing\_index\_group\_stats** | Сводные данные о группах отсутствующих индексов. |
| **sys.dm\_db\_missing\_index\_groups** | Данные о том, какие отсутствующие индексы содержались в указанной группе. |
| **INDEXKEY\_PROPERTY** | Положение индексного столбца в индексе и порядок сортировки столбца (ASC или DESC). |
| **INDEXPROPERTY** | Тип индекса, количество уровней и текущие параметры индекса, хранящиеся в метаданных. |
| **INDEX\_COL** | Имя ключевого столбца указанного индекса. |

## Создание индексов

Создание индексов осуществляется командой CREATE INDEX.

CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index\_name

    ON <object> ( column [ ASC | DESC ] [ ,...n ] )

    [ INCLUDE ( column\_name [ ,...n ] ) ]

[ WHERE <filter\_predicate> ]

    [ WITH ( <relational\_index\_option> [ ,...n ] ) ]

    [ ON { filegroup\_name | default } ]

<object> ::=

{

    [ database\_name. [ schema\_name ] . | schema\_name. ]

        table\_or\_view\_name

}

<relational\_index\_option> ::=

{

    PAD\_INDEX = { ON | OFF }

  | FILLFACTOR = fillfactor

  | SORT\_IN\_TEMPDB = { ON | OFF }

  | IGNORE\_DUP\_KEY = { ON | OFF }

  | STATISTICS\_NORECOMPUTE = { ON | OFF }

  | DROP\_EXISTING = { ON | OFF }

  | ONLINE = { ON | OFF }

  | ALLOW\_ROW\_LOCKS = { ON | OFF }

  | ALLOW\_PAGE\_LOCKS = { ON | OFF }

  | MAXDOP = max\_degree\_of\_parallelism

}

**UNIQUE**

Создает уникальный индекс для таблицы или представления. Уникальным является индекс, в котором не может быть двух строк с одним и тем же значением ключа индекса. Кластеризованный индекс представления должен быть уникальным.

Компонент Database Engine не позволяет создать уникальный индекс по столбцам, уже содержащим повторяющиеся значения, даже если параметру IGNORE\_DUP\_KEY присвоено значение ON. При попытке создания такого индекса компонент Database Engine выдает сообщение об ошибке. Прежде чем создавать уникальный индекс по такому столбцу или столбцам, необходимо удалить все повторяющиеся значения. Столбцы, используемые в уникальном индексе, должны иметь свойство NOT NULL, т. к. при создании индекса значения NULL рассматриваются как повторяющиеся.

**CLUSTERED**

Создает кластеризованный индекс, при создании кластеризованного индекса все существующие некластеризованные индексы таблицы перестраиваются. Если аргумент CLUSTERED не указан, создается некластеризованный индекс.

**NONCLUSTERED**

Создание индекса, задающего логическое упорядочение для таблицы. Логический порядок строк в некластеризованном индексе не влияет на их физический порядок.

Для индексированных представлений некластеризованные индексы могут создаваться только в случае, если уже определен уникальный кластеризованный индекс.

По умолчанию, используется значение NONCLUSTERED.

**index\_name**

Имя индекса. Имена индексов должны быть уникальными в пределах таблицы или представления, но необязательно должны быть уникальными в пределах базы данных.

**column**

Столбец или столбцы, на которых основан индекс. Имена одного или нескольких столбцов для создания комбинированного индекса. Столбцы, которые должны быть включены в составной индекс, указываются в скобках за аргументом table\_or\_view\_name в порядке сортировки.

Столбцы с типами данных для больших объектов **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml** или **image** не могут быть ключевыми столбцами для индекса. Кроме того, определение представления не может включать столбцы типов **ntext**, **text** и **image**, даже если они указаны в инструкции CREATE INDEX.

**[ ASC | DESC ]**

Определяет сортировку значений заданного столбца индекса: по возрастанию или по убыванию. Значение по умолчанию **-** ASC.

**INCLUDE (column[ ,*...* n])**

Указывает неключевые столбцы, добавляемые на конечный уровень некластеризованного индекса. Некластеризованный индекс может быть уникальным или неуникальным.

**WHERE <filter\_predicate>**

Отфильтрованный индекс является оптимизированным некластеризованным индексом, предназначенным для запросов, выбирающих небольшой процент строк таблицы. Чтобы проиндексировать часть данных таблицы, в нем используется предикат фильтра. Правильно составленный отфильтрованный индекс может увеличить скорость выполнения запроса, уменьшить стоимость хранения и обслуживания.

Предикат фильтра использует простую логику сравнения и не может ссылаться на вычисляемый столбец, столбец определяемого пользователем типа, столбец пространственного типа данных или столбец типа **hierarchyID**. Сравнения с помощью литералов NULL с операторами сравнения недопустимы. Вместо этого используются операторы IS NULL и IS NOT NULL.

При создании и использовании отфильтрованных индексов обязательно должны быть установлены следующие значения параметров.

|  |  |
| --- | --- |
| **Параметры SET** | **Обязательное значение** |
| ANSI\_NULLS | ON |
| ANSI\_PADDING | ON |
| ANSI\_WARNINGS | ON |
| ARITHABORT | ON |
| CONCAT\_NULL\_YIELDS\_NULL | ON |
| NUMERIC\_ROUNDABORT | OFF |
| QUOTED\_IDENTIFIER | ON |

**ON filegroup\_name**

Создает заданный индекс в указанной файловой группе. Если местоположение не указано и таблица или представление не секционированы, индекс использует ту же файловую группу, что и базовая таблица или базовое представление. Файловая группа должна существовать.

**ON "default"**

Создает заданный индекс в файловой группе, используемой по умолчанию.

**<object>::=**

Полное или неполное имя индексируемого объекта.

**<relational\_index\_option>::=**

Указывает параметры, которые могут использоваться при создании индекса.

**PAD\_INDEX = { ON | OFF }**

Определяет разреженность индекса. Значение по умолчанию — OFF.

ON

Процент свободного места, определяемый аргументом **fillfactor**, применяется к страницам индекса промежуточного уровня.

OFF или **fillfactor** не указан

Страницы промежуточного уровня заполняются почти полностью, при этом остается достаточно места по крайней мере для одной строки максимального размера, возможного в этом индексе при заданном наборе ключей на промежуточных страницах.

Параметр PAD\_INDEX имеет смысл только в случае, если указан параметр FILLFACTOR, так как использует процентное значение, указанное в нем. Если процент, заданный аргументом FILLFACTOR, недостаточно велик для размещения одной строки, компонент Database Engine внутренне переопределит это значение, чтобы обеспечить минимум. Количество строк на странице индекса промежуточного уровня никогда не бывает менее двух даже при самых малых значениях аргумента **fillfactor**.

Для обратной совместимости синтаксиса аргумент WITH PAD\_INDEX эквивалентен аргументу WITH PAD\_INDEX = ON.

**FILLFACTOR =fillfactor**

Указывает значение в процентах, показывающее, насколько полным компонент Database Engine должен сделать конечный уровень каждой страницы индекса во время создания или перестроения индекса. Параметр **fillfactor** должен быть целым числом от 1 до 100. Значение по умолчанию равно 0. Если параметр **fillfactor** равен 100 или 0, компонент Database Engine создает индексы с полностью заполненными страницами конечного уровня.

Аргумент FILLFACTOR действует только при создании или перестройке индекса. Компонент Database Engine не сохраняет динамически указанный процентный объем свободного места на страницах. Значение коэффициента заполнения можно увидеть в представлении каталога **sys.indexes**.

**SORT\_IN\_TEMPDB = { ON | OFF }**

Указывает, сохранять ли временные результаты сортировки в базе данных **tempdb**. Значение по умолчанию - **OFF**.

**ON**

Промежуточные результаты сортировки, которые используются при индексировании, хранятся в базе данных **tempdb**. Это может уменьшить время, необходимое для создания индекса, если база данных **tempdb** и база данных пользователя находятся на разных наборах дисков. Однако это увеличивает использование места на диске, которое используется при индексировании.

**OFF**

Промежуточные результаты сортировки хранятся в той же базе данных, где и индекс.

**IGNORE\_DUP\_KEY = { ON | OFF }**

Определяет ответ на ошибку, случающуюся, когда операция вставки пытается вставить в уникальный индекс повторяющиеся значения ключа. Параметр **IGNORE\_DUP\_KEY** применяется только к операциям вставки, производимым после создания или перестроения индекса. Значение по умолчанию - **OFF**.

**ON**

Если в уникальный индекс вставляются повторяющиеся значения ключа, выводится предупреждающее сообщение. С ошибкой завершаются только строки, нарушающие ограничение уникальности.

**OFF**

Если в уникальный индекс вставляются повторяющиеся значения ключа, выводится сообщение об ошибке. Будет выполнен откат всей операции INSERT.

**STATISTICS\_NORECOMPUTE = { ON | OFF }**

Указывает, будет ли выполняться автоматический перерасчет статистики распределения. Значение по умолчанию - **OFF**.

**ON**

Устаревшие статистики не пересчитываются автоматически.

**OFF**

Автоматическое обновление статистических данных включено.

**DROP\_EXISTING = { ON | OFF }**

Указывает, что названный существующий кластеризованный или некластеризованный индекс удаляется и перестраивается. Значение по умолчанию **-** **OFF**.

**ON**

Существующий индекс удаляется и перестраивается. Указанное имя индекса должно совпадать с уже существующим индексом, но определение индекса может быть изменено. Например, можно указать другие столбцы, порядок сортировки, схему секционирования или параметры индекса.

**OFF**

Выдается ошибка, если индекс с указанным именем уже существует.

Тип индекса не может быть изменен с помощью аргумента **DROP\_EXISTING**.

**ONLINE = { ON | OFF }**

Определяет, будут ли базовые таблицы и связанные индексы доступны для запросов и изменения данных во время операций с индексами (только в выпусках SQL Server Enterprise и Developer). Значение по умолчанию - **OFF**.

**ALLOW\_ROW\_LOCKS = { ON | OFF }**

Указывает, разрешена ли блокировка строк. Значение по умолчанию — ON.

**ON**

Блокировки строк допустимы при доступе к индексу. Необходимость в блокировке строк определяет компонент Database Engine.

**OFF**

Блокировки строк не используются.

**ALLOW\_PAGE\_LOCKS = { ON | OFF }**

Указывает, разрешена ли блокировка страниц. Значение по умолчанию - **ON**.

**ON -** блокировки страниц возможны при доступе к индексу. Необходимость в блокировке страниц определяет компонент Database Engine.

**OFF -** блокировки страниц не используются.

**MAXDOP = max\_degree\_of\_parallelism**

**MAXDOP** можно использовать для ограничения числа процессоров, используемых при параллельном выполнении планов. Параллельные операции с индексами доступны только в редакциях Developer, Enterprise и DataCenter. Максимальное число процессоров завис от редакции (Enterprise – 64, DataCenter – 256). Значение по умолчанию 0 - в зависимости от текущей рабочей нагрузки системы использует реальное или меньшее число процессоров.

**Вычисляемые столбцы**

Индексы могут создаваться на вычисляемых столбцах. Вычисляемые столбцы могут иметь свойство PERSISTED - это значит, что компонент Database Engine хранит вычисленные значения в таблице и обновляет их при обновлении любых столбцов, от которых зависит вычисляемый столбец. Компонент Database Engine использует эти сохраняемые значения, когда создает индекс по столбцу и когда запрос обращается к индексу.

Для индексации вычисляемого столбца этот вычисляемый столбец должен быть детерминированным и точным. Детерминированные выражения каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных. Недетерминированные выражения могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных. К точным типам относятся точные числовые типы (целочисленные и десятичные), денежные типы, символьные строки и другие точные типы, перечисленные в модуле 3.

Для материализованных вычисляемых столбцов необходимо, чтобы следующие параметры SET имели значения, указанные выше в разделе «Обязательные параметры SET для индексированных представлений».

Вычисляемые столбцы, производные от типов данных **image**, **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)** и **xml**, могут индексироваться как ключевые или включенные неключевые столбцы, если тип данных вычисляемого столбца приемлем как тип данных для ключевого столбца индекса или неключевого столбца. Если размер ключа индекса превышает 900 байт, выдается предупреждение.

Создание индекса на вычисляемом столбце может привести к ошибке в операциях вставки или обновления, которые до этого успешно выполнялись. Такое неудачное завершение возможно, если вычисляемый столбец вызывает арифметическую ошибку. Например, вычисляемый столбец **"c"** в следующей таблице вызывает арифметическую ошибку, но инструкция INSERT работает.

CREATE TABLE T (a int, b int, c AS a/b);

INSERT INTO T VALUES (1, 0);

Если же после создания таблицы создать индекс на вычисляемом столбце **"c"**, та же инструкция INSERT будет заканчиваться ошибкой.

CREATE TABLE T (a int, b int, c AS a/b);

CREATE UNIQUE CLUSTERED INDEX IdxT ON T(c);

INSERT INTO T VALUES (1, 0);

Примеры:

**Создание простого кластеризованного индекса**

В примере создается некластеризованный индекс по столбцу **CountryID** таблицы **Countries**.

CREATE INDEX IX\_**Countries**\_**CountryID**

ON **Countries** (**CountryID**);

**Создание составного некластеризованного индекса**

В этом примере создается некластеризованный составной индекс по столбцам **SalesQuota** и **SalesYTD** таблицы **Sales.SalesPerson**.

CREATE NONCLUSTERED INDEX IX\_SalesPerson\_SalesQuota\_SalesYTD

ON Sales.SalesPerson (SalesQuota, SalesYTD);

**Создание уникального некластеризованного индекса**

В следующем примере создается уникальный некластеризованный индекс по столбцу **Name** таблицы **Production.UnitMeasure**. Индекс требует уникальности данных, вставляемых в столбец **Name**.

CREATE UNIQUE INDEX AK\_UnitMeasure\_Name

ON Production.UnitMeasure(Name);

Следующий запрос проверяет ограничение уникальности данных при попытке вставить строку с тем же значением, что и в уже существующей строке.

SELECT Name FROM Production.UnitMeasure WHERE Name = N'Ounces';

GO

INSERT INTO Production.UnitMeasure (UnitMeasureCode, Name, ModifiedDate)

VALUES ('OC', 'Ounces', GetDate());

В результате выдается сообщение об ошибке:

Сервер: Сообщение 2601, уровень 14, состояние 1, строка 1

Невозможно вставить повторяющуюся строку ключей в объект UnitMeasure с уникальным индексом AK\_UnitMeasure\_Name. Выполнение данной инструкции было прервано.

**Создание индекса с включенными (неключевыми) столбцами**

В этом примере создается некластеризованный индекс с одним ключевым столбцом (**PostalCode**) и четырьмя неключевыми столбцами (**AddressLine1**, **AddressLine2**, **City**, **StateProvinceID**).

CREATE NONCLUSTERED INDEX IX\_Address\_PostalCode

ON Person.Address (PostalCode)

INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);

## Изменение индексов

Инструкция ALTER INDEX устраняет фрагментацию существующих индексов таблицы либо изменяет их посредством отключения или настройки индексных параметров.

Эта инструкция не может использоваться для изменения определения индекса, в том числе добавления или удаления столбцов или изменения порядка столбцов. Для выполнения этих операций следует использовать инструкцию CREATE INDEX с предложением DROP\_EXISTING.

ALTER INDEX { index\_name | ALL }

    ON <object>

    { REBUILD

        [ [PARTITION = ALL]

          [ WITH ( <rebuild\_index\_option> [ ,...n ] ) ]

          | [ PARTITION = partition\_number

                [ WITH ( <single\_partition\_rebuild\_index\_option>

                        [ ,...n ] )

                ]

            ]

        ]

    | DISABLE

    | REORGANIZE

        [ PARTITION = partition\_number ]

  | SET ( <set\_index\_option> [ ,...n ] )

    }

<object> ::=

{

    [ database\_name. [ schema\_name ] . | schema\_name. ]

        table\_or\_view\_name

}

<rebuild\_index\_option > ::=

{

    PAD\_INDEX = { ON | OFF }

  | FILLFACTOR = fillfactor

  | SORT\_IN\_TEMPDB = { ON | OFF }

  | IGNORE\_DUP\_KEY = { ON | OFF }

  | STATISTICS\_NORECOMPUTE = { ON | OFF }

  | ONLINE = { ON | OFF }

  | ALLOW\_ROW\_LOCKS = { ON | OFF }

  | ALLOW\_PAGE\_LOCKS = { ON | OFF }

  | MAXDOP = max\_degree\_of\_parallelism

  | DATA\_COMPRESSION = { NONE | ROW | PAGE }

     [ ON PARTITIONS ( { <partition\_number\_expression> | <range> }

     [ , ...n ] ) ]

}

<range> ::=

<partition\_number\_expression> TO <partition\_number\_expression>

}

<single\_partition\_rebuild\_index\_option> ::=

{

    SORT\_IN\_TEMPDB = { ON | OFF }

  | MAXDOP = max\_degree\_of\_parallelism

  | DATA\_COMPRESSION = { NONE | ROW | PAGE } }

}

<set\_index\_option>::=

{

    ALLOW\_ROW\_LOCKS = { ON | OFF }

  | ALLOW\_PAGE\_LOCKS = { ON | OFF }

  | IGNORE\_DUP\_KEY = { ON | OFF }

  | STATISTICS\_NORECOMPUTE = { ON | OFF }

}

**index\_name**

Имя индекса, настройки которого требуется изменить.

**ALL**

Указывает все индексы, связанные с таблицей или представлением, независимо от типа индекса.

**database\_name**

Имя базы данных.

**schema\_name**

Имя схемы, к которой принадлежит таблица или представление.

**table\_or\_view\_name**

Имя таблицы или представления, связанного с индексом.

**REBUILD [ WITH (<rebuild\_index\_option> [ ,... n]) ]**

Указывает, что индекс будет перестроен с использованием тех же столбцов, типов индекса, атрибута уникальности и порядка сортировки. **REBUILD** включает отключенный индекс. При перестройке кластеризованного индекса не перестраиваются ассоциированные некластеризованные индексы, если только не указано ключевое слово **ALL**. Если параметры индекса не заданы, то применяется существующий параметр индекса, который хранится в таблице **sys.indexes**. Для любого параметра индекса, значение которого не хранится в таблице **sys.indexes**, применяется значение по умолчанию, указанное в определении аргумента.

При перестроении XML-индекса или пространственного индекса параметры **ONLINE = ON** и **IGNORE\_DUP\_KEY = ON** недопустимы.

Если указано ключевое слово ALL, а базовая таблица реализована в виде кучи, операция перестроения не воздействует на таблицу. Перестраиваются все некластеризованные индексы, ассоциированные с таблицей.

Возможно минимальное протоколирование операции перестроения, если модель восстановления базы данных настроена на массовый или простой режим.

**PARTITION**

Указывает, что только одна секция индекса будет перестроена или реорганизована. PARTITION не может быть указана, если аргумент index\_name — несекционированный индекс.

**PARTITION = ALL**, перестроение всех секций.

**partition\_number**

Номер секции секционированного индекса, который предстоит перестроить или реорганизовать partition\_number. Секция partition\_number должна существовать, иначе выполнение инструкции завершится с ошибкой.

**WITH (<single\_partition\_rebuild\_index\_option>)**

**SORT\_IN\_TEMPDB**, **MAXDOP** и **DATA\_COMPRESSION** - параметры, которые могут быть указаны при перестроении одиночной секции (**PARTITION = n**).

**DISABLE**

Помечает индекс как отключенный и недоступный для использования компонентом Database Engine. Любой индекс может быть отключен, при этом определение отключенного индекса остается в системном каталоге без базовых индексных данных. Отключение кластеризованного индекса блокирует доступ пользователя к данным базовой таблицы. Чтобы активировать индекс, следует использовать инструкцию ALTER INDEX REBUILD или CREATE INDEX WITH DROP\_EXISTING.

**REORGANIZE**

Указывает, что конечный уровень индекса будет реорганизован. Инструкция ALTER INDEX REORGANIZE всегда выполняется в режиме в сети. Это означает, что долгосрочные блокировки таблицы не удерживаются и запросы или обновления базовой таблицы могут продолжаться во время выполнения транзакции ALTER INDEX REORGANIZE. REORGANIZE не может быть вызвана для индекса с ALLOW\_PAGE\_LOCKS со значением OFF.

**SET ( <set\_index option> [ ,... n] )**

Указывает параметры индекса без перестройки или реорганизации индекса. SET нельзя указать для отключенного индекса.

**PAD\_INDEX = { ON | OFF }**

Определяет разреженность индекса. Значение по умолчанию — OFF.

ON

Процент свободного места, определяемый аргументом **fillfactor**, применяется к страницам индекса промежуточного уровня.

OFF или **fillfactor** не указан

Страницы промежуточного уровня заполняются почти полностью, при этом остается достаточно места по крайней мере для одной строки максимального размера, возможного в этом индексе при заданном наборе ключей на промежуточных страницах.

Параметр PAD\_INDEX имеет смысл только в случае, если указан параметр FILLFACTOR, так как использует процентное значение, указанное в нем. Если процент, заданный аргументом FILLFACTOR, недостаточно велик для размещения одной строки, компонент Database Engine внутренне переопределит это значение, чтобы обеспечить минимум. Количество строк на странице индекса промежуточного уровня никогда не бывает менее двух даже при самых малых значениях аргумента **fillfactor**.

Для обратной совместимости синтаксиса аргумент WITH PAD\_INDEX эквивалентен аргументу WITH PAD\_INDEX = ON.

**FILLFACTOR =fillfactor**

Указывает значение в процентах, показывающее, насколько полным компонент Database Engine должен сделать конечный уровень каждой страницы индекса во время создания или перестроения индекса. Параметр **fillfactor** должен быть целым числом от 1 до 100. Значение по умолчанию равно 0. Если параметр **fillfactor** равен 100 или 0, компонент Database Engine создает индексы с полностью заполненными страницами конечного уровня.

Аргумент FILLFACTOR действует только при создании или перестройке индекса. Компонент Database Engine не сохраняет динамически указанный процентный объем свободного места на страницах. Значение коэффициента заполнения можно увидеть в представлении каталога **sys.indexes**.

**SORT\_IN\_TEMPDB = { ON | OFF }**

Указывает, сохранять ли временные результаты сортировки в базе данных **tempdb**. Значение по умолчанию - **OFF**.

**ON**

Промежуточные результаты сортировки, которые используются при индексировании, хранятся в базе данных **tempdb**. Это может уменьшить время, необходимое для создания индекса, если база данных **tempdb** и база данных пользователя находятся на разных наборах дисков. Однако это увеличивает использование места на диске, которое используется при индексировании.

**OFF**

Промежуточные результаты сортировки хранятся в той же базе данных, где и индекс.

**IGNORE\_DUP\_KEY = { ON | OFF }**

Определяет ответ на ошибку, случающуюся, когда операция вставки пытается вставить в уникальный индекс повторяющиеся значения ключа. Параметр **IGNORE\_DUP\_KEY** применяется только к операциям вставки, производимым после создания или перестроения индекса. Значение по умолчанию - **OFF**.

**ON**

Если в уникальный индекс вставляются повторяющиеся значения ключа, выводится предупреждающее сообщение. С ошибкой завершаются только строки, нарушающие ограничение уникальности.

**OFF**

Если в уникальный индекс вставляются повторяющиеся значения ключа, выводится сообщение об ошибке. Будет выполнен откат всей операции INSERT.

**STATISTICS\_NORECOMPUTE = { ON | OFF }**

Указывает, выполнялся ли перерасчет статистики распределения. Значение по умолчанию - **OFF**.

**ON**

Устаревшие статистики не пересчитываются автоматически.

**OFF**

Автоматическое обновление статистических данных включено.

**DROP\_EXISTING = { ON | OFF }**

Указывает, что названный существующий кластеризованный или некластеризованный индекс удаляется и перестраивается. Значение по умолчанию **-** **OFF**.

**ON**

Существующий индекс удаляется и перестраивается. Указанное имя индекса должно совпадать с уже существующим индексом, но определение индекса может быть изменено. Например, можно указать другие столбцы, порядок сортировки, схему секционирования или параметры индекса.

**OFF**

Выдается ошибка, если индекс с указанным именем уже существует.

Тип индекса не может быть изменен с помощью аргумента **DROP\_EXISTING**.

**ONLINE = { ON | OFF }**

Определяет, будут ли базовые таблицы и связанные индексы доступны для запросов и изменения данных во время операций с индексами (только в редакциях SQL Server Enterprise и Developer). Значение по умолчанию - **OFF**.

**ALLOW\_ROW\_LOCKS = { ON | OFF }**

Указывает, разрешена ли блокировка строк. Значение по умолчанию — ON.

**ON**

Блокировки строк допустимы при доступе к индексу. Необходимость в блокировке строк определяет компонент Database Engine.

**OFF**

Блокировки строк не используются.

