

Основные понятия и определения

Термин «база данных» (database) был введен в обиход в области вычислительной техники примерно в 1962 году. Этот термин страдает от обилия различных интерпретаций. Примеры определения термина «база данных» из авторитетных источников:

База данных (БД) - совокупность данных, организованных по определенным правилам, предусматривающая общие принципы описания, хранения и манипулирования данными, независимая от прикладных программ. Является информационной моделью предметной области.

[К. Дейт, Введение в системы баз данных, 1980]

База данных – совокупность хранимых **операционных** данных, используемых прикладными системами некоторого предприятия.

[К. Дейт, Введение в системы баз данных, 6-е издание, 2000]

База данных состоит из некоторого набора **постоянных** данных, которые используются прикладными системами для какого-либо предприятия.

[К. Дейт, Введение в системы баз данных, 7-е издание, 2001]

База данных – это некоторый набор **перманентных** (постоянных) данных, используемых прикладными системами какого-либо предприятия.

1. Операционные данные – мир **OnLine Transaction Processing** (Оперативная обработка транзакций) или **OLTP**.
2. Перманентные данные – мир **OnLine Analytical Processing** (Аналитическая обработка в реальном времени) или **OLAP**. OLAP используется в **системах поддержки принятия решений** (Decision Support System – DSS) и **управленческих информационных системах** (Executive Information System – EIS). Базу данных для поддержки принятия решения обычно называют **хранилищем данных** (data warehouse).

[Джеффри Д. Ульман, Дженифер Уидом, Введение в системы баз данных, 2000]

По сути дела БД – это просто множество информации, существующее долгое время, часто в течении многих лет. Обычно термином "база данных" обозначается множество данных, управляемое СУБД, или просто СБД.

[Д. Кренке, Теория и практика построения баз данных, 2003]

База данных – это **самодокументированное** собрание **интегрированных** записей. Важно понять обе части этого определения.

1. БД является самодокументированной (self describing): она содержит описание собственной структуры. Это описание называется **словарем данных** (data dictionary), **каталогом данных** (data directory) или **метаданными** (metadata).
2. БД – это собрание интегрированных записей: она содержит:
 - a. **Файлы данных**,
 - b. **Метаданные**,
 - c. **Индексы** (indexes), которые представляют связи между данными, а также служат для повышения производительности приложений базы данных.
 - d. Может содержать **метаданные приложений** (application metadata).
3. БД является **информационной моделью** пользовательской модели (user model) предметной области.

База данных - совокупность взаимосвязанных данных некоторой предметной области, хранимых в памяти ЭВМ и организованных таким образом, что эти данные могут быть использованы для решения многих задач многими пользователями. Основные требования к организации данных:

1. Неизбыточность данных - каждое данное присутствует в БД в единственном экземпляре.
2. Совместное использование данных многими пользователями.
3. Эффективность доступа к БД - высокое быстродействие, т. е. малое время отклика на запрос.
4. Целостность данных - соответствие имеющейся в БД информации её внутренней логике, структуре и всем явно заданным правилам.
5. Безопасность данных – защита данных от преднамеренного или непреднамеренного искажения или разрушения данных.
6. Восстановление данных после программных и аппаратных сбоев.
7. Независимость данных от прикладных программ.

Существует множество других определений, отражающих скорее субъективное мнение тех или иных авторов о том, что означает база данных в их понимании, однако общепризнанная единая формулировка отсутствует.

Система управления базами данных (СУБД) - приложение, обеспечивающее создание, хранение, обновление и поиск информации в базах данных.

Система баз данных (или **банк данных** в отечественной терминологии) - совокупность одной или нескольких баз данных и комплекса информационных, программных и технических средств, обеспечивающих накопление, обновление, корректировку и многоаспектное использование данных в интересах пользователей. (См. Григорьев Ю.А., Ревунков Г.И. Банки данных. – М.: МГТУ, 2002.)

Состав системы баз данных

- Одна или несколько баз данных.
- Аппаратное обеспечение.
- Программное обеспечение.
 - СУБД.
 - Прочие компоненты: утилиты, средства разработки приложений, средства проектирования, генераторы отчетов, диспетчер транзакций и т. д.
- Пользователи.
 - Прикладные программисты, которые отвечают за написание прикладных пакетных и интерактивных программ.
 - Конечные пользователи, которые получают доступ к базе данных, применяя одно из интерактивных приложений, интерфейс, интегрированный в программное обеспечение самой СУБД, командный интерфейс, реализуемый процессором языка запросов или некомандный интерфейс, основанный на меню и формах.
 - Администраторы базы данных.

Основные функции СУБД

- **Непосредственное управление данными во внешней памяти.** В развитых СУБД пользователи не обязаны знать, как организованы файлы данных.
- **Управление буферами оперативной памяти.** Для увеличения скорости обмена данными с внешней памятью используется буферизация данных в оперативной памяти. В развитых СУБД поддерживается собственный набор буферов оперативной памяти с собственной дисциплиной замены буферов, даже если операционная система производит общесистемную буферизацию. Существует даже отдельное направление СУБД, которое ориентировано на постоянное присутствие в оперативной памяти всей БД. Это направление основывается на предположении, что объем оперативной памяти компьютеров настолько велик, что можно не беспокоиться о буферизации.
- **Управление транзакциями.** Транзакция – это последовательность операций над данными, рассматриваемая СУБД как единое целое. Реализуется принцип «либо все, либо ничего». Поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД. Но понятие транзакции гораздо более важно в многопользовательских СУБД. То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает удобным использование понятия транзакции как единицы активности пользователя по отношению к БД.
- **Журнализация.** Одним из основных требований к СУБД является требование надежного хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризуемые потерей информации на носителях внешней памяти. В любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений БД. Журнал – это особая часть БД, недоступная пользователям СУБД (иногда поддерживаются две копии журнала, располагаемые на разных физических носителях), в которую поступают записи обо всех изменениях основной части БД.
- **Поддержка языков БД.** Для работы с БД используются специальные языки, называемые языками баз данных. В ранних СУБД (иерархических и сетевых) поддерживалось несколько специализированных по своим функциям языков. В современных СУБД (реляционных) поддерживается язык SQL (Structured Query Language), содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс.

Основные компонентами СУБД

- **Ядро**, которое отвечает за управление данными во внешней и оперативной памяти и журнализацию.
- **Процессор языка базы данных**, обеспечивающий оптимизацию запросов на извлечение и изменение данных, и создание, как правило, машинно-независимого исполняемого внутреннего кода,
- **Подсистема поддержки времени исполнения**, которая интерпретирует программы манипуляции данными, создающие пользовательский интерфейс с СУБД.
- **Сервисные программы** (внешние утилиты), обеспечивающие ряд дополнительных возможностей по обслуживанию информационной системы.

Классификация СУБД

Классификация СУБД может выполняться по нескольким признакам:

1. По модели данных:

a. Дореляционные

- i. Инвертированные списки (файлы)
- ii. Иерархические
- iii. Сетевые

b. Реляционные

c. Постреляционные

БД на основе инвертированных списков представляет собой совокупность файлов, содержащих записи (таблиц). Для записей в файле определен некоторый порядок, диктуемый физической организацией данных. Для каждого файла может быть определено произвольное число других упорядочений на основании значений некоторых полей записей (инвертированных списков). Обычно для этого используются индексы. В такой модели данных отсутствуют ограничения целостности как таковые. Все ограничения на возможные экземпляры БД задаются теми программами, которые работают с БД. Одно из немногих ограничений, которое все-таки может присутствовать - это ограничение, задаваемое уникальным индексом.

Иерархическая модель БД состоит из объектов с указателями от родительских объектов к потомкам, соединяя вместе связанную информацию. Иерархические БД могут быть представлены как дерево.

К основным понятиям сетевой модели БД относятся: элемент (узел), связь. Узел — это совокупность атрибутов данных, описывающих некоторый объект. Сетевые БД могут быть представлены в виде графа. В сетевой БД логика процедуры выборки данных зависит от физической организации этих данных. Поэтому эта модель не является полностью независимой от приложения. Другими словами, если необходимо изменить структуру данных, то нужно изменить и приложение.

Реляционная модель данных включает следующие компоненты:

- **Структурный** (данные в базе данных представляют собой набор отношений),
- **Целостностный** (отношения (таблицы) отвечают определенным условиям целостности),
- **Манипуляционный** (манипулирования отношениями осуществляется средствами реляционной алгебры и/или реляционного исчисления). Кроме того, в состав реляционной модели данных включают теорию нормализации.

В 1985 году доктор Кодд сформулировал двенадцать правил, которым должна соответствовать настоящая реляционная база данных. Они являются полуофициальным определением понятия реляционная база данных. Строгое изложение теории реляционных баз данных (реляционной модели данных) в современном понимании можно найти в книге К. Дж. Дейта.

2. По архитектуре организации хранения данных

- a. **локальные** СУБД (все части локальной СУБД размещаются на одном компьютере)
- b. **распределенные** СУБД (части СУБД могут размещаться на двух и более компьютерах)

3. По способу доступа к БД (а может быть по местоположению БД):

- a. **Файл-серверные.** При работе в архитектуре "файл-сервер" БД и приложение расположены на файловом сервере сети. Возможна многопользовательская работа с одной и той же БД, когда каждый пользователь со своего компьютера запускает приложение, расположенное на сетевом сервере. Тогда на компьютере пользователя запускается копия приложения. По каждому запросу к БД из приложения данные из таблиц БД перегоняются на компьютер пользователя, независимо от того, сколько реально нужно данных для выполнения запроса. После этого выполняется запрос. Каждый пользователь имеет на своем компьютере локальную копию данных, время от времени обновляемых из реальной БД, расположенной на сетевом сервере. При этом изменения, которые каждый пользователь вносит в БД, могут быть до определенного момента неизвестны другим пользователям, что делает актуальной задачу систематического обновления данных на компьютере пользователя из реальной базы данных. Другой актуальной задачей является блокирование записей, которые изменяются одним из пользователей; это необходимо для того, чтобы в это время другой пользователь не внес изменений в те же данные. В архитектуре "файл-сервер" вся тяжесть выполнения запросов к базе данных и управления целостностью базы данных ложится на приложение пользователя. База данных на сервере является пассивным источником данных.
- b. **Клиент-серверные.** Клиент-сервер - сетевая архитектура, в которой устройства являются либо *клиентами*, либо *серверами*. Клиентом (front end) является запрашивающая машина, сервером (back end) - машина, которая отвечает на запрос. Оба термина (клиент и сервер) могут применяться как к физическим устройствам, так и к программному обеспечению. Характерной особенностью архитектуры "клиент-сервер" является перенос вычислительной нагрузки на сервер базы данных (sql-сервер) и максимальная разгрузка приложения клиента от вычислительной работы, а также существенное укрепление безопасности данных – как от злонамеренных, так и просто ошибочных изменений. БД в этом случае помещается на сетевом

сервере, как и в архитектуре "файл-сервер", однако прямого доступа к базе данных из приложений не происходит. Функцию прямого обращения к базе данных осуществляет СУБД. При этом ресурсы клиентского компьютера не участвуют в физическом выполнении запроса; клиентский компьютер лишь отсылает запрос к СУБД и получает результат, после чего интерпретирует его необходимым образом и представляет пользователю. Так как клиентскому приложению посыпается результат выполнения запроса, по сети "путешествуют" только те данные, которые необходимы клиенту. В итоге снижается нагрузка на сеть. Кроме того, СУБД, если это возможно, оптимизирует полученный запрос таким образом, чтобы он был выполнен в минимальное время с наименьшими накладными расходами. При выполнении запросов сервером существенно повышается степень безопасности данных, поскольку правила целостности данных определяются в базе данных на сервере и являются едиными для всех приложений, использующих эту БД. Таким образом, исключается возможность определения противоречивых правил поддержания целостности. Аппарат транзакций, поддерживаемый СУБД, позволяет исключить одновременное изменение одних и тех же данных различными пользователями и предоставляет возможность откатов к первоначальным значениям при внесении в БД изменений, закончившихся аварийно.

- c. **Встраиваемые.** Встраиваемая СУБД — библиотека, которая позволяет унифицированным образом хранить большие объёмы данных на локальной машине. Доступ к данным может происходить через SQL либо через особые функции СУБД. Встраиваемые СУБД быстрее обычных клиент-серверных и не требуют установки сервера, поэтому востребованы в локальном ПО, которое имеет дело с большими объёмами данных.
- d. **Сервисно-ориентированные.** БД является хранилищем сообщений, промежуточных состояний, метаинформации об очередях сообщений и сервисах. Отправка сообщений в очередь и прием сообщений из очереди производится в одной транзакции с изменением данных, что обеспечивает транзакционную целостность системы. Так как очереди сообщений и данные хранятся и обрабатываются в базе единообразно, это обеспечивает гарантированную доставку и обработку сообщений в случае сбоев оборудования или питания с таким же успехом, как и прочих данных, хранящихся в той же базе данных. Кроме этого, в базе данных хранится информация о самих сервисах и обрабатываемых ими очередях сообщений, что обеспечивает восстановление после сбоя состояний не только данных и сообщений, но и настроек сервисов и очередей сообщений.
- e. **Прочие.** Пространственная (spatial database), Временная, или темпоральная (temporal database), Пространственно-временная (spatial-temporal database).

Реляционная модель данных

Введение

Основоположником теории реляционных баз данных является британский учёный Эдгар Кодд, который в 1970 году опубликовал первую работу по реляционной модели данных. Наиболее распространенная трактовка реляционной модели данных принадлежит Кристоферу Дейту. Согласно Дейту, реляционная модель состоит из трех частей: *структурной, целостностной и манипуляционной*.

Структурная часть реляционной модели

Структурная часть реляционной модели описывает, из каких объектов состоит реляционная модель. Постулируется, что основной структурой данных, используемой в реляционной модели, являются нормализованные «п-арные» отношения. Основными понятиями структурной части реляционной модели являются *тип данных, домен, атрибут, схема отношения, схема базы данных, кортеж, отношение, потенциальный, первичный и альтернативные ключи, реляционная база данных*.

Понятие *типа данных* в реляционной модели полностью адекватно понятию типа данных в языках программирования.

Понятие *домена* можно считать уточнением типа данных. Домен можно рассматривать как подмножество значений некоторого типа данных, имеющих определенный смысл. Домен характеризуется следующими свойствами:

- домен имеет уникальное имя (в пределах базы данных),
- домен определен на некотором типе данных или на другом домене,
- домен может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для данного домена,
- домен несет определенную смысловую нагрузку.

Говорят, что домен отражает семантику, определенную предметной областью. Может быть несколько доменов, совпадающих как подмножества, но несущие различный смысл. Основное значение доменов состоит в том, что домены ограничивают сравнения. Некорректно, с логической точки зрения, сравнивать значения из различных доменов, даже если они имеют одинаковый тип.

Понятие домена помогает правильно моделировать предметную область. Не все домены обладают логическим условием, ограничивающим возможные значения домена. В таком случае множество возможных значений домена совпадает с множеством возможных значений типа данных.

Атрибут отношения – это пара вида *<имя_атрибута, имя_домена>*. Имена атрибутов должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов.

Схема отношения – это именованное множество упорядоченных пар *<имя_атрибута, имя_домена>*. *Степенью* или «арностью» схемы отношения является мощность этого множества. *Схема базы данных* в реляционной модели – это множество именованных схем отношений. Понятие схемы отношения близко к понятию структурного типа в языках программирования (например, **record** в языке Pascal или **struct** в языке C).

Кортеж, соответствующий данной схеме отношения, – это множество упорядоченных пар *<имя_атрибута, значение_атрибута>*, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. Значение атрибута должно быть допустимым значением домена, на котором определен данный атрибут. *Степень* или «арность» кортежа совпадает с «арностью» соответствующей схемы отношения.

Отношение, определенное на множестве из *n* доменов (не обязательно различных), содержит две части: заголовок (схему отношения) и тело (множество из *m* кортежей). Значения *n* и *m* называются соответственно *степенью* и *кардинальностью отношения*. Отношения обладают следующими свойствами.

- В отношении нет одинаковых кортежей. Действительно, тело отношения есть множество кортежей и, как всякое множество, не может содержать неразличимые элементы.
- Кортежи не упорядочены (сверху вниз). Причина следующая – тело отношения есть множество, а множество не упорядочено.
- Атрибуты не упорядочены (слева направо). Т.к. каждый атрибут имеет уникальное имя в пределах отношения, то порядок атрибутов не имеет значения. В таблицах в отличие от отношений столбцы упорядочены.

- Каждый кортеж содержит ровно одно значение для каждого атрибута. Отношение, удовлетворяющее этому свойству, называется нормализованным или представленным в первой нормальной форме (1NF).
- Все значения атрибутов атомарны, т. е. не обладают структурой. Трактовка этого свойства в последнее время претерпела существенные изменения. Исторически в большинстве публикаций по базам данных считалось недопустимым использовать атрибуты со структуризованными значениями. (А в большинстве изданий это считается таковым и поныне.) В настоящее время принимается следующая точка зрения: тип данных атрибута может быть произвольным, а, следовательно, возможно существование отношения с атрибутами, значениями которых также являются отношения. Именно так в некоторых постреляционных базах данных реализована работа со сколь угодно сложными типами данных, создаваемых пользователями.

В реляционной модели каждый кортеж любого отношения должен отличаться от любого другого кортежа этого отношения (т.е. любое отношение должно обладать уникальным ключом).

Непустое подмножество множества атрибутов схемы отношения будет *потенциальным ключом* тогда и только тогда, когда оно будет обладать свойствами *уникальности* (в отношении нет двух различных кортежей с одинаковыми значениями потенциального ключа) и *неизбыточности* (никакое из собственных подмножеств множества потенциального ключа не обладает свойством уникальности).

В реляционной модели по традиции один из потенциальных ключей должен быть выбран в качестве *первичного ключа*, а все остальные потенциальные ключи будут называться *альтернативными*.

Реляционная база данных – это набор отношений, имена которых совпадают с именами схем отношений в схеме базы данных.

Целостностная часть реляционной модели

В целостностной части реляционной модели фиксируются два базовых требования целостности, которые должны выполняться для любых отношений в любых реляционных базах данных. Это *целостность сущностей* и *ссылочная целостность* (или *целостность внешних ключей*).

Простой объект реального мира представляется в реляционной модели как кортеж некоторого отношения. Требование целостности сущностей заключается в следующем: каждый кортеж любого отношения должен отличаться от любого другого кортежа этого отношения (т.е. любое отношение должно обладать потенциальным ключом). Вполне очевидно, что если данное требование не соблюдается (т.е. кортежи в рамках одного отношения не уникальны), то в базе данных может храниться противоречивая информация об одном и том же объекте. Поддержание целостности сущностей обеспечивается средствами СУБД. Это осуществляется с помощью двух ограничений:

- 1) при добавлении записей в таблицу проверяется уникальность их первичных ключей,
- 2) не позволяет изменение значений атрибутов, входящих в первичный ключ.

Сложные объекты реального мира представляются в реляционной модели данных в виде кортежей нескольких нормализованных отношений, связанных между собой. При этом

- 1) связи между данными отношениями описываются в терминах *функциональных зависимостей*,
- 2) для отражения функциональных зависимостей между кортежами разных отношений используется дублирование первичного ключа одного отношения (родительского) в другое (дочернее). Атрибуты, представляющие собой копии ключей родительских отношений, называются *внешними ключами*.

Внешний ключ в отношении R2 – это непустое подмножество атрибутов FK этого отношения, такое, что: а) существует отношение R1 (причем отношения R1 и R2 не обязательно различны) с потенциальным ключом CK; б) каждое значение внешнего ключа FK в текущем значении отношения R2 обязательно совпадает со значением ключа CK некоторого кортежа в текущем значении отношения R1.

Требование ссылочной целостности состоит в следующем:

- для каждого значения внешнего ключа, появляющегося в дочернем отношении, в родительском отношении должен найтись кортеж с таким же значением первичного ключа.

Как правило, поддержание ссылочной целостности также возлагается на СУБД. Например, она может не позволить пользователю добавить запись, содержащую внешний ключ с несуществующим (неопределенным) значением.

Манипуляционная часть реляционной модели

Манипуляционная часть реляционной модели описывает два эквивалентных способа манипулирования реляционными данными – *реляционную алгебру* и *реляционное исчисление*. Принципиальное различие между реляционной алгеброй и

реляционным исчислением заключается в следующем: реляционная алгебра в явном виде предоставляет набор операций, а реляционное исчисление представляет систему обозначений для определения требуемого отношения в терминах данных отношений. Формулировка запроса в терминах реляционной алгебры носит предписывающий характер, а в терминах реляционного исчисления – описательный характер. Говоря неформально, реляционная алгебра носит процедурный характер (пусть на очень высоком уровне), а реляционное исчисление – непроцедурный характер.

Реляционная алгебра

Реляционная алгебра является основным компонентом реляционной модели, опубликованной Коддом, и состоит из восьми операторов, составляющих две группы по четыре оператора:

- 1) Традиционные операции над множествами: *объединение* (UNION), *пересечение* (INTERSECT), *разность* (MINUS) и *декартово произведение* (TIMES). Все операции модифицированы, с учетом того, что их operandами являются отношения, а не произвольные множества.
- 2) Специальные реляционные операции: *ограничение* (WHERE) , *проекция* (PROJECT), *соединение* (JOIN) и *деление* (DIVIDE BY).

Результат выполнения любой операции реляционной алгебры над отношениями также является отношением. Эта особенность называется свойством *реляционной замкнутости*. Утверждается, что поскольку реляционная алгебра является замкнутой, то в реляционных выражениях можно использовать вложенные выражения сколь угодно сложной структуры. Если рассматривать свойство реляционной замкнутости строго, то каждая реляционная операция должна быть определена таким образом, чтобы выдавать результат с надлежащим типом отношения (в частности, с соответствующим набором атрибутов или заголовком). Для достижения этой цели вводится новый оператор *переименование* (RENAME), предназначенный для переименования атрибутов в определенном отношении.

В качестве основы для последующих обсуждений рассмотрим упрощенный синтаксис выражений реляционной алгебры в форме БНФ.

реляционное_выражение ::=

унарное_выражение | бинарное_выражение

унарное_выражение ::=

переименование | ограничение | проекция

переименование ::=

терм RENAME имя_атрибута AS имя_атрибута

терм ::=

имя_отношения | (реляционное_выражение)

ограничение ::=

терм WHERE логическое_выражение

проекция ::=

терм | терм [список_имен_атрибутов]

бинарное_выражение ::=

проекция бинарная_операция реляционное_выражение

бинарная_операция ::=

UNION | INTERSECT | MINUS | TIMES | JOIN | DIVIDE BY

По приведенной грамматике можно сделать следующие замечания.

- 1) Реляционные операторы UNION, INTERSECT и MINUS требуют, чтобы отношения были совместимыми по типу, т. е. имели идентичные заголовки.
- 2) Реляционные операторы UNION, INTERSECT, TIMES и JOIN ассоциативны и коммутативны.
- 3) Если отношения A и B не имеют общих атрибутов, то операция соединения A JOIN B эквивалентна операции A TIMES B, т. е. в таком случае соединение вырождается в декартово произведение. Такое соединение называют естественным.
- 4) Другой допустимый синтаксис для синтаксической категории переименования таков: (терм RENAME список_переименований). Здесь каждый из элементов списка переименований представляет собой выражение имя_атрибута AS имя_атрибута.