**ALLOW\_PAGE\_LOCKS = { ON | OFF }**

Указывает, разрешена ли блокировка страниц. Значение по умолчанию - **ON**.

**ON -** блокировки страниц возможны при доступе к индексу. Необходимость в блокировке страниц определяет компонент Database Engine.

**OFF -** блокировки страниц не используются.

**MAXDOP = max\_degree\_of\_parallelism**

**MAXDOP** можно использовать для ограничения числа процессоров, используемых при параллельном выполнении планов. Параллельные операции с индексами доступны только в редакциях Developer, Enterprise и DataCenter. Максимальное число процессоров завис от редакции (Enterprise – 64, DataCenter – 256). Значение по умолчанию 0 - в зависимости от текущей рабочей нагрузки системы использует реальное или меньшее число процессоров.

**DATA\_COMPRESSION**

Задает режим сжатия данных для указанного индекса, номера секции или диапазона секций. Ниже приведены доступные параметры.

**NONE** - индекс или заданные секции не сжимаются.

**ROW** - для индекса или заданных секций производится сжатие строк.

**PAGE** - для индекса или заданных секций производится сжатие страниц.

**ON PARTITIONS ( { <partition\_number\_expression> | <range> } [,...n] )**

Указывает секции, к которым применяется параметр **DATA\_COMPRESSION**. Если индекс не секционирован, аргумент **ON PARTITIONS** создаст ошибку. Если не указано предложение **ON PARTITIONS**, то параметр **DATA\_COMPRESSION** применяется ко всем секциям секционированного индекса.

< **partition\_number\_expression** > можно указать одним из следующих способов:

* Указать номер секции, например ON PARTITIONS (2).
* Указать номера нескольких секций через запятые, например ON PARTITIONS (1, 5).
* Указать диапазоны и отдельные секции: ON PARTITIONS (2, 4, 6 TO 8).

В следующем примере снимается ограничение **PRIMARY KEY** путем отключения индекса **PRIMARY KEY**. Соответствующее первичному ключу ограничение **FOREIGN KEY** в базовой таблице автоматически отключается, и выводится предупреждение.

ALTER INDEX PK\_Department\_DepartmentID

ON Department

DISABLE;

В следующем примере устанавливаются несколько параметров для индекса **AK\_SalesOrderHeader\_SalesOrderNumber**.

ALTER INDEX AK\_SalesOrderHeader\_SalesOrderNumber ON

Sales.SalesOrderHeader

SET ( STATISTICS\_NORECOMPUTE = ON,

IGNORE\_DUP\_KEY = ON,

ALLOW\_PAGE\_LOCKS = ON );

## Восстановление индексов

Компонент SQL Server Database Engine автоматически поддерживает состояние индексов при выполнении операций вставки, обновления или удаления в отношении базовых данных. Со временем эти изменения могут привести к тому, что данные в индексе окажутся фрагментированными (разбросанными по базе данных). Существуют два типа фрагментации:

* внутренняя – внутренняя фрагментация означает неполное заполнение страниц индекса, что приводит к увеличению высоты сбалансированного дерева и, как следствие, росту количества операций ввода-вывода.
* внешняя - имеет место в тех случаях, когда в индексах содержатся страницы, для которых логический порядок, основанный на значении ключа, не совпадает с физическим порядком в файле данных.

Значительно фрагментированные индексы могут серьезно снижать производительность запросов и служить причиной замедления откликов приложения.

Устранить фрагментацию можно путем реорганизации или перестроения индекса. Реорганизация индекса дефрагментирует конечный уровень кластеризованных и некластеризованных индексов по таблицам и представлениям, физически переупорядочивая страницы концевого уровня в соответствии с логическим порядком конечных узлов. При перестроении старый индекс удаляется, и создается новый.

Для секционированных индексов, построенных на основе схемы секционирования, можно использовать любой из этих методов в отношении всего индекса целиком или отдельной его секции.

**Выявление фрагментации**

Первым шагом в определении, какой метод дефрагментации следует использовать, будет анализ индекса на предмет степени фрагментации. Системная функция **sys.dm\_db\_index\_physical\_stats** позволяет выявить фрагментацию конкретного индекса, всех индексов в таблице или индексированном представлении, всех индексов в базе данных или всех индексов во всех базах данных. Для секционированных индексов **sys.dm\_db\_index\_physical\_stats** также предоставляет сведения о фрагментации каждой секции.

Результирующий набор, возвращаемый функцией **sys.dm\_db\_index\_physical\_stats**, включает следующие столбцы:

|  |  |
| --- | --- |
| **Столбец** | **Описание** |
| **avg\_fragmentation\_in\_percent** | Процентная доля логической фрагментации (неупорядоченные страницы в индексе). |
| **fragment\_count** | Число фрагментов (физически последовательные конечные страницы) в индексе. |
| **avg\_fragment\_size\_in\_pages** | Среднее число страниц в одном фрагменте индекса. |

Выяснив степень фрагментации, используя нижеследующую можно выбрать наиболее подходящий метод устранения фрагментации.

|  |  |
| --- | --- |
| **avg\_fragmentation\_in\_percent** | **Корректирующая инструкция** |
| > 5 % и <= 30 % | ALTER INDEX REORGANIZE |
| > 30% | ALTER INDEX REBUILD |

Обычно фрагментация небольших индексов является неконтролируемой. Страницы индексов малого размера хранятся в смешанных экстентах. Смешанные экстенты могут совместно использоваться восемью объектами, поэтому фрагментация в небольшом индексе может не сократиться после реорганизации или перестроения индекса.

**Пример:**

Следующий пример запрашивает через функцию динамического управления **sys.dm\_db\_index\_physical\_stats** среднюю фрагментацию для всех индексов в таблице **Cources**. В соответствии с предыдущей таблицей, рекомендуемым решением проблемы будет реорганизация **IX\_Id\_Unique\_Clustered** и перестроение **IX\_Unique\_Code**.

SELECT a.index\_id, name, avg\_fragmentation\_in\_percent

FROM sys.dm\_db\_index\_physical\_stats (DB\_ID(), OBJECT\_ID(N'Cources'), NULL, NULL, NULL) AS a

JOIN sys.indexes AS b ON a.object\_id = b.object\_id AND a.index\_id = b.index\_id;

GO

Результат

index\_id name avg\_fragmentation\_in\_percent

------------------------------------------------------------------------------------------

1 IX\_Id\_Unique\_Clustered 15.076923076923077

3 IX\_Unique\_Code 99.666666666666657

(2 row(s) affected)

**Реорганизация индекса**

Реорганизация индексов производится при помощи инструкции ALTER INDEX с предложением REORGANIZE. Это предложение эквивалентно **DBCC INDEXDEFRAG**.

Реорганизация индекса дефрагментирует конечный уровень кластеризованных и некластеризованных индексов по таблицам и представлениям, физически переупорядочивая страницы концевого уровня в соответствии с логическим порядком (слева направо) конечных узлов. Расположение страниц в правильном порядке улучшает производительность сканирования индекса. Индекс реорганизуется в пределах выделенных для него страниц, новые страницы не выделяются. Если индекс состоит из более чем одного файла, файлы реорганизуются поочередно. Страницы не перемещаются между файлами.

Кроме того, реорганизация сжимает страницы индекса. Пустые страницы, возникшие в результате этого сжатия, удаляются, освобождая место на диске. Сжатие основывается на коэффициенте заполнения указанном для индекса.

Процесс реорганизации использует минимальные системные ресурсы. Кроме того, реорганизация автоматически осуществляется в режиме в сети. Процесс не требует долговременных блокировок и потому не препятствует выполнению запросов или обновлений.

**Перестроение индексов**

Реорганизация индексов производится при помощи инструкции ALTER INDEX с предложением REBUILD. Это предложение эквивалентно **DBCC DBREINDEX**.

При перестроении старый индекс удаляется, и создается новый. Таким образом, устраняется фрагментация, восстанавливается место на диске путем сжатия страниц с учетом указанного или существующего коэффициента заполнения, переупорядочиваются индексные строки в последовательных страницах. Если указывается ключевое слово ALL, то все индексы для таблицы удаляются и перестраиваются в одной транзакции.

Индекс может быть перестроен как в режиме в сети, так и в режиме вне сети. При перестроении индекса в режиме вне сети таблица, над которой данный индекс определен, будет заблокирована от изменений на время перестроения индекса.

Чтобы добиться доступности, подобной варианту с реорганизацией, следует перестраивать индексы в режиме в сети – это означает, что базовые таблицы и связанные индексы будут доступны для запросов и изменения данных во время операций с индексами. Режим перестроения индекса в сети доступен только в редакциях SQL Server Enterprise и Developer.

При удалении или перестроении больших индексов (более чем из 128 экстентов) Database Engine откладывает фактическое освобождение страниц, избегая блокировки размещения, требуемой для удаления больших объектов, с помощью разделения процесса на две стадии: логическую и физическую.

На логическом этапе существующие единицы распределения помечаются как освобождаемые и блокируются до момента фиксации транзакции, строки данных копируются и сортируются, а затем перемещаются в новые единицы распределения, созданные для хранения перестроенного индекса. При откате происходит откат только данной логической стадии.

Физическая стадия удаления начинается после фиксации транзакции. Единицы распределения, помеченные как освобождаемые, физически удаляются одним пакетом. Эти операции удаления проводятся в коротких транзакциях, выполняемых в фоновом режиме и не требующих большого количества памяти.

В силу того, что физическая стадия начинается после фиксации транзакции, дисковое пространство таблицы или индекса может оставаться недоступным в течение некоторого времени.

**Пример 1:**

В следующем примере перестраивается один индекс в таблице Employee.

ALTER INDEX PK\_Employee\_BusinessEntityID ON HumanResources.Employee

REBUILD;

GO

**Пример 2:**

В следующем примере указывается ключевое слово **ALL**. Тем самым выполняется перестроение всех индексов, связанных с таблицей.

ALTER INDEX ALL ON Production.Product

REBUILD WITH (FILLFACTOR = 80, SORT\_IN\_TEMPDB = ON,

STATISTICS\_NORECOMPUTE = ON);

GO

## Удаление индексов

При удалении некластеризованного индекса его определение удаляется из метаданных, а страницы данных сбалансированного дерева индекса удаляются из файлов базы данных. При удалении кластеризованного индекса определение индекса удаляется из метаданных, а строки данных, которые хранились на конечном уровне кластеризованного индекса, сохраняются в результирующей неупорядоченной таблице - куче. Все пространство, ранее занимаемое индексом, освобождается. Оно может быть впоследствии использовано любым объектом базы данных.

Команда DROP INDEX удаляет один или несколько индексов из текущей базы данных.

DROP INDEX index\_name ON table\_ name

    [ WITH ( <drop\_clustered\_index\_option> [ ,...n ] ) ]

<drop\_clustered\_index\_option> ::=

{

    MAXDOP = max\_degree\_of\_parallelism

    | ONLINE = { ON | OFF }

   | MOVE TO { partition\_scheme\_name ( column\_name )

            | filegroup\_name

            | "default" }

}

**index\_name**

Имя индекса, который необходимо удалить.

**table\_name**

Имя таблицы, на которой создан индекс.

**<drop\_clustered\_index\_option>**

Управляет параметрами кластеризованного индекса. Эти параметры неприменимы к другим типам индексов.

**MAXDOP = max\_degree\_of\_parallelism**

**MAXDOP** можно использовать для ограничения числа процессоров, используемых при параллельном выполнении планов. Параллельные операции с индексами доступны только в редакциях Developer, Enterprise и DataCenter. Максимальное число процессоров завис от редакции (Enterprise – 64, DataCenter – 256). Значение по умолчанию 0 - в зависимости от текущей рабочей нагрузки системы использует реальное или меньшее число процессоров.

**ONLINE = { ON | OFF }**

Определяет, будут ли базовые таблицы и связанные индексы доступны для запросов и изменения данных во время операций с индексами (только в выпусках SQL Server Enterprise и Developer). Значение по умолчанию - **OFF**.

**MOVE TO { partition\_scheme\_name(column\_name) | filegroup\_name | "default" }**

Определяет размещение, куда будут перемещаться строки данных, находящиеся на конечном уровне кластеризованного индекса. Данные перемещаются в новое расположение со структурой типа куча. В качестве нового расположения можно указать файловую группу или схему секционирования, но они должны уже существовать. Параметр **MOVE TO** недопустим для индексированных представлений и некластеризованных индексов. Если ни схема секционирования, ни файловая группа не указаны, результирующая таблица помещается в схему секционирования или файловую группу, которая определена для кластеризованного индекса.

Если кластеризованный индекс удаляется с указанием параметра **MOVE TO**, то все некластеризованные индексы базовых таблиц создаются заново, но остаются в исходных файловых группах или схемах секционирования.

**partition\_scheme\_name(column\_name)**

Указывает схему секционирования, в которой будет размещена результирующая таблица. Схема секционирования должна быть создана заранее. Если схема секционирования не указана и таблица секционирована, таблица включается в ту же схему секционирования, где размещен существующий кластеризованный индекс.

Имя столбца в схеме не обязательно должно соответствовать столбцам из определения индекса. Можно указать любой столбец базовой таблицы.

**filegroup\_name**

Указывает файловую группу, в которую будет помещена результирующая таблица. Если размещение не указано и таблица не секционирована, тогда результирующая таблица включается в ту файловую группу, где размещен существующий кластеризованный индекс. Файловая группа должна существовать.

**"default"**

Указывает размещение по умолчанию для результирующей таблицы.

Следующий пример иллюстрирует удаление индекса **IX\_ProductVendor\_VendorID** в таблице **ProductVendor**.

DROP INDEX IX\_ProductVendor\_BusinessEntityID

ON Purchasing.ProductVendor;

**Лекция № 15**

**Ограничения целостности данных**

## Назначение

Ограничения позволяют задать метод, с помощью которого компонент Database Engine автоматически обеспечивает целостность базы данных. Microsoft SQL Server предоставляет два механизма для реализации бизнес-правил и обеспечения целостности данных: ограничения и триггеры.

## Типы ограничений целостности данных

Планирование и создание таблиц требует указания допустимых значений для столбцов и определения способов принудительного обеспечения целостности данных в них. SQL Server предоставляет следующие механизмы для принудительного обеспечения целостности данных в столбце.

* Определения DEFAULT значений
* Разрешение значений NULL
* Ограничения PRIMARY KEY
* Ограничения FOREIGN KEY
* Ограничения UNIQUE
* Ограничения CHECK

Ограничения задают правила допустимости определенных значений в столбцах и представляют собой стандартный механизм обеспечения целостности. Оптимизатор запросов также использует определения ограничений для построения высокопроизводительных планов выполнения запросов.

### Определение DEFAULT значений

Задает значение по умолчанию, которым Database Engine будет заполнять столбец, если при вставке строки для этого столбца значение не указано. Значение по умолчанию могут быть любым выражением, результат которого является константой, например собственно константой соответствующего типа или встроенной функцией.

Значения по умолчанию создаются с помощью ключевого слова DEFAULT инструкции CREATE TABLE. При этом константное выражение будет присвоено столбцу как значение по умолчанию.

В следующем примере создается таблица, в которой используются различные типы выражений по умолчанию. Здесь объект по умолчанию, назначенный столбцу, создается и связывается со столбцом. Затем производится тестовая вставка данных без указания значений для столбцов со значениями по умолчанию, а также выборка этой тестовой строки для проверки того, что были использованы все значения по умолчанию.

CREATE TABLE test\_defaults

(date\_ins datetime DEFAULT getdate(), --Preferred default definition

mathcol smallint DEFAULT 10 \* 2, --Preferred default definition

char2 char(3) DEFAULT 'xyz') --Preferred default definition;

### Ограничение NOT NULL

Указывает, что в столбце недопустимы значения NULL.

Допустимость значения NULL в столбце определяет, могут ли строки таблицы содержать значения NULL для этого столбца. Значение NULL отличается от нуля (0), пробела или символьной строки нулевой длины, например "". Значение NULL обозначает, что поле не было заполнено. Присутствие NULL, как правило, подразумевает, что значение неизвестно или неопределенно.

Когда в таблицу вставляется строка без указания значения для столбца, допускающего значения NULL, компонент Database Engine присваивает ему значение NULL, если отсутствует определение DEFAULT или объект по умолчанию. Столбец, определенный с помощью ключевого слова NULL, также поддерживает явно заданное пользователем значение NULL, независимо от типа данных и наличия соответствующего значения по умолчанию. Значение NULL не следует заключать в кавычки, так как в этом случае оно будет рассматриваться как символьная строка "NULL".

Запрет значений NULL в столбце обеспечивает целостность данных, гарантируя, что столбец в строке будет всегда содержать данные. Если значения NULL запрещены, то пользователь, заполняя таблицу данными, должен ввести значения в столбцы, иначе строка таблицы не будет записана в базу данных.

### Ограничения CHECK

Обеспечивают целостность домена путем ограничения значений, которые могут быть помещены в столбец.   
Ограничение CHECK задает логическое условие (принимающие значение TRUE, FALSE или unknown), которое применяется ко всем значениям, вставляемым в столбец. Все значения, для которых получается значение FALSE, отбрасываются. Для каждого столбца можно указать несколько ограничений CHECK. Кроме того, можно применять одно проверочное ограничение к нескольким столбцам. Для этого ограничение нужно создать на уровне таблицы.

При задании условия ограничения CHECK допустимо использовать константы, столбцы этой же таблицы и встроенные функции.

В следующем примере показано создание ограничения chk\_id. Это ограничение, среди прочего, следит за соблюдением домена первичного ключа, обеспечивая, чтобы в качестве значений ключа вводились только числа указанного диапазона.

CREATE TABLE cust\_sample

(

cust\_id int PRIMARY KEY,

cust\_name char(50),

cust\_address char(50),

cust\_credit\_limit money,

CONSTRAINT chk\_id CHECK (cust\_id BETWEEN 0 and 10000 )

)

### Ограничения UNIQUE

Обеспечивают уникальность значений в наборе столбцов.   
Ограничение UNIQUE не допускает использования одного и того же значения в двух строках столбца. Допускается использование значения NULL в качестве одного из уникальных значений (как и всякое другое значение столбца с ограничением UNIQUE, NULL может встречаться только один раз).

При создании ограничения UNIQUE создается уникальный некластеризованный индекс для принудительного применения ограничение UNIQUE по умолчанию. Если кластеризованный индекс в таблице еще не создан, то можно указать чтобы создавался уникальный кластеризованный индекс.

Для таблицы можно задать несколько ограничений UNIQUE.

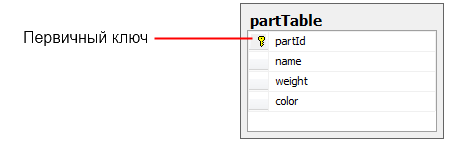
### Ограничения PRIMARY KEY

Используются для указания столбца или набора столбцов, которые имеют значения, уникально идентифицирующие строку в таблице.

Обычно в таблице есть столбец или комбинация столбцов, содержащих значения, уникально определяющие каждую строку таблицы. Такой столбец или сочетание столбцов, называются потенциальными ключами. На основе этих столбцов можно создать первичный ключ, задав ограничение PRIMARY KEY при создании или изменении таблицы.

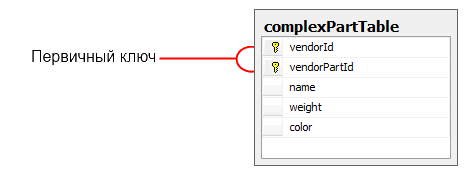
В таблице может быть только одно ограничение PRIMARY KEY, и столбец с данным ограничением не может содержать значения NULL. Из-за того, что ограничения PRIMARY KEY гарантируют уникальность данных, они часто определяются в столбце идентификаторов.

При задании для таблицы ограничения PRIMARY KEY компонент Database Engine гарантирует уникальность данных созданием уникального индекса для столбцов первичных ключей. Этот индекс также обеспечивает быстрый доступ к данным при использовании первичного ключа в запросах. Таким образом, выбранные первичные ключи должны соответствовать правилам создания уникальных индексов.



Если ограничение PRIMARY KEY задано более чем для одного столбца, то значения могут дублироваться в пределах одного столбца, но каждая комбинация значений всех столбцов в определении ограничения PRIMARY KEY должна быть уникальной.

Как показано на следующем рисунке, столбцы **vendorID** и **vendorPartID** таблицы **complexPartTable** формируют составное ограничение PRIMARY KEY для данной таблицы. Это гарантирует уникальность комбинации значений столбцов **vendorID** и **vendorPartID**.



В столбце первичного ключа значения NULL не допускаются.

При создании ограничения PRIMARY KEY автоматически создается уникальный индекс по столбцам первичного ключа. По умолчанию этот индекс кластеризован. но если в таблице уже есть кластеризованный индекс уникальный индекс первичного ключа будет создан как некластеризованный. Также указать, что индекс будет некластеризованным, можно в команде создания первичного ключа, добавив ключевое слово **NONCLUSTERED**.

Согласно принципам построения кластеризованных индексов рассмотренных в модуле 4, рекомендуется использовать небольшой столбец с целочисленными значениями в качестве первичного ключа. Это позволит минимизировать затраты дискового пространства для хранения кластеризованного и всех некластеризованных индексов таблицы, поскольку значения столбцов кластеризованного индекса дублируются в узлах некластеризованных индексов.

Хотя уникальность значений ограничения UNIQUE и PRIMARY KEY гарантируют в равной степени, в случае, когда необходимо обеспечить уникальность в столбце или комбинации столбцов, которые не являются первичными ключевыми, вместо ограничения PRIMARY KEY следует использовать ограничение UNIQUE.

В следующем примере создается таблица **partTable** с указанием поля **partId** в качестве первичного ключа. Индекс, создаваемый по столбцу **partId** первичного ключа, будет некластеризованным.

CREATE TABLE partTable

(partId int PRIMARY KEY NONCLUSTERED,

name char(30),

weight decimal(6,2),

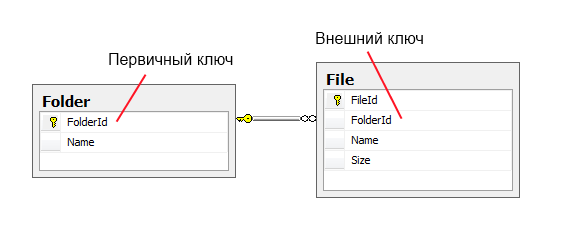
color char(15) );

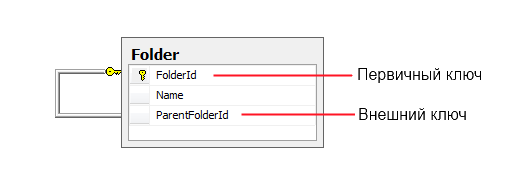
### **Ограничения FOREIGN KEY**

Задают и обеспечивают связи между таблицами. Если один или несколько столбцов, в которых находится первичный ключ для одной таблицы, упоминается в одном или нескольких столбцах другой таблицы, то в ссылке внешнего ключа создается связь между двумя таблицами. Этот столбец становится внешним ключом во второй таблице.

Внешний ключ применяется для принудительного установления связи между данными в двух таблицах. В столбцах, включенных во внешний ключ, могут использоваться только значения из столбцов первичного ключа.

Ограничение FOREIGN KEY может ссылаться на столбцы в таблицах этой же базы данных или столбцы этой же самой таблицы.





Ограничение FOREIGN KEY не обязательно должно быть связано с ограничением PRIMARY KEY в другой таблице. Кроме того, с помощью этого ограничения могут указываться столбцы ограничения UNIQUE в другой таблице.

Ограничение FOREIGN KEY может содержать значения NULL. Впрочем, если любой из столбцов сложного ограничения FOREIGN KEY содержит значения NULL, то при проверке будут пропущены все значения, составляющие ограничение FOREIGN KEY. Чтобы проверялись все значения сложного ограничения FOREIGN KEY, укажите для всех участвующих столбцов параметр NOT NULL.

### Ссылочная целостность

Хотя главная задача ограничения FOREIGN KEY - управление данными, которые могут быть сохранены в таблице внешнего ключа, это ограничение также контролирует изменение данных в таблице первичного ключа.

Ограничение обеспечивает целостность ссылок следующим образом: оно запрещает изменение данных в таблице первичного ключа, если такие изменения сделают недопустимой ссылку в таблице внешнего ключа. Если при попытке удалить строку в таблице первичного ключа или изменить значение этого ключа окажется, что удаленному или измененному значению первичного ключа соответствует значение в ограничении FOREIGN KEY в другой таблице, то действие выполнено не будет. Для успешного изменения или удаления строки с ограничением FOREIGN KEY необходимо сначала удалить данные внешнего ключа в соответствующей таблице либо изменить данные в таблице внешнего ключа, которые связывают внешний ключ с данными другого первичного ключа. Microsoft SQL Server позволяет автоматизировать эти действия, путем указания двух дополнительных предложений **ON DELETE** и **ON UPDATE**.

Предложение **ON DELETE** определяет действия, предпринимаемые при попытке удаления строки, на которую указывает существующий внешний ключ. В предложении ON DELETE предусмотрены следующие параметры:

* **NO ACTION**: удаления не произойдет и будет выведено сообщение об ошибке.
* **CASCADE**: все строки с внешними ключами, указывающими на удаленную строку, также будут удалены.
* **SET NULL**: всем строкам с внешними ключами, указывающими на удаленную строку, присваивается значение NULL.
* **SET DEFAULT**: всем строкам с внешними ключами, указывающим на удаленную строку, присваивается установленное для них значение по умолчанию.

Предложение **ON UPDATE** определяет действия, предпринимаемые при попытке обновления значения потенциального ключа, на которое указывает существующий внешний ключ. Это предложение также поддерживает параметры **NO ACTION**, **CASCADE**, **SET NULL** и **SET DEFAULT**.

### Индексирование ограничений FOREIGN KEY

Необходимость создать индекс внешних ключей часто возникает по следующим причинам:

С помощью ограничений FOREIGN KEY в связанных таблицах проверяются изменения ограничений PRIMARY KEY.

Столбцы внешних ключей часто используются для объединения критериев при соединении данных из связанных таблиц в запросах. Это делается путем сопоставления столбца или столбцов в ограничении FOREIGN KEY в одной таблице с одним или несколькими столбцами первичного или уникального ключей в другой таблице. Индекс позволяет компоненту Database Engine быстро находить связанные данные в таблице внешних ключей. Впрочем, создание индекса не является обязательным. Данные из двух связанных таблиц можно объединять, даже если в таблицах нет соответствующих ограничений PRIMARY KEY или FOREIGN KEY, однако связь внешних ключей между двумя таблицами означает, что эти две таблицы оптимизированы для соединения в запросе, где ключи используются в качестве критериев.

### Количество ограничений FOREIGN KEY в таблице

В SQL Server не предусмотрено ни максимальное количество ограничений FOREIGN KEY в одной таблице (со ссылками на другие таблицы), ни максимальное число ограничений FOREIGN KEY из других таблиц, ссылающихся на одну таблицу. Тем не менее, фактическое максимальное число ограничений FOREIGN KEY ограничено конфигурацией оборудования и структуры базы данных и приложения. Рекомендуется, чтобы таблица содержала не более 253 ограничений FOREIGN KEY, а также чтобы на нее ссылалось не более 253 ограничений FOREIGN KEY. При разработке базы данных и приложений следует учитывать стоимость принудительных ограничений FOREIGN KEY.

Внешний ключ одной таблицы указывает на потенциальный ключ в другой таблице. В следующем примере в таблице **orderTable** создается внешний ключ, который ссылается на таблицу **partTable**, определенную ранее.

CREATE TABLE orderTable

(orderId int,

partId int

FOREIGN KEY REFERENCES partTable(partId)

ON DELETE NO ACTION,

quantity int);

Вставить строку со значением внешнего ключа, для которого отсутствует потенциальный ключ с таким значением, нельзя, за исключением строки со значением NULL.

### Ограничения столбцов и таблиц

Ограничения могут относиться к столбцам или к таблицам. Ограничение столбца указывается в описании столбца и применяется только к данному столбцу. Ограничения, показанные в предыдущих примерах, являются ограничениями столбцов. Ограничение таблицы объявляется независимо от описаний столбцов и может быть применено к нескольким столбцам таблицы. Ограничения таблиц используются при необходимости включить в ограничение несколько столбцов.

Например, если таблица имеет два или более столбцов в первичном ключе, для включения в ограничение всех столбцов первичного ключа необходимо использовать ограничение таблицы.

CREATE TABLE complexPartTable

(

vendorId int,

vendorPartId int,

name char(30),

weight decimal(6,2),

color char(15) ,

CONSTRAINT [PK\_complexPartTable] PRIMARY KEY CLUSTERED

(

vendorId,

vendorPartId

)

)

## Применение триггеров

Триггер DML - это поименованный фрагмент T-SQL кода (хранимая процедура), выполняемый автоматически при возникновении событий языка обработки данных (DML) в базе данных. DML-события возникают при выполнении инструкций INSERT, UPDATE или DELETE, изменяющих данные в указанной таблице или представлении. DML-триггеры могут обращаться к другим таблицам и содержать сложные инструкции Transact-SQL. Триггер и инструкция, при выполнении которой он срабатывает, считаются одной транзакцией, которую можно откатить назад внутри триггера. Также при обнаружении серьезной ошибки (например, нехватки пространства на диске) вся транзакция автоматически откатывается назад.

DML-триггеры удобно использовать в следующих случаях:

* Для каскадных изменений в связанных таблицах базы данных; в большинстве случаев эти изменения можно более эффективно выполнить при помощи каскадных ограничений ссылочной целостности.
* Для предотвращения случайных или неправильных операций INSERT, UPDATE и DELETE и реализации других более сложных ограничений, чем те, которые определены при помощи ограничения CHECK.   
  В отличие от ограничений CHECK, DML-триггеры могут ссылаться на столбцы других таблиц. Например, триггер может использовать инструкцию SELECT для сравнения вставленных или обновленных данных и выполнения других действий, например изменения данных или отображения пользовательского сообщения об ошибке.
* Чтобы оценить состояние таблицы до и после изменения данных и предпринять действия на основе этого различия.

Несколько DML-триггеров одинакового типа (INSERT, UPDATE или DELETE) для таблицы позволяют предпринять несколько различных действий в ответ на одну инструкцию изменения данных.

## Типы триггеров DML

### Триггеры AFTER

Триггеры AFTER выполняются после выполнения инструкций INSERT, UPDATE или DELETE. Предложение AFTER по смыслу эквивалентно предложению FOR, которое является единственно доступным выбором для пользователей более ранних версий Microsoft SQL Server. Триггеры AFTER могут быть определены только в таблицах.

### Триггеры INSTEAD OF

Триггеры INSTEAD OF выполняются вместо обычных действий INSERT, UPDATE или DELETE. Триггеры INSTEAD OF также могут быть определены в представлениях с одной или более базовыми таблицами, где они могут расширить типы поддерживаемых представлением обновлений.

## Рекурсивные триггеры

При выполнении триггером действия, инициирующего другой триггер, триггеры становятся вложенными. Эти действия могут инициировать другие триггеры и т.д. Вложенность триггеров может составлять до 32 уровней. Можно разрешать или запрещать вложенность триггеров AFTER с помощью параметра конфигурации сервера **nested triggers**.Вложенность триггеров INSTEAD OF не зависит от этого параметра.

Если вложенные триггеры разрешены, и триггер в цепочке начинает бесконечный цикл, это превышает предел уровней вложенности и триггер завершается.

Так как триггеры исполняются в пределах транзакции, сбой на любом уровне набора вложенных триггеров приведет к отмене всей транзакции, а также будет выполнен откат всех изменений данных. Чтобы определить, где именно происходит сбой, удобно использовать в триггерах инструкцию PRINT, выводящую дополнительные данные о последовательности выполнения триггеров.

Триггер AFTER не вызывает самого себя рекурсивно, если только не установлен параметр базы данных RECURSIVE\_TRIGGERS.

Существует два типа рекурсии:

* **Прямая рекурсия**   
  Такая рекурсия происходит, когда триггер срабатывает и выполняет действие, вызывающее повторное срабатывание того же триггера. Например, приложение обновляет таблицу **T3**; это вызывает срабатывание триггера **Trig3**. Триггер **Trig3** снова обновляет таблицу **T3**, при этом триггер **Trig3** срабатывает еще раз.   
  В SQL Server 2008 прямая рекурсия может также возникать, когда повторно вызывается тот же триггер, но лишь после того, как вызван триггер другого типа (AFTER или INSTEAD OF). Другими словами, прямая рекурсия триггера INSTEAD OF может возникать, когда один и тот же триггер INSTEAD OF вызывается второй раз, даже если между двумя его вызовами вызывается один или несколько триггеров AFTER. Аналогичным образом прямая рекурсия триггера AFTER может возникать, когда один и тот же триггер AFTER вызывается второй раз, даже если между двумя его вызовами вызывается один или несколько триггеров INSTEAD OF. Например, пусть приложение использует таблицу **T4**. Данное обновление приводит к срабатыванию триггера **Trig4** типа INSTEAD OF. **Trig4** обновляет таблицу **T5**. Данное обновление приводит к срабатыванию триггера **Trig5** типа AFTER. **Trig5** обновляет таблицу **T4**, и это обновление приводит к повторному срабатыванию триггера **Trig4** типа INSTEAD OF. Данная цепь событий считается прямой рекурсией триггера **Trig4**.
* **Косвенная рекурсия**   
  Косвенная рекурсия возникает, когда триггер срабатывает и выполняет действие, которое вызывает срабатывание другого триггера того же типа (AFTER или INSTEAD OF). Второй триггер выполняет действие, вызывающее повторное срабатывание исходного триггера. Другими словами, косвенная рекурсия может возникать, когда триггер INSTEAD OF вызывается второй раз, но лишь после того, как между этими двумя вызовами вызывается другой триггер того же типа INSTEAD OF. Аналогичным образом косвенная рекурсия может возникать, когда триггер AFTER вызывается второй раз, но лишь после того, как между этими двумя вызовами вызывается другой триггер того же типа AFTER. Например, пусть приложение использует таблицу **T1**. Данное обновление приводит к срабатыванию триггера **Trig1** типа AFTER. Триггер **Trig1** обновляет таблицу **T2**; при этом обновлении срабатывает триггер **Trig2** типа AFTER. Триггер **Trig2**, в свою очередь, обновляет таблицу **T1**, что приводит к повторному срабатыванию триггера **Trig1** типа AFTER.

Когда для параметра базы данных RECURSIVE\_TRIGGERS устанавливается значение OFF, предотвращается только прямая рекурсия триггеров AFTER. Чтобы отключить косвенную рекурсию триггеров AFTER, необходимо присвоить параметру сервера **nested triggers** значение **0**.

## Программирование триггеров

Инструкции триггеров используют две особые таблицы: **deleted** (удаленные значения) и **inserted** (вставленные значения). По своей структуре они подобны таблице, на которой определен триггер, то есть таблице, к которой применяется действие пользователя. SQL Server автоматически создает и управляет ими. Эти временные таблицы, находящиеся в оперативной памяти, используются для проверки результатов изменений данных и для установки условий срабатывания триггеров. Изменять данные в этих таблицах или выполнять над ними операции языка DDL, например инструкцию CREATE INDEX, нельзя.

В триггерах DML таблицы **inserted** и **deleted** в основном используются для выполнения следующих операций.

* Расширение ссылочной целостности между таблицами.
* Вставка или обновление данных в базовых таблицах соответствующего представления.
* Проверка на ошибки и принятие соответствующих мер в связи с появлением ошибок.
* Поиск различий между состояниями таблицы до и после изменения данных и принятие соответствующих мер в зависимости от наличия или отсутствия различий.

В таблице **deleted** находятся копии строк, с которыми работали инструкции DELETE или UPDATE. При выполнении инструкции DELETE или UPDATE происходит удаление строк из таблицы триггера и их перенос в таблицу **deleted**. У таблицы **deleted** обычно нет общих строк с таблицей триггера.

В таблице **inserted** находятся копии строк, с которыми работали инструкции INSERT или UPDATE. При выполнении транзакции вставки или обновления происходит одновременное добавление строк в таблицу триггера и в таблицу **inserted**. Строки таблицы **inserted** являются копиями новых строк таблицы триггера.

Транзакция обновления аналогична выполнению операции удаления с последующим выполнением операции вставки; сначала старые строки копируются в таблицу **deleted**, а затем новые строки копируются в таблицу триггера и в таблицу **inserted**.

Следующие правила относятся к инструкциям INSERT или UPDATE, ссылающимся на таблицу с триггером INSTEAD OF, которые должны предоставлять такие значения для столбцов, как если бы в таблице не было триггера INSTEAD OF.

* Не могут быть заданы значения для вычисляемых столбцов и для столбцов с типом данных **timestamp**.
* Если параметр **IDENTITY\_INSERT** для этой таблицы не равен **ON**, то значения для столбцов со свойством **IDENTITY** не могут быть заданы. Когда значение параметра **IDENTITY\_INSERT** равно **ON**, инструкции INSERT должны сами задавать это значение.
* Инструкции INSERT должны определять значения для всех столбцов со свойством **NOT NULL**, не имеющих ограничений **DEFAULT**.
* Для любого столбца, кроме вычисляемых столбцов, **IDENTITY** и столбов типа **timestamp**, определение значений является необязательным, если разрешены **NULL** значения или если какой-либо столбец со свойством **NOT NULL** имеет ограничение **DEFAULT**.

Если инструкция INSERT, UPDATE или DELETE ссылается на представление, для которого определен триггер INSTEAD OF, компонент Database Engine вызывает триггер вместо того, чтобы предпринять какое-либо прямое действие по отношению к таблице. Триггер использует сведения, представленные в таблицах **inserted** и **deleted**, для создания инструкций, необходимых для выполнения требуемых действий в базовых таблицах, даже в том случае, если формат данных в таблицах **inserted** и **deleted**, созданных для представления, отличается от формата данных базовой таблицы.

Формат таблиц **inserted** и **deleted** триггера INSTEAD OF, заданного для представления, совпадает со списком выборки инструкции SELECT, заданной для представления.

Например:

CREATE VIEW EmployeeNames (EmployeeID, LName, FName)

AS

SELECT e.EmployeeID, c.LastName, c.FirstName

FROM AdventureWorks.HumanResources.Employee e

JOIN AdventureWorks.Person.Contact c

ON e.ContactID = c.ContactID

Результирующий набор для данного представления состоит из трех столбцов: одного столбца типа **int** и двух столбцов типа **nvarchar**. Таблицы **inserted** и **deleted** триггера INSTEAD OF, заданного для представления, также содержат столбец типа **int** с именем **EmployeeID**, столбец типа **nvarchar** с именем **LName** и столбец типа **nvarchar** с именем **FName**.

Список выборки представления также может содержать выражения, которые не сопоставлены напрямую с каким-либо одним столбцом базовой таблицы. Некоторые выражения представления, такие как вызов функции или константы, могут не ссылаться на столбцы и просто пропускаться. Сложные выражения могут ссылаться на несколько столбцов, однако таблицы **inserted** и **deleted** содержат только по одному значению для каждой вставляемой строки. Такие же проблемы появляются и в простых выражениях представления, если они ссылаются на вычисляемый столбец со сложным выражением.

Если действия триггера зависят от количества строк, данные в которых были изменены, можно воспользоваться проверками системных переменных (например, проверкой @@ROWCOUNT) .

## Создание триггеров

Триггер DML нельзя создать для временной или системной таблицы, хотя он может ссылаться на временные таблицы. Создание триггеров в базе данных осуществляется командой **CREATE TRIGGER**, данная команда должна быть первой инструкцией пакета; все остальные инструкции пакета интерпретируются как часть определения инструкции **CREATE TRIGGER.**

Несмотря на то, что инструкция TRUNCATE TABLE похожа на инструкцию DELETE без предложения WHERE (которая удаляет все строки), она не приводит к срабатыванию триггеров DELETE, потому что инструкция TRUNCATE TABLE не регистрируется.

Триггеры INSTEAD OF DELETE и INSTEAD OF UPDATE нельзя определить для таблицы, которая имеет внешний ключ, определенный с действием DELETE или UPDATE.

CREATE TRIGGER trigger\_name

ON { table | view }

[ WITH [ ENCRYPTION ] [ EXECUTE AS Clause ] ]

{ FOR | AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

AS { sql\_statement [ ; ] [ ,...n ] }

**trigger\_name**

Имя триггера.

**table | view**

Таблица или представление, в которых выполняется триггер DML. На представление может ссылаться только триггер INSTEAD OF.

**WITH ENCRYPTION**

Затемняет текст инструкции CREATE TRIGGER хранящийся в базе данных. Непривилегированные пользователи не смогут получить исходный текст триггера.

**EXECUTE AS**

Указывает контекст безопасности, в котором выполняется триггер. Позволяет управлять учетной записью пользователя, используемой экземпляром SQL Server для проверки разрешений на любые объекты базы данных, используемые триггером.

**FOR | AFTER**

Тип AFTER указывает, что триггер срабатывает только после успешного выполнения всех операций в инструкции SQL, запускаемой триггером. Все каскадные действия и проверки ограничений, на которые имеется ссылка, должны быть успешно завершены, прежде чем триггер сработает.

Если единственным заданным ключевым словом является FOR, аргумент AFTER используется по умолчанию.

Триггеры AFTER не могут быть определены на представлениях.

**INSTEAD OF**

Указывает, что триггер DML срабатывает вместо инструкции SQL, используемой триггером, переопределяя таким образом действия выполняемой инструкции триггера.

На каждую инструкцию INSERT, UPDATE или DELETE в таблице или представлении может быть определено не более одного триггера INSTEAD OF.

**{ [ DELETE ] [ , ] [ INSERT ] [ , ] [ UPDATE ] }**

Определяет инструкции изменения данных, по которым срабатывает триггер DML, если он применяется к таблице или представлению. Необходимо указать как минимум одну инструкцию. В определении триггера разрешены любые их сочетания в любом порядке.

Для триггеров INSTEAD OF параметр DELETE не разрешен в таблицах, имеющих ограничение целостности внешнего ключа с указанием каскадного действия ON DELETE. Точно так же параметр UPDATE не разрешен в таблицах, имеющих ограничение целостности внешнего ключа с указанием каскадного действия ON UPDATE.

**sql\_statement**

Действия триггера, указанные в инструкциях языка Transact-SQL.

**Пример:**

В следующем примере создается триггер. Этот триггер проверяет уровень кредитоспособности поставщика при попытке добавить новый заказ на покупку в таблицу **PurchaseOrderHeader**. Чтобы получить оценку кредитоспособности поставщика, связанного с заказом на покупку, который только что был добавлен, таблица **inserted** должна ссылаться на таблицу **Vendor** и быть связана с ней. В случае слишком низкой кредитоспособности выводится соответствующее сообщение и вставка не производится.

CREATE TRIGGER LowCredit ON PurchaseOrderHeader

AFTER INSERT

AS

DECLARE @creditrating tinyint,

@vendorid int

SELECT @creditrating = v.CreditRating, @vendorid = p.VendorID

FROM Purchasing.PurchaseOrderHeader AS p

INNER JOIN inserted AS i

ON p.PurchaseOrderID = i.PurchaseOrderID

JOIN Purchasing.Vendor AS v

ON v.VendorID = i.VendorID

IF @creditrating = 5

BEGIN

RAISERROR ('Низкая кредитоспособность покупателя.', 16, 1)

ROLLBACK TRANSACTION

END

GO

## Изменение триггеров

Изменение существующих триггеров в базе данных осуществляется командой **ALTER TRIGGER**.

ALTER TRIGGER trigger\_name

ON { table | view }

[ WITH [ ENCRYPTION ] [ EXECUTE AS Clause ] ]

{ FOR | AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

AS { sql\_statement [ ; ] [ ,...n ] }

Параметры команды **ALTER TRIGGER** идентичны параметрам команды **CREATE TRIGGER**.

Если триггер был создан с опцией шифрования текста инструкции CREATE TRIGGER (**WITH ENCRYPTION**), при изменении триггера, если требуется сохранить текст инструкции в зашифрованном состоянии, должна быть также указана опция **WITH ENCRYPTION**.

В следующем примере изменяется триггер, созданный ранее:

ALTER TRIGGER LowCredit ON PurchaseOrderHeader

AFTER INSERT

AS

DECLARE @creditrating tinyint,

@vendorid int

SELECT @creditrating = v.CreditRating, @vendorid = p.VendorID

FROM Purchasing.PurchaseOrderHeader AS p

INNER JOIN inserted AS i

ON p.PurchaseOrderID = i.PurchaseOrderID

JOIN Purchasing.Vendor AS v

ON v.VendorID = i.VendorID

IF @creditrating = 10

BEGIN

RAISERROR ('Низкая кредитоспособность покупателя.', 16, 1)

ROLLBACK TRANSACTION

END

GO

## Удаление триггеров

Удаление триггеров осуществляется командой **DROP TRIGGER**.

DROP TRIGGER trigger\_name

**trigger\_name**

Имя триггера.

В следующем примере удаляется триггер, созданный ранее:

DROP TRIGGER LowCredit

**Лекция № 16**

**Представления**

## Что такое представление?

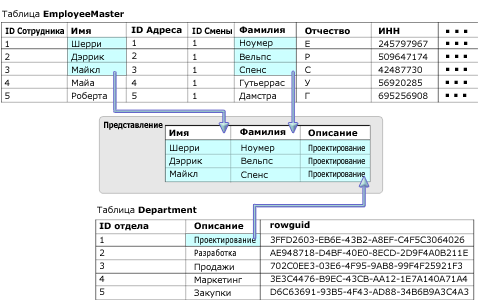
Представление можно считать виртуальной таблицей или хранимым запросом. Если представление не индексировано, его данные не хранятся в базе данных в виде отдельного объекта. В базе данных хранится только инструкция SELECT. Результирующий набор инструкции SELECT формирует виртуальную таблицу, возвращаемую представлением.

Как и настоящая таблица, представление состоит из совокупности именованных столбцов и строк данных. Пока представление не будет проиндексировано, оно не существует в базе данных как хранимая совокупность значений. Строки и столбцы данных извлекаются из таблиц, указанных в определяющем представление запросе и динамически создаваемых при обращениях к представлению.

Представление выполняет функцию фильтра базовых таблиц, на которые оно ссылается. Определяющий представление запрос может быть инициирован в одной или нескольких таблицах или в других представлениях текущей или других баз данных. Кроме того, для определения представлений с данными из нескольких разнородных источников можно использовать распределенные запросы. Это полезно, например, если нужно объединить структурированные подобным образом данные, относящиеся к разным серверам, каждый из которых хранит данные конкретного отдела организации.

На запросы данных посредством представлений не налагаются никакие ограничения; есть только несколько ограничений на изменение данных при помощи представлений.

На следующем рисунке показано представление, созданное на основе двух таблиц.



При выполнении запросов через представление компонент Database Engine проверяет, существуют ли все указанные в инструкции объекты базы данных, верны ли они в контексте инструкции и соответствуют ли инструкции модификации данных правилам обеспечения целостности данных. Если проверка завершается ошибкой, возвращается сообщение об ошибке. При успешной проверке операция преобразуется в операцию над базовой таблицей или таблицами.

Если представление зависит от удаленной таблицы или представления, компонент Database Engine в ответ на попытку использования представления возвращает сообщение об ошибке. Если создана новая таблица или представление, а структура таблицы не изменилась по сравнению с предыдущей базовой таблицей для замены удаленной, то представление можно использовать снова. Если из базовой таблицы были удалены столбцы, используемые в представлении, представление необходимо удалить и создать заново.

## Использование представлений

Представления обычно используются для направления, упрощения и настройки восприятия каждым пользователем информации базы данных. Представления могут использоваться как механизмы безопасности, давая возможность пользователям обращаться к данным через представления, но не предоставляя им разрешений на непосредственный доступ к базовым таблицам, лежащим в основе представлений. Представления могут использоваться для обеспечения интерфейса обратной совместимости, моделирующего таблицу, которая существует, но схема которой изменилась. Представления могут также использоваться для копирования данных на Microsoft SQL Server и с него для повышения производительности и секционирования данных.

**Сосредоточение на определенных данных**

Представления дают возможность пользователям сконцентрироваться на определенных, интересующих их данных и на определенных задачах, за которые они отвечают. Ненужные или конфиденциальные данные могут быть выведены из представления.

**Упрощение обработки данных**

Представления могут упростить работу пользователей с данными. Можно определить часто используемые соединения, перспективы, запросы UNION и SELECT как представления, благодаря чему с пользователей снимается необходимость задавать все условия и характеристики каждый раз при выполнении дополнительной операции на этих данных. Например, сложный запрос, используемый для составления отчетов и выполняющий подзапросы, внешние соединения и статистическую обработку для извлечения данных из группы таблиц, может быть создан как представление. Представление упрощает доступ к данным, так как лежащий в его основе запрос не должен записываться или предъявляться каждый раз при формировании отчета; вместо этого выполняется запрос к представлению.

**Обеспечение обратной совместимости**

Представления дают возможность создать интерфейс, обеспечивающий обратную совместимость с таблицей, в которой изменяется схема.