- 5) Несмотря на большие возможности, предоставляемые операторами реляционной алгебры, существует несколько типов запросов, которые нельзя выразить этими средствами. Для таких случаев необходимо использовать процедурные расширения реляционных языков.

В алгебре Кодда не все операторы являются независимыми, т. е. некоторые из реляционных операторов могут быть выражены через другие реляционные операторы.

Оператор естественного соединения по атрибуту **Y** определяется через оператор декартового произведения и оператор ограничения:

$$\mathbf{A \text{ JOIN } B = ((A \text{ TIMES } (B \text{ RENAME } Y \text{ AS } Y1)) \text{ WHERE } Y=Y1)[X, Y, Z]}$$

Оператор пересечения выражается через вычитание следующим образом:

$$\mathbf{A \text{ INTERSECT } B = A \text{ MINUS } (A \text{ MINUS } B)}$$

Оператор деления выражается через операторы вычитания, декартового произведения и проекции следующим образом:

$$\mathbf{A \text{ DIVIDE BY } B = A[X] \text{ MINUS } ((A[X] \text{ TIMES } B) \text{ MINUS } A)[X]}$$

Оставшиеся реляционные операторы (объединение, вычитание, декартово произведение, ограничение, проекция) являются примитивными операторами – их нельзя выразить друг через друга.

В качестве примера рассмотрим запросы на языке реляционной алгебры для схемы базы данных «Поставщики и детали», представленной следующими схемами отношений:

S(Sno: integer, Sname: string, Status: integer, City: string)
P(Pno: integer, Pname: string, Color: string, Weight: real, City: string)
SP(Sno: integer, Pno: integer, Qty: integer)

В данном примере имена доменов представлены именами типов, имена типов отделяются от имен атрибутов двоеточием, первичные ключи выделены подчеркиванием, а имена внешних ключей схемы отношения **SP** (ПОСТАВКА) совпадают с именами первичных ключей схем отношений **S** (ПОСТАВЩИК) и **P** (ДЕТАЛЬ).

- 1) Получить имена поставщиков, которые поставляют деталь под номером 2.

$$((\mathbf{SP \text{ JOIN } S}) \text{ WHERE } Pno = 2) [Sname]$$

- 2) Получить имена поставщиков, которые поставляют по крайней мере одну красную деталь.

$$(((\mathbf{P \text{ WHERE } Color = 'Красный'}) \text{ JOIN } SP) [Sno] \text{ JOIN } S) [Sname]$$

Другая формулировка того же запроса:

$$(((\mathbf{P \text{ WHERE } Color = 'Красный'}) [Pno] \text{ JOIN } SP) \text{ JOIN } S) [Sname]$$

Этот пример подчеркивает одно важное обстоятельство: возможность сформулировать один и тот же запрос несколькими способами.

- 3) Получить имена поставщиков, которые поставляют все детали.

$$((\mathbf{SP} [Sno, Pno] \text{ DIVIDE BY } P [Pno] \text{ JOIN } S) [Sname]$$

- 4) Получить номера поставщиков, поставляющих по крайней мере все те детали, которые поставляет поставщик под номером 2.

$$\mathbf{SP} [Sno, Pno] \text{ DIVIDE BY } (\mathbf{SP \text{ WHERE } Sno = 2}) [Pno]$$

- 5) Получить все пары номеров поставщиков, размещенных в одном городе

$$(((\mathbf{S \text{ RENAME } Sno \text{ AS } FirstSno}) [FirstSno, City] \text{ JOIN } \\ (\mathbf{S \text{ RENAME } Sno \text{ AS } SecondSno}) [SecondSno, City])$$

WHEPE FirstSno < SecondSno) [FirstSno, SecondSno]

6) Получить имена поставщиков, которые не поставляют деталь под номером 2.

((S[Sno] MINUS (SP WHEPE Pno = 2) [Sno]) JOIN S) [Sname]

Вычислительные возможности реляционной алгебры можно увеличить путем введения дополнительных операторов.

Дополнительные операторы реляционной алгебры

Многие авторы предлагали новые алгебраические операторы после определения Коддом первоначальных восьми. Рассмотрим несколько таких операторов:

SEMIJOIN (полусоединение),
SEMIMINUS (половычитание),
EXTEND (расширение),
SUMMARIZE (обобщение) и
TCLOSE (транзитивное замыкание).
TRANSFORM
ROLLUP
CUBE

Синтаксис этих операторов выглядит следующим образом.

```
<полусоединение> ::= <реляционное выражение> SEMIJOIN <реляционное выражение>
< полувычитание > ::= <реляционное выражение> SEMIMINUS <реляционное выражение>
< расширение> ::= EXTEND <реляционное выражение> ADD
<список добавляемых расширений>
<добавляемое расширение> ::= <выражение> AS <имя атрибута>
<обобщение> ::= SUMMARIZE <реляционное выражение> PER <реляционное выражение>
ADD <список добавляемых обобщений>
<добавляемое обобщение> ::= <тип обобщения> [ ( <скалярное выражение> ) ] AS <имя атрибута>
<тип обобщения> ::= COUNT | SUM | AVG | MAX | MIN | ALL | ANY | COUNTD | SUMD | AVGD
<транзитивное замыкание> ::= TCLOSE <реляционное выражение>
```

Операция расширения

С помощью **операции расширения** из определенного отношения (по крайней мере, концептуально) создается новое отношение, которое содержит дополнительный атрибут, значения которого получены посредством некоторых скалярных вычислений. Примеры.

```
EXTEND S ADD 'Supplier' AS TAG
EXTEND P ADD (Weight * 454 ) AS GMWT
( EXTEND P ADD (Weight * 454 ) AS GMWT ) WHERE GMWT > Weight ( 10000.0 ) { ALL BUT GMWT }
EXTEND ( P JOIN SP ) ADD (Weight * Qty ) AS SHIPWT
( EXTEND S ADD City AS SCity ) { ALL BUT City }
EXTEND P ADD Weight * 454 AS GMW, Weight * 16 AS OZWT )
EXTEND S ADD COUNT ( ( SP RENAME SNo AS X ) WHERE X = SNo ) AS NP
```

Рассмотрим вкратце обобщающие функции. Общее назначение этих функций состоит в том, чтобы на основе значений некоторого атрибута определенного отношения получить скалярное значение. В языке Tutorial D параметр <вызов обобщающей функции> является особым случаем параметра <скаларное выражение> и в общем случае имеет следующий вид.

<имя функции> (<реляционное выражение> [, <имя атрибута>])

Если параметр <имя функции> имеет значение COUNT, то параметр <имя атрибута> недопустим и должен быть опущен. В остальных случаях параметр <имя атрибута> может быть опущен тогда и только тогда, когда параметр <реляционное выражение> задает отношение со степенью, равной единице.

Операция обобщения

В реляционной алгебре операция расширения позволяет выполнять "горизонтальные" вычисления в отношении отдельных строк. **Оператор обобщения** выполняет аналогичную функцию для "вертикальных" вычислений в отношении отдельного столбца. Примеры.

SUMMARIZE SP PER SP { PNo } ADD SUM (Qty) AS TOTQTY

В результате его вычисления создается отношение с заголовком {P#, TOTQT Y}, содержащее один кортеж для каждого значения атрибута P# в проекции SP{P#}. Каждый из этих кортежей содержит значение атрибута P# и соответствующее общее количество деталей. Другими словами, концептуально исходное отношение P «перегруппировано» в множество групп кортежей (по одной группе для каждого уникального значения атрибута P#), после чего для каждой полученной группы сгенерирован один кортеж, помещаемый в окончательный результат.

В общем случае значение выражения

SUMMARIZE A PER B ADD <обобщение> AS

Определяется следующим образом

1. Отношение B должно иметь такой же тип, как и некоторая проекция отношения A, Т.е. каждый атрибут отношения B должен одновременно присутствовать в отношении A. Примем, что атрибутами этой проекции (или, что эквивалентно, атрибутами отношения B) являются атрибуты A1, A2, ..., An.
2. Результатом вычисления данного выражения будет отношение с заголовком {A1, A2, ..., An, Z}, где Z является новым добавленным атрибутом.
3. Тело результата содержит все кортежи t, где t является кортежем отношения B, расширенным значением нового атрибута Z. Это значение нового атрибута Z подсчитывается посредством вычисления *обобщающего выражения* по всем кортежам отношения A, которое имеет те же значения для атрибутов A1, A2, ..., An, что и кортеж t. (Разумеется, если в отношении A нет кортежей, принимающих те же значения для атрибутов A1, A2, ..., An, что и кортеж t, то *обобщающее выражение* будет вычислено для пустого множества.) Отношение B не должно содержать атрибут с именем Z, а *обобщающее выражение* не должно ссылаться на атрибут Z. Заметьте, что кардинальность результата равна кардинальности отношения B, а степень результата равна степени отношения B плюс единица. Типом переменной Z в этом случае будет тип *обобщающего выражения*.

Вот еще один пример.

SUMMARIZE (P JOIN SP) PER P { City } ADD COUNT AS NSP

Легко заметить, что оператор SUMMARIZE не примитивен – его можно моделировать с помощью оператора EXTEND. Рассмотрим следующее выражение

SUMMARIZE SP PER S { SNo } ADD COUNT AS NP

По сути, это сокращенная запись представленного ниже более сложного выражения

```
{ EXTEND S { SNo } ADD (( SP RENAME SNo AS X ) WHERE X=SNo ) AS Y, COUNT ( Y ) AS NP } { SNo, NP }
```

Группирование и разгруппирование

Поскольку значениями атрибутов отношений могут быть другие отношения, было бы желательным наличие дополнительных реляционных операторов, называемых операторами группирования и разгруппирования. Рассмотрим пример

SP GROUP (PNo, Qty) AS PQ

который можно прочесть как «сгруппировать отношение SP по атрибуту SNo», поскольку атрибут SNo является единственным атрибутом отношения SP, не упомянутым в предложении GROUP. В результате получится отношение, заголовок которого выглядит так

```
{ SNo SNo, PQ RELATION { PNo PNo, Qty Qty } }
```

Другими словами, он состоит из атрибута PQ, принимающего в качестве значений отношения (PQ, в свою очередь, имеет атрибуты PNo и Qty), а также из всех остальных атрибутов отношения SP (в нашем случае "все остальные атрибуты" - это атрибут SNo). Тело этого отношения содержит ровно по одному кортежу для всех различных значений атрибута SNo исходного отношения SP.

Перейдем теперь к операции разгруппирования. Пусть SPQ - это отношение, полученное в результате группирования. Тогда выражение

SPQ UNGROUP PQ

Возвращает нас к отношению SP (как и следовало ожидать). Точнее, оно выдает в качестве результата отношение, заголовок которого выглядит так

(SNo SNo, PNo PNo, Qty Qty)

Реляционные сравнения

Реляционная алгебра в том виде, в котором она была изначально определена, не поддерживает прямого сравнения двух отношений (например, проверки их равенства или того, является ли одно из них подмножеством другого). Это упущение легко исправляется следующим образом. Сначала определяется новый вид условия - реляционное сравнение - со следующим синтаксисом.

```
<реляционное выражение> <отношение> <реляционное выражение>
<отношение> ::= 
    > -- Собственное супермножество
    | >= -- Супермножество
    | < -- Собственное подмножество
    | <= -- Подмножество
    | = -- Равно
    | <> -- Не равно
```

Здесь параметр <реляционное выражение> в обоих случаях выражения реляционной алгебры, представляющие совместимые по типу отношения. Примеры.

S (City) = P (City)

Смысл выражения: совпадает ли проекция отношения поставщиков S по атрибуту City с проекцией отношения деталей P по атрибуту City?

S (SNo) > SPJ (SNo)

Смысл выражения: есть ли поставщики, вообще не поставляющие деталей?

На практике часто требуется определить, является ли данное отношение пустым. Соответствующий оператор, возвращающий логическое значение имеет вид

IS_EMPTY (<реляционное выражение>)

Не менее часто требуется проверить, присутствует ли данный кортеж t в данном отношении R. Для этой цели подойдет следующее реляционное сравнение.

RELATION { t } <= R

Однако, с точки зрения пользователя, удобнее применять следующее сокращение

t IN R

Реляционное исчисление

Реляционное исчисление основано на разделе математической логики, который называется исчислением предикатов. Реляционное исчисление существует в двух формах: *исчисление кортежей* и *исчисление доменов*. Основное различие

между ними состоит в том, что переменные исчисления кортежей являются переменными кортежей (они изменяются на отношении, а их значения являются кортежами), в то время как переменные исчисления доменов являются переменными доменов (они изменяются на доменах, а их значения являются скалярами).

Исчисление кортежей

Реляционное исчисление является альтернативой реляционной алгебре. Внешне два подхода очень отличаются – исчисление описательное, а алгебра предписывающая, но на более низком уровне они представляют собой одно и то же, поскольку любые выражения исчисления могут быть преобразованы в семантически эквивалентные выражения в алгебре и наоборот.

Исчисление существует в двух формах: исчисление кортежей и исчисление доменов. Основное различие между ними состоит в том, что переменные исчисления кортежей являются переменными кортежей (они изменяются на отношении, а их значения являются кортежами), в то время как переменные исчисления доменов являются переменными доменов (они изменяются на доменах, а их значения являются скалярами; в этом смысле, действительно, "переменная домена" - не очень точный термин).

Выражение исчисления кортежей содержит заключенный в скобки список целевых элементов и выражение WHERE, содержащее формулу WFF ("правильно построенную формулу"). Такая формула WFF составляется из кванторов (EXISTS и FORALL), свободных и связанных переменных, литералов, операторов сравнения, логических (булевых) операторов и скобок. Каждая свободная переменная, которая встречается в формуле WFF, должна быть также перечислена в списке целевых элементов.

Синтаксис исчисления кортежей

Упрощенный синтаксис выражений исчисления кортежей в форме БНФ имеет вид.

объявление-кортежной-переменной ::=

RANGE OF *переменная IS* *список-областей*

область ::=

отношение |
реляционное-выражение

реляционное-выражение ::=

(*список-целевых-элементов*) [**WHERE** *wff*]

целевой-элемент ::=

переменная |
переменная.атрибут [**AS** *атрибут*]

wff ::=

условие |
NOT *wff* /
условие AND wff /
условие OR wff /
IF *условие THEN wff* /
EXISTS *переменная (wff)* /
FORALL *переменная (wff)* /
(*wff*)

условие ::=

(*wff*) /
компанд операция-отношения компанд

По приведенной грамматике можно сделать следующие замечания.

- 1) Квадратные скобки здесь указывают на компоненты, которые по умолчанию могут быть опущены.
- 2) Категории *отношение*, *атрибут* и *переменная* – это идентификаторы (т. е. имена).
- 3) Реляционное выражение содержит заключенный в скобки список целевых элементов и выражение WHERE, содержащее формулу wff («правильно построенную формулу»). Такая формула wff составляется из кванторов (EXISTS и FORALL), свободных и связанных переменных, констант, операторов сравнения, логических (булевых)

операторов и скобок. Каждая свободная переменная, которая встречается в формуле *wff*, должна быть также перечислена в списке целевых элементов.

- 4) Категория **условие** представляет или формулу *wff*, заключенную в скобки, или простое скалярное сравнение, где каждый **компанд** оператора сравнения – это либо скалярная константа, либо значение атрибута в форме *переменная.атрибут*.

Пусть кортежная переменная *T* определяются следующим образом:

RANGE OF T IS R1, R2, ..., Rn

Тогда отношения *R1, R2, ..., Rn* должны быть совместимы по типу т. е. они должны иметь идентичные заголовки, и кортежная переменная *T* изменяется на объединении этих отношений, т. е. её значение в любое заданное время будет некоторым текущим кортежем, по крайней мере одного из этих отношений.

Примеры объявлений кортежных переменных.

RANGE OF SX IS S

RANGE OF SPX IS SP

RANGE OF SY IS

(SX) **WHERE** SX.City = 'Смоленск'

(SX) **WHERE EXISTS** SPX (SPX.Sno = SX.Sno **AND** SPX.Pno = 1)

Здесь переменная кортежа *SY* может принимать значения из множества кортежей *S* для поставщиков, которые или размещены в Смоленске, или поставляют деталь под номером 1, или и то и другое.

Для сравнения с реляционной алгеброй рассмотрим некоторые запросы на языке исчисления кортежей, которые соответствуют рассмотренным ранее.

Для сравнения с реляционной алгеброй некоторые примеры соответствуют рассмотренным ранее. Любые примеры можно расширить, включив в них завершающий шаг *присвоения*, присвоив значение выражения некоторому именованному отношению; этот шаг для краткости опускается.

- 1) Получить номера поставщиков из Смоленска со статусом больше 20

(SX.Sno) **WHERE** SX.City = 'Смоленск' **AND** SX.Status > 20

- 2) Получить все такие пары номеров поставщиков, что два поставщика размещаются в одном городе

(SX.Sno **AS** FirstSno, SY.Sno **AS** SecondSno) **WHERE** SX.City = SY.City **AND** SX.Sno < SY.Sno

Замечание. Спецификации «**AS FirstSno**» и «**AS SecondSno**» дают имена атрибутам *результата*; следовательно, такие имена недоступны для использования во фразе **WHERE** и потому второе сравнение во фразе **WHERE** «**SX.Sno < SY.Sno**», а не «**FirstSno < SecondSno**».

- 3) Получить имена поставщиков, которые поставляют деталь с номером 2

SX.Sname **WHERE EXISTS** SPX (SPX.Sno = SX.Sno **AND** SPX.Pno = 2)

- 4) Получить имена поставщиков, которые поставляют по крайней мере одну красную деталь

SX.Sname **WHERE EXISTS** SPX (SX.Sno = SPX.Sno **AND EXISTS** PX (PX.Pno = SPX.Pno **AND** PX.Color = 'Красный'))

Или эквивалентная формула (но в предваренной нормальной форме, в которой все кванторы записываются в начале формулы WFF):

SX.Sname **WHERE EXISTS** SPX (**EXISTS** PX (SX.Sno = SPX.Sno **AND** SPX.Pno = PX.Pno **AND** PX.Color = 'Красный'))

Предваренная нормальная форма не является более или менее правильной по сравнению с другими формами, но немного попрактиковавшись можно убедиться, что в большинстве случаев это наиболее естественная формулировка запросов. Кроме того, эта форма позволяет уменьшить количество скобок, как показано ниже.

Формулу WFF

квантор_1 переменная_1 (квантор_2 переменная_2 (wff))

(где каждый из кванторов *квантор_1* и *квантор_2* представляет или квантор **EXISTS**, или квантор **FORALL**, *переменная_1* и *переменная_2* - имена переменных) по умолчанию можно однозначно сократить к виду

квантор_1 переменная_1 квантор_2 переменная_2 (wff)

Таким образом, приводимое выше выражение исчисления можно переписать (при желании) следующим образом:

SX.Sname **WHERE EXISTS SPX EXISTS PX** (SX.Sno = SPX.Sno **AND** SPX.Pno = PX.Pno **AND** PX.Color = 'Красный')

Однако для ясности мы будем продолжать показывать все скобки во всех остальных примерах.

- 5) Получить имена поставщиков, которые поставляют по крайней мере одну деталь, поставляемую поставщиком под номером 2

SX.Sname **WHERE EXISTS SPX (EXISTS SPY (SX.Sno = SPX.Sno **AND** SPX.Pno = SPX.Pno **AND** SPY.sno = 2))**

- 6) Получить имена поставщиков, которые поставляют все детали

SX.SNAME **WHERE FORALL PX (EXISTS SPX (SPX.Sno = SX.Sno **AND** SPX.Pno = PX.Pno))**

Или равносильное выражение без использования квантора **FORALL**:

SX. SNAME **WHERE NOT EXISTS PX (NOT EXISTS SPX (SPX.Sno = SX.Sno **AND** SPX.Pno = PX.Pno))**

Отметим, что содержание этого запроса – «имена поставщиков, которые поставляют все детали» – является точным на 100% (в отличие от случая с алгебраическим аналогом при использовании оператора **DIVIDE BY**).

- 7) Получить имена поставщиков, которые не поставляют деталь под номером 2

SX.Sname **WHERE NOT EXISTS SPX (SPX.Sno = SX.Sno **AND** SPX.Pno = 2)**

Обратите внимание, как просто это решение можно получить из примера 3.

- 8) Получить номера поставщиков, которые поставляют по крайней мере все детали, поставляемые поставщиком с номером 2

SX.Sno **WHERE FORALL SPY (SPY.Sno <> 2 **OR** EXISTS SPZ (SPZ.Sno = SX.Sno **AND** SPZ.Pno = SPY.Pno))**

Переформулируем запрос в соответствии с приведенным в выражении: «Получить номера поставщиков, скажем **SX**, таких, что для всех поставок **SPY** поставка осуществляется либо не от поставщика с номером **2**, либо если от поставщика с номером **2**, то существует поставка **SPZ** детали **SPY** от поставщика **SX**».

Введем другое синтаксическое соглашение, чтобы облегчить формулирование таких сложных запросов, как этот, а именно: явную синтаксическую форму для оператора логической импликации.

Если f и g - формулы WFF, то выражение логической импликации

IF f THEN g

также будет формулой WFF с семантикой, идентичной семантике формулы

(NOT f) OR g

Таким образом, выражение, приведенное выше, может быть переписано (при желании) так:

SX.Sno WHERE FORALL SPY (IF SPY.Sno = 2 THEN EXISTS SPZ (SPZ.Sno = SX.Sno AND SPZ.Pno = SPY.Pno))

Дадим словесную формулировку: "Получить номера поставщиков, скажем **SX**, таких, что для всех поставок **SPY** в случае поставки от поставщика с номером **2** существует поставка **SPZ** детали **SPY** от поставщика **SX**".

- 9) Получить номера деталей, которые или весят более 16 ед., или поставляются поставщиком с номером 2, или и то и другое

**RANGE OF PU IS PX.Pno WHERE PX.Weight > 16, SPX.Pno WHERE SPX.Sno = 2;
PU.Pno**

В реляционном алгебраическом эквиваленте здесь могло бы использоваться явное объединение. Альтернативная формулировка этого запроса имеет вид.

PX.Pno WHERE PX.Weight > 16 OR EXISTS SPX (SPX.Pno = PX.Pno AND SPX.Sno = 2)

Однако тот факт, что каждый номер детали в отношении **SP** присутствует и в отношении **P**, делает эту вторую формулировку не в «стиле объединения».

Вычислительные возможности исчисления кортежей

Добавить вычислительные возможности в исчисление довольно просто: необходимо расширить определение компарандов и целевых элементов так, чтобы они включали новую категорию - скалярные выражения, в которых операнды, в свою очередь, могут включать литералы, ссылки на атрибуты и (или) ссылки на итоговые функции. Для целевых элементов также требуется использовать спецификацию вида "**AS attribute**" для того, чтобы дать подходящее имя результирующему атрибуту, если нет очевидного наследуемого имени.

Т. к. что смысл скалярного выражения легко воспринимается, подробности опускаются. Однако синтаксис для ссылок на итоговые функции будет показан:

aggregate_function (expression [, attribute])

где

aggregate_function - это **COUNT**, **SUM**, **AVG**, **MAX** или **MIN** (возможны, конечно, и некоторые другие функции),

expression - это выражение исчисления кортежей (вычисляющее отношение),

attribute - это такой атрибут результирующего отношения, по которому подсчитывается итог.

Для функции **COUNT** аргумент *attribute* не нужен и опускается; для других итоговых функций его можно опустить по умолчанию, если и только если вычисление аргумента *expression* дает отношение степени один и в таком случае единственный атрибут результата вычисления выражения *expression* подразумевается по умолчанию. Обратите внимание, что ссылка на итоговую функцию возвращает скалярное значение и поэтому допустима в качестве операнда скалярного выражения.

Из сказанного можно сделать следующие выводы:

1. Итоговая функция действует в некоторых отношениях как новый тип квантора. В частности, если аргумент *expression* в данной ссылке на итоговую функцию представляется в виде "(*tic*) **WHERE** *f*", где *tic*- целевой элемент списка (*target_item_comma_list*), а *f* - это формула WFF, и если экземпляр переменной кортежа *T* свободен в формуле *f*, то такой экземпляр переменной *T* связан в ссылке на итоговую функцию

*aggregate_function ((tic) WHERE *f* [, attribute])*

2. Пользователи, владеющие языком SQL, могут заметить, что с помощью двух следующих аргументов в ссылке на итоговую функцию, *expression* и *attribute*, можно избежать необходимых для SQL специальных приемов

использования оператора **DISTINCT** для исключения дублирующих кортежей, если это требуется, перед выполнением итоговой операции. Вычисление аргумента *expression* дает отношение, из которого повторяющиеся кортежи всегда исключаются по определению. Аргумент *attribute* обозначает атрибут такого отношения, по которому выполняются итоговые вычисления, а дублирующие значения перед подсчетом итога из такого атрибута не удаляются. Конечно, атрибут может не содержать никаких дублирующих значений в любом случае, в частности, если такой атрибут является первичным ключом.

Примеры

Чтобы облегчить сравнение с реляционной алгеброй, некоторые примеры соответствуют рассмотренным ранее.

- 1) Получить номера деталей и их вес всех типов деталей, вес которых превышает 10 ед.

(PX.Pno, PX.Weight AS GMWT) **WHERE** PX.Weight > 10

Обратите внимание, что спецификация **AS GMWT** дает имя соответствующему атрибуту результата. Поэтому такое имя недоступно для использования во фразе **WHERE** и потому выражение **PX.Weight** записывается в двух местах.

- 2) Получить всех поставщиков, добавив для каждого литеральное значение "Поставщик"

(SX, 'Поставщик' AS TAG)

- 3) Получить каждую поставку с полными данными о входящих в нее деталях и общим весом поставки

(SPX.Sno, SPX.Qty, PX, PX.Weight * SPX.Qty AS SHIPWT) **WHERE** PX.Pno = SPX.Pno

- 4) Для каждой детали получить ее номер и общее поставляемое количество

(PX.Pno, SUM (SPX **WHERE** SPX.Pno = PX.Pno, Qty) AS TOTQTY)

Заметьте, что вместо фразы вида "**BY (Pno)**" ссылка на итоговую функцию включает в качестве' первого аргумента выражение исчисления; вычисление этого выражения (для данной детали) дает точное множество кортежей, по которому подсчитывается итог. Также стоит отметить, что результат будет включать один кортеж для каждой детали, даже для деталей, которые в данное время не поставляются никакими поставщиками (количество для каждой такой детали будет равно нулю).

- 5) Получить общее количество поставляемых деталей

(SUM (SPX, Qty) AS GRANDTOTAL)

в отличие от своего эквивалента - оператора **SUMMARIZE**, это выражение возвращает корректный результат (а именно нуль), если отношение **SP** пустое.

- 6) Для каждого поставщика получить его номер и общее количество поставляемых им деталей

(SX.Sno, COUNT (SPX **WHERE** SPX.Sno = SX.Sno) AS NUMBER_OF_PARTS)

В отличие от своего эквивалента - оператора **SUMMARIZE**, это выражение дает результат, который включает кортеж для поставщика с номером 5 (с нулевым количеством).

- 7) Получить города, в которых хранится а) более пяти красных деталей; б) не более пяти красных деталей

PX.City **WHERE** COUNT (PY **WHERE** PY.City = PX.City **AND** PY.Color = 'Красный') > 5

Это выражение представляет собой решение задачи для части (а). В отличие от своего эквивалента - оператора **SUMMARIZE**, это решение может быть преобразовано в решение для части (б) простой заменой знака ">" на "<=".

Исчисление доменов

Реляционное исчисление, ориентированное на домены (или исчисление доменов), отличается от исчисления кортежей тем, что в нем используются переменные доменов вместо переменных кортежей, т. е. переменные, принимающие свои значения в пределах домена, а не отношения. (Замечание. "Переменную домена" было бы лучше называть "скалярной переменной", так как ее значения - это элементы домена, т.е. скаляры, а не сами домены.)

С практической точки зрения большинство очевидных различий между версиями исчисления для доменов и кортежей основано на том, что версия для доменов поддерживает дополнительную форму условия, которое мы будем называть **условием принадлежности**. В общем виде условие принадлежности можно записать так:

R (pair, pair, ...),

где **R** - это имя отношения, а каждая пара **pair** имеет вид **A : v** (где **A** - атрибут отношения **R**, а **v** - или переменная домена, или литерал). Проверка условия дает значение истина, если и только если существует кортеж в отношении **R**, имеющий определенные значения для определенных атрибутов. Например, вычисление выражения

SP (Sno : 1, Pno : 1)

дает значение истина, если и только если в отношении **SP** существует кортеж со значением **Sno**, равным **1**, и значением **Pno**, равным **1**. Аналогично, условие принадлежности

SP (Sno : SX, Pno : PX)

принимает значение истина, если и только если в отношении **SP** существует кортеж со значением **Sno**, эквивалентным текущему значению переменной домена **SX** (какому бы то ни было), и значением **Pno**, эквивалентным текущему значению переменной домена **PX** (опять же какому бы то ни было).

Далее будем подразумевать существование переменных доменов с именами: образуемыми добавлением букв **X, Y, Z, ...** к соответствующим именам доменов. Напомним, что в базе данных поставщиков и деталей каждый атрибут имеет такое же имя, как и соответствующий ему домен, за исключением атрибутов **Sname** и **Pname**, для которых соответствующий домен называется просто **Name**.

Примеры выражений исчисления доменов:

1. **(SX)**
2. **(SX) WHERE S (Sno : SX)**
3. **(SX) WHERE S (Sno : SX, City : 'Смоленск')**
4. **(SX, CityX) WHERE S (Sno : SX, City : CityX) AND SP (Sno : SX, Pno : 2)**
5. **(SX, PX) WHERE S (Sno : SX, City : CityX) AND P (Pno : PX, City : CityY) AND CityX <> CityY**

Семантика выражений запросов:

1. Множество всех номеров поставщиков.
2. Множество всех номеров поставщиков отношения **S**.
3. Подмножество номеров поставщиков из города **Смоленск**.
4. Получить номера и города поставщиков, поставляющих деталь под номером 2. (Обратите внимание, что для этого запроса, выраженного в терминах исчисления кортежей, требовался квантор существования; заметьте также, что это первый из приведенных здесь примеров, где действительно необходимы скобки для списка целевых элементов).
5. Получить такие пары 'номер поставщика' – 'номер детали', что поставщики и детали размещены в одном городе".

Примеры

Для сравнения с реляционной алгеброй некоторые примеры соответствуют рассмотренным ранее.

- 1) Получить номера поставщиков из Смоленска со статусом, большим 20

SX WHERE EXISTS StatusX (StatusX > 20 AND S (Sno : SX, Status : StatusX, City : 'Смоленск'))

Обратите внимание, что кванторы все еще требуются. Этот пример несколько неуклюжий по сравнению с его аналогом, выраженным в терминах исчисления кортежей. С другой стороны, конечно, есть случаи, когда верно обратное; смотрите, в частности, более сложные примеры, приведенные ниже.

2) Получить все такие пары номеров поставщиков, что два поставщика размещаются в одном городе

(SX AS FirstSno, SY AS SecondSno) **WHERE EXIST** CityZ (S (Sno : SX, City : CityZ) **AND** S (Sno : SY, City : CityZ) **AND** SX < SY)

3) Получить имена поставщиков, которые поставляют по крайней мере одну красную деталь

NameX **WHERE EXISTS** SX **EXISTS** PX (S (Sno : SX, Sname : NameX) **AND** SP (Sno : SX, Pno : PX) **AND** P (Pno : PX, Color : 'Красный'))

4) Получить имена поставщиков, которые поставляют по крайней мере одну деталь, поставляемую поставщиком с номером 2

NameX **WHERE EXISTS** SX **EXISTS** PX (S (Sno : SX, Sname : NameX) **AND** SP (Sno : SX, Pno : PX) **AND** SP (Sno : 2, Pno : PX))

5) Получить имена поставщиков, которые поставляют все типы деталей

NameX **WHERE EXISTS** SX (S (Sno : SX, Sname : NameX) **AND** **FORALL** PX (**IF** P (Pno : PX) **THEN** SP (Sno : SX, Pno : PX)))

6) Получить имена поставщиков, которые не поставляют деталь с номером 2

NameX **WHERE EXISTS** SX (S (Sno : SX, Sname : NameX) **AND** **NOT** SP (Sno : SX, Pno : 2))

7) Получить номера поставщиков, которые поставляют по крайней мере все типы деталей, поставляемых поставщиком с номером 2

SX **WHERE FORALL** PX (**IF** SP (Sno : 2, Pno : PX) **THEN** SP (Sno : SX, Pno : PX))

8) Получить номера деталей, которые или весят более 16 фунтов, или поставляются поставщиком с номером 2, или и то, и другое

PX **WHERE EXISTS** WeightX (P (Pno : PX, Weight : WeightX) **AND** WeightX > 16) **OR** SP (Sno : 2, Pno : PX)

Исчисление доменов, как и исчисление кортежей, формально эквивалентно реляционной алгебре (т.е. оно реляционно полно). Для доказательства можно сослаться, например, на работы Ульмана (Ullman).

Реляционное исчисление и реляционная алгебра

Ранее утверждалось, что реляционная алгебра и реляционное исчисление в своей основе эквивалентны. Обсудим это утверждение более подробно. Вначале Кодд показал в своей статье, что алгебра, по крайней мере, мощнее исчисления. (Термин "исчисление" будет использоваться для обозначения исчисления кортежей.) Он сделал это, придумав алгоритм, называемый "алгоритмом редукции Кодда", с помощью которого любое выражение исчисления можно преобразовать в семантически эквивалентное выражение алгебры. Мы не станем приводить здесь полностью этот алгоритм, а ограничимся примером, иллюстрирующим в общих чертах, как этот алгоритм функционирует.

В качестве основы для нашего примера используется база данных поставщиков, деталей и проектов. Для удобства приводится набор примерных значений для этой базы данных.

Таблица Поставщики (S)

Sno	Sname	Status	City
1	Алмаз	20	Смоленск
2	Циклон	10	Владимир
3	Дельта	30	Владимир
4	Орион	20	Смоленск
5	Аргон	30	Ярославль

Таблица Детали (P)

Pno	Pname	Color	Weight	City
1	Гайка	Красный	12	Смоленск
2	Болт	Зеленый	17	Владимир
3	Винт	Синий	17	Рязань
4	Винт	Красный	14	Смоленск
5	Шайба	Синий	12	Владимир
6	Шпунт	Красный	19	Смоленск

Таблица Проекты (J)

Jno	Jname	City
1	Ангара	Владимир
2	Алтай	Рязань
3	Енисей	Ярославль
4	Амур	Ярославль
5	Памир	Смоленск
6	Чегет	Тверь
7	Эльбрус	Смоленск

Таблица Поставки (SPJ)

Sno	Pno	Jno	Qty
1	1	1	200
1	1	4	700
2	3	1	400

2	3	2	200
2	3	3	200
2	3	4	500
2	3	5	600
2	3	6	400
2	3	7	800
2	5	2	100
3	3	1	200
3	4	2	500
4	6	3	300
4	6	7	300
5	2	2	200
5	2	4	100
5	5	5	500
5	5	7	100
5	6	2	200
5	1	4	100
5	3	4	200
5	4	4	800
5	5	4	400
5	6	4	500

База данных поставщиков, деталей и проектов (значения для примера)

Рассмотрим запрос: "Получить имена и города поставщиков, обеспечивающих по крайней мере один проект в городе Ярославль с поставкой по крайней мере 50 штук каждой детали". Выражение исчисления для этого запроса следующее:

(SX.Name, SX.City) **WHERE EXISTS JX FORALL PX EXISTS SPJX (JX.City = 'Ярославль' AND JX.Jno = SPJX.Jno AND PX.Pno = SPJX.Pno AND SX.Sno = SPJX.Sno AND SPJX.Qty >= 50)**

Здесь **SX, PX, JX** и **SPJX** - переменные кортежей, берущие свои значения из отношений **S, P, J** и **SPJ** соответственно. Теперь покажем, как вычислить это выражение, чтобы добиться необходимого результата.

Шаг 1. Для каждой переменной кортежа выбираем ее область значений (т.е. набор всех значений для этой переменной) если это возможно. Выражение «выбираем, если возможно» подразумевает, что существует условие выборки, встроенное в фразу WHERE, которую можно использовать, чтобы сразу исключить из рассмотрения некоторые кортежи. В нашем случае выбираются следующие наборы кортежей:

SX : Все кортежи отношения **S** 5 кортежей

PX : Все кортежи отношения **P** 6 кортежей

JX : Кортежи отношения **J**, в которых **City** = 'Ярославль' 2 кортежа

SPJX : Кортежи отношения **SPJ**, в которых **Qty** ≥ 50 24 кортежа

Шаг 2. Строим декартово произведение диапазонов, выбранных на первом шаге. Получим ...

Для экономии места таблица не приводится. Полное произведение содержит $5 \cdot 6 \cdot 2 \cdot 24 = 1440$ кортежей.

Шаг 3. Осуществляем выборку из произведения, построенного на втором шаге в соответствии с частью «условие соединения» фразы **WHERE**. В нашем примере эта часть следующая:

JX.Jno = SPJX.Jno AND PX.Pno = SPJX.Pno AND SX.Sno = SPJX.Sno AND

Поэтому из произведения исключаются кортежи, для которых значение **Sno** поставщика не равно значению **Sno** поставки, значение **Pno** детали не равно значению **Pno** поставки, значение **Jno** проекта не равно значению **Jno** поставки, после чего получаем подмножество декартова произведения, состоящее только из 10 кортежей.

Шаг 4. Применяем кванторы справа налево следующим образом:

- Для квантора «**EXISTS RX**» (где **RX** - переменная кортежа, принимающая значение на некотором отношении **R**) проецируем текущий промежуточный результат, чтобы исключить все атрибуты отношения **R**.
- Для квантора «**FORALL RX**» делим текущий промежуточный результат на отношение «выбранной области значений», соответствующее **RX**, которое было получено выше. При выполнении этой операции также будут исключены все атрибуты отношения **R**.

В нашем примере имеем следующие кванторы:

EXISTS JX FORALL PX EXISTS SPJX

Выполняем соответствующие операции.

1. **EXISTS SPJX.** Проецируем, исключая атрибуты отношения **SPJ** (**SPJ.Sno**, **SPJ.Pno**, **SPJ.Jno** и **SPJ.Qty**). В результате получаем:

Sno	Sname	Status	City	Pno	Pname	Color	Weight	City	Jno	Jname	City
1	Алмаз	20	Смоленск	1	Гайка	Красный	12	Смоленск	4	Амур	Ярославль
2	Циклон	10	Владимир	3	Винт	Синий	17	Рязань	3	Енисей	Ярославль
2	Циклон	10	Владимир	3	Винт	Синий	17	Рязань	4	Амур	Ярославль
4	Орион	20	Смоленск	6	Шпунт	Красный	19	Смоленск	3	Енисей	Ярославль
5	Аргон	30	Ярославль	2	Болт	Зеленый	17	Владимир			Ярославль
5	Аргон	30	Ярославль	1	Гайка	Красный	12	Смоленск	4	Амур	Ярославль
5	Аргон	30	Ярославль	3	Винт	Синий	17	Рязань	4	Амур	Ярославль
5	Аргон	30	Ярославль	4	Винт	Красный	14	Смоленск	4	Амур	Ярославль
5	Аргон	30	Ярославль	5	Шайба	Синий	12	Владимир	4	Амур	Ярославль
5	Аргон	30	Ярославль	6	Шпунт	Красный	19	Смоленск	4	Амур	Ярославль

2. **FORALL PX.** Делим на отношение **P**. В результате получаем:

Sno	Sname	Status	City	Jno	Jname	City
5	Аргон	30	Ярославль	4	Амур	Ярославль

(Теперь у нас есть место, чтобы показать отношение полностью, без сокращений.)

3. **EXISTS JX.** Проецируем, исключая атрибуты отношения **J** (**J.Jno**, **J.name** и **J.City**). В результате получаем:

Sno	Sname	Status	City
5	Аргон	30	Ярославль

Шаг 5. Проецируем результат шага 4 в соответствии со спецификациями в целевом списке элементов. В нашем примере целевым элементом списка будет: **SX.Sname**, **SX.City**. Следовательно, конечный результат таков:

Sname	City
Аргон	Ярославль

Из сказанного выше следует, что начальное выражение исчисления семантически эквивалентно определенному вложенному алгебраическому выражению, а если быть более точным, то проекции от проекции деления проекции выборки из произведения четырех выборок (!). Этим завершаем пример. Конечно, можно намного улучшить алгоритм, хотя многие подробности скрыты в наших пояснениях; но вместе с тем, необходим адекватный пример, предлагающий общую идею.

Теперь можно объяснить одну из причин (и не только одну) определения Коддом ровно восьми алгебраических операторов. Эти восемь операторов обеспечивают соглашение целевого языка, как средства возможной реализации исчисления. Другими словами, для данного языка, такого как QUEL, который основывается на исчислении, одно из возможных применений заключается в том, чтобы можно было брать запрос в том виде, в каком он предоставляется пользователем (являющийся, по существу, просто выражением исчисления), и применять к нему алгоритм получения эквивалентного алгебраического выражения. Это алгебраическое выражение, конечно, содержит множество алгебраических операций, которые согласно определению по своей природе выполнимы. (Следующий шаг состоит в продолжении оптимизации этого алгебраического выражения.)

Также необходимо отметить, что восемь алгебраических операторов Кодда являются мерой выразительной силы любого языка баз данных (существующего или предлагаемого). Обсудим этот вопрос подробнее.

Во-первых, язык называется реляционно полным, если он по своим возможностям, по крайней мере, не уступает реляционному исчислению, т.е. любое отношение, которое можно определить с помощью реляционного исчисления, также можно определить и с помощью некоторого выражения рассматриваемого языка. «Реляционно полный» - значит, не уступающий по возможностям алгебре, а не исчислению, но это то же самое. По сути, из существования алгоритма преобразования Кодда немедленно следует, что реляционная алгебра обладает реляционной полнотой.)

Реляционную полноту можно рассматривать как основную меру возможностей выборки и выразительной силы языков баз данных вообще. В частности, так как исчисление и алгебра - реляционно полные, они могут служить базисом для проектирования языков, не уступающих им по выразительности, без необходимости пересортировки для использования циклов - особенно важное замечание, если язык предназначается конечным пользователям, хотя оно также применимо, если язык предназначается прикладным программистам.

Далее, поскольку алгебра реляционно полная, то, чтобы доказать, что некоторый язык L обладает реляционной полнотой, достаточно показать, что в языке L есть аналоги всех восьми алгебраических операций (на самом деле достаточно показать, что в нем есть аналоги пяти примитивных операций) и что operandами любой операции языка L могут быть любые выражения этого языка. Язык SQL - это пример языка, реляционную замкнутость которого можно показать таким способом. Язык QUEL – еще один такой пример. В действительности на практике часто проще показать, что в данном языке есть эквиваленты алгебраических операций, чем найти в нем эквиваленты выражений исчисления. Именно поэтому реляционная полнота обычно определяется в терминах алгебраических выражений, а не выражений исчисления.

Семантическое моделирование данных

Реляционная модель данных достаточна для моделирования предметных областей. Однако проявляется ограниченность реляционной модели данных в следующих аспектах:

- Модель не предоставляет достаточных средств для представления смысла данных.
- Для многих приложений трудно моделировать предметную область на основе плоских таблиц.
- Хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не предоставляет каких-либо средств для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области («сущностей») и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо аппарата для разделения сущностей и связей.

Указанные ограничения вызвали к жизни направление семантических (концептуальных, инфологических) моделей данных. Любая развитая семантическая модель данных, как и реляционная модель, включает структурную, манипуляционную и целостную части. Главным назначением семантических моделей является обеспечение возможности выражения семантики данных. На практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем

- a) Либо вручную преобразуется к реляционной (или какой-либо другой) схеме.
- b) Либо реализуется автоматизированная компиляция концептуальной схемы в реляционную.
- c) Либо происходит работа с базой данных в семантической модели, т.е. под управлением СУБД, основанных на семантических моделях данных. (Третья возможность еще не вышла за пределы исследовательских и экспериментальных проектов.)

Наиболее известным представителем класса семантических моделей предметной области является модель «сущность-связь» или *ER-модель*, предложенная Питером Ченом в 1976 году. Модель сущность-связь — модель данных, позволяющая описывать концептуальные схемы предметной области. Предметная область — часть реального мира, рассматриваемая в пределах данного контекста. Под контекстом здесь может пониматься, например, область исследования или область, которая является объектом некоторой деятельности. ER-модель используется при высокуюровневом (концептуальном) проектировании баз данных. С её помощью можно выделить ключевые сущности и обозначить связи, которые могут устанавливаться между этими сущностями. Во время проектирования баз данных происходит преобразование ER-модели в конкретную схему базы данных на основе выбранной модели данных (реляционной, объектной, сетевой или др.). ER-модель представляет собой формальную конструкцию, которая сама по себе не предписывает никаких графических средств её визуализации. В качестве стандартной графической нотации, с помощью которой можно визуализировать ERM, была предложена диаграмма сущность-связь (entity-relationship diagram, ERD). На практике понятия *ER-модель* и *ER-диаграмма* часто не различают, хотя для визуализации ER-моделей предложены и другие графические нотации. Основными понятиями ER-модели являются *сущность*, *связь* и *атрибут* (*свойство*).