Например, приложение могло обращаться к ненормализованной таблице, имеющей следующую схему:

Employee(Name, BirthDate, Salary, Department, BuildingName)

Чтобы избежать хранения повторяющихся данных в базе данных, таблица была нормализована и раздалена на следующие две таблицы:

Employee2(Name, BirthDate, Salary, DeptId)

Department(DeptId, BuildingName)

Чтобы обеспечить работу приложений, которые по-прежнему ссылаются на данные из таблицы Employee, пользователь может удалить старую таблицу Employee и заменить ее следующим представлением:

CREATE VIEW Employee AS

SELECT Name, BirthDate, Salary, BuildingName

FROM Employee2 e, Department d

WHERE e.DeptId = d.DeptId

Приложения, которые раньше выполняли запросы к таблице Employee, теперь могут получать свои данные из представления Employee. Если приложение только считывает данные из Employee, то изменять его не нужно. Кроме того, иногда для поддержки приложений, обновляющих таблицу Employee, нужно добавить триггеры INSTEAD OF в новое представление для сопоставления операций INSERT, DELETE и UPDATE в представлении с аналогичными операциями над базовыми таблицами. Как было рассмотрено в модуле 5, SQL Server поддерживает два типа DML-триггеров выполняемые после определенного DML действия или вместо этого действия. Над представлениями можно создать только триггеры, выполняемые вместо DML действия (INSTEAD OF).

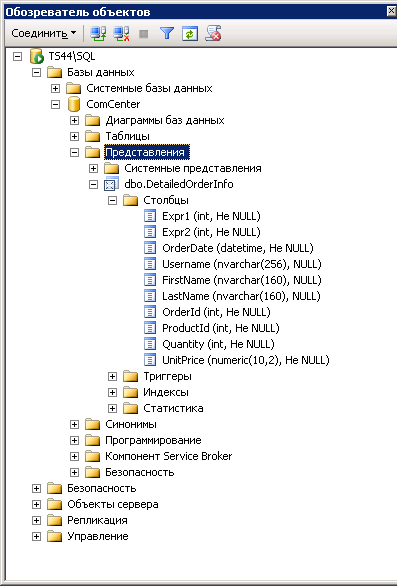
**Настройка данных**

Представление дает пользователям возможность видеть данные различным образом, даже если они используют одни и те же данные в одно и то же время. Это особенно полезно, когда пользователи, имеющие разные интересы и обладающие разным уровнем квалификации, работают с одной и той же базой данных.

Например, может быть создано представление, которое извлекает данные о заказчиках, с которыми работает менеджер. Представление может определять, какие данные необходимо извлечь на основании идентификатора входа менеджера, использующего это представление.

## Источники информации о представлениях

**Обозреватель объектов SQL Server Management Studio**



Представления каталога

|  |  |
| --- | --- |
| **Представление каталога** | **Отображение сведений о** |
| **sys.views** | Информация обо всех представлениях в базе данных |
| **sys.columns** | Информация о столбцах таблиц и представлений, типе данных столбцов |
| **sys.sql\_expression\_dependencies** | Информация о зависимостях между объектами следующих типов:   * привязанные к схеме сущности; * сущности без привязки к схеме; * зависимости на уровне столбцов в сущностях, привязанных к схеме. |
| **sys.sql\_modules** | Содержит текст инструкции SELECT всех представлений |

### Функции динамического управления

|  |  |
| --- | --- |
| **Функция** | **Отображение сведений о** |
| **sys.dm\_sql\_referenced\_entities** | Информация об объектах следующих типов, упоминаемых заданной сущностью:   * сущности, привязанные к схеме; * сущности без привязки к схеме; * зависимости уровня столбца, как связанные, так и не связанные со схемами; * определяемые пользователем типы; * коллекции XML-схем; * функции секционирования. |

## Создание предавлений

Также как и таблица, представление может включать не более 1 024 столбцов.

Создание представления осуществляется при помощи команды **CREATE VIEW**:

CREATE VIEW [ schema\_name . ] view\_name [ (column [ ,...n ] ) ]

[ WITH [ ENCRYPTION ] [ SCHEMABINDING ] ]

AS select\_statement

[ WITH CHECK OPTION ]

**schema\_name**

Имя схемы, которой принадлежит представление.

**view\_name**

Имя представления. Имена представлений должны соответствовать требованиям, предъявляемым к идентификаторам. Указывать имя владельца представления не обязательно.

**column**

Имя, которое будет иметь столбец в представлении. Имя столбца требуется только в тех случаях, когда столбец формируется на основе арифметического выражения, функции или константы, если два или более столбцов могут по иной причине получить одинаковые имена (как правило, в результате соединения) или если столбцу представления назначается имя, отличное от имени столбца, от которого он произведен. Назначать столбцам имена можно также в инструкции SELECT.

Если аргумент column не указан, столбцам представления назначаются такие же имена, которые имеют столбцы в инструкции SELECT.

**AS**

Определяет действия, которые должны быть выполнены в представлении.

**select\_statement**

Инструкция SELECT, которая определяет представление. Представление не обязательно является простым подмножеством строк и столбцов одной конкретной таблицы. С помощью предложения SELECT можно создавать представление, использующее более одной таблицы, или другие представления любой степени сложности.

В аргументе **select\_statement** можно использовать функции и множественные инструкции SELECT, разделенные оператором UNION или UNION ALL.

**CHECK OPTION**

Обеспечивает соответствие всех выполняемых для представления инструкций модификации данных критериям, заданным при помощи аргумента **select\_statement**. Если строка изменяется посредством представления, предложение WITH CHECK OPTION гарантирует, что после фиксации изменений доступ к данным из представления сохранится, т.е. строка не может быть изменена таким образом, что перестанет подпадать под критерий выборки. Любые обновления, произведенные непосредственно в базовых таблицах представления, не проверяются в контексте представления — даже в том случае, если указано предложение CHECK OPTION.

**ENCRYPTION**

Затемняет текст инструкции CREATE VIEW хранящийся в базе данных. Непривилегированные пользователи не смогут получить исходный текст триггера.

**SCHEMABINDING**

Привязывает представление к схеме базовой таблицы или таблиц. Если аргумент **SCHEMABINDING** указан, нельзя изменить базовую таблицу или таблицы таким способом, который может повлиять на определение представления. Сначала нужно изменить или удалить само представление для сброса зависимостей от таблицы, которую требуется изменить. При использовании аргумента SCHEMABINDING инструкция **select\_statement** должна включать двухкомпонентные (schema***.***object) имена таблиц, представлений или пользовательских функций, упоминаемых в предложении. Все указанные в инструкции объекты должны находиться в одной базе данных.

Как было рассмотрено в начале модуля, если представление зависит от удаленной таблицы или представления, компонент Database Engine в ответ на попытку использования представления возвращает сообщение об ошибке. Также ошибка произойдет при попытке использования представления, если из базовой таблицы были удалены столбцы, используемые в представлении. SQL Server не препятствует такому изменению или удалению таблиц, участвующих в представлении, и ошибка будет обнаружена только при обращении к представлению. Указание аргумента SCHEMABINDING позволяет избежать подобных ошибок, поскольку представления или таблицы, входящие в представление, созданное при помощи предложения SCHEMABINDING, не могут быть удалены, пока это представление не будет удалено или изменено таким образом, чтобы оно более не было привязано к схеме. В противном случае компонент Database Engine выдаст ошибку. Кроме того, выполнение инструкций ALTER TABLE для таблиц, которые входят в представления, привязанные к схемам, завершается ошибкой, если эти инструкции влияют на определение представления.

Аргумент SCHEMABINDING нельзя указывать, если представление содержит столбцы с псевдонимами типов данных.

**Пример:**

В следующем примере создается представление **EmployeeHireDate**, содержащее фамилии и имена всех сотрудников, а также даты их приема на работу.

CREATE VIEW EmployeeHireDate

AS

SELECT p.FirstName, p.LastName, e.HireDate

FROM HumanResources.Employee AS e JOIN Person.Person AS p

ON e.BusinessEntityID = p.BusinessEntityID ;

## Изменение представлений

Инструкция ALTER VIEW позволяет изменять ранее созданные представления, и имеет синтаксис аналогичный инструкции создания представления CREATE VIEW.

ALTER VIEW [ schema\_name . ] view\_name [ ( column [ ,...n ] ) ]

[ WITH [ ENCRYPTION ] [ SCHEMABINDING ] ]

AS select\_statement

[ WITH CHECK OPTION ]

Если предыдущее определение представления было создано с использованием предложения WITH ENCRYPTION или CHECK OPTION, эти параметры будут действовать только в том случае, если они включены в инструкцию ALTER VIEW.

**Пример:**

В следующем примере представление **EmployeeHireDate**, содержащее фамилии и имена всех сотрудников, а также даты их приема на работу, модифицируется таким образом, чтобы из базы данных извлекались сведения только о сотрудниках, принятых на работу до 2009 года.

ALTER VIEW HumanResources.EmployeeHireDate

AS

SELECT p.FirstName, p.LastName, e.HireDate

FROM HumanResources.Employee AS e JOIN Person.Person AS p

ON e.BusinessEntityID = p.BusinessEntityID

WHERE HireDate < CONVERT(DATETIME,'20090101',101) ;

GO

## Удаление представлений

Удаление представлений из базы производится инструкцией **DROP VIEW**:

DROP VIEW [ schema\_name . ] view\_name [ ...,n ]

**schema\_name**

Имя схемы, к которой принадлежит представление.

**view\_name**

Имя удаляемого представления.

**Пример:**

В следующем примере удаляется представление **EmployeeHireDate** из реляционной схемы **HumanResources**.

DROP VIEW HumanResources.EmployeeHireDate;

## Обновление данных в представлении

Можно изменять данные базовой таблицы через представление до тех пор, пока выполняются следующие условия:

* Любые изменения, в том числе инструкции UPDATE, INSERT и DELETE, должны ссылаться на столбцы только одной базовой таблицы.
* Изменяемые в представлении столбцы должны непосредственно ссылаться на данные столбцов базовой таблицы. Столбцы нельзя сформировать каким-либо другим образом, в том числе:
  + при помощи агрегатной функции: AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR и VARP;
  + на основе вычисления. Столбец нельзя вычислить по выражению, включающему другие столбцы. Столбцы, сформированные при помощи операторов UNION, UNION ALL, CROSSJOIN, EXCEPT и INTERSECT, считаются вычисляемыми и также не являются обновляемыми.
* Предложения GROUP BY, HAVING и DISTINCT не влияют на изменяемые столбцы.
* Предложение TOP не используется нигде в инструкции **SELECT** представления вместе с предложением WITH CHECK OPTION.

Вышеназванные ограничения относятся ко всем подзапросам представления в предложении FROM, равно как и к самому представлению. То есть, компонент Database Engine должен иметь возможность однозначно проследить изменения от определения представления до одной базовой таблицы.

Если вышеуказанные ограничения не позволяют изменить данные через представление напрямую, для него можно создать триггеры INSTEAD OF. Триггер INSTEAD OF выполняется вместо инструкции модификации данных, для которой он определен. Этот триггер позволяет пользователю указать набор действий, которые должны быть выполнены для обработки инструкции модификации данных. Таким образом, если для представления создан триггер INSTEAD OF, связанный с конкретной инструкцией модификации данных (INSERT, UPDATE или DELETE), соответствующее представление можно обновлять при помощи этой инструкции.

## Типы представлений

### Стандартные представления

Сочетание данных из одной или нескольких таблиц с помощью стандартного представления позволяет использовать почти все преимущества представлений. Сюда входит фокусировка на конкретных данных и упрощение управления ими. Эти преимущества подробнее описываются в разделе Сценарии для использования представлений.

### Индексированные представления

Индексированным называется материализованное представление. Это значит, что оно вычислено и сохранено. Индексировать представление можно, создав для него уникальный кластеризованный индекс. Индексированные представления значительно повышают производительность некоторых типов запросов. Индексированные представления эффективнее всего использовать в запросах, группирующих множество строк. Они не очень хорошо подходят для часто обновляющихся базовых наборов данных.

### Секционированные представления

Секционированным называется представление, соединяющее горизонтально секционированные данные набора базовых таблиц, находящихся на одном или нескольких серверах. При этом данные выглядят так, как будто находятся в одной таблице. Объединение объединения результатов нескольких запросов из отдельных таблиц в единый результирующий набор осуществляется при помощи оператора языка T-SQL UNION.

Представление, соединяющее базовые таблицы одного экземпляра SQL Server, называется локальным секционированным представлением. Если представление соединяет данные с разных серверов, оно называется распределенным секционированным представлением. Распределенные секционированные представления используются для реализации федераций серверов баз данных. Федерация - это группа серверов, которые администрируются независимо, но при этом взаимодействуют с целью разделения нагрузки на систему. Формирование федерации серверов баз данных секционированием данных – это механизм, позволяющий распределить нагрузку на набор серверов для удовлетворения требований обработки данных больших многоуровневых веб-узлов.

Если таблицы расположены на разных серверах или на компьютере с несколькими процессорами, все таблицы, участвующие в запросе, могут просматриваться параллельно. Это может повысить производительность выполнения запроса. Наряду с этим, задачи обслуживания, такие как повторное создание индексов или резервное копирование таблицы, могут выполняться гораздо быстрее.

## Секционированные представления

При конструировании секционированных представлений исходная таблица заменяется несколькими меньшими таблицами-элементами. Каждая таблица-элемент и исходная таблица имеют одинаковое количество столбцов, у каждого столбца такие же атрибуты, как и у соответствующего ему столбца в исходной таблице (тип данных, размер, параметры сортировки). При создании распределенного секционированного представления каждая таблица-элемент находится на отдельном сервере, запросы на выборку данных из распределенного секционированного представления будут затрагивать только те сервера, на которых лежат данные подпадающие под условие запроса. Таким образом, при правильной организации распределенного секционированного представления, нагрузка на отдельные сервера может быть существенно снижена.

Создание секционированного представления состоит из нескольких этапов.

### Создание таблиц-элементов

Таблицы-элементы конструируются так, чтобы в каждой таблице хранился горизонтальный фрагмент исходной таблицы, основанный на интервале значений ключа. Интервалы основаны на значениях данных в столбце секционирования. Диапазон значений в каждой таблице-элементе определяется проверочным ограничением на столбце секционирования, и интервалы не пересекаются. Например, нельзя, чтобы в одной таблице был интервал от 1 до 200000, а в другой – от 15000 до 300000, иначе будет неясно, какая таблица содержит значения от 15000 до 200000.

Например, таблица **Customer** расположенная на **Server1** секционируется на три таблицы **Customers\_33**, **Customers\_66** и **Customers\_99**, расположенные на **Server1**, **Server2** и **Server3** соответственно. Для этих таблиц будет задано следующее проверочное ограничение:

-- На Server1:

CREATE TABLE Customers\_33

(CustomerID INTEGER PRIMARY KEY

CHECK (CustomerID BETWEEN 1 AND 32999),

... -- прочие колонки)

-- На Server2:

CREATE TABLE Customers\_66

(CustomerID INTEGER PRIMARY KEY

CHECK (CustomerID BETWEEN 33000 AND 65999),

... -- прочие колонки)

-- На Server3:

CREATE TABLE Customers\_99

(CustomerID INTEGER PRIMARY KEY

CHECK (CustomerID BETWEEN 66000 AND 99999),

... -- прочие колонки)

### Определение распределенных секционированных представлений

После создания таблиц-элементов необходимо определить собственно секционированное представление, которое будет обеспечивать доступ к тем же данным, что и исходная таблица. При этом нагрузка на каждый из серверов, содержащих таблицы-элементы, будет снижена.

Для предыдущего примера на **Server1** определяется распределенное секционированное представление **Customers**. Это позволяет запросам, которые ранее использовали таблицу **Customers**, продолжать работать без каких-либо изменений. Распределенное секционированное представление поможет распределить нагрузку между тремя серверами при доступе к данным представления.

-- для Server1

CREATE VIEW Customers AS

SELECT \* FROM CompanyDatabase.TableOwner.Customers\_33

UNION ALL

SELECT \* FROM Server2.CompanyDatabase.TableOwner.Customers\_66

UNION ALL

SELECT \* FROM Server3.CompanyDatabase.TableOwner.Customers\_99

### Правила таблиц

Таблицы-элементы определяются в предложении FROM в каждой инструкции SELECT в определении представления. Каждая таблица-элемент должна соответствовать следующим правилам:

* На таблицы-элементы нельзя ссылаться в представлении больше одного раза.
* У таблиц-элементов не может быть индексов, созданных на любых вычисляемых столбцах.
* Все ограничения первичного ключа таблиц-элементов должны быть на одинаковом количестве столбцов.
* У таблиц-элементов должна быть одна и та же настройка ANSI\_PADDING.

### Правила столбцов

Столбцы определяются в списке выборки каждой инструкции SELECT в определении представления. Каждый столбец должен соответствовать следующим правилам.

* Все столбцы в каждой таблице-элементе должны быть включены в список выборки. Допустимый синтаксис: SELECT \* FROM <таблица-элемент>.
* На столбцы нельзя ссылаться в списке выборки больше одного раза.
* В списке выборки у столбцов должен быть одинаковый порядок следования.
* Тип столбцов в списке выборки каждой инструкции SELECT должен совпадать. Это относится к типам данных, точности, масштабу и параметрам сортировки. Например, следующее определение представления будет ошибочным, поскольку первый столбец в обоих выражениях SELECT имеет разный тип данных:

CREATE VIEW NonUpdatable

AS

SELECT IntPrimaryKey, IntPartNmbr

FROM FirstTable

UNION ALL

SELECT NumericPrimaryKey, IntPartNmbr

FROM SecondTable

### Правила столбцов секционирования

Для секционирования может использоваться только один столбец, и он должен присутствовать в каждой таблице-элементе. Проверочные ограничения определяют данные, доступные в каждой таблице-элементе. Применяются следующие дополнительные правила:

* Диапазон ключа проверочных ограничений в таблице не должен пересекаться с диапазонами любой другой таблицы. Любое значение столбца секционирования должно быть сопоставленным только с одной таблицей. Проверочные ограничения могут использовать только следующие операторы: BETWEEN, IN, AND, OR, <, <=, >, >=, =.
* Столбец секционирования не может быть столбцом с автоматическим приращением, столбцом со значением по умолчанию или столбцом типа **timestamp**.
* Столбец секционирования должен находиться в одинаковом порядковом расположении в списках выборки всех инструкций SELECT в представлении.
* Столбец секционирования не может содержать значения NULL.
* Столбец секционирования должен быть частью первичного ключа таблицы.
* Столбец секционирования не может быть вычисляемым столбцом.
* У столбца секционирования может быть только одно ограничение. Если задано несколько ограничений, то SQL Server игнорирует их все и не учитывает их при выяснении того, является ли это представление секционированным.
* Ограничения для обновления столбца секционирования отсутствуют.

Столбец секционирования, соответствующий всем этим правилам, будет поддерживать все оптимизации, которые поддерживаются оптимизатором запросов.

**Условия изменения данных в секционированных представлениях**

На инструкции, изменяющие секционированные представления, распространяются следующие ограничения:

* В инструкции INSERT должны быть указаны значения для всех столбцов представления, даже если в базовых таблицах-элементах действует ограничение DEFAULT для этих столбцов или они поддерживают значения NULL. Для тех столбцов таблиц-элементов, которые имеют значения по умолчанию, в инструкциях нельзя явно использовать ключевое слово DEFAULT.
* Значение, вставляемое в столбец секционирования, должно отвечать хотя бы одному из базовых ограничений; в противном случае операция вставки завершится ошибкой из-за нарушения ограничений.
* В предложении SET инструкции UPDATE в качестве значения не может быть указано ключевое слово DEFAULT, даже если столбец имеет значение DEFAULT, определенное в соответствующей таблице-элементе.
* Столбцы PRIMARY KEY не могут быть изменены при помощи инструкции UPDATE, если таблицы-элементы включают столбцы типов **text**, **ntext** или **image**.
* Столбец представления, который является столбцом идентификаторов в одной или нескольких таблицах-элементах, не может быть изменен при помощи инструкции INSERT или UPDATE.
* Если одна из таблиц-элементов содержит столбец **timestamp**, представление не может быть изменено при помощи инструкции INSERT или UPDATE.
* Если одна из таблиц-элементов содержит триггер, ограничение ON UPDATE CASCADE/SET NULL/SET DEFAULT или ограничение ON DELETE CASCADE/SET NULL/SET DEFAULT, то представление не может быть изменено.
* Выполнение операций INSERT, UPDATE и DELETE для секционированного представления не допускается, если осуществляется самосоединение с тем же представлением или с какой-либо из таблиц-элементов, указанных в инструкции.
* Массовый импорт данных в секционированное представление не поддерживается программой **bcp** и инструкциями BULK INSERT и INSERT... SELECT \* FROM OPENROWSET(BULK...).

**Детерминированные и недетерминированные функции**

* [SQL Server 2008](http://msdn.microsoft.com/ru-ru/library/ms178091(v=SQL.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/ru-ru/library/ms178091(v=SQL.90).aspx)

Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных.

Определяемые пользователями функции имеют несколько свойств, определяющих способность ядра SQL Server Database Engine индексировать результаты функции как с помощью индексов вычисляемых столбцов, вызывающих функцию, так и с помощью индексированных представлений, которые на нее ссылаются. Детерминизм функции - одно из таких свойств. Например, в представлении нельзя создать кластеризованный индекс, если оно ссылается на какие-либо недетерминированные функции.

Детерминизм встроенных функций

На детерминизм встроенных функций повлиять нельзя. Каждая из них детерминирована или недетерминирована в зависимости от реализации в SQL Server.

Все статистические и строковые встроенные функции детерминированы

Следующие встроенные функции, отличные от функций статистических и строковых, всегда детерминированы.

|  |  |  |
| --- | --- | --- |
| ABS | DATEDIFF | POWER |
| ACOS | DAY | RADIANS |
| ASIN | DEGREES | ROUND |
| ATAN | EXP | SIGN |
| ATN2 | FLOOR | SIN |
| CEILING | ISNULL | SQUARE |
| COALESCE | ISNUMERIC | SQRT |
| COS | LOG | TAN |
| COT | LOG10 | YEAR |
| DATALENGTH | MONTH |  |
| DATEADD | NULLIF |  |

Следующие функции не всегда детерминированы, их можно использовать в индексированных представлениях или индексах вычисляемых столбцов, только если они заданы детерминированным образом.

|  |  |
| --- | --- |
| CAST | Детерминирована, кроме случаев использования с **datetime**, **smalldatetime** или **sql\_variant**. |
| CONVERT | Детерминирована, кроме следующих случаев:   * Исходный тип - **sql\_variant**. * Конечный тип - **sql\_variant**, и его исходный тип недетерминирован. * Исходный или конечный тип — **datetime** или **smalldatetime**, другой исходный или конечный тип — строка символов, и задан недетерминированный стиль. Чтобы быть детерминированным, параметр стиля должен быть константой. |
| CHECKSUM | Детерминирована, за исключением CHECKSUM(\*). |
| ISDATE | Детерминирована, только если используется с функцией CONVERT, при этом задан аргумент стиля CONVERT. |
| RAND | Функция RAND детерминирована, только если аргумент **seed** определен. |

Все функции конфигурации, курсора, метаданных, безопасности и системные статистические - недетерминированные.

Функции, вызывающие расширенные хранимые процедуры, недетерминированы, так как расширенные хранимые процедуры могут оказать на базу данных побочное действие. Побочные действия - это такое изменение глобального состояния базы данных, как обновление таблицы, внешнего сетевого ресурса или файла.

## Индексированные представления

Представления часто называют виртуальными таблицами, потому что возвращаемый представлением результирующий набор имеет такую же общую форму, как и таблица со столбцами и строками, и потому что в инструкциях SQL ссылаться на представления можно так же, как на таблицы. Результирующий набор стандартного представления не хранится в базе данных длительное время. Каждый раз, когда в запросе упоминается стандартное представление, SQL Server подставляет в запрос определение представления до тех пор, пока не будет сформирован запрос, ссылающийся только на базовые таблицы. После этого итоговый запрос выполняется как обычно.

Если работа стандартного представления связана со сложной обработкой большого количества строк, например со статистической обработкой больших объемов данных или соединением многих строк, трата ресурсов на динамическое создание результирующего набора при каждом запросе может оказаться довольно существенной. Если такие представления часто упоминаются в запросах, можно повысить производительность системы, создав для представления уникальный кластеризованный индекс. В этом случае результирующий набор будет храниться в базе данных подобно таблице с кластеризованным индексом.

Другое преимущество создания индекса для представления заключается в том, что оптимизатор будет использовать этот индекс в запросах, в которых представление не упоминается в предложении FROM непосредственно. Извлечение данных из индексированного представления может привести к повышению эффективности уже имеющихся запросов, при этом их не нужно переписывать.

Индексированные представления наиболее эффективны, если данные базовых таблиц обновляются довольно редко. Расходы на поддержание индексированного представления могут оказаться более высокими, чем расходы, связанные с поддержанием индекса таблицы. Если базовые данные обновляются часто, расходы на поддержание данных индексированного представления могут существенно перевесить преимущества от его использования. Если базовые данные обновляются периодически и пакетами, а в интервалах между операциями обновления используются преимущественно только для чтения, в некоторых случаях имеет смысл перед обновлением удалять все индексированные представления, а после обновления - создавать их заново. Это может повысить эффективность обновлений.

Индексированные представления повышают эффективность запросов следующих типов:

* Соединения и статистические вычисления, в ходе которых обрабатывается большое число строк.
* Соединения и статистические вычисления, которые часто выполняются несколькими запросами.   
  Например, при работе с базой данных OLTP, хранящей информацию о материально-технических запасах, скорее всего, потребуется часто соединять таблицы **ProductMaster**, **ProductVendor** и **VendorMaster**. Даже если в каждом запросе с соединением обрабатывается небольшое число строк, общая трата ресурсов на выполнение сотен тысяч таких запросов может оказаться существенной. Так как подобные связи обычно обновляются нечасто, общую производительность всей системы можно повысить, определив индексированное представление, хранящее результаты операций соединения.
* Запросы систем поддержки принятия решений.   
  Одной из особенностей аналитических систем является то, что в них хранятся статистические данные, обновляемые сравнительно редко. Во многих запросах систем поддержки принятия решений выполняется дополнительная статистическая обработка данных и соединение многих строк. Кроме того, системы поддержки принятия решений иногда содержат крупные таблицы с большим количеством крупных столбцов. Эффективность запросов, ссылающихся на подмножество этих столбцов, можно повысить, создав индексированное представление, включающее только те столбцы, которые упоминаются в запросе, или слегка расширенное надмножество этих столбцов. Создание представлений с узким индексом, содержащих подмножество столбцов одной таблицы, называется стратегией **вертикального секционирования**, потому что при этом таблица делится по вертикали.

Рассмотрим, например, следующую таблицу и индексированное представление:

CREATE TABLE wide\_tbl(a int PRIMARY KEY, b int, ..., z int)

CREATE VIEW v\_abc WITH SCHEMABINDING AS

SELECT a, b, c

FROM dbo.wide\_tbl

WHERE a BETWEEN 0 AND 1000

CREATE UNIQUE CLUSTERED INDEX i\_abc ON v\_abc(a)

Чтобы обработать следующий запрос, достаточно представления **v\_abc**:

SELECT b, count\_big(\*), SUM(c)

FROM wide\_tbl

WHERE a BETWEEN 0 AND 1000

GROUP BY b

Представление **v\_abc** занимает гораздо меньше страниц, чем таблица **wide\_tbl**. Таким образом, при разрешении данного запроса оптимизатор, наиболее вероятно, выберет представление, а не таблицу.   
Однако предпочтительным способов вертикального секционирования таблицы или ее подмножества является не индексированное представление, а некластеризованный индекс таблицы, указавывающий столбцы, которые должны войти в результат, при помощи предложения INCLUDE.

Эффективность запросов следующих типов индексированные представления обычно не повышают:

* Запросы систем OLTP, включающие большое число операций записи.
* Операции над часто обновляемыми базами данных.
* Запросы, не включающие статистических выражений или операций соединения.
* Статистическая обработка данных с большим количеством элементов ключа GROUP BY. Если количество элементов ключа велико, это означает, что ключ содержит много разных значений. Уникальный ключ имеет максимально возможное количество элементов, потому что каждый ключ имеет свое значение. Индексированные представления повышают эффективность работы с такими ключами, уменьшая число обрабатываемых строк. Если результирующий набор представления включает почти столько же строк, сколько содержится в базовой таблице, выгода от использования представления невелика.

Например, следующий запрос данных из таблицы, включающей 1 000 строк:

SELECT PriKey, SUM(SalesCol)

FROM ExampleTable

GROUP BY PriKey

Если количество элементов ключа таблицы равно 100, индексированное представление, созданное на основе результатов этого запроса, включало бы только 100 строк. При работе с данным представлением для выполнения запросов требовалось бы в среднем в десять раз меньше операций чтения данных из базовой таблицы. Если бы ключ этой таблицы был уникальным, его количество элементов было бы равно 1000 и результирующий набор представления **содержал** бы 1 000 строк. Если представление и базовая таблица включают строки одинакового размера, использование индексированного представления вместо чтения данных непосредственно из базовой таблицы выгоды не приносит.

* Соединения с расширением. К этой категории относятся представления, результирующие наборы которых превышают объем данных в базовых таблицах.

## Создание индексированных представлений

Перед созданием кластеризованного индекса для представления оно должно удовлетворять следующим требованиям.

* При выполнении инструкции CREATE VIEW параметры ANSI\_NULLS и QUOTED\_IDENTIFIER должны быть установлены в ON. Для представлений эти сведения доступны через функцию OBJECTPROPERTY со свойствами **ExecIsAnsiNullsOn** и **ExecIsQuotedIdentOn**.
* При создании инструкцией CREATE TABLE таблиц, на которые ссылается представление, параметр ANSI\_NULLS должен быть установлен в ON.
* Представление не должно ссылаться ни на какие другие представления, ссылки должны указывать только на базовые таблицы.
* Все базовые таблицы, на которые ссылается представление, должны находиться в той же базе данных, что и представление, и иметь того же владельца, что и представление.
* Представление должно быть создано с параметром SCHEMABINDING. Это позволяет привязать представление к схеме базовых таблиц.
* Пользовательские функции, на которые ссылается представление, должны быть созданы с параметром SCHEMABINDING.
* Таблицы и пользовательские функции, на которые ссылается представление, должны указываться по именам из двух элементов. Имена из одного, трех или четырех элементов недопустимы.
* Все функции, на которые ссылаются выражения в представлении, должны быть детерминированными.
* Неявное преобразование символьных данных не в Юникоде между различными параметрами сортировки также считается недетерминированным, если только не установлен уровень совместимости 80 или менее.   
  При уровне совместимости 90 создание индексов представлений, содержащих эти выражения, недопустимо. Это, однако, не относится к существующим представлениям, содержащим такие выражения из обновленной базы данных. Если используются индексированные представления, содержащие неявное преобразование строк в дату, необходимо убедиться в том, что значения параметров LANGUAGE и DATEFORMAT в базах данных и приложениях согласованы, чтобы избежать возможности повреждения индексированных представлений.
* Если в определении представления используется статистическая функция, список SELECT должен также включать в себя COUNT\_BIG (\*).
* Свойство доступа к данным пользовательской функции должно быть установлено в NO SQL, а свойство внешнего доступа в NO.
* Инструкция SELECT в представлении не может содержать следующие синтаксические элементы языка Transact-SQL.
  + Синтаксис \* или table\_name**.**\* для задания столбцов. Имена столбцов должны быть указаны явно.
  + Имя столбца, указываемое в качестве простого выражения, не может быть задано более чем в одном столбце представления. Ссылки на столбец могут выполняться несколько раз, при условии, что все (или все, кроме одной) ссылки на столбец являются частью сложного выражения или параметром функции.

Например, следующий список выборки недопустим:

SELECT ColumnA, ColumnB, ColumnA

Следующий список выборки допустим:

SELECT SUM(ColumnA) AS SumColA, ColumnA % ColumnB AS ModuloColAColB, COUNT\_BIG(\*) AS cBig FROM dbo.T1 GROUP BY ModuloColAColB

* + Выражение для столбца, указанного в предложении GROUP BY, или выражение для результата выполнения статистической функции.
  + Операторы UNION, EXCEPT или INTERSECT.
  + Вложенные запросы.
  + Внешние соединения или самосоединение.
  + Предложение TOP.
  + Предложение ORDER BY.
  + Ключевое слово DISTINCT.
  + COUNT (COUNT\_BIG(\*) допустимо).
  + Статистические функции AVG, MAX, MIN, STDEV, STDEVP, VAR или VARP.
  + Функция SUM, ссылающаяся на выражение, результатом которого может быть значение NULL.

Первым индексом, создаваемым для представления, должен быть уникальный кластеризованный индекс. После этого могут быть созданы дополнительные некластеризованные индексы. При именовании индексов представлений применяются те же соглашения, что и для индексов таблиц. Единственная разница состоит в том, что имя таблицы заменяется именем представления.

Инструкция CREATE INDEX должна удовлетворять перечисленным ниже требованиям, а также обычным требованиям для CREATE INDEX.

* Пользователь, выполняющий инструкцию CREATE INDEX, должен быть владельцем представления.
* При выполнении инструкции CREATE INDEX должны быть установлены в значение ON следующие параметры SET.
  + ANSI\_NULLS
  + ANSI\_PADDING
  + ANSI\_WARNINGS
  + CONCAT\_NULL\_YIELDS\_NULL
  + QUOTED\_IDENTIFIER
* Параметр NUMERIC\_ROUNDABORT должен быть установлен в OFF. Это установка по умолчанию.
* Если база данных работает при уровне совместимости 80 или ниже, то параметр ARITHABORT должен быть установлен в значение ON.
* При создании кластеризованного или некластеризованного индекса параметр IGNORE\_DUP\_KEY должен быть установлен в OFF (установка по умолчанию).
* Представление не может содержать столбцы типа **text**, **ntext** или **image**, если даже на них нет ссылок в инструкции CREATE INDEX.
* Если инструкция SELECT в определении представления содержит предложение GROUP BY, ключ уникального кластеризованного индекса может ссылаться только на столбцы, которые заданы в предложении GROUP BY.
* Выражение с потерей точности, формирующее значение ключевого столбца индекса, должно ссылаться на хранимый столбец в базовой таблице данного представления. Этот столбец может быть либо обычным, либо материализованным вычисляемым столбцом. Никакие другие выражения с потерей точности не могут быть частью ключевого столбца индексированного представления.

После создания кластеризованного индекса любое соединение, которое пытается изменить базовые данные представления, должно иметь такие же установки параметров, какие необходимы для создания индекса. Если соединение, выполняющее инструкцию, имеет неверные установки параметров, SQL Server выдает ошибку и выполняет откат инструкций INSERT, UPDATE или DELETE, которые выполняются в отношении результирующего набора представления.

При удалении представления удаляются также и все его индексы. При удалении кластеризованного индекса удаляются все некластеризованные индексы и автоматически созданные для представления статистики. Статистики, созданные пользователем, сохраняются. Некластеризованные индексы могут удаляться по отдельности. При удалении кластеризованного индекса представления удаляется сохраненный результирующий набор, и оптимизатор снова начинает работать с ним, как с обычным представлением.

Хотя в инструкции CREATE UNIQUE CLUSTERED INDEX задаются только столбцы, образующие кластеризованный индексный ключ, в базе данных хранится весь результирующий набор представления. Как и в кластеризованном индексе базовой таблицы, структура сбалансированного дерева кластеризованного индекса содержит только ключевые столбцы, а в результирующем наборе представления строки данных содержат все столбцы.

Если необходимо добавить индексы к представлениям в существующей системе, то необходимо привязать к схеме представление, для которого создается индекс. Это можно сделать следующим образом.

* Удалить и вновь создать представление с указанием WITH SCHEMABINDING.
* Создать второе представление с тем же текстом, что и в существующем представлении, но под другим именем. Оптимизатор пользуется индексами нового представления даже в том случае, если в запросе предложение FROM на него непосредственно не ссылается.

Новое представление должно удовлетворять всем требованиям, предъявляемым к индексированным представлениям.

В следующем примере создаются представление и индекс этого представления. Включено два запроса, использующих созданное индексированное представление.

--Установка опций необходимых для создания индексированного представления

SET NUMERIC\_ROUNDABORT OFF;

SET ANSI\_PADDING, ANSI\_WARNINGS, CONCAT\_NULL\_YIELDS\_NULL, ARITHABORT, QUOTED\_IDENTIFIER, ANSI\_NULLS ON;

GO

--Создание представления с опцией SCHEMABINDING

CREATE VIEW Sales.vOrders

WITH SCHEMABINDING

AS

SELECT SUM(UnitPrice\*OrderQty\*(1.00-UnitPriceDiscount)) AS Revenue,

OrderDate, ProductID, COUNT\_BIG(\*) AS COUNT

FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o

WHERE od.SalesOrderID = o.SalesOrderID

GROUP BY OrderDate, ProductID;

GO

--Создание уникального кластеризованного индекса для представления

CREATE UNIQUE CLUSTERED INDEX IDX\_V1

ON Sales.vOrders (OrderDate, ProductID);

GO

**Лекция № 17**

**Хранимые процедуры и функции**

## Преимущества хранимых процедур

Хранимая процедура — это сохраненная коллекция инструкций языка Transact-SQL, которая может принимать и возвращать предоставленные пользователем параметры.

Использование хранимых процедур обеспечивает следующие преимущества:

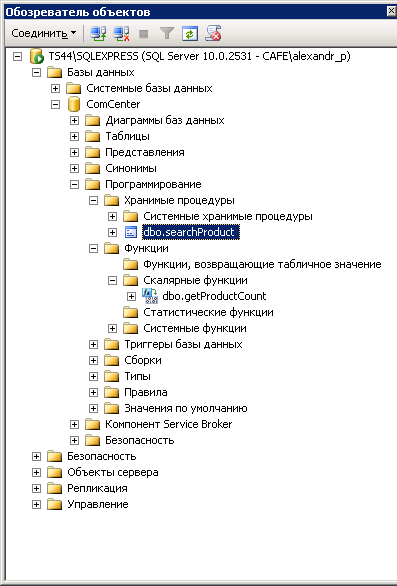
* **Делают возможным модульное программирование.**   
  Можно, однажды создав хранимую процедуру, сохранить ее в базе данных, а затем любое число раз вызывать из своей программы.
* **Позволяют ускорить выполнение.**   
  Хранимые процедуры снижают стоимость компиляции кода Transact-SQL, кэшируя и повторно используя планы выполнения. Это означает, что для хранимых процедур нет необходимости выполнять повторный синтаксический анализ и оптимизацию при каждом вызове, что значительно ускоряет их выполнение.
* **Позволяют уменьшить сетевой трафик.**   
  Вместо передачи на сервер тестов больших запросов Transact-SQL, запросы можно оформить в виде хранимых процедур. При этом для их выполнения будут передаваться только имена процедур и значения параметров.

Хранимые процедуры нельзя использовать в запросах SELECT напрямую.

Хранимые процедуры могут быть вложенными - одна хранимая процедура может вызывать другую. Уровень вложенности увеличивается на 1, когда начинается выполнение вызванной процедуры, и уменьшается на 1, когда вызванная процедура завершается. Уровень вложенности хранимых процедур может достигать 32.

## Источники информации о хранимых процедурах

### Обозреватель объектов SQL Server Management Studio



### Представления каталога

|  |  |
| --- | --- |
| **Представление каталога** | **Отображение сведений о** |
| **sys.procedures** | Информация о хранимых процедурах. |
| **sys.parameters** | Информация о параметрах функции. |
| **sys.sql\_modules** | Позволяет просмотреть определение хранимой процедуры, то есть инструкции Transact-SQL, которые были использованы для ее создания. |
| **sys.sql\_expression\_dependencies** | Информация о зависимостях между объектами следующих типов:   * привязанные к схеме сущности; * сущности без привязки к схеме; * зависимости на уровне столбцов в сущностях, привязанных к схеме. |

### Функции динамического управления

|  |  |
| --- | --- |
| **Функция** | **Отображение сведений о** |
| **sys.dm\_sql\_referenced\_entities** | Информация об объектах следующих типов, упоминаемых заданной сущностью:   * сущности, привязанные к схеме; * сущности без привязки к схеме; * зависимости уровня столбца, как связанные, так и не связанные со схемами; * определяемые пользователем типы; * коллекции XML-схем; * функции секционирования. |

## Создание хранимых процедур

Создание хранимой процедуры осуществляется вызовом команды **CREATE PROCEDURE**.

При этом можно создавать хранимые процедуры, ссылающиеся на еще не существующие объекты (таблицы, представления и т.д.). Во время создания хранимой процедуры выполняется только проверка синтаксиса. Сама хранимая процедура не компилируется до первого обращения к ней. Ссылки на все используемые в хранимой процедуре объекты разрешаются только во время компиляции. Таким образом, ничто не мешает создать синтаксически корректную хранимую процедуру, ссылающуюся на несуществующие таблицы, однако если эти таблицы будут отсутствовать во время выполнения хранимой процедуры, она завершится с ошибкой.

CREATE { PROC | PROCEDURE } procedure\_name

    [

{ @parameter data\_type }

[ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]

    ] [ ,...n ]

[ WITH [ ENCRYPTION ] [ RECOMPILE ] [ EXECUTE AS ]]

AS

[ BEGIN ] <statements> [ END ]

**procedure\_name**

Имя новой хранимой процедуры. Bмя хранимой процедуры не может иметь длину более 128 символов.

**@parameter**

Параметр процедуры. Имя параметра всегда должно начинаться со знака @. В одной инструкции CREATE PROCEDURE можно объявить один или более параметров. При выполнении процедуры значение каждого из объявленных параметров должно быть указано пользователем, если для параметра не определено значение по умолчанию или значение не задано равным другому параметру. Хранимая процедура может иметь не более 2 100 параметров.

Параметры являются локальными в пределах процедуры; в разных процедурах могут быть использованы одинаковые имена параметров. По умолчанию параметры могут использоваться только в качестве константных выражений; они не могут быть использованы вместо имен таблиц, столбцов или других объектов базы данных.

**data\_type**

Тип данных параметра. Все типы данных Transact-SQL могут использоваться в качестве параметра хранимой процедуры. Также можно использовать определяемый пользователем табличный тип, чтобы объявить входной параметр хранимой процедуры Transact-SQL. Параметры табличного типа могут быть только **входными** и должны сопровождаться ключевым словом **READONLY**. Тип данных **cursor** можно использовать только в качестве **выходного** параметра. При указании типа данных **cursor** нужно также указать ключевые слова VARYING и OUTPUT. Выходных параметров типа **cursor** может быть несколько.

**VARYING**

Указывает результирующий набор, поддерживаемый в качестве выходного параметра. Этот параметр динамически формируется хранимой процедурой, и его содержимое может различаться. Применяется только к параметрам типа **cursor**.

**default**

Значение по умолчанию для параметра. Если значение **default** определено, процедуру можно выполнить без указания значения соответствующего параметра. Значение по умолчанию должно быть константой или может равняться NULL. Если в процедуре используется параметр с ключевым словом LIKE, он может включать символы-шаблоны %, \_, [] и [^].

**OUTPUT**

Показывает, что параметр процедуры является выходным. Значение этого параметра можно получить после вызова хранимой процедуры. Выходные параметры используются для возврата значений коду, вызвавшему процедуру. Определяемый пользователем табличный тип не может быть указан в качестве выходного параметра хранимой процедуры.

**READONLY**

Указывает, что параметр не может быть обновлен или изменен в тексте процедуры. Если тип параметра является определяемым пользователем табличным типом, ключевое слово **READONLY** должно быть указано.

**RECOMPILE**

Показывает, что компонент Database Engine не кэширует план выполнения процедуры и что процедура компилируется во время выполнения.

**ENCRYPTION**

Показывает, что SQL Server выполнит затемнение исходного текста инструкции CREATE PROCEDURE. Результат затемнения не виден непосредственно ни в одном из представлений каталогов SQL Server. Пользователи, не имеющие доступа к системным таблицам или файлам баз данных, не смогут получить скрытый текст. Однако этот текст будет доступен привилегированным пользователям.

**EXECUTE AS**

Определяет контекст безопасности, в котором должна быть выполнена хранимая процедура, т.е. какую учетную запись компонент Database Engine использует при проверке разрешений на объекты, на которые ссылается хранимая процедура. Это повышает гибкость и безопасность управления разрешениями на цепочки владения между пользовательскими хранимыми процедурами и объектами, на которые они ссылаются.

Пользователям необходимо будет предоставлять только разрешения на саму хранимую процедуру, без выдачи явных разрешений на объекты, на которые он ссылается. Только пользователь, от имени которого выполняется модуль, должен будет иметь разрешения на объекты, к которым этот модуль обращается.

Допустимые значения **EXECUTE AS { CALLER | SELF | OWNER | 'user\_name' }**.

**CALLER**

Является значением по умолчанию и указывает, что инструкции, содержащиеся в хранимой процедуре, выполняются в контексте пользователя, ее вызывающего. Пользователь, выполняющий хранимую процедуру, должен иметь соответствующие разрешения не только на саму хранимую процедуру, но также и на объекты базы данных, на которые имеются ссылки из нее.

**SELF**

EXECUTE AS SELF эквивалентно EXECUTE AS **user\_name**, где указанный пользователь - это тот, кто создает или изменяет модуль.

**OWNER**

Указывает, что инструкции, содержащиеся в хранимой процедуре, выполняются в контексте текущего владельца этой хранимой процедуре.

**'user\_name'**

Указывает, что инструкции, содержащиеся в хранимой процедуре, выполняются в контексте пользователя, указываемого аргументом **user\_name**. Разрешения на объекты, на которые ссылается хранимая процедура, проверяются для пользователя **user\_name**.

Пользователь **user\_name** должен присутствовать в текущей базе данных и не должен относиться к учетной записи группы. В качестве аргумента **user\_name** нельзя указывать роль, сертификат, ключ или встроенную учетную запись (например, NT AUTHORITY\LocalService, NT AUTHORITY\NetworkService или NT AUTHORITY\LocalSystem).

**<statements>**

Одна или несколько инструкций языка Transact-SQL, которые будут включены в состав процедуры.

Инструкцию CREATE PROCEDURE нельзя объединять с другими инструкциями Transact-SQL в одном пакете.

По умолчанию параметры могут принимать значения NULL. Если параметр, имеющий значение NULL, используется в инструкции CREATE TABLE или ALTER TABLE при обращении к столбцу, не поддерживающему значения NULL, то компонент Database Engine возвращает ошибку. Чтобы предотвратить передачу значений NULL столбцу, который их не поддерживает, следует реализовать в процедуре соответствующую логику или передать столбцу значение по умолчанию при помощи ключевого слова DEFAULT инструкции CREATE TABLE или ALTER TABLE.

Каждая хранимая процедура имеет возвращаемое значение типа **int**. Возвращаемое значение можно передать вызывающему коду при помощи конструкции **RETURN value**. Возвращаемое значение можно использовать совместно с выходными параметрами, например, для индикации успешности вызова процедуры.

При создании или изменении хранимой процедуры Transact-SQL компонент Database Engine сохраняет значения **SET QUOTED\_IDENTIFIER** и **SET ANSI\_NULLS**. Эти первоначальные значения используются при выполнении хранимой процедуры. Таким образом, пока хранимая процедура выполняется, любые значения **SET QUOTED\_IDENTIFIER** и **SET ANSI\_NULLS**, задаваемые во время клиентского сеанса, не учитываются. Другие параметры **SET**, такие как **SET ARITHABORT** или **SET ANSI\_PADDINGS**, при создании или изменении хранимой процедуры не сохраняются. Если логика хранимой процедуры зависит от конкретного значения параметра, следует включить инструкцию **SET** в начало процедуры, чтобы гарантировать нужное значение. Если инструкция **SET** выполняется из хранимой процедуры, устанавливаемое ею значение действует только до завершения хранимой процедуры. После этого оно принимает значение, которое имело место при вызове хранимой процедуры. Это позволяет клиентам задавать нужные им параметры без влияния на логику хранимой процедуры.

## Вызов хранимых процедур

Выполнение созданной хранимой процедуры осуществляется вызовом команды EXEC с указанием хранимой процедуры в качестве аргумента:

EXEC procedure\_name

Если в хранимой процедуре предусмотрена обработка параметров, можно указать значения аргументов.

Предоставленное значение должно быть константой или переменной; нельзя указывать имя функции в качестве значения аргумента. Переменные могут быть пользовательскими или системными, например **@@spid**.

EXEC procedure\_name value1, value2 ...

**Примеры:**

В следующих примерах демонстрируется передача значений аргументов хранимой процедуре uspSaleProduct. Процедура ожидает значений для двух входных аргументов: кода продукта и даты.

**Пример 1:**

Передача констант в качестве значений параметров

EXEC dbo.uspSaleProduct 819, '20050225';

**Пример 2:**

Передача переменных в качестве значений параметров

DECLARE @ProductID int, @SaleDate datetime;

SET @ProductID = 819;

SET @SaleDate = '20050225';

EXEC dbo.uspSaleProduct @ProductID, @SaleDate;

**Пример 3:**

Передача имени функции в качестве значений параметров. Данный вызов порождает ошибку, поскольку имя функции нельзя указывать в качестве значения аргумента

EXEC dbo.uspSaleProduct 819, GETDATE();

**Пример 3:**

Передача результата вызова функции в качестве значения параметра, используя переменные

DECLARE @SaleDate datetime;

SET @SaleDate = GETDATE();

EXEC dbo.uspSaleProduct 819, @SaleDate;

Если необходимо указывать параметры в порядке, отличном от определенного в хранимой процедуре, их необходимо именовать, то есть указывать в виде **@аргумент =** **значение**. Можно также опустить аргументы, для которых представлены значения по умолчанию. При указании одного аргумента в виде **@аргумент = value** необходимо предоставить все последующие аргументы тем же способом.