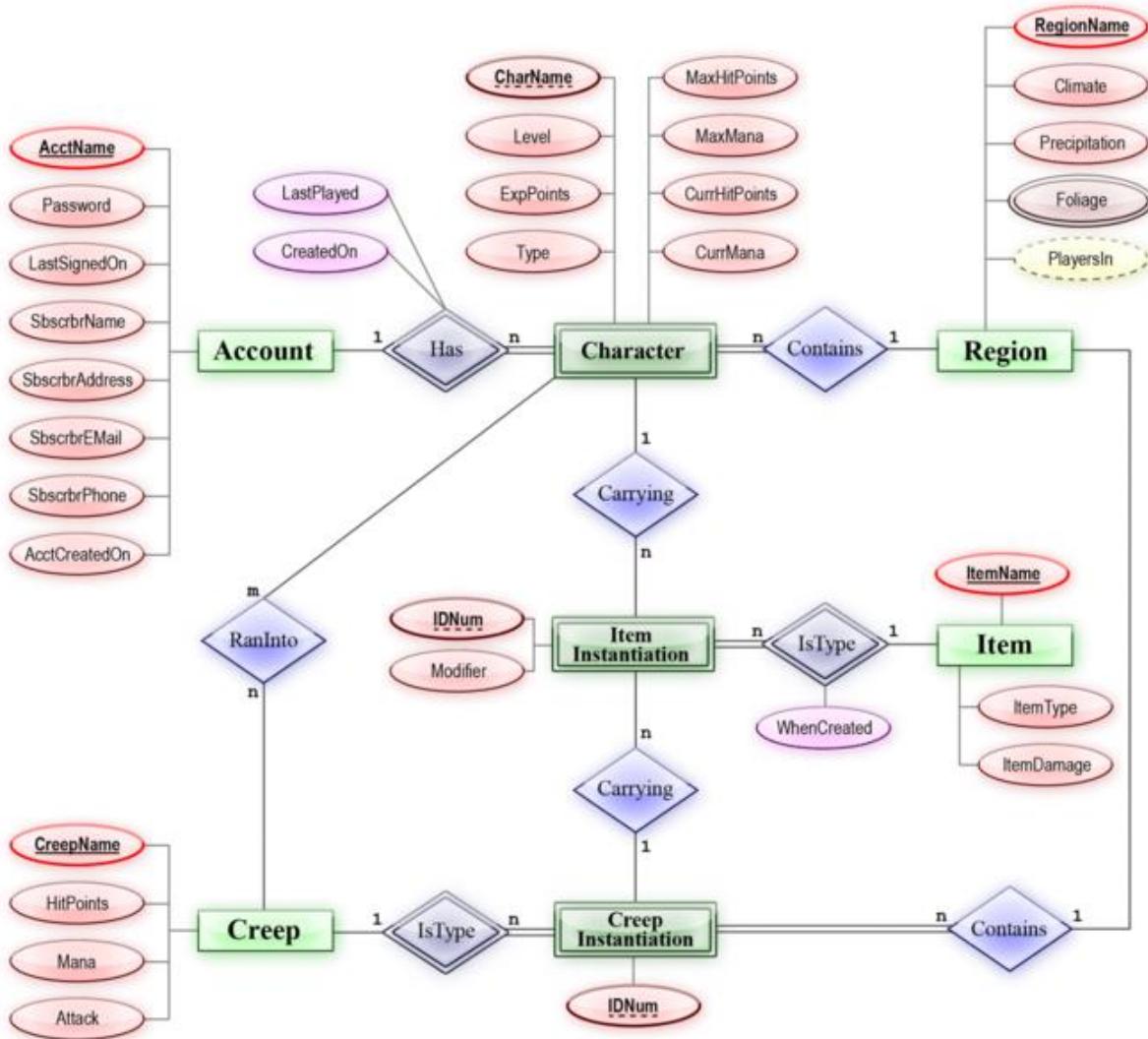
Сущность - это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа. Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных объектов этого типа. Каждый экземпляр сущности должен быть отличим от любого другого экземпляра той же сущности (это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах). Сущности подразделяются на *сильные* и *слабые*. Сильные сущности существуют сами по себе, а существование слабых сущностей зависит от существования сильных.

Связь - это ассоциация, устанавливаемая между сущностями. Эта ассоциация может существовать между разными сущностями или между сущностью и ей же самой (рекурсивная связь). Сущности, включенные в связь, называются ее *участниками*, а количество участников связи называется ее *степенью*. Участие сущности в связи может быть как *полным*, так и *частичным*. Связи в ER-модели могут иметь тип «один к одному», «один ко многим», «многие ко многим». Именно тип связи «многие ко многим» является единственным типом, представляющим истинную связь, поскольку это единственным тип связи, который требует для своего представления отдельного отношения. Связи типа «один к одному» и «один ко многим» всегда могут быть представлены с помощью механизма внешнего ключа, помещаемого в одно из отношений.

Свойством сущности (и связи) является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности (или связи). Значения свойств каждого типа извлекаются из соответствующего множества значений, которое в реляционной терминологии называется доменом. Свойства могут быть *простыми* или *составными*, *ключевыми*, *однозначными* или *многозначными*, *опущенными* (т. е. «неизвестными») или *непредставленными*), *базовыми* или *производными*.

Более сложными элементами ER-модели являются подтипы и супертипы сущностей. Как в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), вводится возможность наследования типа сущности, исходя из одного или нескольких супертипов.

На ER-диаграммах множества сущностей изображаются в виде прямоугольников, множества отношений изображаются в виде ромбов. Слабый тип сущности изображают в виде прямоугольника с двойным контуром. Слабый тип связи изображают в виде ромба с двойным контуром. Если сущность участвует в отношении, они связаны линией. Тип связи с частичным участием изображают двойной линией. Вид типа связи обозначается над линиями в виде соответствующих надписей возле типов сущностей. Например, если это вид бинарной связи «один ко многим», то делаются надписи 1, n (или m), соответственно, возле соответствующих типов сущностей. Атрибуты изображаются в виде овалов и связываются линией с одним отношением или с одной сущностью. Именование сущности обычно выражается уникальным существительным, именование связи обычно выражается глаголом, именование атрибута обычно выражается существительным. Неизбыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности, являются ключом сущности.



Существует множество инструментов для работы с ER-моделями, вот некоторые из них: Microsoft Visio, ERwin, Oracle Designer, PowerDesigner, Rational Rose. В справочниках приводятся сведения о 25 таких инструментах.

Получение реляционной схемы из ER-схемы осуществляется с помощью следующей пошаговой процедуры.

Шаг 1. Каждая простая сущность превращается в таблицу. Простая сущность - сущность, не являющаяся подтипов и не имеющая подтипов. Имя сущности становится именем таблицы.

Шаг 2. Каждый свойство (атрибут) становится возможным столбцом с тем же именем; может выбираться более точный формат. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, - не могут.

Шаг 3. Компоненты уникального идентификатора сущности превращаются в первичный ключ таблицы. Если имеется несколько возможных уникальных идентификаторов, выбирается наиболее используемый. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно). Для именования этих столбцов используются имена концов связей и/или имена сущностей.

Шаг 4. Связи «многие к одному» (и «один к одному») становятся внешними ключами. Т.е. делается копия уникального идентификатора с конца связи «один», и соответствующие столбцы составляют внешний ключ. Необязательные связи соответствуют столбцам, допускающим неопределенные значения; обязательные связи - столбцам, не допускающим неопределенные значения.

Шаг 5. Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.

Шаг 6. Если в концептуальной схеме присутствовали подтипы, то возможны два способа:

- a) все подтипы в одной таблице,
- b) для каждого подтипа - отдельная таблица.

Теория проектирования реляционных баз данных

Введение

При проектировании базы данных решаются две основные проблемы:

- Каким образом отобразить объекты предметной области в абстрактные объекты модели данных, чтобы это отображение не противоречило семантике предметной области, и было, по возможности, лучшим (эффективным, удобным и т. д.)? Часто эту проблему называют проблемой логического проектирования баз данных.
- Как обеспечить эффективность выполнения запросов к базе данных? Этую проблему обычно называют проблемой физического проектирования баз данных.

В случае реляционных баз данных нет общих рецептов по части физического проектирования. Здесь слишком много зависит от используемой СУБД. Поэтому ограничимся только существенными вопросами логического проектирования реляционных баз данных. Более того, не будем касаться определения ограничений целостности общего вида, а ограничимся ограничениями первичного и внешнего ключей. Будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том, из каких отношений должна состоять базы данных, и какие атрибуты должны быть у этих отношений.

Классический подход к проектированию реляционных баз данных заключается в том, что сначала предметная область представляется в виде одного или нескольких отношений, а далее осуществляется процесс *нормализации* схемы отношений, причем каждая следующая нормальная форма обладает свойствами лучшими, чем предыдущая. Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойствам ей набору ограничений. Примером набора ограничений является ограничение первой нормальной формы – значения всех атрибутов отношения атомарны. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1НФ или 1NF);
- вторая нормальная форма (2НФ или 2NF);
- третья нормальная форма (3НФ или 3NF);
- нормальная форма Бойса-Кодда (НФБК или BCNF);
- четвертая нормальная форма (4НФ или 4NF);
- пятая нормальная форма, или нормальная форма проекции-соединения (5НФ или 5NF или PJ/NF).

Основные свойства нормальных форм такие:

- каждая следующая нормальная форма в некотором смысле лучше предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных свойств сохраняются.

Процесс проектирования реляционной базы данных на основе метода нормализации преследует две основные цели:

- избежать избыточности хранения данных;
- устранить аномалии обновления отношений.

Эти цели являются актуальными для информационных систем оперативной обработки транзакций (On-Line Transaction Processing – OLTP), которым свойственны частые обновления базы данных, и потому аномалии обновления могут сильно вредить эффективности приложения. В информационных системах оперативной аналитической обработки (On-Line Analytical Processing – OLAP), в частности, в системах поддержки принятия решений, базы данных в основном используются для выборки данных. Поэтому аномалиями обновления можно пренебречь. Из этого не следует, что принципы нормализации непригодны при проектировании баз данных OLAP-приложений. Даже если схема такой базы данных должна быть денормализована по соображениям эффективности, то чтобы получить правильную денормализованную схему, нужно сначала понять, как выглядит нормализованная схема.

В основе метода нормализации лежит *декомпозиция* отношения, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы. Считаются правильными такие декомпозиции отношения, которые обратимы, т. е. имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*.

Функциональные зависимости

Для демонстрации основных идей данной темы, будет использоваться несколько измененная версия отношения поставок SP, содержащая атрибут City, представляющий город соответствующего поставщика. Это измененное отношение будет называться SCP.

Sno	City	Pno	Qty
1	Смоленск	1	100
1	Смоленск	2	100
2	Владимир	1	200
2	Владимир	2	200
3	Владимир	2	300
4	Смоленск	2	400
4	Смоленск	4	400
4	Смоленск	5	400

Следует четко различать:

- значение этого отношения в определенный момент времени;
- и набор всех возможных значений, которые данное отношение может принимать в различные моменты времени.

Определение 1. Пусть R - это отношение, а X и Y - произвольные подмножества множества атрибутов отношения R. Тогда Y **функционально зависито** от X, что в символическом виде записывается как $X \rightarrow Y \Leftrightarrow \forall$ значение множества X связано в точности с одним значением множества Y.

Примеры ФЗ, которым удовлетворяет отношение SCP в данном состоянии:

$$\begin{array}{lll} \{ Sno \} & \rightarrow & \{ City \} \\ \{ Sno, Pno \} & \rightarrow & \{ Qty \} \\ \{ Sno, Pno \} & \rightarrow & \{ City \} \\ \{ Sno, Pno \} & \rightarrow & \{ City, Qty \} \\ \{ Sno, Pno \} & \rightarrow & \{ Sno \} \\ \{ Sno, Pno \} & \rightarrow & \{ Sno, Pno, City, Qty \} \\ \{ Sno \} & \rightarrow & \{ Qty \} \\ \{ Qty \} & \rightarrow & \{ Sno \} \end{array}$$

Левая и правая стороны ФЗ будут называться **дeterminантой** и **зависимой частью** соответственно.

Определение 2. Пусть R является переменной-отношением, а X и Y - произвольными подмножествами множества атрибутов переменной-отношения R. Тогда $X \rightarrow Y \Leftrightarrow \forall$ **допустимого значения отношения R** \forall значение X связано в точности с одним значением Y.

Определение 2a. Пусть R(A1, A2, ..., An) - схема отношения. Функциональная зависимость, обозначаемая $X \rightarrow Y$ между двумя наборами атрибутов X и Y, которые являются подмножествами R определяет ограничение на возможность существования кортежа в некотором отношении г. Ограничение означает, что для любых двух кортежей t1 и t2 в г, для которых имеет место $t1[X] = t2[X]$, также имеет место $t1[Y] = t2[Y]$.

- Если ограничение на схеме отношения R утверждает, что не может быть более одного кортежа со значением атрибутов X в любом отношении экземпляре отношения г, то X является потенциальным ключом R. Это означает, что $X \rightarrow Y$ для любого подмножества атрибутов Y из R. Если X является потенциальным ключ R, то $X \rightarrow R$.
- Если $X \rightarrow Y$ в R, это не означает, что $Y \rightarrow X$ в R.

Функциональная зависимость является семантическим свойством, т. е. свойством значения атрибутов.

Примеры безотносительных ко времени ФЗ для переменной-отношения SCP:

$$\begin{array}{lll} \{ Sno, Pno \} & \rightarrow & \{ Qty \} \\ \{ Sno, Pno \} & \rightarrow & \{ City \} \\ \{ Sno, Pno \} & \rightarrow & \{ City, Qty \} \\ \{ Sno, Pno \} & \rightarrow & \{ Sno \} \\ \{ Sno, Pno \} & \rightarrow & \{ Sno, Pno, City, Qty \} \\ \{ Sno \} & \rightarrow & \{ City \} \end{array}$$

Следует обратить внимание на ФЗ, которые выполняются для отношения SCP, но не выполняются «всегда» для переменной-отношения SCP:

$$\begin{array}{l} \{ \text{Sno} \} \rightarrow \{ \text{Qty} \} \\ \{ \text{Qty} \} \rightarrow \{ \text{Sno} \} \end{array}$$

Если X является потенциальным ключом переменной-отношения R, то все атрибуты Y переменной-отношения R должны быть обязательно ФЗ от X (это следует из определения потенциального ключа). Если переменная-отношение R удовлетворяет ФЗ $X \rightarrow Y$ и X **не** является потенциальным ключом, то R будет характеризоваться некоторой избыточностью. Например, в случае отношения SCP сведения о том, что каждый данный поставщик находится в данном городе, будут повторяться много раз (это хорошо видно из таблицы).

Эта избыточность приводит к разным **аномалиям обновления**, получившим такое название по историческим причинам. Под этим понимаются определенные трудности, появляющиеся при выполнении операций обновления INSERT, DELETE и UPDATE. Рассмотрим избыточность, соответствующую ФЗ ($\text{Sno} \rightarrow \text{City}$). Ниже поясняются проблемы, которые возникнут при выполнении каждой из указанных операций обновления.

- Операция INSERT. Нельзя поместить в переменную-отношение SCP информацию о том, что некоторый поставщик находится в определенном городе, не указав сведения хотя бы об одной детали, поставляемой этим поставщиком.
- Операция DELETE. Если из переменной-отношения SCP удалить кортеж, который является единственным для некоторого поставщика, будет удалена не только информация о поставке поставщиком некоторой детали, но также информация о том, что этот поставщик находится в определенном городе. В действительности проблема заключается в том, что в переменной-отношении SCP содержится слишком много собранной в одном месте информации, поэтому при удалении некоторого кортежа теряется слишком много информации.
- Операция UPDATE. Название города для каждого поставщика повторяется в переменной-отношении SCP несколько раз, и эта избыточность приводит к возникновению проблем при обновлении.

Для решения всех этих проблем, как предлагалось выше, необходимо выполнить декомпозицию переменной-отношения SCP на две следующие переменные-отношения.

$$\begin{array}{l} S' \{ \text{Sno}, \text{City} \} \\ SP \{ \text{Sno}, \text{Pno}, \text{Qty} \} \end{array}$$

Даже если ограничиться рассмотрением ФЗ, которые выполняются «всегда», множество ФЗ, выполняющихся для всех допустимых значений данного отношения, может быть все еще очень большим. Поэтому встает задача сокращения множества ФЗ до компактных размеров. Важность этой задачи вытекает из того, что ФЗ являются ограничениями целостности. Поэтому при каждом обновлении данных в СУБД все они должны быть проверены. Следовательно, для заданного множества ФЗ S желательно найти такое множество T, которое (в идеальной ситуации) было бы гораздо меньшего размера, чем множество S, причем каждая ФЗ множества S могла бы быть заменена ФЗ множества T. Если бы такое множество T было найдено, то в СУБД достаточно было бы использовать ФЗ из множества T, а ФЗ из множества S подразумевались бы автоматически.

Очевидным способом сокращения размера множества ФЗ было бы исключение **тривиальных зависимостей**, т. е. таких, которые не могут не выполняться. Примером тривиальной ФЗ для отношения SCP может быть $\{ \text{Sno}, \text{Pno} \} \rightarrow \{ \text{Sno} \}$.

Определение 3. ФЗ $(X \rightarrow Y)$ *тривиальная* $\Leftrightarrow Y \subseteq X$.

Одни ФЗ могут подразумевать другие ФЗ. Например, зависимость

$$\{ \text{Sno}, \text{Pno} \} \rightarrow \{ \text{City}, \text{Qty} \}$$

подразумевает следующие ФЗ

$$\begin{array}{l} \{ \text{Sno}, \text{Pno} \} \rightarrow \text{City} \\ \{ \text{Sno}, \text{Pno} \} \rightarrow \text{Qty} \end{array}$$

В качестве более сложного примера можно привести переменную-отношение R с атрибутами A, B и C, для которых выполняются ФЗ $(A \rightarrow B)$ и $(B \rightarrow C)$. В этом случае также выполняется ФЗ $(A \rightarrow C)$, которая называется **транзитивной** ФЗ.

Определение 4. Множество всех ФЗ, которые задаются данным множеством ФЗ S, называется **замыканием** S и обозначается символом S^+ .

Из сказанного выше становится ясно, что для выполнения сформулированной задачи следует найти способ вычисления S^+ на основе S.

Первая попытка решить эту проблему принадлежит Армстронгу (Armstrong), который предложил набор *правил вывода* новых ФЗ на основе заданных (эти правила также называются аксиомами Армстронга).

Пусть в перечисленных ниже правилах A, B и C – произвольные подмножества множества атрибутов заданной переменной-отношения R, а символическая запись AB означает { A, B }. Тогда правила вывода определяются следующим образом.

1. Правило *рефлексивности*: $(B \subseteq A) \Rightarrow (A \rightarrow B)$.
2. Правило *дополнения*: $(A \rightarrow B) \Rightarrow AC \rightarrow BC$.
3. Правило *транзитивности*: $(A \rightarrow B) \text{ и } (B \rightarrow C) \Rightarrow (A \rightarrow C)$.

Каждое из этих правил может быть непосредственно доказано на основе определения ФЗ. Более того, эти правила являются *полными* в том смысле, что для заданного множества ФЗ S минимальный набор ФЗ, которые подразумевают все зависимости из множества S, может быть выведен из S на основе этих правил. Они также являются *исчерпывающими*, поскольку никакие дополнительные ФЗ (т.е. ФЗ, которые не подразумеваются ФЗ множества S) с их помощью не могут быть выведены. Иначе говоря, эти правила могут быть использованы для получения замыкания S^+ .

Из трех описанных выше правил для упрощения задачи практического вычисления замыкания S^+ можно вывести несколько дополнительных правил. (Примем, что D – это другое произвольное подмножество множества атрибутов R.)

4. Правило *самоопределения*: $A \rightarrow A$.
5. Правило *декомпозиции*: $(A \rightarrow BC) \Rightarrow (A \rightarrow B) \text{ и } (A \rightarrow C)$.
6. Правило *объединения*: $(A \rightarrow B) \text{ и } (A \rightarrow C) \Rightarrow (A \rightarrow BC)$.
7. Правило *композиции*: $(A \rightarrow B) \text{ и } (C \rightarrow D) \Rightarrow (AC \rightarrow BD)$.

Кроме того, Дарвен (Darwen) доказал следующее правило, которое назвал *общей теоремой объединения*:

8. $(A \rightarrow B) \text{ и } (C \rightarrow D) \Rightarrow (A(C-B) \rightarrow BD)$.

Упражнение. Пусть дана некоторая переменная-отношение R с атрибутами A, B, C, D, E, F и следующими ФЗ:

A	->	BC
B	->	E
CD	->	EF

Показать, что для переменной-отношения R также выполняется ФЗ $(AD \rightarrow F)$, которая вследствие этого принадлежит замыканию заданного множества ФЗ.

1. $A \rightarrow BC$ (дано)
2. $A \rightarrow C$ (следует из п. 1 согласно правилу декомпозиции)
3. $AD \rightarrow CD$ (следует из п. 2 согласно правилу дополнения)
4. $CD \rightarrow EF$ (дано)
5. $AD \rightarrow EF$ (следует из п. 3 и 4 согласно правилу транзитивности)
6. $AD \rightarrow F$ (следует из п. 5 согласно правилу декомпозиции) // Конец упр.

Хотя эффективный алгоритм для вычисления замыкания S^+ на основе множества ФЗ S так и не сформулирован, можно описать алгоритм, который определяет, будет ли данная ФЗ находиться в данном замыкании. Для этого, прежде всего, следует дать определение понятию *суперключ*.

Определение 5. Суперключ переменной-отношения R – это множество атрибутов переменной-отношения R, которое содержит в виде подмножества (но не обязательно собственного подмножества), по крайней мере, один потенциальный ключ.

Из этого определения следует, что суперключи для данной переменной-отношения R – это такие подмножества K множества атрибутов переменной-отношения R, что ФЗ $(K \rightarrow A)$ истинна для каждого атрибута A переменной-отношения R.

Предположим, что известны некоторые ФЗ, выполняющиеся для данной переменной-отношения, и требуется определить потенциальные ключи этой переменной-отношения. По определению потенциальными ключами называются неприводимые суперключи. Таким образом, выясняя, является ли данное множество атрибутов K суперключом, можно в значительной степени продвинуться к выяснению вопроса, является ли K потенциальным ключом. Для этого нужно определить, будет ли набор атрибутов переменной-отношения R множеством всех атрибутов, функционально зависящих от K. Таким образом, для заданного множества зависимостей S, которые выполняются для переменной-отношения R, необходимо найти способ определения множества всех атрибутов переменной-отношения R, которые функционально

зависимы от K, т.е. так называемое замыкание K^+ множества K для S. Простой алгоритм вычисления этого замыкания имеет вид.

```

function Closure(K: SetOfAttr; S: SetOfFD): SetOfAttr;
var
    Jold, Jnew: SetOfAttr;
begin
    Jnew := K;
    repeat
        Jold := Jnew;
        foreach ((X -> Y) in S) do
            if (X ⊆ Jnew) then Jnew = Jnew ∪ Y;
        until (Jold = Jnew);
        return Jold;
end; // of Closure

```

Упражнение. Пусть дана переменная-отношение R с атрибутами A, B, C, D, E и F и следующими ФЗ:

A	->	BC
E	->	CF
B	->	E
CD	->	EF

Вычислить замыкание $\{A, B\}^+$ множества атрибутов $\{A, B\}$, исходя из заданного множества ФЗ.