EXEC procedure\_name @arg1=value1, @arg2= value2 ...

При выполнении хранимой процедуры сервер отвергает любые аргументы, не включенные в список аргументов в процессе создания процедуры. Любой аргумент, переданный ссылкой (явно передающей имя аргумента) не принимается, если имя аргумента не совпадает.

**Примеры:**

В следующих примерах демонстрируется передача значений аргументов хранимой процедуре uspSaleProduct в виде **@аргумент = value**. Процедура ожидает значений для двух входных аргументов: кода продукта и даты.

**Пример 1:**

Передача параметров в порядке следования в команде **CREATE PROCEDURE**

EXEC dbo.uspSaleProduct @SaleDate='20050225', @ProductID=819;

**Пример 2:**

Передача параметров в обратном порядке

EXEC dbo.uspSaleProduct @ProductID=819, @SaleDate='20050225';

Хотя разрешается опустить аргументы, для которых предоставлены значения по умолчанию, список аргументов можно лишь подвергнуть усечению (т.е. не передавать значения параметров идущих в конце списка параметров и имеющих DEFAULT значение). Например, когда в хранимой процедуре присутствует пять аргументов, можно опустить четвертый и пятый, но нельзя пропустить четвертый и включить пятый, если только аргументы не были предоставлены в формате **@аргумент =** **value**.

Значение по умолчанию, определенное для аргумента в хранимой процедуре, используется, когда:

* не указано значение для аргумента при выполнении хранимой процедуры;
* в качестве значения для аргумента указывается ключевое слово DEFAULT.

## Перекомпиляция хранимых процедур

После изменения базы данных из-за добавления индексов или изменения данных в индексируемых столбцах нужно заново выполнить оптимизацию исходных планов запросов, использовавшихся для доступа к таблицам, с помощью перекомпиляции. Оптимизация проводится автоматически во время первого выполнения хранимой процедуры после перезапуска Microsoft SQL Server. Она также проводится при изменении базовой таблицы, используемой хранимой процедурой. Но если добавляется индекс, предоставляющий хранимой процедуре преимущества, оптимизация не проводится до следующего выполнения этой процедуры после перезапуска Microsoft SQL Server. В этой ситуации будет полезно выполнить принудительную перекомпиляцию при следующем запуске хранимой процедуры.

Другая причина для принудительного перекомпилирования хранимой процедуры - это нейтрализации, если это необходимо, пробного сохранения параметров при компиляции процедуры. Когда SQL Server выполняет хранимые процедуры, значения всех используемых ими параметров при компиляции включаются как часть формируемого плана запроса. Если эти значения типичны для последующих вызовов процедуры, компиляция и выполнение хранимой процедуры с этим планом запроса происходит быстрее. Иначе производительность может снизиться.

SQL Server предоставляет три способа вызвать принудительную перекомпиляцию хранимой процедуры:

* Выполнение системной хранимой процедуры **sp\_recompile** приводит к перекомпиляции хранимой процедуры при следующем ее запуске.
* Создание хранимой процедуры с параметром WITH RECOMPILE в определении указывает, что SQL Server не будет кэшировать план этой процедуры; хранимая процедура будет перекомпилироваться при каждом запуске. Параметр WITH RECOMPILE полезен, если хранимая процедура принимает параметры, чьи значения сильно меняются между ее выполнениями, что приводит к созданию новых планов выполнения каждый раз. Этот параметр используется редко и замедляет выполнение хранимых процедур, так как они должны перекомпилироваться при каждом своем запуске.
* Если нужно перекомпилировать не всю хранимую процедуру, а только отдельные запросы, можно указать подсказку RECOMPILE внутри каждого запроса, требующего перекомпиляции. При перекомпиляции инструкций с помощью подсказки RECOMPILE кроме текущих значений параметров процедуры также используются значения локальных переменных внутри хранимой процедуры. Этот режим используется, если нетипичные или временные значения встречаются только в части запросов, входящих в хранимую процедуру.
* Принудительно перекомпилировать хранимую процедуру можно, задав параметр WITH RECOMPILE при ее выполнении. Используйте этот параметр, только если предоставляемые параметры нетипичны или если с момента создания хранимой процедуры данные изменились существенно.

## Преимущества определяемых пользователем функций

Подобно функциям языков программирования, определяемые пользователем функции Microsoft SQL Server являются подпрограммами, которые принимают параметры, выполняют действия, такие как сложные вычисления, и затем возвращают результат выполнения в виде значения. Возвращаемое значение может быть либо единичным скалярным значением, либо результирующим набором.

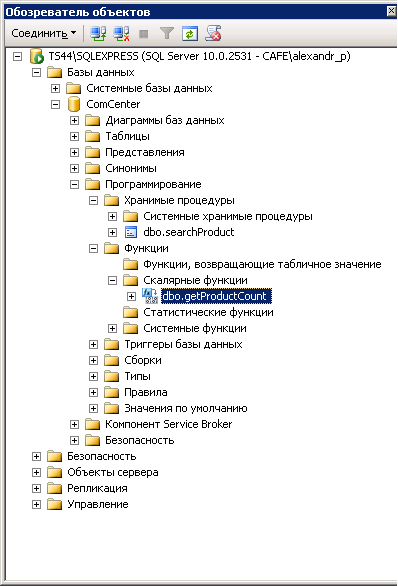
SQL Server 2008 поддерживает определяемые пользователем функции и встроенные (системные) функции.

Использование определяемых пользователем функций SQL Server обеспечивает следующие преимущества.

* **Делают возможным модульное программирование.**   
  Можно, однажды создав функцию, сохранить ее в базе данных, а затем любое число раз вызывать из своей программы. Определяемые пользователем функции могут быть изменены независимо от исходного кода программы.
* **Позволяют ускорить выполнение.**   
  Как и хранимые процедуры, определяемые пользователем функции Transact-SQL снижают стоимость компиляции кода Transact-SQL, кэшируя и повторно используя планы выполнения. Это означает, что для определяемых пользователем функций нет необходимости выполнять повторный синтаксический анализ и оптимизацию при каждом вызове, что значительно ускоряет их выполнение.
* **Позволяют уменьшить сетевой трафик.**   
  Операция, которая фильтрует данные на основе какого-нибудь сложного ограничения и не может быть выражена одним скалярным выражением, может быть реализована в виде функции. Ее можно вызвать из предложения WHERE, чтобы уменьшить число строк, возвращаемых клиенту.

## Источники информации о пользовательских функциях

### Обозреватель объектов SQL Server Management Studio



### Представления каталога

|  |  |
| --- | --- |
| **Представление каталога** | **Отображение сведений о** |
| **sys.objects** | Информация о функциях. |
| **sys.parameters** | Информация о параметрах функции. |
| **sys.sql\_modules** | Позволяет просмотреть определение функции, то есть инструкции Transact-SQL, которые были использованы для создания определяемой пользователем функции. |
| **sys.sql\_expression\_dependencies** | Информация о зависимостях между объектами следующих типов:   * привязанные к схеме сущности; * сущности без привязки к схеме; * зависимости на уровне столбцов в сущностях, привязанных к схеме. |

### Функции динамического управления

|  |  |
| --- | --- |
| **Функция** | **Отображение сведений о** |
| **sys.dm\_sql\_referenced\_entities** | Информация об объектах следующих типов, упоминаемых заданной сущностью:   * сущности, привязанные к схеме; * сущности без привязки к схеме; * зависимости уровня столбца, как связанные, так и не связанные со схемами; * определяемые пользователем типы; * коллекции XML-схем; * функции секционирования. |

## Типы определяемых пользователем функции

Любая определяемая пользователем функция состоит из двух частей: заголовка и текста. Функция принимает ноль и более параметров и возвращает либо скалярное значение, либо таблицу.

Заголовок определяет:

* имя функции с необязательным именем схемы или владельца;
* имя и тип данных параметров функции;
* тип данных возвращаемого значения и необязательное имя;

Текст определяет действие или логику, которую выполняет функция, и может содержать одну или несколько инструкций Transact-SQL, реализующих логику функции.

Пользовательские функции MS SQL Server подразделяются на 3 категории:

* Скалярные функции
* Возвращающие табличное значение функции
* Встроенные функцииСкалярные функции

Пользовательские скалярные функции возвращают одно значение типа данных, заданного в предложении RETURNS. Встроенная скалярная функция не имеет тела, скалярное значение является результатом одной инструкции. Скалярная функция из нескольких инструкций имеет тело, ограниченное блоком BEGIN...END, и содержит последовательность инструкций Transact-SQL, возвращающих одно значение. Такие функции могут возвращать любые типы данных, кроме **text**, **ntext**, **image**, **cursor** и **timestamp**.

CREATE FUNCTION function\_name

(

[ { @parameter\_name [ AS ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

]

)

RETURNS return\_data\_type

    [ WITH [ ENCRYPTION ] [ SCHEMABINDING ] [ EXECUTE AS Clause ]]

    [ AS ]

    BEGIN

                function\_body

        RETURN scalar\_expression

    END

**function\_name**

Имя пользовательской функции. Скобки после имени функции обязательны даже при отсутствии параметров

**@parameter\_name**

Параметр пользовательской функции. Может быть объявлен один или несколько параметров.

Для функций допускается не более 1 024 параметров. При выполнении функции значение каждого из объявленных параметров должно быть указано пользователем, если для этого параметра не определено значение по умолчанию.

Имя параметра всегда должно начинаться со знака @. Параметры локальны в пределах функции, то есть в разных функциях могут быть использованы одинаковые имена параметров. Аргументы могут использоваться только вместо констант. Они не могут использоваться вместо имен таблиц, имен столбцов или имен других объектов базы данных.

**parameter\_data\_type**

Тип данных параметра. Для параметров функций Transact-SQL допустимы любые типы данных за исключением типа данных **timestamp**, нескалярные типы **cursor** и **table** также не могут быть указаны в качестве типов данных параметров.

**[ = default ]**

Значение параметра по умолчанию. Если определено значение **default**, функция выполняется даже в том случае, если для данного параметра значение не указано.

Если параметр функции имеет значение по умолчанию, то для него должно быть указано ключевое слово **DEFAULT** для получения функцией значения по умолчанию. Применение ключевого слова **DEFAULT** следует отличать от использования аргументов со значениями по умолчанию в хранимых процедурах, когда не указанный аргумент неявно принимает значение по умолчанию.

**return\_data\_type**

Тип возвращаемого значения скалярной пользовательской функции. Для возвращаемых значений функций Transact-SQL допустимы любые типы данных за исключением типа данных **timestamp**, нескалярные типы **cursor** и **table** также не могут быть указаны в качестве типов данных параметров.

**ENCRYPTION**

Указывает, что компонент Database Engine преобразует исходный текст инструкции CREATE FUNCTION в скрытый формат. Пользователи, не имеющие доступа к системным таблицам или файлам баз данных, не смогут получить скрытый текст.

**SCHEMABINDING**

Указывает, что функция привязана к объектам базы данных, которые содержат ссылки на нее. Если аргумент SCHEMABINDING указан, нельзя изменить базовые объекты таким способом, который может повлиять на определение функции. Сначала нужно изменить или удалить само определение функции, чтобы удалить зависимости от объекта, который требуется изменить.

Привязка функции к объектам, на которые она ссылается, удаляется только в следующих случаях:

* При удалении функции.
* При изменении функции инструкцией ALTER, если не указан параметр SCHEMABINDING.

**EXECUTE AS**

Определяет контекст безопасности, в котором должна быть выполнена функция, т.е. какую учетную запись компонент Database Engine использует при проверке разрешений на объекты, на которые ссылается функция. Это повышает гибкость и безопасность управления разрешениями на цепочки владения между пользовательскими функциями и объектами, на которые они ссылаются.

Пользователям необходимо будет предоставлять только разрешения на саму функцию, без выдачи явных разрешений на объекты, на которые она ссылается. Только пользователь, от имени которого выполняется модуль, должен будет иметь разрешения на объекты, к которым этот модуль обращается.

Допустимые значения **EXECUTE AS { CALLER | SELF | OWNER | 'user\_name' }**.

**function\_body**

Указывает серию инструкций Transact-SQL, которая в совокупности не вызывает побочных эффектов вроде изменения содержимого таблиц и формирует возвращаемое значение функции.

Для скалярных функций аргумент function\_body представляет собой серию инструкций Transact-SQL, которые в совокупности вычисляют скалярное значение.

**scalar\_expression**

Указывает скалярное значение, возвращаемое скалярной функцией.

**Пример:**

Следующий пример демонстрирует создание скалярной функции из нескольких инструкций. Функция имеет один входной параметр **ProductCategoryID** и возвращает одно значение – максимальную цену товара указанной категории.

CREATE FUNCTION getMaxPrice (@ProductCategoryID int)

RETURNS int

AS

BEGIN

DECLARE @ret int;

SELECT @ret = MAX(Price)

FROM Product

WHERE ProductCategoryID = @ProductCategoryID

IF (@ret IS NULL)

SET @ret = 0;

RETURN @ret;

END;

--Использование созданной функции

SELECT ProductCategoryName, getMaxPrice (ProductCategoryID) AS MaxPrice

FROM ProductCategory

WHERE ProductCategoryID = 1;

### Возвращающие табличное значение определяемые пользователем функции

Определяемые пользователем функции, возвращающие тип данных **table**, могут быть полноценной альтернативой представлениям. Возвращающая табличное значение функция, определяемая пользователем, может быть использована там, где в запросах Transact-SQL разрешены выражения представлений. В то время как представления ограничены одной инструкцией SELECT, определяемые пользователем функции могут содержать дополнительные инструкции, обеспечивающие более сложную логику.

Возвращающая табличное значение функция, определяемая пользователем, также может заменять хранимые процедуры, возвращающие один результирующий набор. На таблицу, возвращаемую определяемой пользователем функцией, можно ссылаться в предложении FROM инструкции Transact-SQL, в котором нельзя ссылаться на хранимые процедуры, возвращающие результирующие наборы.

CREATE FUNCTION function\_name

( [ { @parameter\_name [ AS ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

  ]

)

RETURNS @return\_variable TABLE < table\_type\_definition >

    [ WITH [ ENCRYPTION ] [ SCHEMABINDING ] [ EXECUTE AS Clause ]]

    [ AS ]

    BEGIN

                function\_body

        RETURN

    END

<table\_type\_definition>:: =

(

{ <column\_definition> <column\_constraint>

| <computed\_column\_definition> }

        [ <table\_constraint> ] [ ,...n ]

)

**function\_name**

Имя пользовательской функции. Скобки после имени функции обязательны даже при отсутствии параметров

**@parameter\_name**

Параметр пользовательской функции. Может быть объявлен один или несколько параметров.

Для функций допускается не более 1 024 параметров. При выполнении функции значение каждого из объявленных параметров должно быть указано пользователем, если для этого параметра не определено значение по умолчанию.

Имя параметра всегда должно начинаться со знака @. Параметры локальны в пределах функции, то есть в разных функциях могут быть использованы одинаковые имена параметров. Аргументы могут использоваться только вместо констант. Они не могут использоваться вместо имен таблиц, имен столбцов или имен других объектов базы данных.

**parameter\_data\_type**

Тип данных параметра. Для параметров функций Transact-SQL допустимы любые типы данных за исключением типа данных **timestamp**, нескалярные типы **cursor** и **table** также не могут быть указаны в качестве типов данных параметров.

**[ = default ]**

Значение параметра по умолчанию. Если определено значение **default**, функция выполняется даже в том случае, если для данного параметра значение не указано.

Если параметр функции имеет значение по умолчанию, то для него должно быть указано ключевое слово DEFAULT для получения функцией значения по умолчанию. Применение ключевого слова DEFAULT следует отличать от использования аргументов со значениями по умолчанию в хранимых процедурах, когда не указанный аргумент неявно принимает значение по умолчанию.

**<table\_type\_definition>**

Определение табличного типа, включающее определение столбцов и ограничений целостности.

**ENCRYPTION**

Указывает, что компонент Database Engine преобразует исходный текст инструкции CREATE FUNCTION в скрытый формат. Пользователи, не имеющие доступа к системным таблицам или файлам баз данных, не смогут получить скрытый текст.

**SCHEMABINDING**

Указывает, что функция привязана к объектам базы данных, которые содержат ссылки на нее. Если аргумент SCHEMABINDING указан, нельзя изменить базовые объекты таким способом, который может повлиять на определение функции. Сначала нужно изменить или удалить само определение функции, чтобы удалить зависимости от объекта, который требуется изменить.

Привязка функции к объектам, на которые она ссылается, удаляется только в следующих случаях:

* При удалении функции.
* При изменении функции инструкцией ALTER, если не указан параметр SCHEMABINDING.

**EXECUTE AS**

Определяет контекст безопасности, в котором должна быть выполнена функция, т.е. какую учетную запись компонент Database Engine использует при проверке разрешений на объекты, на которые ссылается функция. Это повышает гибкость и безопасность управления разрешениями на цепочки владения между пользовательскими функциями и объектами, на которые они ссылаются.

Пользователям необходимо будет предоставлять только разрешения на саму функцию, без выдачи явных разрешений на объекты, на которые она ссылается. Только пользователь, от имени которого выполняется модуль, должен будет иметь разрешения на объекты, к которым этот модуль обращается.

Допустимые значения **EXECUTE AS { CALLER | SELF | OWNER | 'user\_name' }**.

**function\_body**

Указывает серию инструкций Transact-SQL, которая в совокупности не вызывает побочных эффектов вроде изменения содержимого таблиц и формирует возвращаемое значение функции.

Для функций, возвращающих табличное значение из нескольких инструкций, аргумент function\_body представляет собой серию инструкций Transact-SQL, заполняющих возвращаемую переменную **@return\_variable**.

**TABLE**

Указывает, что возвращаемым значением функции, возвращающей табличное значение, является таблица. Функциям, возвращающим табличное значение, могут передаваться только константы и **локальные переменные**.

В функциях, возвращающих табличное значение из нескольких инструкций, переменной **@return\_variable** является переменная TABLE, используемая для сохранения данных и накопления строк, которые будут возвращены в качестве значения функции.

Ни одна из инструкций Transact-SQL в возвращающей табличное значение функции не может возвращать результирующий набор непосредственно пользователю. Единственные данные, которые функция может вернуть пользователю, это таблица, состоящая из строк, вставленных в переменную TABLE, возврат происходит при выполнении инструкции RETURN.

**Пример:**

В следующем примере создается функция **ufnGetUserInfo**. В этой функции именем локальной возвращаемой переменной является **@retUserInfo**. Инструкции в теле функции вставляют строки в эту переменную для создания табличных результатов, возвращаемых этой функцией.

CREATE FUNCTION ufnGetUserInfo (@LoginName nvarchar(50))

RETURNS @retUserInfo TABLE

(

UserID int PRIMARY KEY NOT NULL,

FirstName nvarchar(50) NULL,

JobTitle nvarchar(2);

)

AS

BEGIN

DECLARE

@UserID nvarchar(50),

@FirstName nvarchar(50),

@JobTitle nvarchar(2);

SELECT

@UserID = UserID,

@FirstName = FirstName

FROM User

WHERE LoginName = @LoginName;

IF @UserID IS NOT NULL

BEGIN

SELECT @JobTitle =

CASE

-- Check for employee

WHEN EXISTS(SELECT \* FROM Person.Person AS p

WHERE p.UserID = @UserID AND p.PersonType = 'EM')

THEN (SELECT JobTitle

FROM Employee AS e

WHERE e.UserID = @UserID)

-- Check for vendor

WHEN EXISTS(SELECT \* FROM Person.Person AS p

WHERE p.UserID = @UserID AND p.PersonType = 'PT')

THEN (SELECT JobTitle

FROM Partner AS pt

WHERE pt.UserID = @UserID)

ELSE NULL

END;

END;

IF @UserID IS NOT NULL

BEGIN

INSERT @retUserInfo

SELECT @UserID, @FirstName, @JobTitle;

END;

RETURN;

END;

SELECT UserID, FirstName, JobTitle

FROM ufnGetUserInfo('admin');

### Определяемые пользователем встроенные функции

Определяемые пользователем встроенные функции представляют собой подмножество определяемых пользователем функций, возвращающих значения типа данных **table**. Встроенные функции могут быть использованы для замены параметризованных представлений. В то время как представления ограничены одной инструкцией SELECT, определяемые пользователем встроенные функции могут содержать более сложную логику фильтрации записей, основанную на параметрах функции.

CREATE FUNCTION function\_name

( [ { @parameter\_name [ AS ] parameter\_data\_type

[ = default ] }

[ ,...n ]

]

)

RETURNS TABLE

[ WITH [ ENCRYPTION ] [ SCHEMABINDING ] ]

[ AS ]

RETURN [ ( ] select\_stmt [ ) ]

**function\_name**

Имя пользовательской функции. Скобки после имени функции обязательны даже при отсутствии параметров

**@parameter\_name**

Параметр пользовательской функции. Может быть объявлен один или несколько параметров.

Для функций допускается не более 1 024 параметров. При выполнении функции значение каждого из объявленных параметров должно быть указано пользователем, если для этого параметра не определено значение по умолчанию.

Имя параметра всегда должно начинаться со знака @. Параметры локальны в пределах функции, то есть в разных функциях могут быть использованы одинаковые имена параметров. Аргументы могут использоваться только вместо констант. Они не могут использоваться вместо имен таблиц, имен столбцов или имен других объектов базы данных.

**parameter\_data\_type**

Тип данных параметра. Для параметров функций Transact-SQL допустимы любые типы данных за исключением типа данных **timestamp**, нескалярные типы **cursor** и **table** также не могут быть указаны в качестве типов данных параметров.

**[ = default ]**

Значение параметра по умолчанию. Если определено значение **dшefault**, функция выполняется даже в том случае, если для данного параметра значение не указано.

Если параметр функции имеет значение по умолчанию, то для него должно быть указано ключевое слово DEFAULT для получения функцией значения по умолчанию. Применение ключевого слова DEFAULT следует отличать от использования аргументов со значениями по умолчанию в хранимых процедурах, когда не указанный аргумент неявно принимает значение по умолчанию.

**TABLE**

Указывает, что возвращаемым значением функции, возвращающей табличное значение, является таблица. Функциям, возвращающим табличное значение, могут передаваться только константы и **локальные переменные**.

Во встроенных функциях, возвращающих табличное значение, возвращаемое значение TABLE определяется при использовании единственной инструкции SELECT. Встроенные функции не имеют соответствующих возвращаемых переменных.

**ENCRYPTION**

Указывает, что компонент Database Engine преобразует исходный текст инструкции CREATE FUNCTION в скрытый формат. Пользователи, не имеющие доступа к системным таблицам или файлам баз данных, не смогут получить скрытый текст.

**SCHEMABINDING**

Указывает, что функция привязана к объектам базы данных, которые содержат ссылки на нее. Если аргумент SCHEMABINDING указан, нельзя изменить базовые объекты таким способом, который может повлиять на определение функции. Сначала нужно изменить или удалить само определение функции, чтобы удалить зависимости от объекта, который требуется изменить.

Привязка функции к объектам, на которые она ссылается, удаляется только в следующих случаях:

* При удалении функции.
* При изменении функции инструкцией ALTER, если не указан параметр SCHEMABINDING.

**select\_stmt**

Единичная инструкция SELECT, определяющая возвращаемое значение встроенной функции, возвращающей табличное значение.

**Пример:**

В следующем примере создается функция **ufnGetEmployeeInfo**. Функция состоит из единственного оператора SELECT, и по сути является параметризованным представлением.

CREATE FUNCTION ufnGetEmployeeInfo (@LoginName nvarchar(50))

RETURNS TABLE

AS

RETURN

(

SELECT u.FirstName, e.JobTitle

FROM Employee AS e INNER JOIN User AS u

ON e.UserID = u.UserID

WHERE u.LoginName = @LoginName;

);

SELECT \*

FROM ufnGetEmployeeInfo ('admin');

Встроенные функции также могут быть использованы для улучшения работы индексированных представлений. Индексированное представление не может использовать параметры в условиях поиска предложения WHERE для настройки сохраненного результирующего набора в соответствии с нуждами конкретных пользователей. Тем не менее, можно задать индексированное представление, в котором будет храниться полный набор данных, соответствующих представлению, а затем создать встроенную функцию, в которой содержатся параметризованные условия поиска, позволяющие пользователям настраивать результаты. Если определение представления является сложным, то работа по составлению результирующего набора в основном сводится к таким операциям, как построение статистических выражений и соединение таблиц, что приводит к созданию кластеризованного индекса представления. При последующем создании встроенной функции, обращающейся к индексированному представлению, в ней могут быть применены пользовательские параметризованные фильтры, что позволит возвращать конкретные строки данных результирующего набора индексированного представления.