1. Присвоим Jold и Jnew начальное значение - множество $\{A, B\}$.
2. Выполним внутренний цикл четыре раза - по одному разу для каждой заданной ФЗ. На первой итерации (для зависимости $A \rightarrow BC$) будет обнаружено, что левая часть действительно является подмножеством замыкания Jnew. Таким образом, к Jnew можно добавить атрибуты B и C. Jnew теперь представляет собой множество $\{A, B, C\}$.
3. На второй итерации (для зависимости $E \rightarrow CF$) обнаруживается, что левая часть не является подмножеством полученного до этого момента результата, который, таким образом, остается неизменным.
4. На третьей итерации (для зависимости $B \rightarrow E$) к Jnew будет добавлено множество E, которое теперь будет иметь вид $\{A, B, C, E\}$.
5. На четвертой итерации (для зависимости $CD \rightarrow EF$) Jnew останется неизменным.
6. Далее внутренний цикл выполняется еще четыре раза. На первой итерации результат останется прежним, на второй он будет расширен до $\{A, B, C, E, F\}$, а на третьей и четвертой - снова не изменится.
7. Наконец после еще одного четырехкратного прохождения цикла замыкание Jnew останется неизменным и весь процесс завершится с результатом $\{A, B\}^+ = \{A, B, C, E, F\}$. // Конец упр.

Выходы.

1. Для заданного множества ФЗ S легко можно указать, будет ли заданная ФЗ $(X \rightarrow Y)$ следовать из S, поскольку это возможно тогда и только тогда, когда множество Y является подмножеством замыкания X^+ множества X для заданного множества S. Таким образом, представлен простой способ определения, будет ли данная ФЗ $(X \rightarrow Y)$ включена в замыкание S^+ множества S.
2. Напомним определение понятия суперключа. Суперключ переменной-отношения R - это множество атрибутов переменной-отношения R, которое в виде подмножества (но необязательно собственного подмножества) содержит по крайней мере один потенциальный ключ. Из этого определения прямо следует, что суперключи для данной переменной-отношения R - это такие подмножества K множества атрибутов переменной-отношения R, что ФЗ $(K \rightarrow A)$ будет истинна для каждого атрибута A переменной-отношения R. Другими словами, множество K является суперключем тогда и только тогда, когда замыкание K^+ для множества K в пределах заданного множества ФЗ является множеством абсолютно всех атрибутов переменной-отношения R. (Кроме того, множество K является потенциальным ключом тогда и только тогда, когда оно является неприводимым суперключом).

Определение 6. Два множества ФЗ S1 и S2 **эквивалентны** тогда и только тогда, когда они **являются покрытиями** друг для друга, т. е. $S1^+ = S2^+$.

Пример

Consider the following two sets of functional dependencies:

$F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$ and $G = \{A \rightarrow CD, E \rightarrow AH\}$.

Check whether or not they are equivalent.

Answer:

To show equivalence, we prove that G is covered by F and F is covered by G.

Proof that G is covered by F:

$\{A\}+ = \{A, C, D\}$ (with respect to F), which covers A \rightarrow CD in G
 $\{E\}+ = \{E, A, D, H, C\}$ (with respect to F), which covers E \rightarrow AH in G

Proof that F is covered by G:

$\{A\}+ = \{A, C, D\}$ (with respect to G), which covers A \rightarrow C in F
 $\{A, C\}+ = \{A, C, D\}$ (with respect to G), which covers AC \rightarrow D in F
 $\{E\}+ = \{E, A, H, C, D\}$ (with respect to G), which covers E \rightarrow AD and E \rightarrow H in F

Конец примера]

Каждое множество ФЗ эквивалентно, по крайней мере, одному **неприводимому** множеству.

Определение 7. Множество ФЗ является **неприводимым** тогда и только тогда, когда оно обладает всеми перечисленными ниже свойствами.

- Каждая ФЗ этого множества имеет одноэлементную правую часть.
- Ни одна ФЗ множества не может быть устранина без изменения замыкания этого множества.
- Ни один атрибут не может быть устранен из левой части любой ФЗ данного множества без изменения замыкания множества.

Если I является неприводимым множеством, которое эквивалентно множеству S, то проверка выполнения ФЗ из множества I автоматически обеспечит выполнение ФЗ из множества S.

Пример. Рассмотрим переменную-отношение деталей P с функциональными зависимостями, перечисленными ниже.

Pno	->	Pname
Pno	->	Color
Pno	->	Weight
Pno	->	City

Нетрудно заметить, что это множество функциональных зависимостей является неприводимым:

- правая часть каждой зависимости содержит только один атрибут,
- левая часть, очевидно, является неприводимой,
- ни одна из функциональных зависимостей не может быть опущена без изменения замыкания множества (т. е. без утраты некоторой информации).

В противоположность этому приведенные ниже множества функциональных зависимостей не являются неприводимыми.

1. Pno \rightarrow { Pname, Color }
Pno \rightarrow Weight
Pno \rightarrow City

(Правая часть первой ФЗ не является одноэлементным множеством.)

2. { Pno, Pname } \rightarrow Color
Pno \rightarrow Pname
Pno \rightarrow Weight
Pno \rightarrow City

(Первую ФЗ можно упростить, опустив атрибут Pname в левой части без изменения замыкания, т. е. она не является неприводимой слева.)

3. Pno \rightarrow Pno
Pno \rightarrow Pname
Pno \rightarrow Color
Pno \rightarrow Weight
Pno \rightarrow City

(Здесь первую ФЗ можно опустить без изменения замыкания.) // Конец прим.

Можно сделать утверждение, что для любого множества ФЗ существует, по крайней мере, одно эквивалентное множество, которое является неприводимым. Это достаточно легко продемонстрировать на следующем примере. Пусть дано исходное множество ФЗ S. Тогда благодаря правилу декомпозиции можно без утраты общности предположить, что каждая ФЗ в этом множестве S имеет одноэлементную правую часть. Далее для каждой ФЗ f из этого множества S следует проверить каждый атрибут A в левой части зависимости f. Если множество S и множество зависимостей,

полученное в результате устраниния атрибута A в левой части зависимости f, эквивалентны, значит этот атрибут следует удалить. Затем для каждой оставшейся во множестве S зависимости f, если множества S и $S \setminus \{f\}$ эквивалентны, следует удалить зависимость f из множества S. Получившееся в результате таких действий множество S является неприводимым и эквивалентно исходному множеству S.

Упражнение. Пусть дана переменная-отношение R с атрибутами A, B, C, D и следующими функциональными зависимостями.

A	->	BC
B	->	C
A	->	B
AB	->	C
AC	->	D

Найти неприводимое множество функциональных зависимостей эквивалентное данному множеству.

1. Прежде всего, следует переписать заданные ФЗ таким образом, чтобы каждая из них имела одноэлементную правую часть.

A	->	B
A	->	C
B	->	C
A	->	B
AB	->	C
AC	->	D

Нетрудно заметить, что зависимость A -> B записана дважды, так что одну из них можно удалить.

2. Затем в левой части зависимости AC -> D может быть опущен атрибут C, поскольку дана зависимость A -> C, из которой по правилу дополнения можно получить зависимость A -> AC. Кроме того, дана зависимость AC -> D, из которой по правилу транзитивности можно получить зависимость A -> D. Таким образом, атрибут C в левой части исходной зависимости AC -> D является избыточным.
3. Далее можно заметить, что зависимость AB -> C может быть исключена, поскольку дана зависимость A -> C, из которой по правилу дополнения можно получить зависимость AB -> CB, а затем по правилу декомпозиции - зависимость AB -> C.
4. Наконец зависимость A -> C подразумевается зависимостями A -> B и B -> C, так что она также может быть отброшена. В результате получается неприводимое множество зависимостей.

A	->	B
B	->	C
A	->	D // Конец упр.

Множество ФЗ I, которое неприводимо и эквивалентно другому множеству ФЗ S, называется неприводимым покрытием множества S. Таким образом, с тем же успехом в системе вместо исходного множества ФЗ S может использоваться его неприводимое покрытие I (здесь следует повторить, что для вычисления неприводимого эквивалентного покрытия I необязательно вычислять замыкание S^+). Однако необходимо отметить, что для заданного множества ФЗ не всегда существует уникальное неприводимое покрытие.

Дополнения к ФЗ

Проектирование базы данных может быть выполнено с использованием двух подходов: снизу вверх (bottom-up) или сверху вниз (top-down).

- Методология проектирования bottom-up (также называемая методологией синтеза) рассматривает основные связи между отдельными атрибутами в качестве отправной точки и использует их, чтобы построить схемы отношений схем. Этот подход не пользуется популярностью на практике, потому что она страдает от проблемы того, чтобы собрать большое число бинарных связей между атрибутами в качестве отправной точки. На практике это сделать почти невозможно.
- Методология проектирования top-down (также называемая методологией анализа) начинается с некоторого набора отношений, состоящих из атрибутов. Затем отношения анализируются отдельно или совместно, в результате чего происходит их декомпозиция до тех пор, пока не будут достигнуты все желаемые свойства.

Неявными целями обеих методологий являются сохранение информации и минимизация избыточности.

Процесс нормализации схем отношений, основанный на операции декомпозиции, должен обладать следующими свойствами:

- неаддитивностью JOIN или JOIN без потерь информации (NJP), которая гарантирует, что в результате декомпозиции не появятся лишние кортежи.
- свойством сохранения зависимостей (DPP), которое гарантирует, что каждая функциональная зависимость будет представлена в каком-либо отдельном отношении после декомпозиции.

Свойство NJP является очень критическим и должно быть достигнуто любой ценой. Свойство DPP желательно, но не всегда достижимо.

Определение. Суперключ в схеме отношения $R(A_1, A_2, \dots, A_n)$ - это набор атрибутов S из R со свойством, что ни для каких двух кортежей t_1 и t_2 в любом отношении над схемой R не будет выполняться равенство $t_1[S] = t_2[S]$.

Определение. Потенциальный ключ K - это суперключ с дополнительным свойством: удаление любого атрибута из K приведет к тому, что K перестанет быть суперключом.

Определение. Атрибут в схеме отношения R называется первичным атрибутом R , если он является членом некоторого потенциального ключа R . Атрибут называется непервичным, если он не является первичным атрибутом, то есть, если он не является членом какого-либо потенциального ключа.

Алгоритм поиска минимального покрытия F для множества функциональных зависимостей E

Вход: Множество функциональных зависимостей E .

1. Пусть $F := E$.
2. Заменить каждую функциональную зависимость $X \rightarrow \{A_1, A_2, \dots, A_n\}$ из F на n функциональных зависимостей $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$.
3. Для каждой функциональной зависимости $X \rightarrow A$ из F

Для каждого атрибута B , который является элементом X
 если $\{F - \{X \rightarrow A\} \cup \{X - \{B\}\} \rightarrow A\}$ эквивалентно F
 заменить $X \rightarrow A$ на $(X - \{B\}) \rightarrow A$ в F .
4. Для каждой оставшейся функциональной зависимости $X \rightarrow A$ из F

если $\{F - \{X \rightarrow A\}\}$ эквивалентно F ,
 удалить $X \rightarrow A$ из F .

Пример.

Let the given set of FDs be $E : \{B > A, D > A, AB > D\}$. We have to find the minimal cover of E .

- All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm and can proceed to step 2. In step 2 we need to determine if $AB > D$ has any redundant attribute on the left-hand side; that is, can it be replaced by $B > D$ or $A > D$?
- Since $B > A$, by augmenting with B on both sides (IR2), we have $BB > AB$, or $B > AB$ (i). However, $AB > D$ as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), $B > D$. Thus $AB > D$ may be replaced by $B > D$.
- We now have a set equivalent to original E , say E' : $\{B > A, D > A, B > D\}$. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.
- In step 3 we look for a redundant FD in E' . By using the transitive rule on $B > D$ and $D > A$, we derive $B > A$. Hence $B > A$ is redundant in E' and can be eliminated.
- Therefore, the minimal cover of E is $\{B > D, D > A\}$.

Этот алгоритм может быть использован для синтеза отношения из заданного множества зависимостей E .

Алгоритм нахождения ключа K схемы отношения R для заданного множества функциональных зависимостей F

Вход: Схема отношения R и множество функциональных зависимостей F на атрибутах R .

1. Пусть $K := R$.
2. Для каждого атрибута из K

{
 вычислить $\text{Closure}(\{K - A\}, F)$;
 если замыкание содержит все атрибуты из R , то установите $K := K - \{A\}$
 };

Заметьте, что алгоритм определяет только один ключ из множества возможных ключей на R ; возвращаемый ключ зависит от порядка, в котором атрибуты удаляются из R на шаге 2.

[Конец дополнений к Ф3]

Нормальные формы, основанные на функциональных зависимостях

Определение 1. Переменная отношения находится в первой нормальной форме (1НФ) тогда и только тогда, когда в любом допустимом значении отношения каждый его кортеж содержит только одно значение для каждого из атрибутов.

Первая нормальная форма - это условие, согласно которому каждый компонент каждого кортежа является атомарным значением. В реляционной модели отношение всегда находится в первой нормальной форме по определению понятия отношения.

Определение 2. Функциональная зависимость $R.X \rightarrow R.Y$ называется полной, если атрибут Y не зависит функционально от любого точного подмножества X .

Определение 3. Функциональная зависимость $R.X \rightarrow R.Y$ называется транзитивной, если существует такой атрибут Z , что имеются функциональные зависимости $R.X \rightarrow R.Z$ и $R.Z \rightarrow R.Y$ и отсутствует функциональная зависимость $R.X \rightarrow R.Y$. (При отсутствии последнего требования мы имели бы "неинтересные" транзитивные зависимости в любом отношении, обладающем несколькими ключами.)

Определение 4. Неключевым атрибутом называется любой атрибут отношения, не входящий в состав потенциального ключа (в частности, первичного).

Определение 5. Два или более атрибута взаимно независимы, если ни один из этих атрибутов не является функционально зависимым от других.

Вторая нормальная форма

Определение 6. (В этом определении предполагается, что единственным ключом отношения является первичный ключ.) Отношение R находится во второй нормальной форме (2НФ) в том и только в том случае, когда оно находится в 1НФ, и каждый неключевой атрибут полностью зависит от первичного ключа.

Если допустить наличие нескольких ключей, то определение 6 примет следующий вид:

Определение 6а. Отношение R находится во второй нормальной форме (2NF) в том и только в том случае, когда оно находится в 1НФ, и каждый неключевой атрибут полностью зависит от каждого ключа R .

Здесь и далее мы не будем приводить примеры для отношений с несколькими ключами. Они слишком громоздки и относятся к ситуациям, редко встречающимся на практике.

Третья нормальная форма

Определение 7. (Снова определение дается в предположении существования единственного ключа.) Отношение R находится в третьей нормальной форме (3НФ) в том и только в том случае, если оно находится в 2НФ и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

Если отказаться от того ограничения, что отношение обладает единственным ключом, то определение 3NF примет следующую форму:

Определение 7а. Отношение R находится в третьей нормальной форме (3НФ) в том и только в том случае, если оно находится в 2НФ, и каждый неключевой атрибут не является транзитивно зависимым от какого-либо ключа R .

На практике третья нормальная форма схем отношений достаточна в большинстве случаев, и приведением к третьей нормальной форме процесс проектирования реляционной базы данных обычно заканчивается. Однако иногда полезно продолжить процесс нормализации.

[Дополнения к 2НФ и 3НФ]

Exercise 1. Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I\}$ and the set of functional dependencies $F = \{ \{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\} \}$. What is the key for R ? Decompose R into 2NF, then 3NF relations.

Answer:

A minimal set of attributes whose closure includes all the attributes in R is a key. Since the closure $\{A, B\}^+ = R$, one key of R is $\{A, B\}$ (in this case, it is the only key).

To normalize R intuitively into 2NF then 3NF, we take the following steps:

First, identify partial dependencies that violate 2NF. These are attributes that are functionally dependent on either parts of the key, $\{A\}$ or $\{B\}$, alone. We can calculate the closures $\{A\}^+$ and $\{B\}^+$ to determine partially dependent attributes:

o $\{A\}^+ = \{A, D, E, I, J\}$, hence $\{A\} \rightarrow \{D, E, I, J\}$ ($\{A\} \rightarrow \{A\}$ is a trivial dependency)

o $\{B\}^+ = \{B, F, G, H\}$, hence $\{B\} \rightarrow \{F, G, H\}$ ($\{B\} \rightarrow \{B\}$ is a trivial dependency)

To normalize into 2NF, we remove the attributes that are functionally dependent on part of the key (A or B) from R and place them in separate relations R1 and R2, along with the part of the key they depend on (A or B), which are copied into each of these relations but also remains in the original relation, which we call R3 below:

o R1 = {A, D, E, I, J},

o R2 = {B, F, G, H},

o R3 = {A, B, C}

The new keys for R1, R2, R3 are underlined.

Next, we look for transitive dependencies in R1, R2, R3. The relation R1 has the transitive dependency $\{A\} \rightarrow \{D\} \rightarrow \{I, J\}$, so we remove the transitively dependent attributes {I, J} from R1 into a relation R11 and copy the attribute D they are dependent on into R11. The remaining attributes are kept in a relation R12. Hence, R1 is decomposed into R11 and R12 as follows:

o R11 = {D, I, J},

o R12 = {A, D, E}

The relation R2 is similarly decomposed into R21 and R22 based on the transitive dependency $\{B\} \rightarrow \{F\} \rightarrow \{G, H\}$:

o R21 = {F, G, H},

o R22 = {B, F}

The final set of relations in 3NF are {R11, R12, R21, R22, R3}

Exercise 2. Repeat exercise 1 for the following different set of functional dependencies

$G = \{ \{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\} \}$.

Answer:

To help in solving this problem systematically, we can first find the closures of all single attributes to see if any is a key on its own as follows:

o $\{A\}^+ = \{A, I\}$,

o $\{B\}^+ = \{B\}$,

o $\{C\}^+ = \{C\}$,

o $\{D\}^+ = \{D\}$,

o $\{E\}^+ = \{E\}$,

o $\{F\}^+ = \{F\}$,

o $\{G\}^+ = \{G\}$,

o $\{H\}^+ = \{H, J\}$,

o $\{I\}^+ = \{I\}$,

o $\{J\}^+ = \{J\}$

Since none of the single attributes is a key, we next calculate the closures of pairs of attributes that are possible keys:

o $\{A, B\}^+ = \{A, B, C, I\}$,

o $\{B, D\}^+ = \{B, D, E, F\}$,

o $\{A, D\}^+ = \{A, D, G, H, I, J\}$

None of these pairs are keys either since none of the closures includes all attributes. But the union of the three closures includes all the attributes:

o $\{A, B, D\}^+ = \{A, B, C, D, E, F, G, H, I\}$

Hence, {A, B, D} is a key.

Based on the above analysis, we decompose as follows, in a similar manner to **Exercise 1**, starting with $R = \{A, B, D, C, E, F, G, H, I\}$.

The first-level partial dependencies on the key (which violate 2NF) are:

o $\{A, B\} \rightarrow \{C, I\}$,

o $\{B, D\} \rightarrow \{E, F\}$,

o $\{A, D\} \rightarrow \{G, H, I, J\}$

Hence, R is decomposed into R1, R2, R3, R4 (keys are underlined):

o R1 = {A, B, C, I},

o R2 = {B, D, E, F},

o R3 = {A, D, G, H, I, J},

o R4 = {A, B, D}

Additional partial dependencies exist in R1 and R3 because $\{A\} \rightarrow \{I\}$. Hence, we remove {I} into R5, so the following relations are the result of 2NF decomposition:

o R1 = {A, B, C},

o R2 = {B, D, E, F},

o R3 = {A, D, G, H, J},

o R4 = {A, B, D},

o R5 = {A, I}

Next, we check for transitive dependencies in each of the relations (which violate 3NF). Only R3 has a transitive dependency $\{A, D\} \rightarrow \{H\} \rightarrow \{J\}$, so it is decomposed into R31 and R32 as follows:

o R31 = {H, J},

o R32 = {A, D, G, H}

The final set of 3NF relations is {R1, R2, R31, R32, R4, R5}

Конец дополнений к 2НФ и 3НФ]

Нормальная форма Бойса-Кодда

Определение 3НФ неадекватно при выполнении следующих условий.

1. Переменная-отношение имеет два (или более) потенциальных ключа.
2. Эти потенциальные ключи являются составными.
3. Два или более потенциальных ключей перекрываются (т. е. имеют по крайней мере один общий атрибут).

Поэтому впоследствии исходное определение 3НФ было заменено более строгим определением нормальной формы Бойса-Кодда (НФБК).

Определение 8. Детерминант - любой атрибут (или группа атрибутов), от которого полностью функционально зависит некоторый другой атрибут.

Определение 9. Отношение R находится в нормальной форме Бойса-Кодда (НФБК) в том и только в том случае, если каждый детерминант является потенциальным ключом.

Отношение в НФБК позволяет исключить все виды аномалий обновления, связанные с функциональными зависимостями. Однако отношение в НФБК все еще может быть плохо структурированным и допускать аномалии обновления. В этом случае причиной аномалий обновления является наличие многозначных зависимостей.

[Упражнение.]

Дана схема отношения R(A, B, C, D, E, F, G, H, I, J), для которой выполняется множество функциональных зависимостей S={AB->C, A->DE, B->F, B->G, F->G, D->HIJ}.

- a) Выполняются ли функциональные зависимости AF->BI и AB->DJ для R? Ответ пояснить.
- b) Найти все потенциальные ключи для R.
- c) Показать этапы преобразования R в BCNF.

Решение:

- a)
 - AF->BI: AF->ADEF->ADEFG->ADEFGHIJ not
 - AB->DJ: AB->ABC->ABCDE->ABCDEF->ABCDEFG->ABCDEFGHIJ yes

b)

The only key is AB:

- super-key: see above
- minimal: A->ADHIJ, B->BFG

A and B do not occur on any right side of a FD. Therefore, they must be part of any key. This shows that AB is the only key.

c)

A->DE violates BCNF:

R1 = (ADE, {A->DE})

R2 = (ABCFGHIJ, {AB->C, B->F, B->G, F->G})

R2 is not in BCNF, since B->FG violates BCNF:

R21 = (BFG, {B->F, B->G, F->G})

R22 = (ABCHIJ, AB->C)

R21 is not in BCNF, since F->G violates BCNF.

R211 = (FG, {F->G})

R212 = (BF, {B->F})

decomposition R={R1, R211, R212, R22} lost dependencies: D->HIJ, B->G

Конец упражнения]

[Дополнение ко всему]

Properties of Relational Decompositions

Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i . Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D . It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F . We now define these concepts more formally.

Definition. Given a set of dependencies F on R , the **projection** of F on R_i , denoted by $\pi_{R_i}(F)$ where R_i is a subset of R , is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i . Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+ , the closure of F , such that all their left- and right-hand-side attributes are in R_i . We say that a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F ; that is, $((\pi_{R_1}(F)) \cup \dots \cup (\pi_{R_m}(F)))^+ = F^+$.

If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all leftand right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN — an option that is not practical.

Claim 1. It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation R_i in D is in 3NF.

Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.

Definition. Formally, a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R has the **lossless (nonadditive) join property** with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F , the following holds, where $*$ is the NATURAL JOIN of all the relations in D : $(\pi_{R_1}(r), \dots, \pi_{R_m}(r))^* = r$.

The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term

nonadditive join because it describes the situation more accurately. Although the term *lossless join* has been popular in the literature, we will henceforth use the term *nonadditive join*, which is self-explanatory and unambiguous. The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information.

Testing Binary Decompositions for the Nonadditive Join Property

There is a special case of a decomposition called a **binary decomposition** — decomposition of a relation R into two relations.

Property NJB (Nonadditive Join Test for Binary Decompositions).

A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R if and only if either

- The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+ , or
- The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+

Successive Nonadditive Join Decompositions

Claim 2 (Preservation of Nonadditivity in Successive Decompositions).