Лекция № 18

## Концепция транзакций

Концепция транзакции была разработана для тех случаев, в которых результирующее состояние базы данных зависит от общей успешности выполнения ряда операций. Это может происходить из-за того, что последовательные операции могут приводить к изменению результатов предыдущих операций. В таких случаях, если какая либо операция завершается ошибкой, результирующее состояние может быть неопределенным.

Чтобы разрешить эту проблему, в транзакциях группируются серии операций способом, при котором обеспечивается определенность окончательного результата. Все операции должны завершиться успешно и затем быть выполнены (записаны в базу данных), в противном случае вся транзакция дает сбой. Отмена транзакции называется откатом. Откат дает возможность восстановить базу данных после изменений и возвратить эту базу данных в то состояние, которое было перед выполнением транзакции.

Например, в автоматизированной банковской транзакции, чтобы правильно выполнить перевод денег со счета **A** на счет **B**, кредитованию счета **B** должно предшествовать списание денег со счета **А**, в противном случае транзакция должна завершиться ошибкой.

Транзакция имеет следующий набор свойств ACID:

* **Атомарность (Atomicity)**   Транзакция является наименьшей неделимой единицей работы и выполняется только один раз, независимо от того вся ли работа выполнена либо она не выполнена вообще.
* **Целостность (Consistency)**   Транзакция сохраняет целостность данных, преобразуя одно согласованное состояние данных в другое согласованное состояние.
* **Изоляция (Isolation)**   Транзакция является единицей изоляции и каждая из них возникает отдельно и независимо от параллельных транзакций.
* **Надежность (Durability)**   Транзакция является единицей восстановления. Если транзакция проходит успешно, то внесенные ей изменения сохраняются даже в случае сбоя или завершения работы системы. Если транзакция дает сбой, система остается в том состоянии, которое было до выполнения транзакции.

Рассмотрим перечисленные свойства на примере банковской транзакции:

* **Атомарность. Данное свойство гарантирует, что списание денег со счета А и кредитование счета В либо произойдут вместе, либо не произойдет ни одного из них.**
* **Целостность. Гарантирует, что целостность данных после завершения транзакции не будет нарушена. Например, если при пополнении счета В произойдет ошибка, списания средств со счета А не произойдет.**
* **Изоляция. Обеспечивает возможность параллельного выполнения независимой транзакции по переводу средств со счета А на счет С, не нарушая при этом целостность данных, то есть счета В и С будут увеличены на указанные для них значения, а счет А уменьшен на сумму этих значений. При этом, если средств недостаточно для операции перевода на счет В и достаточно для перевода на счет С, транзакция по кредитованию счета С завершится успешно, а для счета В транзакция завершится с ошибкой и будет отменена.**
* **Надежность. Гарантирует, что если во время выполнения транзакции после списания средств со счета А и до внесения соответствующей суммы на счет В произойдет сбой системы, то после восстановления средства со счета А списаны не будут. Если же сбой произойдет после успешной фиксации транзакции, счет А будет уменьшен, а счет В пополнен на указанную сумму.**

## Эффекты одновременного доступа

Изменение данных пользователями может оказывать влияние на других пользователей, считывающих или изменяющих эти же данные в этот же момент времени. В этом случае говорят, что пользователи получают одновременный доступ к этим данным. Если в системе хранения данных отсутствует управление одновременным доступом, то могут наблюдаться следующие побочные эффекты.

* Потерянные обновления.
* Незафиксированная зависимость («грязные» операции чтения).
* Несогласованность анализа (неповторяющееся чтение).
* Фантомные операции чтения.

Негативные эффекты конкурентного доступа перечислены в порядке возрастания накладных расходов для борьбы с ними и уменьшения возможности одновременного доступа пользователей к данным.

**Потерянные обновления**

Потерянные обновления возникают, когда две или более транзакций выбирают одну и ту же строку и изменяют ее на основании ее исходного значения. Каждая транзакция не знает о других транзакциях. Последнее обновление изменяет данные других транзакций, что приводит к потере данных.

Например, у двух редакторов есть электронные копии одного и того же документа. Каждый редактор изменяет свою копию независимо и затем сохраняет ее, перезаписывая исходный документ. Редактор, сохраняющий свою измененную копию, перезаписывает изменения, сделанные другим редактором. Этой проблемы можно избежать, если у одного редактора не будет доступа к этому файлу до завершения и фиксации транзакции другого редактора.

**Незафиксированная зависимость (грязные операции чтения)**

Незафиксированная зависимость возникает, когда вторая транзакция считывает строку, изменяемую в данный момент другой транзакцией, таким образом вторая транзакция будет считывать данные, которые еще не зафиксированы и могут быть изменены первой транзакцией.

В Модуле 2, при изучении механизма записи данных на диск, рассматривались грязные страницы данных, то есть страницы измененные в кэше SQL Server, но еще не записанные на диск. Грязные операции чтения предполагают, что пользователь читает данные из страниц, измененных незафиксированными транзакциями, изменения которых еще могут быть отменены. Таким образом чтение происходят с грязных страниц, однако не все операции чтения с грязных страниц являются грязными. Фактически страницы могут содержать данные измененные зафиксированной транзакцией, при этом еще не сохраненные из кэша на диск.

Например, редактор изменяет электронный документ. В это время второй редактор снимает копию этого документа, включающую уже сделанные изменения, и распространяет этот документ предполагаемой аудитории. Затем первый редактор решает, что сделанные изменения неправильны, удаляет их и сохраняет документ. Распространенный документ содержит изменения, которые уже не существуют и должны считаться никогда не существовавшими. Этой проблемы можно избежать, если никто не сможет считать измененный документ, пока первый редактор не сохранит окончательную версию изменений и не зафиксирует эту транзакцию.

**Несогласованность анализа (неповторяющаяся операция чтения)**

Несогласованность анализа возникает, когда вторая транзакция осуществляет доступ к одной строке несколько раз, и каждый раз считывает разные данные. Несогласованность анализа сходна с незафиксированной зависимостью, когда транзакция изменяет данные, считываемые другой транзакцией. Однако при несогласованности анализа данные, считываемые второй транзакцией, были зафиксированы транзакцией, которая сделала изменения. Кроме того, несогласованность анализа подразумевает несколько операций чтения (две и более) одной строки, при которых каждый раз данные изменяются другой транзакцией, это называется также неповторяющейся операцией чтения.

Например, редактор считывает один и тот же документ дважды, но между этими операциями писатель перезаписывает этот документ. Когда редактор считывает документ во второй раз, он уже изменен. Первую операцию чтения повторить нельзя. Этой проблемы можно избежать, если писатель не сможет изменять документ, пока редактор не считает его в последний раз.

**Фантомные операции чтения**

Фантомные операции чтения возникают, когда операция вставки или удаления применяются к строке, которая принадлежит диапазону строк, считываемых некоторой транзакцией. Первая транзакция считывает набор строк, содержащий строку, которая уже не будет существовать при второй и последующих операциях чтения в результате ее удаления другой транзакцией. Аналогично вторая и последующие транзакции считают строку, которая не существовала при первом чтении и была вставлена другой транзакцией.

Например, редактор изменяет документ, представленный писателем, но когда издательский отдел вставляет эти изменения в главную копию документа, они обнаруживают, что автор добавил в документ новый неотредактированный материал. Аналогично ситуации с неповторяющейся операцией чтения этой проблемы можно избежать, если никто не сможет добавлять к документу новый материал до окончания работы с исходным документом редактора и издательского отдела.

## Блокировки и управление версиями строк

Компонент Database Engine использует блокировки и управление версиями строк для гарантии целостности транзакций и поддержания согласованности базы данных, когда несколько пользователей обращаются к одним и тем же данным в одно и то же время.

Блокировка и управление версиями строк не дают пользователям считывать незафиксированные данные и не дают нескольким пользователям менять одни и те же данные в одно и то же время. Без этих механизмов запросы к таким данным могли бы возвращать непредвиденные результаты, например данные, которые еще не были зафиксированы в базе данных.

* **Блокировка**  
  Каждая транзакция запрашивает блокировку разных типов ресурсов, например строк, страниц или таблиц, от которых эта транзакция зависит. Блокировка не дает другим транзакциям изменять ресурсы, чтобы избежать ошибок в транзакции, запросившей блокировку. Каждая транзакция освобождает свои блокировки, если больше не зависит от блокируемого ресурса.
* **Управление версиями строк**  
  Компонент Database Engine имеет возможность хранить версии каждой измененной строки. Приложения могут указать, что транзакция будет использовать версии строк для просмотра данных, существовавших до ее начала или до начала запроса, вместо того, чтобы защищать все операции чтения блокировками. При управлении версиями строк вероятность того, что операция чтения будет блокировать другие транзакции, значительно снижается.

Блокировка - это механизм, с помощью которого компонент Microsoft SQL Server Database Engine синхронизирует одновременный доступ нескольких пользователей к одному фрагменту данных.

Прежде чем транзакция сможет распоряжаться текущим состоянием фрагмента данных, например, для чтения или изменения данных, она должна защититься от изменений этих данных другой транзакцией. Для этого транзакция запрашивает блокировку фрагмента данных. Существует несколько режимов блокировки, например общая или монопольная. Режим блокировки определяет уровень подчинения данных транзакции. Ни одна транзакция не может получить блокировку, которая противоречит другой блокировке этих данных, предоставленной другой транзакции. Если транзакция запрашивает режим блокировки, противоречащий предоставленной ранее блокировке тех же данных, экземпляр компонента Database Engine приостанавливает ее работу до тех пор, пока первая блокировка не освободится.

При изменении фрагмента данных транзакция удерживает блокировку, защищая изменения до конца транзакции. Продолжительность блокировки, полученной для защиты операций чтения, зависит от уровня изоляции транзакции. Все блокировки, удерживаемые транзакцией, освобождаются после ее завершения (при фиксации или откате).

За управление блокировками отвечает внутренний компонент Database Engine, называемый диспетчером блокировок. Когда экземпляр компонента Database Engine обрабатывает инструкцию Transact-SQL, обработчик запросов компонента Database Engine определяет, к каким ресурсам требуется доступ. Обработчик запросов определяет, какие типы блокировок требуются для защиты каждого ресурса, в зависимости от типа доступа и уровня изоляции транзакции. Затем обработчик запросов запрашивает соответствующую блокировку у диспетчера блокировок. Диспетчер блокировок предоставляет блокировку, если она не противоречит блокировкам, удерживаемым другими транзакциями.

## Типы управления одновременным доступом

Теория управления одновременным доступом предлагает два способа осуществления управления:

* **Пессимистичное управление**Система блокировок не допускает, чтобы изменение данных одними пользователями влияло на других пользователей. После того как действие пользователя приводит к блокировке, до тех пор пока инициатор ее не снимет, другие пользователи не могут выполнять действия, которые могут вызвать конфликт с блокировкой. Это называется пессимистичным управлением, поскольку в основном применяется в средах с большим количеством конфликтов данных, где затраты на защиту данных с помощью блокировок меньше затрат на откат изменений в случае конфликтов одновременного доступа.
* **Оптимистичное управление**  
  При оптимистическом управлении пользователи не блокируют данные на период чтения. Когда пользователь обновляет данные, система проверяет, вносил ли другой пользователь в них изменение после считывания. Если другой пользователь изменял данные, возникает ошибка. Как правило, при получении сообщения об ошибке пользователь откатывает изменения и начинает процесс заново. Это называется оптимистичным управлением, поскольку в основном применяется в средах с небольшим количеством конфликтов данных, где затраты на периодический откат транзакции меньше затрат на блокировку данных при считывании.

Microsoft SQL Server поддерживает ряд средств управления одновременным доступом. Пользователи задают тип управления посредством выбора уровней изоляции транзакций.

## Режимы блокировки

Для обеспечения безопасного одновременного доступа транзакций к ресурсам компонент Microsoft SQL Server Database Engine блокирует ресурсы с помощью различных режимов блокировки.

В следующей таблице показаны режимы блокировки ресурсов, применяемые компонентом Database Engine.

|  |  |
| --- | --- |
| **Режим блокировки** | **Описание** |
| **Совмещаемая блокировка (S)** | Используется для операций считывания, которые не меняют и не обновляют данные, такие как инструкция SELECT. Данный тип блокировки реализует пессимистичное управление одновременным доступом, эту блокировку могут одновременно захватывать несколько транзакций читающие данные, но транзакции изменяющие данные не могут получить на них блокировку до тех пор, пока кем-либо удерживается совмещаемая блокировка чтения. |
| **Блокировка обновления (U)** | Применяется к тем ресурсам, которые потенциально могут быть обновлены транзакцией. Данный тип блокировки может захватываться только одной транзакцией, при этом, если транзакция действительно обновляет данные, блокировка преобразуется в **Монопольную**. Использование блокировки обновления позволяет избежать возникновения взаимоблокировки, когда несколько сеансов считывают данные используя **Совмещаемую** блокировку, а затем пытаются получить **Монопольную** блокировку для обновления данных. |
| **Монопольная блокировка (Х)** | Данный тип блокировки запрещает транзакциям одновременный доступ к ресурсу и используется для операций модификации данных, таких как инструкции INSERT, UPDATE или DELETE. Гарантирует, что несколько обновлений не будет выполнено одновременно для одного ресурса, при этом операции считывания из других транзакций будут допускаться только при наличии подсказки NOLOCK или уровня изоляции обеспечивающего доступ для чтения к нефиксированным данным. |
| **Блокировка с намерением** | Используется для создания иерархии блокировок. Типы намеренной блокировки: с намерением совмещаемого доступа (IS), с намерением монопольного доступа (IX), а также совмещаемая с намерением монопольного доступа (SIX). |
| **Блокировка схемы** | Используется во время выполнения операции, зависящей от схемы таблицы. Пока удерживается блокировка изменения схемы, одновременный доступ к таблице запрещен, то есть любые операции вне блокировки изменения схемы будут запрещены до снятия блокировки.  Существует два типа блокировки схем: блокировка изменения схемы (Sch-S) и блокировка стабильности схемы (Sch-M). |
| **Блокировка массового обновления (BU)** | Используется, если выполняется массовое копирование данных в таблицу и указана подсказка **TABLOCK**. Данный тип блокировки позволяет поддерживать несколько одновременных потоков массовой загрузки данных в одну и ту же таблицу и при этом запрещать доступ к таблице любым другим процессам, отличным от массовой загрузки данных. |
| **Блокировка диапазона ключей** | Защищает диапазон строк, неявно включенный в набор записей, считываемый инструкцией языка Transact-SQL. На время блокировки диапазона ключей конкурентные транзакции не могут изменить данные таблицы таким образом, что в первоначально полученном запросом наборе строк пропадут или появятся новые строки. |

**Совмещаемые блокировки**

Данный тип блокировки реализует пессимистичное управление одновременным доступом, эту блокировку могут одновременно захватывать несколько транзакций читающие данные, но транзакции изменяющие данные не могут получить на них блокировку до тех пор, пока кем-либо удерживается совмещаемая блокировка чтения. Совмещаемые блокировки (S) ресурса снимаются по завершении операции считывания, если только уровень изоляции транзакции не установлен на повторяющееся чтение или более высокий уровень, а также если совмещаемые блокировки (S) не продлены на все время транзакции с помощью подсказки блокировки.

**Блокировки обновления**

Применяется к тем ресурсам, которые потенциально могут быть обновлены транзакцией. Данный тип блокировки может захватываться только одной транзакцией, при этом, если транзакция действительно обновляет данные, блокировка преобразуется в **Монопольную**.

Использование блокировки обновления позволяет избежать возникновения взаимоблокировки, когда несколько сеансов считывают данные используя **Совмещаемую** блокировку, а затем пытаются получить **Монопольную** блокировку для обновления данных. Например, если две транзакции запрашивают совмещаемую блокировку на ресурс и затем пытаются одновременно обновить данные, то одна из транзакций пытается преобразовать блокировку в монопольную (**X**). Преобразование совмещаемой блокировки в монопольную потребует некоторого времени, поскольку монопольная блокировка для одной транзакции несовместима с совмещаемой блокировкой для другой транзакции. Начнется ожидание блокировки. Вторая транзакция попытается получить монопольную (**X**) блокировку для обновления. Поскольку обе транзакции выполняют преобразование в монопольную (**X**) блокировку и при этом каждая из транзакций ожидает, пока вторая снимет совмещаемую блокировку, то в результате возникает взаимоблокировка.

**Монопольные блокировки**

Монопольная (**X**) блокировка запрещает транзакциям одновременный доступ к ресурсу и гарантирует, что несколько обновлений не будет выполнено одновременно для одного ресурса, при этом операции считывания из других транзакций будут допускаться только при наличии подсказки NOLOCK или уровня изоляции обеспечивающего доступ для чтения к нефиксированным данным.

Изменяющие данные инструкции, такие как INSERT, UPDATE или DELETE, соединяют как операции изменения, так и операции считывания. Чтобы выполнить необходимые операции изменения данных, инструкция сначала должна получить соответствующие данные с помощью операций считывания. Например, инструкция UPDATE может изменять строки в одной таблице, основанной на соединении данных из другой таблицы. В этом случае инструкция UPDATE кроме монопольной блокировки обновляемых строк запрашивает также совмещаемые блокировки для строк, считываемых в соединенной таблице.

Блокировки с намерением

В компоненте Database Engine блокировки с намерением применяются для защиты размещения совмещаемой (**S**) или монопольной (**X**) блокировки ресурса на более низком уровне иерархии. Блокировки с намерением должны захватываться до блокировок более низкого уровня, то есть они обозначают намерение поместить блокировку на более низком уровне.

Блокировка с намерением выполняет две функции:

* предотвращает изменение ресурса более высокого уровня другими транзакциям таким образом, что это сделает недействительной блокировку более низкого уровня;
* повышает эффективность компонента Database Engine при распознавании конфликтов блокировок на более высоком уровне гранулярности.

Например, в таблице требуется блокировка с намерением совмещаемого доступа до того, как для страниц или строк этой таблицы будет запрошена совмещаемая (**S**) блокировка. Если задать блокировку с намерением на уровне таблицы, то другим транзакциям будет запрещено получать монопольную (**X**) блокировку для таблицы, содержащей эту страницу.

Компонент Database Engine проверяет наличие блокировок с намерением только на уровне таблицы, чтобы определить, может ли транзакция безопасно получить для этой таблицы совмещаемую блокировку. Благодаря этому нет необходимости проверять блокировки в каждой строке и на каждой странице, чтобы убедиться, что транзакция может заблокировать всю таблицу, это позволяет добиться большей производительности.

В состав блокировок с намерением входят блокировка с намерением совмещаемого доступа (**IS**), блокировка с намерением монопольного доступа (**IX**), а также совмещаемая блокировка с намерением монопольного доступа (**SIX**).

Режимы блокировки с намерениями:

|  |  |
| --- | --- |
| **Режим блокировки** | **Описание** |
| **Блокировка с намерением совмещаемого доступа (IS)** | Защищает запрошенные или полученные совмещаемые блокировки на некоторых (но не на всех) ресурсах на более низком уровне иерархии. |
| **Блокировка с намерением монопольного доступа (IX)** | Защищает запрошенные или полученные монопольные блокировки на некоторых (но не на всех) ресурсах на более низком уровне иерархии. Режим **IX** является расширенным режимом **IS**, кроме того, он защищает запрос на совмещаемые блокировки на ресурсах более низкого уровня. |
| **Совмещаемая блокировка с намерением монопольного доступа (SIX)** | Защищает запрошенные или полученные совмещаемые блокировки на всех ресурсах более низкого уровня иерархии, а также блокировки с намерением на некоторых (но не всех) ресурсах более низкого уровня. На ресурсах верхнего уровня допускаются одновременные блокировки **IS**. Например, запрос блокировки **SIX** для таблицы запрашивает блокировку с намерением монопольного доступа для всех изменяемых страниц и монопольную блокировку изменяемых строк. Одновременно для одного ресурса может быть установлена только одна блокировка **SIX**, что предотвращает обновление ресурса другими транзакциями, хотя эти транзакции могут считывать данные с ресурсов более низкого уровня в иерархии, получая блокировки **IS** уровня таблицы. |
| **Блокировка с намерением обновления (IU)** | Защищает запрошенные или полученные блокировки обновления на всех ресурсах более низкого уровня в иерархии. Блокировки **IU** применяются только на страничных ресурсах. Если выполняется операция обновления, то блокировки **IU** преобразуются в **IX**. |
| **Совмещаемая блокировка с намерением обновления (SIU)** | Сочетание блокировок **S** и **IU** в результате раздельного запрашивания этих блокировок и одновременного удержания их обеих. |
| **Блокировка обновления с намерением монопольного доступа (UIX)** | Сочетание блокировок **U** и **IX** в результате раздельного запрашивания этих блокировок и одновременного удержания их обеих. |

Блокировки схем

В компоненте Database Engine блокировка изменения схемы (**Sch-M**) применяется с операциями языка определения схемы (DDL) для таблиц, например при добавлении столбца или очистке таблицы. Пока удерживается блокировка изменения схемы, одновременный доступ к таблице запрещен, то есть любые операции вне блокировки изменения схемы будут запрещены до снятия блокировки.

Существует два типа блокировки схем:

* Блокировка изменения схемы (**Sch-M**) применяется с некоторыми операциями языка обработки данных, например усечением таблиц, чтобы предотвратить одновременный доступ к таблице.
* Блокировка стабильности схемы (**Sch-S**) применяется компонентом Database Engine при компиляции и выполнении запросов. Блокировка стабильности схемы (**Sch-S**) не влияет на блокировки транзакций, включая монопольные (**X**) блокировки, поэтому другие транзакции (даже транзакции с монопольной блокировкой (**X**) для таблицы) могут продолжать работу во время компиляции запроса. Однако, одновременные операции DDL и DML, которые запрашивают блокировки изменения схемы (**Sch-M**), над таблицей выполняться не могут.

Блокировки массового обновления

Блокировки массового обновления (**BU**) применяются компонентом Database Engine при массовом копировании данных в таблицу. Данный тип блокировки позволяет поддерживать несколько одновременных потоков массовой загрузки данных в одну и ту же таблицу и при этом запрещать доступ к таблице любым другим процессам, отличным от массовой загрузки данных.

Блокировки диапазона ключа

Блокировки диапазона ключа защищает диапазон строк, неявно включенный в набор записей, считываемый инструкцией языка Transact-SQL. На время блокировки диапазона ключей конкурентные транзакции не могут изменить данные таблицы таким образом, что в первоначально полученном запросом наборе строк пропадут или появятся новые строки.

Использование различных режимов блокировки позволяет обеспечить высокопроизводительный конкурентный доступ к данным, не нарушая их целостность. За счет использования подсказок в запросах и возможности выбора желаемого уровня изоляции транзакций, пользователи имеют возможность гранулярного управления достоверностью получаемых данных и скоростью доступа к ним.

## Взаимоблокировка

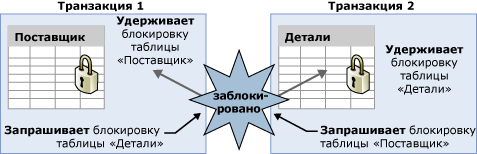
Взаимоблокировка возникает, когда две и более задач постоянно блокируют друг друга в ситуации, когда у каждой задачи заблокирован ресурс, который пытаются заблокировать другие задачи. Например:

* Транзакция А создает общую блокировку строки 1.
* Транзакция Б создает общую блокировку строки 2.
* Транзакция А теперь запрашивает монопольную блокировку строки 2 и блокируется до того, как транзакция Б закончится и освободит общую блокировку строки 2.
* Транзакция Б теперь запрашивает монопольную блокировку строки 1 и блокируется до того, как транзакция A закончится и освободит общую блокировку строки 1.

Транзакция А не может завершиться до того, как завершится транзакция Б, а транзакция Б заблокирована транзакцией А. Такое условие также называется цикличной зависимостью: транзакция А зависит от транзакции Б, а транзакция Б зависит от транзакции А и этим замыкает цикл.

Обе транзакции находятся в состоянии взаимоблокировки и будут всегда находиться в состоянии ожидания, если взаимоблокировка не будет разрушена внешним процессом. Монитор взаимоблокировок компонента Microsoft SQL Server Database Engine периодически проверяет задачи на состояние взаимоблокировки. Если монитор обнаруживает цикличную зависимость, то выбирается одна задача, для которой транзакция будет завершена с ошибкой. Это позволяет другой задаче завершить свою транзакцию. Позднее приложение может повторно выполнить транзакцию, которая завершилась с ошибкой, обычно после того как другая транзакция (бывшая в состоянии взаимоблокировки) завершится.