If a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R , and if a decomposition $D_i = \{Q_1, Q_2, \dots, Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i , then the decomposition $D = \{R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1}, \dots, R_m\}$ of R has the nonadditive join property with respect to F .

Algorithms for Relational Database Schema Design

We now give three algorithms for creating a relational decomposition from a universal relation. Each algorithm has specific properties, as we discuss next.

Dependency-Preserving Decomposition into 3NF Schemas

Algorithm creates a dependency-preserving decomposition $D = \{R_1, R_2, \dots, R_m\}$ of a universal relation R based on a set of functional dependencies F , such that each R_i in D is in 3NF. It guarantees only the dependency-preserving property; it does *not* guarantee the nonadditive join property. The first step of Algorithm is to find a minimal cover G for F . Note that multiple minimal covers may exist for a given set F . In such cases the algorithms can potentially yield multiple alternative designs.

Algorithm. Relational Synthesis into 3NF with Dependency Preservation

Input: A universal relation R and a set of functional dependencies F on the attributes of R .

1. Find a minimal cover G for F ;
2. For each left-hand-side X of a functional dependency that appears in G , create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ are the only dependencies in G with X as the left-hand-side (X is the key of this relation);
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

Example of Algorithm. Consider the following universal relation:

$U(\text{Emp_ssn}, \text{Pno}, \text{Esal}, \text{Ephone}, \text{Dno}, \text{Pname}, \text{Plocation})$

Emp_ssn , Esal , Ephone refer to the Social Security number, salary, and phone number of the employee.

Pno , Pname , and Plocation refer to the number, name, and location of the project.

Dno is department number.

The following dependencies are present:

FD1: $\text{Emp_ssn} \rightarrow \{\text{Esal}, \text{Ephone}, \text{Dno}\}$

FD2: $\text{Pno} \rightarrow \{\text{Pname}, \text{Plocation}\}$

FD3: $\{\text{Emp_ssn}, \text{Pno}\} \rightarrow \{\text{Esal}, \text{Ephone}, \text{Dno}, \text{Pname}, \text{Plocation}\}$

By virtue of FD3, the attribute set {Emp_ssn, Pno} represents a key of the universal relation. Hence F , the set of given FDs includes {Emp_ssn \rightarrow {Esal, Ephone, Dno}; Pno \rightarrow {Pname, Plocation}; {Emp_ssn, Pno} \rightarrow {Esal, Ephone, Dno, Pname, Plocation}}.

By applying the minimal cover Algorithm, in step 3 we see that Pno is a redundant attribute in Emp_ssn, Pno \rightarrow {Esal, Ephone, Dno}. Moreover, Emp_ssn is redundant in {Emp_ssn, Pno} \rightarrow {Pname, Plocation}. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows (if we group attributes with the same left-hand side into one FD):

Minimal cover G: {Emp_ssn \rightarrow {Esal, Ephone, Dno}; Pno \rightarrow {Pname, Plocation}}

By applying Algorithm to the above Minimal cover G, we get a 3NF design consisting of two relations with keys Emp_ssn and Pno as follows:

R1 (Emp_ssn, Esal, Ephone, Dno)

R2 (Pno, Pname, Plocation)

An observant reader would notice easily that these two relations have lost the original information contained in the key of the universal relation U (namely, that there are certain employees working on certain projects in a many-to-many relationship).

Thus, while the algorithm does preserve the original dependencies, it makes no guarantee of preserving all of the information. Hence, the resulting design is a *lossy* design.

Claim 3. Every relation schema created by Algorithm is in 3NF. (We will not provide a formal proof here; the proof depends on G being a minimal set of dependencies.)

Algorithm is called a **relational synthesis algorithm**, because each relation schema R_i in the decomposition is synthesized (constructed) from the set of functional dependencies in G with the same left-hand-side X.

Nonadditive Join Decomposition into BCNF Schemas

The next algorithm decomposes a universal relation schema $R = \{A_1, A_2, \dots, A_n\}$ into a decomposition $D = \{R_1, R_2, \dots, R_m\}$ such that each R_i is in BCNF and the decomposition D has the lossless join property with respect to F. Algorithm utilizes Property NJB and Claim 2 (preservation of nonadditivity in successive decompositions)

to create a nonadditive join decomposition $D = \{R_1, R_2, \dots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in BCNF.

Algorithm. Relational Decomposition into BCNF with Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Set $D := \{R\}$;
2. While there is a relation schema Q in D that is not in BCNF do
{
 choose a relation schema Q in D that is not in BCNF;
 find a functional dependency $X \rightarrow Y$ in Q that violates BCNF;
 replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
};

Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

By now we know that it is *not possible to have all three of the following:*

- (1) guaranteed nonlossy design,
- (2) guaranteed dependency preservation, and
- (3) all relations in BCNF.

Algorithm. Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F.
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ are the only dependencies in G with X as left-hand-side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R.
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S.

Example of Algorithm. Let us revisit the example given earlier. The minimal cover G holds as before. The second step produces relations R1 and R2 as before. However, now in step 3, we will generate a relation corresponding to the key {Emp_ssn, Pno}. Hence, the resulting design contains:

R1 (Emp_ssn, Esal, Ephone, Dno)

R2 (Pno, Pname, Plocation)

R3 (Emp_ssn, Pno)

This design achieves both the desirable properties of dependency preservation and nonadditive join.

Конец дополнения ко всему]

Многозначные зависимости и четвертая нормальная форма

В качестве примера рассмотрим переменную-отношение **HCTX**, содержащую информацию о курсах обучения, преподавателях и учебниках. В этой переменной-отношении атрибуты, описывающие преподавателей и учебники, принимают в качестве значений *отношения*. Каждый кортеж переменной-отношения **HCTX** состоит из атрибутов названия курса (**COURSE**), а также атрибута-отношения с именами преподавателей (**TEACHERS**) и атрибута-отношения с названиями учебников (**TEXTS**). Смысл каждого кортежа состоит в том, что соответствующий курс может преподаваться любым из указанных преподавателей с использованием всех указанных учебников. Предположим, что для заданного курса может быть определено произвольное количество соответствующих преподавателей и учебников. Более

того, допустим, что преподаватели и рекомендуемые учебники совершенно независимы друг от друга. Это значит, что независимо от того, кто преподает данный курс, всегда используется один и тот же набор учебников. Наконец, допустим, что определенный преподаватель или определенный учебник может быть связан с любым количеством курсов.

```
<HCTX>
  <Course>
    <Name>Физика</Name>
    <Teachers>
      <Teacher>Проф. Фейнман</Teacher>
      <Teacher>Проф. Арнольд</Teacher>
    </Teachers>
    <Texts>
      <Text>Механика</Text>
      <Text>Оптика</Text>
    </Texts>
  </Course>
  <Course>
    <Name>Математика</Name>
    <Teachers>
      <Teacher>Проф. Арнольд</Teacher>
    </Teachers>
    <Texts>
      <Text>Механика</Text>
      <Text>Анализ</Text>
      <Text>Алгебра</Text>
    </Texts>
  </Course>
</HCTX>
```

Пусть необходимо исключить атрибуты, принимающие в качестве значений отношения. Один из способов заключается в простой замене переменной-отношения **HCTX** переменной-отношением **CTX** с тремя скалярными атрибутами { **Course**, **Teacher** и **Text** }, как показано в таблице.

CTX

Course	Teacher	Text
Физика	Проф. Фейнман	Механика
Физика	Проф. Фейнман	Оптика
Физика	Проф. Арнольд	Механика
Физика	Проф. Арнольд	Оптика
Математика	Проф. Арнольд	Механика
Математика	Проф. Арнольд	Анализ
Математика	Проф. Арнольд	Алгебра

Очевидно, что переменная-отношение **CTX** характеризуется значительной **избыточностью**, вследствие чего возникнут **аномалии обновления**. Например, для добавления информации о том, что курс ‘Физика’ может читаться новым преподавателем, необходимо создать два новых кортежа, по одному для каждого используемого учебника. Как можно избежать появления таких проблем? Ситуацию можно улучшить, если выполнить декомпозицию переменной-отношения **CTX** на две проекции (например, с именами **CT** и **CX**) с атрибутами { **Course**, **Teacher** } и { **Course**, **Text** } соответственно. При этом переменная-отношение **CTX** может быть восстановлена путем обратного соединения проекций **CT** и **CX**, и поэтому данная декомпозиция была выполнена без потерь.

CT

Course	Teacher
Физика	Проф. Фейнман
Физика	Проф. Арнольд
Математика	Проф. Арнольд

CX

Course	Text
Физика	Механика
Физика	Оптика
Математика	Механика
Математика	Анализ

С неформальной точки зрения, очевидно, что переменная-отношение **CTX** спроектирована неудачно и ее декомпозиция на проекции **CT** и **CX** является более удачным решением. Но с формальной точки зрения это совсем неочевидно. Переменная-отношение **CTX** вообще не имеет функциональных зависимостей (за исключением таких тривиальных, как **Course -> Course**). Фактически переменная-отношение **CTX** находится в **НФБК**, поскольку все ее атрибуты входят в состав ее ключа, а любая, подобная переменная-отношение обязательно находится в **НФБК**. Проекции **CT** и **CX** также являются полностью ключевыми, а потому находятся в **НФБК**.

Способы разрешения «проблем», которые связаны с переменными-отношениями в **НФБК**, подобными переменной-отношению **CTX**, были сформулированы Фейгином (Fagin) в строгом теоретическом виде с использованием понятия **многозначной зависимости (МЗ)**, которую можно считать обобщением **Ф3** в том смысле, что каждая **Ф3** также является **МЗ**. Точнее говоря, **Ф3** - это **МЗ**, в которой множество зависимых значений, соответствующее заданному значению детерминанта, всегда является однозначным множеством. В переменной-отношении **CTX** есть две **МЗ**, которые не являются **Ф3**:

Course -> Teacher

Course -> Text

Формальное определение МЗ. Пусть **A**, **B** и **C** являются произвольными подмножествами множества атрибутов переменной-отношения **R**. Тогда подмножество **B** **многозначно зависит** от подмножества **A**, что символически выражается записью **A -> B**, тогда и только тогда, когда множество значений **B**, соответствующее заданной паре (значение **A**, значение **C**) переменной-отношения **R**, зависит от **A**, но не зависит от **C**.

Другое определение МЗ. Многозначная зависимостей **X -> Y** в схеме отношения **R**, где **X** и **Y** подмножества **R**, указывает следующее ограничение на любое отношении **r** над схемой **R**: если два кортежа **t1** и **t2** существуют в **r** такие, что **t1[X] = t2[X]**, то должны существовать также два кортежа **t3** и **t4** в **r** над **R** со следующими свойствами, где **Z = (R - (X -> Y))**:

- **t3[X] = t4[X] = t1[X] = t2[X]**.
- **t3[Y] = t1[Y] and t4[Y] = t2[Y]**.
- **t3[Z] = t2[Z] and t4[Z] = t1[Z]**.

Всякий раз, когда имеет место **X -> Y**, то говорят, что **X** многозначно определяет **Y**. В силу симметрии в определении, когда имеет место **X -> Y** в **R**, то имеет место и **X -> Z**. Следовательно, **X -> Y** влечет **X -> Z**, и поэтому иногда пишут **X -> Y | Z**.

Лемма Фейгина. Для данной переменной-отношения **R { A, B, C }** **МЗ A ->> B** выполняется тогда и только тогда, когда также выполняется **МЗ A ->> C**.

Таким образом, **МЗ** всегда образуют связанные пары, поэтому обычно их представляют вместе в символическом виде **A ->> B | C**.

Возвращаясь к исходной задаче с переменной-отношением **CTX**, можно отметить следующее: проблема с переменной-отношением **CTX** возникает из-за того, что она содержит **МЗ**, которые не являются **Ф3**. Проекции **CT** и **CX** не содержат **МЗ**, а потому они действительно представляют собой некоторое усовершенствование исходной структуры. Замена исходной переменной-отношения **CTX** двумя проекциями **CT** и **CX** является правомочной в соответствии с теоремой Фейгина.

Теорема Фейгина. Пусть **A**, **B** и **C** являются множествами атрибутов переменной-отношения **R { A, B, C }**. Переменная-отношение **R** будет равна соединению ее проекций **{ A, B }** и **{ A, C }** тогда и только тогда, когда для переменной-отношения **R** выполняется **МЗ A ->> B | C**.

Эта теорема является более строгой версией теоремы Хита. Теперь, следуя работе Фейгина, можно дать определение **четвертой нормальной формы**.

Определение тривиальной МЗ. **МЗ A ->> B** называется тривиальной, если выполняется хотя бы одно из условий:

1. Множество **A** является надмножеством **B**;
2. Объединение **A** и **B** образует весь заголовок отношения.

Определение 4НФ. Переменная-отношение **R** находится в **четвертой нормальной форме (4НФ)** тогда и только тогда, когда в случае существования таких подмножеств **A** и **B** атрибутов этой переменной-отношения **R**, для которых

выполняется нетривиальная **M3** $A \rightarrow\!\!\!> B$, все атрибуты переменной-отношения **R** также функционально зависят от атрибута **A**.

Это определение можно также сформулировать в следующей эквивалентной форме: переменная-отношение **R** находится в **4НФ**, если она находится **НФБК** и все **M3** в переменной-отношении **R** фактически представляют собой **Ф3** от ее ключей.

Переменная-отношение **СТХ** не находится в **4НФ**, поскольку содержит **M3**, которые не являются **Ф3**, не говоря уже о том, что последняя должна быть еще и **Ф3** от ключа. Однако, обе ее проекции, **СТ** и **СХ**, находятся в **4НФ**. Следовательно, **4НФ** обеспечивает лучшую структуру данных по сравнению с **НФБК**. Фейгин показал, что **4НФ** всегда является достижимой, но в реальной практике такая декомпозиция (или даже декомпозиция до **НФБК**) не всегда оказывается полезной и нужной.

В заключение можно констатировать следующее:

- полностью ключевое отношение всегда находится в НФБК поскольку оно не имеет Ф3.
- полностью ключевое отношение, которое не имеет Ф3 но имеет М3, не находится в 4НФ.
- отношение, которое не находится в 4НФ в связи с нетривиальным М3 должны быть декомпозировано, чтобы преобразовать его в набор отношений в 4НФ.
- декомпозиция устраняет избыточность, вызванную М3.

[Дополнение к 4НФ]

Определение. Многозначная зависимостей $X \rightarrow\!\!\!> Y$ в схеме отношения **R**, где **X** и **Y** подмножества **R**, указывает следующее ограничение на любое отношении **r** над схемой **R**: если два кортежа **t1** и **t2** существуют в **r** такие, что $t1[X] = t2[X]$, то должны существовать также два кортежа **T3** и **T4** в **r** над **R** со следующими свойствами, где $Z = (R - (X \rightarrow Y))$:

- $t3[X] = t4[X] = t1[X] = t2[X]$.
- $t3[Y] = t1[Y] \text{ and } t4[Y] = t2[Y]$.
- $t3[Z] = t2[Z] \text{ and } t4[Z] = t1[Z]$.

Всякий раз, когда имеет место $X \rightarrow\!\!\!> Y$, то говорят, что **X** multidetermines **Y**. В силу симметрии в определении, когда имеет место $X \rightarrow\!\!\!> Y$ в **R**, то имеет место и $X \rightarrow\!\!\!> Z$. Следовательно, $X \rightarrow\!\!\!> Y$ влечет $X \rightarrow\!\!\!> Z$, и поэтому иногда пишут $X \rightarrow\!\!\!> Y | Z$.

Определение. Схема отношения **R** в 4NF по отношению к множеству зависимостей **F** (которая включает в себя функциональные зависимости и многозначных зависимостей), если для каждой нетривиальной многозначной зависимости $X \rightarrow\!\!\!> Y$ в F^+ **X** является суперключом для **R**.

Можем констатировать следующее:

- полностью ключевое отношение всегда находится в BCNF поскольку оно не имеет FD.
- полностью ключевое отношение, которое не имеет FDs но имеет MVD, не находится в 4NF.
- отношение, которое не находится в 4NF в связи с нетривиальным MVD должны быть декомпозировано, чтобы преобразовать его в набор отношений в 4NF.
- декомпозиция устраняет избыточность, вызванную MVD.

[Конец дополнения к 4НФ]

Зависимости соединения и пятая нормальная форма

До сих пор предполагалось, что единственной необходимой или допустимой операцией в процессе нормализации является замена переменной-отношения по правилам декомпозиции без потерь *только двумя* ее проекциями. Однако существуют переменные-отношения, для которых нельзя выполнить декомпозицию без потерь на две проекции, но которые *можно* подвергнуть декомпозиции без потерь на три или более проекций. Подобные переменные-отношения обозначаются термином "п-декомпозируемая переменная-отношение".

В качестве примера можно рассмотреть переменную-отношение **SPJ** из базы данных «Поставщики, детали и проекты» (в целях упрощения изложения атрибут **Qty** исключен). Эта переменная-отношение состоит, только из ключевых атрибутов, не содержит нетривиальных **Ф3** и **М3** и потому находится в **4НФ**. На рисунках показаны следующие компоненты:

- Исходное отношение **SPJ**.
- Три бинарные проекции, **SP**, **PJ** и **JS**, переменной-отношения **SPJ**.
- Результат соединения проекций **SP** и **PJ** по атрибуту **Pno**.
- Соединение этого результата с проекцией **JS** по комбинации атрибутов (**Jno**, **Sno**).

SPJ (Исходное отношение)

Sno	Pno	Jno
1	1	2
1	2	1
2	1	1
1	1	1

SP

Sno	Pno
1	1
1	2
2	1

PJ

Pno	Jno
1	2
2	1
1	1

JS

Jno	Sno
2	1
1	1
1	2

SPJ[^] (Соединение **SP** и **PJ** по атрибуту **Pno**)

Sno	Pno	Jno
1	1	2
1	2	1
2	1	1
2	1	2
1	1	1

SPJ (Соединение **SPJ[^]** и **JS** по комбинации атрибутов (**Jno**, **Sno**))

Sno	Pno	Jno
1	1	2
1	2	1
2	1	1
1	1	1

В результате первого соединения получается копия исходной переменной-отношения **SPJ[^]** с одним дополнительным (излишним) кортежем, а в результате второго соединения этот лишний кортеж исключается. Иначе говоря, исходная переменная-отношение **SPJ** является 3-декомпозируемой. Переменная-отношение **SPJ** может быть получена только в результате соединения всех трех ее бинарных проекций, но не любых двух из них.

Представленный пример выполнен в терминах *отношений*, а не *переменных-отношений*. 3-декомпозируемость переменной-отношения **SPJ** может быть более фундаментальным и не зависящим от времени свойством (т.е. свойством, которое удовлетворяется для всех допустимых значений данной переменной-отношения), если данная переменная-отношение удовлетворяет определенному не зависящему от времени ограничению целостности:

ЕСЛИ кортежи **(s1, p1, j2), (s2, p1, j1), (s1, p2, j1)** присутствуют в **SPJ**,
ТО кортеж **(s1, p1, j1)** также присутствует в **SPJ**.

Следует обратить внимание на циклическую структуру этого ограничения. *Переменная-отношение будет n-декомпозируемой для n>2 тогда и только тогда, когда она удовлетворяет некоторому циклическому ограничению*. Ограничение 3-декомпозируемости для краткости принято называть **3Д-ограничением**. Поскольку 3Д-ограничение удовлетворяется тогда и только тогда, когда переменная-отношение равносильно соединению некоторых ее проекций, такое ограничение называется **зависимостью соединения (3С)**.

Определение зависимости соединения. Пусть **R** является переменной-отношением, а **A, B, ..., Z** - произвольными подмножествами множества ее атрибутов. Переменная-отношение **R** удовлетворяет **зависимости соединения** *{ **A, B, ..., Z**

} (читается «звездочка A, B, \dots, Z ») тогда и только тогда, когда любое допустимое значение переменной-отношения R эквивалентно соединению ее проекций по подмножествам атрибутов A, B, \dots, Z .

Например, если использовать сокращенную символьную запись SP для подмножества $\{ Sno, Pno \}$ множества атрибутов переменной-отношения SPJ и аналогично использовать сокращения PJ и JS для двух других подмножеств, то переменная-отношение SPJ будет удовлетворять зависимости соединения $*\{ SP, PJ, JS \}$. Отсюда ясно, что переменная-отношение SPJ с зависимостью соединения $*\{ SP, PJ, JS \}$ может быть 3-декомпозируемой.

Теорема Фейгина (которая рассматривалась ранее) утверждает, что переменная-отношение $R \{ A, B, C \}$ может быть декомпозирована без потерь на проекции с атрибутами $\{ A, B \}$ и $\{ A, C \}$ тогда и только тогда, когда для переменной-отношения R выполняются **МЗ** $A \rightarrow\!\!\!> B$ и $A \rightarrow\!\!\!> C$.

Теперь теорема Фейгина может быть сформулирована иначе. Переменная-отношение $R \{ A, B, C \}$ удовлетворяет зависимости соединения $*\{ AB, AC \}$ тогда и только тогда, когда она удовлетворяет многозначной зависимости $A \rightarrow\!\!\!> B | C$.

Поскольку эту теорему можно использовать в качестве определения многозначной зависимости, то либо многозначная зависимость является частным случаем зависимости соединения, либо (что эквивалентно) зависимость соединения является обобщение понятия многозначной зависимости. Формально получается следующее $A \rightarrow\!\!\!> B | C = *\{ AB, AC \}$.

Рассмотренный выше пример обнаруживает следующую проблему: переменная-отношение SPJ содержит зависимость соединения, которая не является ни многозначной, ни функциональной. Такую переменную-отношение можно декомпозировать на меньшие компоненты, а именно – на проекции, определяемые зависимостью соединения. Данный процесс декомпозиции может повторяться до тех пор, пока все результирующие переменные-отношения не будут находиться в **пятой нормальной форме**.

Определение 5НФ. Переменная-отношение R находится в **пятой нормальной форме (5НФ)**, которую иногда иначе называют **проекционно-соединительной нормальной формой (ПСНФ)**, тогда и только тогда, когда каждая нетривиальная 3С в переменной-отношении R подразумевается ее потенциальными ключами.

Определение тривиальной 3С. 3С $*\{ A, B, \dots, Z \}$ называется **тривиальной** тогда и только тогда, когда одна из проекций A, B, \dots, Z является проекцией, идентичной R (т.е. проекцией по всем атрибутам переменной-отношения R).

Определение. 3С $*\{ A, B, \dots, Z \}$ подразумевается потенциальными ключами тогда и только тогда, когда каждое подмножество атрибутов A, B, \dots, Z фактически является суперключом для данной переменной-отношения.

Переменная-отношение SPJ не находится в **5НФ**. Она удовлетворяет некоторой зависимости соединения, а именно – **ЗД-ограничению**, которое, конечно же, не подразумевается ее единственным потенциальным ключом (этот ключ является комбинацией всех ее атрибутов). Иначе говоря, переменная-отношение SPJ не находится в **5НФ**, поскольку она может быть 3-декомпозирована и возможность такой декомпозиции не подразумевается тем фактом, что комбинация атрибутов $\{ Sno, Pno, Jno \}$ является ее потенциальным ключом. Наоборот, после 3-декомпозиции проекции SP, PJ, JS находятся в **5НФ**, поскольку в них вовсе нет нетривиальных зависимостей соединения.

Возникает вопрос, следует ли выполнять такую декомпозицию? Ответ: если переменная-отношение находится в **5НФ**, то гарантируется, что она не содержит аномалий, которые могут быть исключены посредством ее разбиения на проекции. Конечно, это не значит, что данная переменная-отношение свободна от всех возможных аномалий. Это всего лишь означает, что она свободна от аномалий, которые могут быть исключены с помощью разбиения на проекции. **5НФ** является окончательной нормальной формой по отношению к операциям проекции и соединения.

Общая схема процедуры нормализации

Каждый этап процесса нормализации заключается в разбиении на проекции переменных-отношений, полученных на предыдущем этапе. При этом на каждом этапе нормализации существующие ограничения используются для выбора тех проекций, которые будут получены в этот раз. Весь процесс можно неформально определить с помощью перечисленных ниже правил.

1. Переменную-отношение в **1НФ** следует разбить на такие проекции, которые позволяют исключить все функциональные зависимости, не являющиеся неприводимыми. В результате будет получен набор переменных-отношений в **2НФ**.
2. Полученные переменные-отношения в **2НФ** следует разбить на такие проекции, которые позволяют исключить все существующие транзитивные функциональные зависимости. В результате будет получен набор переменных-отношений в **3НФ**.
3. Полученные переменные-отношения в **3НФ** следует разбить на проекции, позволяющие исключить любые оставшиеся функциональные зависимости, в которых детерминанты не являются потенциальными ключами. В результате такого приведения будет получен набор переменных-отношений в **НФБК**. Замечание. Правила 1-3 могут

быть объединены в одно: "Исходную переменную-отношение следует разбить на проекции, позволяющие исключить все функциональные зависимости, в которых детерминанты не являются потенциальными ключами".

4. Полученные переменные-отношения в **НФБК** следует разбить на проекции, позволяющие исключить любые многозначные зависимости, которые не являются функциональными. В результате будет получен набор переменных-отношений в **4НФ**.
5. Полученные переменные-отношения в **4НФ** следует разбить на проекции, позволяющие исключить любые зависимости соединения, которые не подразумеваются потенциальными ключами. В результате будет получен набор переменных-отношений в **5НФ**.

Введение в SQL

Краткая историческая справка

В начале 70-х годов доктор Э.Ф.Кодд, работавший в компании IBM, разработал теорию реляционных баз данных. Для воплощения идей реляционной модели он разработал язык реляционных баз данных и назвал его Alpha. IBM предпочла передать дальнейшую разработку группе программистов, неподконтрольной доктору Кодду. Нарушив некоторые принципы реляционной модели, они реализовали её в экспериментальной реляционной СУБД IBM System R, для которой затем был создан специальный язык SEQUEL (Structured English QUERy Language — «структурированный английский язык запросов»), позволявший относительно просто управлять данными в этой СУБД. Поскольку SEQUEL было уже зарегистрированной торговой маркой, название сократили до SQL, и таким оно осталось по сей день. Язык ориентирован **главным образом** на удобную и понятную пользователям формулировку запросов к реляционным БД. В действительности SQL обеспечивает и другие возможности.

В настоящее время язык SQL реализован во всех коммерческих реляционных СУБД. Все компании-производители провозглашают соответствие своей реализации стандарту SQL, но на самом деле реализуют диалекты SQL.

Стандартизация SQL

Формальное название стандарта SQL - это ISO/IEC 9075 "Database Language SQL" (Язык баз данных SQL). Время от времени выпускается пересмотренная версия этого стандарта; наиболее свежее обновление было выпущено в 2011 году. Версия 2011 известна как ISO/IEC 9075:2011 или просто как SQL:2011. Версиями до неё были: SQL-86 (SQL1), SQL-92 (SQL2), SQL:1999 (SQL3), SQL:2003, SQL:2006 и SQL:2008. Каждая версия замещает предыдущую, так что требования соответствия стандарту более ранних версий не имеют официальной силы. Стандарт SQL:2011 не является свободно доступным. Полный стандарт можно приобрести у организации ISO. Общая организация стандарта SQL:2011 имеет вид:

- 9075-1, SQL/Framework;
- 9075-2, SQL/Foundation;
- 9075-3, SQL/CLI; (спецификация интерфейса уровня вызова)
- 9075-4, SQL/PSM; (спецификация хранимых процедур)
- 9075-9, SQL/MED; (управление внешними данными)
- 9075-10, SQL/OLB; (связывание с объектно-ориентированными языками программирования)
- 9075-11, SQL/Schemata;
- 9075-13, SQL/JRT; (использование подпрограмм и типов SQL в языке программирования Java)
- 9075-14, SQL/XML. (спецификации языковых средств, позволяющих работать с XML-документами в среде SQL)

Несмотря на наличие международного стандарта, многие производители СУБД вносят изменения в язык SQL, тем самым отступая от стандарта. В результате у разных производителей СУБД в ходе разные диалекты SQL, в общем случае между собой несовместимые. В настоящее время проблема совместимости решается так: описание языка имеет модульную структуру, основная часть стандарта вынесена в раздел «SQL/Foundation», все остальные выведены в отдельные модули, остался только один уровень совместимости – «Core», что означает поддержку этой основной части. Поддержка остальных возможностей оставлена на усмотрение производителей СУБД.

При всех своих изменениях, SQL остаётся единственным механизмом связи между прикладным программным обеспечением и базой данных. В тоже время, современные СУБД предоставляют пользователю развитые средства визуального построения запросов. Хотя SQL и задумывался как средство работы конечного пользователя, в конце концов, он стал настолько сложным, что превратился в инструмент профессионального программиста.

Transact-SQL (T-SQL)

Transact-SQL (T-SQL) – расширение языка SQL, созданное компанией Microsoft (для Microsoft SQL Server) и Sybase (для Sybase ASE). **Замечание.** Последняя версия Sybase – «SAP ADAPTIVE SERVER ENTERPRISE 16».

Классификация инструкций T-SQL

Инструкции T-SQL принято делить на следующие категории:

- **инструкции языка описания данных** (Data Definition Language, DDL):
 - CREATE создает объект БД (саму базу, таблицу, представление, пользователя и т. д.) [54]
 - ALTER изменяет объект [49]
 - DROP удаляет объект [52]
 - ENABLE TRIGGER включает триггер DML, DDL или logon
 - DISABLE TRIGGER отключает триггер

- TRUNCATE TABLE удаляет все строки в таблице, не записывая в журнал удаление отдельных строк
 - UPDATE STATISTICS обновляет статистику оптимизации запросов для таблицы или индексированного представления
- **инструкции языка обработки данных** (Data Manipulation Language, DML):
 - SELECT считывает данные, удовлетворяющие заданным условиям
 - INSERT добавляет новые данные
 - UPDATE изменяет существующие данные
 - DELETE удаляет данные
 - MERGE выполняет операции вставки, обновления или удаления для целевой таблицы на основе результатов соединения с исходной таблицей
 - BULK INSERT выполняет импорт файла данных в таблицу или представление базы данных в формате, указанном пользователем
 - READTEXT считывает значения text, ntext или image из столбцов типа text, ntext или image начиная с указанной позиции
 - WRITETEXT обновляет и заменяет все поле text, ntext или image
 - UPDATETEXT обновляет часть поля text, ntext или image
- **инструкции безопасности** (ранее инструкции языка доступа к данным - Data Control Language, DCL):
 - GRANT предоставляет пользователю разрешения на определенные операции с объектом
 - REVOKE отзывает ранее выданые разрешения
 - DENY задает запрет, имеющий приоритет над разрешением
 - ADD SIGNATURE добавляет цифровую подпись для хранимой процедуры, функции, сборки или триггера
 - OPEN MASTER KEY открывает главный ключ в текущей базе данных
 - CLOSE MASTER KEY закрывает главный ключ в текущей базе данных
 - OPEN SYMMETRIC KEY расшифровывает симметричный ключ и делает его доступным для использования
 - CLOSE SYMMETRIC KEY закрывает симметричный ключ или все симметричные ключи, открытые в текущем сеансе
 - EXECUTE AS контекст выполнения сеанса переключается на заданное имя входа и имя пользователя
 - REVERT переключает контекст выполнения в контекст участника, вызывавшего последнюю инструкцию EXECUTE AS
 - SETUSER позволяет члену предопределенной роли сервера sysadmin или члену предопределенной роли базы данных db_owner олицетворять другого пользователя
- **инструкции управления транзакциями** (Transaction Control Language, TCL):
 - BEGIN DISTRIBUTED TRANSACTION запускает распределенную транзакцию, управляемую координатором распределенных транзакций
 - BEGIN TRANSACTION отмечает начальную точку явной локальной транзакции
 - COMMIT TRANSACTION отмечает успешное завершение явной или неявной транзакции
 - COMMIT WORK действует так же, как и инструкция COMMIT TRANSACTION
 - ROLLBACK TRANSACTION откатывает явные или неявные транзакции до начала или до точки сохранения транзакции
 - ROLLBACK WORK действует так же, как и инструкция ROLLBACK TRANSACTION
 - SAVE TRANSACTION устанавливает точку сохранения внутри транзакции
- **инструкции управления потоком** (Control-of-Flow Language, CFL):
 - BEGIN...END
 - BREAK
 - CONTINUE
 - GOTO
 - IF...ELSE
 - RETURN
 - THROW
 - TRY...CATCH
 - WAITFOR
 - WHILE
 - PRINT
 - EXECUTE
 - RAISERROR
- **инструкции курсоров:**
 - DECLARE CURSOR определяет атрибуты серверного курсора
 - OPEN открывает серверный курсор и заполняет его
 - FETCH получает определенную строку из серверного курсора
 - CLOSE закрывает открытый курсор, высвобождая текущий результирующий набор и снимая блокировки курсоров для строк, на которых установлен курсор
 - DEALLOCATE удаляет ссылку курсора и освобождает структуры данных, составляющие курсор
- **инструкции BACKUP и RESTORE** - позволяют создавать резервные копии баз данных и восстанавливать базы данных

- **команды управления:**
 - CHECKPOINT создает ручную контрольную точку в базе данных SQL Server, с которой в данный момент установлено соединение
 - DBCC выступают в качестве консольных команд базы данных для SQL Server
 - KILL прерывает пользовательский процесс, определяемый идентификатором сеанса или единицей работы
 - KILL QUERY NOTIFICATION SUBSCRIPTION удаляет подписки на уведомления о запросах из экземпляра SQL Server
 - KILL STATS JOB останавливает асинхронное задание обновление статистики
 - RECONFIGURE изменяет значение параметра конфигурации с помощью хранимой системной процедуры sp_configure
 - SHUTDOWN немедленно останавливает SQL Server
- **инструкции SET** - изменяют текущий сеанс, управляя специфическими данными. Инструкции SET группируются в следующие категории:
 - **Инструкции даты и времени**
 - SET DATEFIRST
 - **SET DATEFORMAT**
 - **Инструкции блокировки**
 - SET DEADLOCK_PRIORITY
 - SET LOCK_TIMEOUT
 - **Прочие инструкции**
 - SET CONCAT_NULL_YIELDS_NULL
 - SET CURSOR_CLOSE_ON_COMMIT
 - SET FIPS_FLAGGER
 - **SET IDENTITY_INSERT**
 - SET LANGUAGE
 - SET OFFSETS
 - **SET QUOTED_IDENTIFIER**
 - **Инструкции выполнения запросов**
 - SET ARITHABORT
 - SET ARITHIGNORE
 - SET FMTONLY
 - **SET NOCOUNT**
 - SET NOEXEC
 - SET NUMERIC_ROUNDABORT
 - SET PARSEONLY
 - SET QUERY_GOVERNOR_COST_LIMIT
 - **SET ROWCOUNT**
 - SET TEXTSIZE
 - **Инструкции настроек ISO**
 - SET ANSI_DEFAULTS
 - SET ANSI_NULL_DFLT_OFF
 - SET ANSI_NULL_DFLT_ON
 - SET ANSI_NULLS
 - SET ANSI_PADDING
 - SET ANSI_WARNINGS
 - **Статистические инструкции**
 - SET FORCEPLAN
 - **SET SHOWPLAN_ALL**
 - **SET SHOWPLAN_TEXT**
 - **SET SHOWPLAN_XML**
 - SET STATISTICS IO
 - SET STATISTICS XML
 - SET STATISTICS PROFILE
 - SET STATISTICS TIME
 - **Инструкции управления транзакциями**
 - SET IMPLICIT_TRANSACTIONS
 - SET REMOTE_PROC_TRANSACTIONS
 - SET TRANSACTION ISOLATION LEVEL
 - SET XACT_ABORT

Полный справочник по инструкциям языка T-SQL находится по адресу
<http://msdn.microsoft.com/ru-ru/library/bb510741.aspx>

Системы типов данных языка SQL

A. Типы данных SQL:2011

Все допустимые в SQL типы данных, которые можно использовать при определении столбцов, разбиваются на следующие категории:

- точные числовые типы (exact numerics);
- приближенные числовые типы (approximate numerics);
- типы символьных строк (character strings);
- типы битовых строк (bit strings);
- типы даты и времени (datetimes);
- типы временных интервалов (intervals);
- булевский тип (Booleans);
- типы коллекций (collection types);
- анонимные строчные типы (anonymous row types);
- типы, определяемые пользователем (user-defined types);
- ссылочные типы (reference types).

Комментарии к некоторым типам данных (при первом знакомстве пропустить)

Булевский тип

При определении столбца булевского типа указывается просто спецификация BOOLEAN. Булевский тип состоит из трех значений: true, false и unknown (соответствующие литералы обозначаются TRUE, FALSE и UNKNOWN). Поддерживается возможность построения булевых выражений, которые вычисляются в трехзначной логике.

Типы коллекций

Под термином коллекция обычно понимается одно из следующих образований: массив, список, множество и мульти множество. В варианте SQL:2011 специфицированы только массивы и мульти множества.

Анонимные строчные типы

Анонимный строчный тип – это конструктор типов ROW, позволяющий производить безымянные типы строк (кортежей). При определении столбца, значения которого должны принадлежать некоторому строчному типу, используется конструкция ROW ($fld_1, fld_2, \dots, fld_n$), где каждый элемент fld_i , определяющий поле строчного типа, задается в виде тройки $fldname, fldtype, fldoptions$. В качестве типа данных поля строчного типа можно использовать любой допустимый в SQL тип данных, включая типы коллекций, определяемые пользователями типы и другие строчные типы. Необязательный элемент $fldoptions$ может задаваться для указания применяемого по умолчанию порядка сортировки, если соответствующий подэлемент $fldtype$ указывает на тип символьных строк, а также должен задаваться, если $fldtype$ указывает на ссылочный тип. Степенью строчного типа называется число его полей.

Типы, определяемые пользователем

Эта категория типов данных связана с объектными расширениями языка SQL. Различают:

- Структурные типы (Structured Types). Можно определить долговременный хранимый, именованный тип данных, включающий один или более атрибутов любого из допустимых в SQL типа данных, в том числе другие структурные типы, типы коллекций, строчные типы и т. д. Стандарт SQL не накладывает ограничений на сложность получаемой в результате структуры данных, однако не запрещает устанавливать такие ограничения в реализации. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа.
- Индивидуальные типы (Distinct Types). Можно определить долговременно хранимый, именованный тип данных, опираясь на единственный предопределенный тип. Например, можно определить индивидуальный тип данных PRICE, опираясь на тип DECIMAL (5, 2). Тогда значения типа PRICE представляются точно так же, как значения типа DECIMAL (5, 2). В существующем стандарте индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Например, чтобы сложить два значения типа PRICE требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями типа DECIMAL (5, 2). Другая возможность состоит в явном определении методов, функций и процедур, связанных с данным индивидуальным типом. Похоже, что в будущих версиях стандарта появятся и другие, более удобные возможности.

Ссылочные типы

Эта категория типов данных связана с объектными расширениями языка SQL. Обеспечивается механизм конструирования типов (ссыльных типов), которые могут использоваться в качестве типов столбцов некоторого вида таблиц (типовизированных таблиц). Фактически значениями ссыльного типа являются строки соответствующей типизированной таблицы. Более точно, каждой строке типизированной таблицы приписывается уникальное значение (нечто вроде первичного ключа, назначаемого системой или приложением), которое может использоваться в методах, определенных для табличного типа, для уникальной идентификации строк соответствующей таблицы. Эти уникальные значения называются ссыльными значениями, а их тип – ссыльным типом. Ссыльный тип может содержать только те значения, которые действительно ссылаются на экземпляры указанного типа (т. е. на строки соответствующей типизированной таблицы).

B. Типы данных SQL Server

- Точные числа: bigint int smallint tinyint bit decimal numeric money smallmoney
- Приблизительные числа: float real
- Дата и время: date, datetime2, datetime, datetimeoffset, smalldatetime, time
- Символьные строки: char varchar text
- Символьные строки в Юникоде: nchar nvarchar ntext
- Двоичные данные: binary varbinary image
- Прочие типы данных: cursor, hierarchyid, sql_variant, table, timestamp, uniqueidentifier, xml, пространственные типы (geography и geometry)

В зависимости от параметров хранения, некоторые типы данных в SQL Server относятся к следующим группам:

- типы данных больших значений: varchar(max), nvarchar(max) и varbinary(max);
- типы данных больших объектов: text, ntext, image, varchar(max), nvarchar(max), varbinary(max) и xml.

Идентификаторы T-SQL

Идентификаторы в SQL Server могут присваиваться любым сущностям. Для большинства объектов идентификаторы необходимы, а для некоторых, например ограничений, необязательны. Существует два класса идентификаторов:

- a) обычные и
- b) с разделителями.

Правила для обычных идентификаторов

1. Первым символом должен быть один из следующих:
 - a) латинская буква,
 - b) подчеркивание '_',
 - c) коммерческое at '@',
 - d) решетка '#'.
2. Определенные символы в начале идентификатора имеют особое значение:
 - a) идентификатор, начинающийся символом @_, означает локальную переменную или параметр,
 - b) идентификатор, начинающийся символом #_, означает локальный временный объект,
 - c) идентификатор, начинающийся двойным символом ##_, означает глобальный временный объект,
 - d) некоторые функции языка T-SQL имеют имена, начинающиеся двойным символом @@_, и во избежание путаницы с этими функциями не следует использовать имена, начинающиеся символами @@_.
3. Последующие символы могут включать:
 - a) латинские буквы,
 - b) десятичные цифры,
 - c) символы @, \$, # и _.

Замечание. Правила записи обычных идентификаторов зависят от уровня совместимости базы данных, который можно установить с помощью инструкции ALTER DATABASE.

Идентификаторы с разделителями могут содержать то же количество символов, что и обычные идентификаторы. Это может быть от 1 до 128 символов, не включая символы-разделители. Идентификаторы локальных временных таблиц могут быть максимум 116 символов.

Правила для идентификаторов с разделителями

Идентификаторы, не соответствующие правилам, могут содержать любую комбинацию символов в текущей кодовой странице и должны заключаться в разделители:

- a) двойные кавычки и
- b) квадратные скобки.

В этом случае SET QUOTED_IDENTIFIER должен быть установлен в состояние ON.

Примеры: "Blanks in Table Name" или [Blanks in Table Name].

Использование идентификаторов в качестве имен объектов

Полное имя объекта состоит из четырех идентификаторов: имени сервера, имени базы данных, имени схемы и имени объекта, которые отображаются в следующем формате:

имя_сервера.[имя_базы_данных].[имя_схемы].имя_объекта

Замечания.

- 1) Большинство ссылок на объекты используют трехкомпонентные имена.
- 2) По умолчанию в качестве *имени_сервера* используется локальный сервер.
- 3) По умолчанию в качестве *имени_базы_данных* используется текущая база данных соединения.
- 4) По умолчанию в качестве *имени_схемы* обычно используется **dbo**, если в явном виде не были заданы иные настройки.
- 5) Четырехсоставные имена обычно используются в распределенных запросах и удаленных вызовах хранимых процедур.
- 6) Для ссылки на столбцы используется дополнительная точечная нотация.
- 7) Для обращения к свойствам столбцов UDT используется дополнительная точечная нотация.

Зарезервированные ключевые слова T-SQL

- список зарезервированных слов SQL Server (185 штук)
- список зарезервированных ключевых слов ODBC (235 штук)
- список зарезервированных ключевых слов на будущее (273 штуки)

Константы T-SQL

Константа, также называемая литералом или скалярным значением, зависит от типа данных.

Символьные строки

Заключаются в одинарные кавычки, например, 'O'Brien'. Если для соединения значение параметра QUOTED_IDENTIFIER было задано как OFF, то символьные константы также могут заключаться в двойные кавычки. Если символьная строка, заключенная в одинарные кавычки, содержит также внедренную одинарную кавычку, то эту кавычку необходимо заключить в дополнительные одинарные кавычки. Данное требование не распространяется на строки, заключенные в двойные кавычки.

Символьные строки в Юникоде

Начинается с идентификатора N, например, N'Русская Редакция'.

Двоичные строки

Начинаются с префикса 0x, за которым следует строка шестнадцатеричных чисел, которая не заключается в кавычки, например, 0x12Ef.

Константы типа bit

Содержат последовательности нулей и единиц и в кавычки не заключаются. Все числа, больше единицы, преобразуются в единицу. Примеры: 0101010101, 000000000.

Константы datetime

Задаются с помощью символьных значений даты специального формата, заключенных в одинарные кавычки.

Константы integer

Состоят из числовых строк, которые не заключаются в кавычки и не содержат десятичного разделителя.

Константы decimal

Состоят из числовых строк, которые не заключаются в кавычки и содержат десятичный разделитель.

Константы типа float и real

Представляются в экспоненциальной форме.

Константы **money**

Состоят из числовых строк, которые не заключаются в кавычки, могут содержать десятичный разделитель и префикс в виде знака валюты.

Константы **uniqueidentifier**

Состоят из строки, представляющей идентификатор GUID. Могут указываться с помощью символьного или двоичного символьного формата.

Константы **xml**

Скалярные выражения T-SQL

Сочетание operandов и операторов, используемое компонентом SQL Server для получения одиночного значения данных. В простейшем случае выражение может быть:

- литералом (константой);
- функцией;
- именем столбца;
- переменной;
- вложенным запросом;
- функцией CASE.

В выражениях символы и значения типа datetime необходимо заключать в одинарные кавычки. Символьные константы, используемые в качестве шаблона для предложения LIKE, должны быть заключены в одинарные кавычки.

```
expression ::=  
{      constant |  
      scalar_function |  
      [ table_name. ] column |  
      variable |  
      ( expression ) |  
      ( scalar_subquery ) |  
      { unary_operator } expression |  
      expression { binary_operator } expression |  
      ranking_windowed_function |  
      aggregate_windowed_function  
}
```

Категории операторов T-SQL

Категория	Операторы
Арифметические операторы	+, -, *, /, %
Логические операторы	ALL, AND, ANY, BETWEEN, EXISTS, IN, LIKE, NOT, OR, SOME
Оператор присваивания	=
Оператор разрешения области	:: (обеспечивает доступ к статическим элементам составного типа данных)
Битовые операторы	&, , ^
Операторы наборов	EXCEPT, INTERSECT, UNION
Операторы сравнения	=, >, <, >=, <=, <>, !=, !<, !>
Оператор объединения строк	+(сцепление строк)
Составные операторы	+=, -=, *=, /=, %=, &=, ^=, =
Унарные операторы	+, -, ~

Приоритет операторов T-SQL

Уровень	Операторы
1	~
2	*, /, %
3	+ (положительное), - (отрицательное), + (сложение), (+ сцепление), - (вычитание), &, ^,
4	=, >, <, >=, <=, <>, !=, !<, !>
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	= (присваивание)

Замечания.