На рисунке транзакция Т1 зависит от транзакции Т2 для ресурса блокировки таблицы **Деталь**. Аналогично транзакция Т2 зависит от транзакции Т1 для ресурса блокировки таблицы **Поставщик**. Так как эти зависимости из одного цикла, возникает взаимоблокировка транзакций T1 и T2.



## Отображение сведений о блокировках

* [SQL Server 2008](http://msdn.microsoft.com/ru-ru/library/ms177475(v=SQL.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/ru-ru/library/ms177475(v=SQL.90).aspx)

SQL Server предоставляет несколько способов получить сведения об активности текущей блокировки на экземпляре компонента SQL Server Database Engine.

|  |  |
| --- | --- |
| **Подраздел** | **Описание** |
| **Server Profiler Locks** | С помощью приложения SQL Server Profiler можно задать категорию событий блокировки, чтобы получать сведения о событиях блокировки в журнал трассировки. |
| **SQL Server, объект Locks** | В системном мониторе можно указать счетчики объекта блокировок, чтобы контролировать уровень блокирования на экземпляре компонента Database Engine. |
| **sys.dm\_tran\_locks (Transact-SQL)** | Можно выполнить запрос к представлению динамического управления **sys.dm\_tran\_locks**, чтобы получить сведения о текущем состоянии блокировки экземпляра компонента Database Engine. |
| **Xevents** | Получать сообщения о блокировках можно при помощи механизма расширенных событий. При использовании **Xevents** наряду с событиями можно отслеживать данные контекста выполнения, такие как стеки вызовов Transact SQL или обработчиков плана выполнения. События **Xevents** могут захватываться несколькими типами различных выходов, включая трассировку событий для Windows (**ETW**). |

## Режимы транзакций

Транзакция является единственной единицей работы. Если транзакция выполнена успешно, все модификации данных, сделанные в течение транзакции, принимаются и становятся постоянной частью базы данных. Если в результате выполнения транзакции происходят ошибки и должна быть произведена отмена или выполнен откат, все модификации данных будут отменены.

SQL Server работает в следующих режимах транзакций.

* **Автоматическое принятие транзакций**

Каждая отдельная инструкция является транзакцией.

* **Явные транзакции**

Каждая транзакция явно начинается с инструкции BEGIN TRANSACTION и явно заканчивается инструкцией COMMIT или ROLLBACK.

* **Неявные транзакции**

Новая транзакция неявно начинается, когда предыдущая транзакция завершена, но каждая транзакция явно завершается инструкцией COMMIT или ROLLBACK.

* **Распределенные транзакции**

Распределенные транзакции выполняются над ресурсами на двух или более серверах.

### Автоматическая фиксация транзакций

Режим автоматической фиксации транзакций используется компонентом Database Engine по умолчанию. После завершения каждая инструкция Transact-SQL фиксируется или откатывается назад. Если инструкция выполняется без ошибок, она фиксируется. В противном случае она откатывается назад. Соединение с экземпляром компонента Database Engine работает в режиме автоматической фиксации везде, где не используются явные или неявные транзакции.

Соединение с экземпляром компонента Database Engine работает в режиме автоматической фиксации до тех пор, пока явная транзакция не запускается инструкцией **BEGIN TRANSACTION** или не включается режим неявной транзакции. Если явная транзакция фиксируется или откатывается назад или выключается режим неявной транзакции, то соединение переходит в режим автоматической фиксации транзакций.

Присвоение параметру **IMPLICIT\_TRANSACTIONS** значения **ON** устанавливает для соединения режим неявных транзакций. Значение **OFF** возвращает соединение в режим с автоматической фиксацией транзакций.

**Ошибки времени компиляции и выполнения**

Иногда в режиме автоматической фиксации экземпляр компонента Database Engine откатывает назад весь пакет вместо одной инструкции SQL. Это происходит, если ошибка возникает во время компиляции, а не во время выполнения. Ошибка компиляции не позволяет компоненту Database Engine построить план выполнения, поэтому пакет не выполняется. Поскольку произошел откат назад всех инструкций, предшествующих неправильной инструкции, нельзя выполнить весь пакет.

В следующем примере ни одна из инструкций INSERT в третьем пакете не выполнится из-за ошибки компиляции. При этом произойдет откат первых двух инструкций INSERT и они не будут выполняться ни при каких условиях.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBatch VALUSE (3, 'ccc'); -- Синтаксическая ошибка.

GO

SELECT \* FROM TestBatch; -- Не возвращает ни одной строки.

GO

В следующем примере третья инструкция INSERT вызывает ошибку повторения первичного ключа во время выполнения. Первые две инструкции INSERT выполняются успешно и фиксируются, поэтому остаются после возникновения ошибки времени выполнения.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBatch VALUES (1, 'ccc'); -- Дублирование первичного ключа.

GO

SELECT \* FROM TestBatch; -- Возвращает строки 1 и 2.

GO

Компонент Database Engine использует отложенное разрешение имен, при котором имена объектов разрешаются только во время выполнения. В следующем примере первые две инструкции INSERT выполняются и фиксируются, а вставленные строки остаются в таблице TestBatch после того, как третья инструкция INSERT вызывает ошибку времени выполнения, ссылаясь на таблицу, которой не существует.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBch VALUES (3, 'ccc'); -- Неверное имя таблицы.

GO

SELECT \* FROM TestBatch; -- Возвращает строки 1 и 2.

GO

### Явные транзакции

Явными транзакциями являются транзакции, для которых явно назначаются запуск и остановка. Сценарии Transact-SQL используют для определения явных транзакций инструкции Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION и ROLLBACK WORK.

**BEGIN TRANSACTION**

Отмечает точку запуска явной транзакции для соединения.

**COMMIT TRANSACTION или COMMIT WORK**

Используется для успешного завершения транзакции, если не было ошибок. Все изменения данных, сделанные в транзакции, становятся постоянной частью базы данных. Ресурсы, заблокированные транзакцией, высвобождаются.

**ROLLBACK TRANSACTION или ROLLBACK WORK**

Используется для удаления транзакции, если были ошибки. Все измененные транзакцией данные возвращаются в то состояние, в котором они были в момент запуска транзакции. Ресурсы, заблокированные транзакцией, высвобождаются.

Режим явной транзакции работает только во время выполнения транзакции. После завершения транзакции соединение возвращается в тот режим транзакции, в котором оно было до запуска явной транзакции, либо в неявный режим, либо в режим автоматической фиксации.

### Неявные транзакции

Если соединение выполняется в режиме неявных транзакций, экземпляр компонента SQL Server Database Engine автоматически начинает новую транзакцию после фиксации или отката текущей. Для запуска таких транзакций ничего делать не нужно - необходимо только фиксировать или выполнять откат каждой транзакции. Режим неявных транзакций формирует непрерывную цепь транзакций.

После установления на соединении режима неявных транзакций экземпляр компонента Database Engine автоматически запускает транзакцию, если вначале выполняет любую из следующих инструкций:

|  |  |
| --- | --- |
| ALTER TABLE | INSERT |
| CREATE | OPEN |
| DELETE | REVOKE |
| DROP | SELECT |
| FETCH | TRUNCATE TABLE |
| GRANT | UPDATE |

Транзакция продолжает оставаться активной до тех пор, пока не будет выдана инструкция COMMIT или ROLLBACK. После фиксации или отката первой транзакции экземпляр компонента Database Engine автоматически запускает новую транзакцию каждый раз, когда на соединении выполняется какая-либо из этих инструкций. Экземпляр продолжает формировать цепь неявных транзакций до тех пор, пока не будет выключен режим неявных транзакций.

Режим неявных транзакций устанавливается или при помощи инструкции языка Transact-SQL SET, или функциями и методами API базы данных.

Инструкция SET IMPLICIT\_TRANSACTIONS OFF используется для отключения режима неявных транзакций. Инструкции COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION или ROLLBACK WORK используются для завершения каждой транзакции.

CREATE TABLE ImplicitTran (ColA int PRIMARY KEY, ColB char(3) NOT NULL);

GO

SET IMPLICIT\_TRANSACTIONS ON;

GO

-- первая неявная транзакции инициируется командой INSERT.

INSERT INTO ImplicitTran VALUES (1, 'aaa');

GO

INSERT INTO ImplicitTran VALUES (2, 'bbb');

GO

-- фиксация первой транзакции.

COMMIT TRANSACTION;

GO

-- вторая неявная транзакции инициируется командой SELECT.

SELECT COUNT(\*) FROM ImplicitTran;

GO

INSERT INTO ImplicitTran VALUES (3, 'ccc');

GO

SELECT \* FROM ImplicitTran;

GO

-- фиксация второй транзакции.

COMMIT TRANSACTION;

GO

SET IMPLICIT\_TRANSACTIONS OFF;

GO

### Распределенные транзакции

Распределенные транзакции выполняются на двух или более серверах, которые называются диспетчерами ресурсов. Управление транзакцией должно координироваться между диспетчерами ресурсов компонентом сервера, который называется диспетчером транзакций. Каждый экземпляр компонента SQL Server Database Engine может действовать как диспетчер ресурсов в распределенных транзакциях, координируемых диспетчерами транзакций, например, координатором распределенных транзакций (MS DTC) или другими диспетчерами транзакций, поддерживающими спецификацию Open Group XA обработки распределенных транзакций.

Транзакция в отдельном экземпляре компонента Database Engine, распространяющаяся на несколько источников данных, в действительности является распределенной транзакцией. Экземпляр управляет распределенной транзакцией на внутреннем уровне, для пользователя она действует как локальная транзакция.

В приложении управление распределенной транзакцией во многом похоже на управление локальной. В конце транзакции приложение запрашивает ее фиксацию или откат. Распределенной фиксацией диспетчер транзакций должен управлять иначе, чтобы свести к минимуму риск сбоя сети, в результате которого одни диспетчеры ресурсов могут фиксировать транзакцию, тогда как другие будут выполнять ее откат. Выход из положения заключается в двухфазном процессе фиксации (фаза подготовки и фаза фиксации), который называется двухфазной фиксацией (2PC).

* **Фаза подготовки**

Когда диспетчер транзакции получает запрос на фиксацию, он отправляет команду подготовки всем диспетчерам ресурсов, занятым в транзакции. Каждый диспетчер ресурсов всемерно обеспечивает устойчивость транзакции, а все буферы, в которых хранятся образы журналов для этой транзакции, записываются на диск. По мере того, как каждый диспетчер ресурсов завершает фазу подготовки, он возвращает диспетчеру транзакций значение успешного или неуспешного завершения подготовки.

* **Фаза фиксации**

Если диспетчер транзакций получает значения успешного завершения подготовки от всех диспетчеров ресурсов, то он отправляет команду фиксации каждому диспетчеру ресурсов. После этого диспетчеры ресурсов могут завершить фиксацию. Если все диспетчеры ресурсов сообщают об успешной фиксации, то диспетчер транзакций отправляет уведомление приложению. Если какой-либо диспетчер ресурсов сообщил о неуспешном завершении подготовки, то диспетчер транзакций отправляет команду отката всем диспетчерам ресурсов и сообщает приложению о сбое фиксации.

## Уровни изоляции в ядре СУБД

Транзакции указывают уровень изоляции, который определяет степень, до которой одна транзакция должна быть изолирована от изменений ресурса или данных, произведенных другими транзакциями.

Уровни изоляции транзакций контролируют следующие параметры:

* Применение и типы блокировки при чтении данных.
* Время удержания блокировок чтения.
* Использование операции чтения ссылок на строки, измененные другой транзакцией.
  + Блокировка до тех пор, пока не будет снята монопольная блокировка строки.
  + Извлечение зафиксированной версии строки, которая существовала в то время, когда началось выполнение инструкции или транзакции.
  + Считывание незафиксированного изменения данных.

Выбор уровня изоляции транзакции не влияет на блокировки, примененные для защиты изменений данных. Транзакция всегда вызывает монопольную блокировку любых данных, которые она изменяет, и держит блокировку до тех пор, пока транзакция не завершится, независимо от уровня изоляции, установленного для транзакции. Для операций чтения уровни изоляции транзакций, в основном, определяют уровень защиты от эффектов изменений, сделанных другими транзакциями.

Более низкий уровень изоляции увеличивает возможность получения доступа к данным несколькими пользователями одновременно, но увеличивает число негативных эффектов одновременного доступа (таких как «грязное» чтение или потерянные обновления), с которыми может столкнуться пользователь. Наоборот, более высокий уровень изоляции уменьшает число негативных эффектов одновременного доступа, с которыми может столкнуться пользователь, но требует больше системных ресурсов и увеличивает шанс того, что одна транзакция блокирует другую. Выбор соответствующего уровня изоляции зависит от баланса между требованиями к целостности данных приложения и издержек каждого уровня изоляции. Самый высокий уровень изоляции — изоляция упорядочиваемых транзакций — гарантирует, что транзакция извлечет в точности те же данные при каждой операции чтения, но достигается это применением уровня блокировки, при котором очень вероятно влияние на других пользователей в многопользовательских системах. Самый низкий уровень изоляции — изоляция незафиксированного чтения — может извлечь данные, которые были изменены, но не зафиксированы другой транзакцией. При изоляции незафиксированного чтения могут проявиться все негативные эффекты одновременного доступа, при таком уровне нет блокировки чтения или управления версиями, так что издержки минимальны.

Стандарт SQL-99 определяет следующие уровни изоляции, каждый из которых поддерживается компонентом Microsoft SQL Server Database Engine:

* изоляция незафиксированного чтения (самый низкий уровень, при котором транзакции изолируются до такой степени, чтобы только уберечь от считывания физически поврежденных данных);
* изоляция зафиксированного чтения (уровень компонента Database Engine по умолчанию);
* изоляция повторяющегося чтения;
* изоляция упорядочиваемых транзакций (самый высокий уровень, при котором транзакции полностью изолированы друг от друга).

SQL Server 2008 также поддерживает два уровня изоляции транзакций, которые используют управление версиями строк. Один является новой реализацией уровня изоляции зафиксированного чтения, а второй **-** изоляция моментального снимка.

* Если параметру базы данных READ\_COMMITTED\_SNAPSHOT присвоено значение ON, уровень изоляции зафиксированного чтения использует управление версиями строк для обеспечения согласованности считывания на уровне инструкций. Операции чтения требуют применения только блокировок уровня таблицы SCH-S и не допускают применения блокировок строк или страниц. Если параметр базы данных READ\_COMMITED\_SNAPSHOT установлен в OFF (значение по умолчанию), то изоляция зафиксированного чтения работает так же, как и в предыдущих версиях SQL Server. Обе реализации согласуются с определением ANSI для уровня изоляции зафиксированного чтения.
* Уровень изоляции моментальных снимков использует управление версиями строк для обеспечения согласованности чтения на уровне транзакций. Операции чтения применяют только блокировки таблицы SCH-S и не применяют блокировок строк или страниц. Если считываемые строки изменены другой транзакцией, то извлекается версия строки, которая существовала в момент запуска транзакции. Изоляция моментальных снимков включается, если параметр базы данных ALLOW\_SNAPSHOT\_ISOLATION установлен в ON. По умолчанию для пользовательских баз данных этот параметр установлен в OFF.

**Установка уровня изоляции транзакции**

Управление поведением блокировки и версиями строк инструкций Transact-SQL, выданных при подключении к SQL Server, осуществляется при помощи команды **SET TRANSACTION ISOLATION LEVEL**.

Одновременно может быть установлен только один параметр уровня изоляции, который продолжает действовать для текущего соединения до тех пор, пока не будет явно изменен. Все операции считывания, выполняемые в рамках транзакции, функционируют в соответствии с правилами уровня изоляции, если только табличная подсказка в предложении FROM инструкции не указывает на другое поведение блокировки или управления версиями строк для таблицы.

SET TRANSACTION ISOLATION LEVEL

    { READ UNCOMMITTED

    | READ COMMITTED

    | REPEATABLE READ

    | SNAPSHOT

    | SERIALIZABLE

    }

**READ UNCOMMITTED**

Указывает, что инструкции могут считывать строки, которые были изменены другими транзакциями, но еще не были зафиксированы.

Транзакции, работающие на уровне READ UNCOMMITTED, не используют разделяемые блокировки, чтобы предотвратить изменение считываемых текущей транзакцией данных другими транзакциями. Транзакции READ UNCOMMITTED также не блокируются монопольными блокировками, которые не позволили бы текущей транзакции считывать измененные другими транзакциями, но не зафиксированные строки. Установка этого параметра позволяет считывать незафиксированные изменения, которые называются недействительными результатами чтения. Значения в данных могут быть изменены и до окончания транзакции строки могут появляться и исчезать в наборе данных. Этот аргумент действует так же, как и настройка NOLOCK всех таблиц во всех инструкциях SELECT в транзакции. Это наименьшее ограничение уровней изоляции.

В SQL Server 2008 конфликты блокировок при защите транзакций от недействительных результатов чтения незафиксированных изменений данных можно сократить с помощью следующего:

* + уровня изоляции READ COMMITTED с параметром базы данных READ\_COMMITTED\_SNAPSHOT, находящимся в состоянии ON;
  + уровня изоляции моментального снимка (SNAPSHOT).

**READ COMMITTED**

Указывает, что инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы. Это предотвращает недействительные результаты чтения. Данные могут быть изменены другими транзакциями между отдельными инструкциями в текущей транзакции, результатом чего будет неповторяющееся считывание или фиктивные данные. Этот параметр в SQL Server установлен по умолчанию.

Поведение READ COMMITTED зависит от настройки аргумента базы данных READ\_COMMITTED\_SNAPSHOT.

* Если параметр READ\_COMMITTED\_SNAPSHOT находится в состоянии OFF (по умолчанию), компонент Database Engine при выполнении операций считывания текущей транзакцией использует разделяемые блокировки для предотвращения изменения строк другими транзакциями. Разделяемые блокировки также блокируют инструкции от считывания строк, измененных другими транзакциями, пока не завершится другая транзакция. По завершении инструкции разделяемые блокировки снимаются.
* Если параметр READ\_COMMITTED\_SNAPSHOT находится в состоянии ON, компонент Database Engine использует управление версиями строк для представления каждой инструкции согласованного на уровне транзакций моментального снимка данных в том виде, который они имели на момент начала выполнения инструкции. Для защиты данных от обновления другими транзакциями блокировки не используются.

При установке параметра READ\_COMMITTED\_SNAPSHOT разрешается только то соединение с базой данных, которое выполняет команду ALTER DATABASE. До завершения инструкции ALTER DATABASE в базе данных не должно быть других открытых соединений. База данных не обязательно должна находиться в однопользовательском режиме.

**REPEATABLE READ**

Указывает на то, что инструкции не могут считывать данные, которые были изменены, но еще не зафиксированы другими транзакциями, а также на то, что другие транзакции не могут изменять данные, читаемые текущей транзакцией, до ее завершения.

Разделяемые блокировки применяются ко всем данным, считываемым любой инструкцией транзакции, и сохраняются до ее завершения. Это запрещает другим транзакциям изменять строки, считываемые текущей транзакцией. Другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведет к считыванию фантома. Учитывая то, что разделяемые блокировки сохраняются до завершения транзакции и не снимаются в конце каждой инструкции, степень совпадений ниже, чем при уровне изоляции по умолчанию READ COMMITTED. Используйте этот аргумент только в случае необходимости.

**SNAPSHOT**

Указывает на то, что данные, считанные любой инструкцией транзакции, будут согласованы на уровне транзакции с версией данных, существовавших в ее начале. Транзакция распознает только те изменения, которые были зафиксированы до ее начала. Инструкции, выполняемые текущей транзакцией, не видят изменений данных, произведенных другими транзакциями после запуска текущей транзакции. Таким образом достигается эффект получения инструкциями в транзакции моментального снимка зафиксированных данных на момент запуска транзакции.

Транзакции моментальных снимков не требуют блокировки при считывании данных, за исключением случаев восстановления базы данных. Считывание данных транзакциями моментальных снимков не блокирует запись данных другими транзакциями. Транзакции, осуществляющие запись данных, не блокируют считывание данных транзакциями моментальных снимков.

На этапе отката восстановления базы данных транзакция моментальных снимков запросит блокировку, если будет предпринята попытка считывания данных, заблокированных другой откатываемой транзакцией. Блокировка транзакции моментальных снимков сохраняется до завершения отката. Блокировка снимается сразу после предоставления.

Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить аргумент базы данных ALLOW\_SNAPSHOT\_ISOLATION в ON. Если транзакция с уровнем изоляции моментальных снимков обращается к данным из нескольких баз данных, аргумент ALLOW\_SNAPSHOT\_ISOLATION должен быть включен в каждой базе данных.

Невозможно изменить на уровень изоляции моментальных снимков уровень изоляции транзакции, запущенной с другим уровнем изоляции; в этом случае транзакция будет прервана. Если транзакция запущена с уровнем изоляции моментальных снимков, ее уровень изоляции можно изменять. Транзакция начинается в момент первого доступа к данным.

Транзакция, работающая с уровнем изоляции моментальных снимков, может просматривать внесенные ею изменения. Например, если транзакция выполняет инструкцию UPDATE, а затем инструкцию SELECT для одной и той же таблицы, измененные данные будут включены в результирующий набор.

**SERIALIZABLE**

Указывает следующее.

* Инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы.
* Другие транзакции не могут изменять данные, считываемые текущей транзакцией, до ее завершения.
* Другие транзакции не могут вставлять новые строки со значениями ключа, которые входят в диапазон ключей, считываемых инструкциями текущей транзакции, до ее завершения.
* Блокировка диапазона устанавливается в диапазоне значений ключа, соответствующих условиям поиска любой инструкции, выполненной во время транзакции. Обновление и вставка строк, удовлетворяющих инструкциям текущей транзакции, блокируется для других транзакций. Это гарантирует, что если какая-либо инструкция транзакции выполняется повторно, она будет считывать тот же самый набор строк. Блокировки диапазона сохраняются до завершения транзакции. Это самый строгий уровень изоляции, поскольку он блокирует целые диапазоны ключей и сохраняет блокировку до завершения транзакции.

Из-за низкого параллелизма этот параметр рекомендуется использовать только при необходимости.

Следующая таблица показывает негативные эффекты одновременного доступа, допускаемые различными уровнями изоляции.

|  |  |  |  |
| --- | --- | --- | --- |
| **Уровень изоляции** | **«Грязное» чтение** | **Неповторяющееся чтение** | **Фантомное чтение** |
| **незафиксированного чтения** | Да | Да | Да |
| **зафиксированного чтения** | Нет | Да | Да |
| **повторяющегося чтения** | Нет | Нет | Да |
| **моментального снимка** | Нет | Нет | Нет |
| **упорядочиваемых транзакций** | Нет | Нет | Нет |

Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания. Разделяемые блокировки, применяемые для **READ COMMITTED** или **REPEATABLE READ**, как правило, являются блокировками строк, но при этом, если в процессе считывания идет обращение к большому числу строк, блокировка строк может быть расширена до блокировки страниц или таблиц. Если строка была изменена транзакцией после считывания, для защиты такой строки транзакция применяет монопольную блокировку, которая сохраняется до завершения транзакции. Например, если транзакция **REPEATABLE READ** имеет разделяемую блокировку строки и при этом изменяет ее, совмещаемая блокировка преобразуется в монопольную.

В любой момент транзакции можно переключиться с одного уровня изоляции на другой, однако есть одно исключение. Это смена уровня изоляции на уровень изоляции **SNAPSHOT**. Такая смена приводит к ошибке и откату транзакции. Однако для транзакции, которая была начата с уровнем изоляции **SNAPSHOT**, можно установить любой другой уровень изоляции.

Когда для транзакции изменяется уровень изоляции, ресурсы, которые считываются после изменения, защищаются в соответствии с правилами нового уровня. Ресурсы, которые считываются до изменения, остаются защищенными с соответствии с правилами предыдущего уровня. Например, если для транзакции уровень изоляции изменяется с **READ COMMITTED** на **SERIALIZABLE**, то совмещаемые блокировки, полученные после изменения, будут удерживаться до завершения транзакции.

Если инструкция **SET TRANSACTION ISOLATION LEVEL** использовалась в хранимой процедуре или триггере, то при возврате управления из них уровень изоляции будет изменен на тот, который действовал на момент их вызова. Например, если уровень изоляции **REPEATABLE READ** устанавливается в пакете, а пакет затем вызывает хранимую процедуру, которая меняет уровень изоляции на **SERIALIZABLE**, при возвращении хранимой процедурой управления пакету, настройки уровня изоляции меняются назад на **REPEATABLE READ**.

В следующем примере устанавливается уровень изоляции для сеанса. Для каждой последующей инструкции Transact-SQL SQL Server сохраняет все разделяемые блокировки до конца транзакции.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

GO

BEGIN TRANSACTION;

GO

SELECT \*

    FROM HumanResources.EmployeePayHistory;

GO

SELECT \*

    FROM HumanResources.Department;

GO

COMMIT TRANSACTION;

GO