1. Если два оператора в выражении имеют один и тот же уровень старшинства, они выполняются в порядке слева направо по мере их появления в выражении.
2. Чтобы изменить приоритет операторов в выражении, следует использовать скобки.

Приоритет типов данных T-SQL

Если оператор связывает два выражения различных типов данных, то по правилам приоритета типов данных определяется, какой тип данных имеет меньший приоритет и будет преобразован в тип данных с большим приоритетом. Если неявное преобразование не поддерживается, возвращается ошибка. Если оба операнда выражения имеют одинаковый тип данных, результат операции будет иметь тот же тип данных.

В SQL Server используется следующий приоритет типов данных:

1. UDT (высший приоритет);
2. sql_variant;
3. xml;
4. datetimeoffset;
5. datetime2;
6. datetime;
7. smalldatetime;
8. date;
9. time;
10. float;
11. real;
12. decimal;
13. money;
14. smallmoney;
15. bigint;
16. int;
17. smallint;
18. tinyint;
19. bit;
20. ntext;
21. text;
22. image;
23. timestamp;
24. uniqueidentifier;
25. nvarchar (включая nvarchar(max));
26. nchar;
27. varchar (включая varchar(max));
28. char;
29. varbinary (включая varbinary(max));
30. binary (низший приоритет).

Приведение и преобразование типов данных

Следующие функции поддерживают приведение и преобразование типов данных:

- CAST и CONVERT
- PARSE
- TRY_CAST
- TRY_CONVERT
- TRY_PARSE

Использование функций CAST и CONVERT для преобразования выражений одного типа в другой

CAST (*выражение AS целевой_тип_данных [(длина)]*)
CONVERT (*целевой_тип_данных [(длина)] , выражение [, стиль]*)

Пример 1. Получить названия деталей и их цены для тех деталей, у которых первая цифра цены по прейскуранту – 2.

```
SELECT PName, Price
```

```
FROM P
WHERE CAST(Price AS int) LIKE '2%';
```

Можно и так

```
SELECT PName, Price
FROM P
WHERE CAST(CAST(Price AS int) AS varchar(10)) LIKE '2%';
```

Пример 2. Объединение несимвольных недвоичных выражений с помощью функции CAST.

```
SELECT ' Цена детали '+Pname+ ' по прейскуранту составляет '+CAST(Price AS varchar(12)) AS [Цена]
FROM P
WHERE Price BETWEEN 10.00 AND 20.00;
```

Пример 3. Показать текущую дату и время, используя функцию CAST для изменения текущей даты и времени в символьный тип данных и затем использовать CONVERT для отображения даты и времени в формате ISO 8901.

```
SELECT
    GETDATE() AS [До преобразования],
    CAST(GETDATE() AS nvarchar(30)) AS [Используя Cast],
    CONVERT(nvarchar(30), GETDATE(), 126) AS [Используя Convert в ISO8601] ;
```

```
2010-11-24 00:33:45.403 Nov 24 2010 12:33AM      2010-11-24T00:33:45.403
```

Пример 4. Частичная противоположность предыдущему примеру. Отобразить дату и время в виде символьных данных, использует функцию CAST для изменения символьных данных в тип данных datetime, а затем использует CONVERT для изменения символьных данных в тип данных datetime.

```
SELECT
    '2006-04-25T15:50:59.997' AS UnconvertedText,
    CAST('2006-04-25T15:50:59.997' AS datetime) AS UsingCast,
    CONVERT(datetime, '2006-04-25T15:50:59.997', 126) AS UsingConvertFrom_ISO8601 ;
```

```
2006-04-25T15:50:59.997 2006-04-25 15:50:59.997 2006-04-25 15:50:59.9970
```

Условие поиска T-SQL

Сочетание одного или нескольких предикатов, в котором используются логические операторы AND, OR и NOT. Параметр *search_condition* в инструкции DELETE, MERGE, SELECT или UPDATE указывает, сколько строк возвращается или обрабатывается инструкцией.

```
<search_condition> ::=  
{ [ NOT ] <predicate> | ( <search_condition> ) } [ { AND | OR } [ NOT ] { <predicate> | ( <search_condition> ) } ]  
[ ,...n ]  
  
<predicate> ::=  
{ expression { = | <> | != | > | >= | !> | < | <= | !< } expression  
| match_expression [ NOT ] LIKE pattern [ ESCAPE 'escape_character' ]  
| expression [ NOT ] BETWEEN expression AND expression  
| expression IS [ NOT ] NULL  
| CONTAINS ( { column | * } , '<contains_search_condition>' )  
| FREETEXT ( { column | * } , 'freetext_string' )  
| test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )  
| expression { = | <> | != | > | >= | !> | < | <= | !< } { ALL | SOME | ANY } ( subquery )  
| EXISTS ( subquery )  
}
```

Комментарий.

- 1) *expression* – скалярное выражение.
- 2) *match_expression* – любое допустимое выражение символьного типа данных.

- 3) *pattern* – конкретная строка символов для поиска в *match_expression*, которая может содержать следующие допустимые символы-шаблоны: %, _, [], [^]. Длина значения *pattern* не может превышать 8000 байт.
- 4) *escape_character* – символ, помещаемый перед символом-шаблоном, чтобы символ-шаблон рассматривался как обычный символ, а не как шаблон.
- 5) CONTAINS – осуществляет поиск столбцов, содержащих символьные данные с заданной точностью (*fuzzy*), соответствующие заданным отдельным словам и фразам на основе похожести словам и точному расстоянию между словами, взвешенному совпадению. Этот параметр может быть использован только в инструкции SELECT.
- 6) FREETEXT – предоставляет простую форму естественного языка ввода запросов на осуществление поиска столбцов, содержащих символьные данные, совпадающие с содержанием предиката не точно, а по смыслу. Этот параметр может быть использован только в инструкции SELECT Список значений необходимо заключать в скобки.
- 7) *subquery* – может рассматриваться как ограниченная инструкция SELECT и являющаяся подобной на *query_expression* в инструкции SELECT. Использование предложений ORDER BY, COMPUTE и ключевого слова INTO не допускается.

Архитектура SQL Server

SQL Server 2014 содержит следующие технологии управления и анализа данных:

- Database Engine - основная служба (реляционное ядро) для хранения, обработки и обеспечения безопасности данных.
- Data Quality Services - построение базы знаний и ее использование для выполнения разнообразных задач по обеспечению качества данных, включая исправление, дополнение, стандартизацию и устранение дубликатов данных.
- Analysis Services - многомерные данные и интеллектуальный анализ данных и совместная работа с PowerPivot, Excel и SharePoint. Для обнаружения в данных закономерностей и тенденций можно применять сочетание этих функций и средств, а затем использовать найденные закономерности и тенденции для принятия обоснованных решений в отношении сложных бизнес-задач.
- Integration Services - интеграцию и преобразование данных, например, экспорт-импорт данных.
- Master Data Services - управления основными данными. Решение, построенное на основе Master Data Services, позволяет обеспечить правильность информации, используемой для построения отчетов и выполнения анализа. С помощью Master Data Services можно создать центральный репозиторий основных данных и поддерживать запись этих данных по мере их изменения, защищенную и доступную для аудита.
- Replication - копирование и распространение данных и объектов баз данных между базами данных, а также синхронизации баз данных для поддержания согласованности.
- Reporting Services - создания отчетов с поддержкой веб-интерфейса

Ключевые компоненты Database Engine:

Query Optimizer (см. Query Optimizer Deep Dive - Part 1..Part 4)

http://sqlblog.com/blogs/paul_white/archive/2012/04/28/query-optimizer-deep-dive-part-1.aspx

Performance|Improving .NET Application Performance and Scalability (Chapter 1.. Chapter 17)

<http://msdn.microsoft.com/en-us/library/ff647813.aspx>

Логические операторы описывают операции реляционной алгебры, используемые для обработки инструкции.

Физические операторы реализуют действия, описанные логическими операторами. Каждый физический оператор является объектом или процедурой, выполняющей операцию.

Оптимизатор запросов использует операторы для построения плана запроса. План запроса — это дерево физических операторов. Можно просмотреть план запроса с помощью инструкций SET SHOWPLAN.

Логические операторы описывают операции реляционной алгебры, используемые для обработки инструкции.

Физические операторы реализуют действия, описанные логическими операторами. Каждый физический оператор является объектом или процедурой, выполняющей операцию.

Справочник по логическим и физическим операторам Showplan (104 штуки)

<http://msdn.microsoft.com/ru-ru/library/ms191158.aspx>

Инструкции языка описания данных (Инструкции DDL)

В реляционной модели постулируется, что основными структурами данных являются домены и нормализованные «пларные» отношения, т.е. таблицы.

Создание, изменение и уничтожение доменов

В стандартном SQL домен является именованным объектом схемы базы данных. Домены можно создавать (CREATE), изменять (ALTER) и ликвидировать (DROP). Имена доменов можно использовать при определении столбцов таблиц. Можно считать, что в SQL определение домена представляет собой вынесенное за пределы определения индивидуальной таблицы «родовое» определение столбца, которое можно использовать для определения различных реальных столбцов реальных базовых таблиц.

Определение домена

Для определения домена в SQL используется оператор CREATE DOMAIN. Общий синтаксис этого оператора следующий:

```
определение_домена ::= CREATE DOMAIN имя_домена [AS] тип_данных
    [ значение_по_умолчанию ]
    [ список_ограничений_целостности ]
```

Раздел *значение_по_умолчанию* имеет вид:

```
DEFAULT { литерал | нульместная_функция | NULL }
```

где нульместная_функция может задаваться в одной из следующих форм:

```
USER
CURRENT_USER
SESSION_USER
SYSTEM_USER
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
```

Элемент списка *список_ограничений_целостности* имеет вид

```
[CONSTRAINT имя_ограничения] CHECK (условное_выражение)
```

Наиболее естественным (и наиболее распространенным) видом ограничения домена является следующий:

```
CHECK (VALUE IN (список_допустимых_значений))
```

Примеры определений доменов

```
CREATE DOMAIN EMP_NO AS INTEGER
    CHECK (VALUE BETWEEN 1 AND 10000);

CREATE DOMAIN SALARY AS NUMERIC (10, 2)
    DEFAULT 10000.00
    CHECK (VALUE BETWEEN 10000.00 AND 20000000.00)
    CONSTRAINT SAL_NOT_NULL CHECK (VALUE IS NOT NULL);
```

Замечание. Transact-SQL не поддерживает концепции домена. При попытке выполнить выше приведенные инструкции будет получено следующее сообщение:

```
Msg 343, Level 15, State 1, Line 1
Unknown object type 'DOMAIN' used in a CREATE, DROP, or ALTER statement.
```

Явные преобразования типов или доменов и оператор CAST

Неявные преобразования типов недостаточно гибки и иногда могут вызывать ошибки. Число допустимых неявных преобразований типов в SQL весьма ограничено. Однако в SQL существует специальный оператор CAST, с помощью которого можно явно преобразовывать типы или домены в более широких пределах допускаемых преобразований. Конструкция имеет следующий синтаксис:

```
CAST ({скалярное выражение | NULL } AS {тип_данных | имя_домена})
```

Примеры явного преобразования типа

A. Простое использование функций CAST

```
CAST ([Цена] AS int)
```

B. Использование функции CAST с арифметическими операторами

В следующем примере вычисляется столбец значений путем деления суммарных продаж за год на проценты комиссионных . Результат преобразуется к типу данных int после округления до ближайшего целого числа.

```
CAST(ROUND([Суммарные продажи за год]/[Проценты комиссионных], 0) AS int)
```

C. Использование функции CAST для склеивания строк

```
'The list price is ' + CAST(ListPrice AS varchar(12))
```

D. Использование функций CAST с типом данных datetime

```
CAST(GETDATE() AS nvarchar(30))
```

E. Использование функции CAST с временной переменной @myval

```
DECLARE @myval decimal (5, 2)
SET @myval = 193.57
SELECT CAST(CAST(@myval AS varbinary(20)) AS decimal(10,5))
```

Замечание. Transact-SQL поддерживает преобразование CAST.

Создание, изменение и удаление базовых таблиц

Базовая таблица SQL-ориентированной базы данных является прямым аналогом переменной отношения реляционной модели данных. Базовые (реально хранимые в базе данных) таблицы создаются (определяются) с использованием оператора CREATE TABLE. Для изменения определения базовой таблицы применяется оператор ALTER TABLE. Уничтожить хранимую таблицу (отменить ее определение) можно с помощью оператора DROP TABLE.

Определение базовой таблицы

Оператор создания базовой таблицы CREATE TABLE имеет следующий синтаксис:

определение_базовой_таблицы ::=
CREATE TABLE *имя_базовой_таблицы* (*список_элементов_базовой_таблицы*)

элемент_базовой_таблицы ::=
определение_столбца | *определение_ограничения_базовой_таблицы*

определение_столбца ::=
имя_столбца
{ *тип_данных* | *имя_домена* }
[*значение_по_умолчанию*]
[*список_определений_ограничений_столбца*]

значение_по_умолчанию ::=

DEFAULT { *литерал* | *нульместная_функция* | NULL }

определение_ограничения_столбца ::=
[CONSTRAINT *имя_ограничения*]
NOT NULL |
{ PRIMARY KEY | UNIQUE } |
ограничение_ссылочной_целостности |
CHECK (*условное_выражение*)

ограничение_ссылочной_целостности ::=
REFERENCES *имя_базовой_таблицы* [(*список_столбцов*)]
[ON DELETE *ссылочное_действие*]
[ON UPDATE *ссылочное_действие*]

Последняя синтаксическая конструкция работает и в случае определения внешнего ключа на уровне таблицы (в одном из определений табличных ограничений целостности). Поэтому обсуждение этого вопроса будет отложено до рассмотрения общего случая.

определение_ограничения_базовой_таблицы задается в следующем синтаксисе:

определение_ограничения_базовой_таблицы ::=
[CONSTRAINT *имя_ограничения*]
{ PRIMARY KEY | UNIQUE } (*список_столбцов*) |
FOREIGN KEY (*список_столбцов*) *ограничение_ссылочной_целостности* |
CHECK (*условное_выражение*)

Поддержка ссылочной целостности и ссылочные действия

В связи с определением ограничения внешнего ключа осталось рассмотреть еще два необязательных раздела – ON DELETE *ссылочное_действие* и ON UPDATE *ссылочное_действие*. Прежде всего, приведем синтаксическое правило:

ссылочное_действие ::=
{ NO ACTION | CASCADE | SET DEFAULT | SET NULL }

Чтобы объяснить, в каких случаях и каким образом выполняются эти действия, требуется сначала определить понятие ссылающейся строки (referencing row). Для данной строки t таблицы T строкой таблицы S, ссылающейся на строку t, называется каждая строка таблицы S, значение внешнего ключа которой совпадает со значением соответствующего потенциального ключа строки t.

Пусть определение ограничения внешнего ключа содержит раздел ON DELETE. Предположим, что предпринимается попытка удалить строку t из таблицы T. Тогда:

- 1) если в качестве требуемого ссылочного действия указано NO ACTION, то операция удаления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа;
- 2) если в качестве требуемого ссылочного действия указано CASCADE, то строка t удаляется, и удаляются все строки, ссылающиеся на t;
- 3) если в качестве требуемого ссылочного действия указано SET DEFAULT, то строка t удаляется, и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку t, проставляется заданное при их определении значение по умолчанию;
- 4) если в качестве требуемого ссылочного действия указано SET NULL, то строка t удаляется, и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку t, проставляется NULL

Пусть определение ограничения внешнего ключа содержит раздел ON UPDATE. Предположим, что предпринимается попытка обновить столбцы соответствующего возможного ключа в строке t из таблицы T. Тогда:

- если в качестве требуемого ссылочного действия указано NO ACTION или RESTRICT, то операция обновления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа;
- если в качестве требуемого ссылочного действия указано CASCADE, то строка t обновляется и соответствующим образом обновляются все строки, ссылающиеся на t (в них должным образом изменяются значения столбцов, входящих в состав внешнего ключа);

- если в качестве требуемого ссылочного действия указано SET DEFAULT, то строка t обновляется, и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы T, всех строк, ссылающихся на строку t, проставляется заданное при их определении значение по умолчанию;
- если в качестве требуемого ссылочного действия указано SET NULL, то строка t обновляется, и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы T, всех строк, ссылающихся на строку t, проставляется NULL.

Классификация ограничений

A. Ограничения 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.

B. Ограничения PRIMARY KEY

- 1) В таблице возможно наличие только одного ограничения по первичному ключу.
- 2) Индекс, формируемый ограничением PRIMARY KEY, не может привести к выходу количества индексов в таблице за пределы в 999 некластеризованных индексов и 1 кластеризованный.
- 3) Если для ограничения PRIMARY KEY не указан параметр CLUSTERED или NONCLUSTERED, применяется параметр CLUSTERED, если для ограничения UNIQUE не определено кластеризованных индексов.
- 4) Все столбцы с ограничением PRIMARY KEY должны иметь признак NOT NULL. Если допустимость значения NULL не указана, то для всех столбцов с ограничением PRIMARY KEY устанавливается признак NOT NULL.
- 5) Если первичный ключ определен на столбце определяемого пользователем типа данных CLR, реализация этого типа должна поддерживать двоичную сортировку.

C. Ограничения UNIQUE

- 1) Если для ограничения UNIQUE не указан параметр CLUSTERED или NONCLUSTERED, по умолчанию применяется параметр NONCLUSTERED.
- 2) Каждое ограничение уникальности создает индекс. Количество ограничений UNIQUE не может привести к выходу количества индексов в таблице за пределы в 999 некластеризованных индексов и 1 кластеризованный.

D. Ограничения FOREIGN KEY

- 1) Если столбцу, имеющему ограничение внешнего ключа, задается значение, отличное от NULL, такое же значение должно существовать и в указываемом столбце; в противном случае будет возвращено сообщение о нарушении внешнего ключа.
- 2) Если не указаны исходные столбцы, ограничения FOREIGN KEY применяются к предшествующему столбцу.
- 3) Ограничения FOREIGN KEY могут ссылаться только на таблицы в пределах той же базы данных на том же сервере. Межбазовую ссылочную целостность необходимо реализовать посредством триггеров.
- 4) Ограничения FOREIGN KEY могут ссылаться на другие столбцы той же таблицы. Это называется самовызовом.
- 5) Предложение REFERENCES ограничения внешнего ключа на уровне столбца может содержать только один ссылочный столбец. Этот столбец должен принадлежать к тому же типу данных, что и столбец, для которого определяется ограничение.
- 6) Предложение REFERENCES ограничения внешнего ключа на уровне таблицы должно содержать такое же число ссылочных столбцов, какое содержится в списке столбцов в ограничении. Тип данных каждого ссылочного столбца должен также совпадать с типом соответствующего столбца в списке столбцов.
- 7) Если частью внешнего или ссылочного ключа является столбец типа **timestamp**, ключевые слова CASCADE, SET NULL и SET DEFAULT указывать нельзя.
- 8) Ключевые слова CASCADE, SET NULL, SET DEFAULT и NO ACTION можно сочетать в таблицах, имеющих взаимные ссылочные связи. Если компонент Database Engine обнаруживает ключевое слово NO ACTION, оно остановит и произведет откат связанных операций CASCADE, SET NULL и SET DEFAULT. Если инструкция DELETE содержит сочетание ключевых слов CASCADE, SET NULL, SET DEFAULT и NO ACTION, то все операции CASCADE, SET NULL и SET DEFAULT выполняются перед поиском компонентом Database Engine операции NO ACTION.
- 9) Компонент Компонент Database Engine не имеет стандартного предела на количество ограничений FOREIGN KEY, содержащихся в таблице, ссылающейся на другие таблицы, или на количество ограничений FOREIGN KEY в других таблицах, ссылающихся на указанную таблицу. Тем не менее фактическое количество ограничений FOREIGN KEY, доступных для использования, ограничивается конфигурацией оборудования, 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. Предел эффективности в конкретном случае может более или менее зависеть от приложения и оборудования. При разработке базы данных и приложений следует учитывать стоимость принудительных ограничений FOREIGN KEY.
- 10) Ограничения FOREIGN KEY не применяются к временным таблицам.
- 11) Ограничения FOREIGN KEY могут ссылаться только на столбцы с ограничениями PRIMARY KEY или UNIQUE в таблице, на которую указывает ссылка, или на столбцы уникального индекса (UNIQUE INDEX) такой таблицы.

- 12) Если внешний ключ определен на столбце определяемого пользователем типа данных CLR, реализация этого типа должна поддерживать двоичную сортировку.
- 13) Столбцы, участвующие в связи по внешнему ключу, должны иметь одинаковую длину и масштаб.

E. Ограничения DEFAULT

- 1) Столбец может иметь только одно определение DEFAULT.
- 2) Ограничение DEFAULT может содержать значения констант, функции, функции без параметров SQL-92 или значение NULL.
- 3) Значение *константное выражение* в определении DEFAULT не может ссылаться на другой столбец таблицы, а также на другие таблицы, представления или хранимые процедуры.
- 4) Определения DEFAULT нельзя создавать для столбцов с типом данных **timestamp** или столбцов со свойством **IDENTITY**.
- 5) Определения DEFAULT нельзя создавать для столбцов с псевдонимами типов данных, если такой тип привязан к определенному по умолчанию объекту.

F. Ограничения CHECK

- 1) Столбец может содержать любое количество ограничений CHECK, а условие может включать несколько логических выражений, соединенных операторами AND и OR. При указании нескольких ограничений CHECK для столбца их проверка производится в порядке создания.
- 2) Условие поиска должно возвращать логическое выражение и не может ссылаться на другую таблицу.
- 3) Ограничение CHECK уровня столбца может ссылаться только на ограничиваемый столбец, а ограничение CHECK уровня таблицы — только на столбцы этой таблицы.
- 4) Правила и ограничения CHECK выполняют одну и ту же функцию проверки данных при выполнении инструкций **INSERT** и **UPDATE**.
- 5) Если для столбца или столбцов задано правило либо одно или несколько ограничений CHECK, применяются все ограничения.
- 6) Ограничения CHECK нельзя определять для столбцов типов данных **text**, **ntext** или **image**.

Примеры определений базовых таблиц

```

CREATE TABLE EMPLOYEE
(
    ID SMALLINT NOT NULL,
    NAME VARCHAR(9),
    DEPT SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
    JOB CHAR(5) CHECK (JOB IN ('Sales', 'Mgr', 'Clerk')),
    HIREDATE DATE,
    SALARY DECIMAL(7, 2),
    COMM DECIMAL(7, 2),
    PRIMARY KEY (ID),
    CONSTRAINT YEARSAL CHECK (YEAR(HIREDATE) > 1986 OR SALARY > 40500)
);

CREATE TABLE [dbo].[Shipping_Methods]
(
    [ShippingMethodID] [int] IDENTITY (1, 1) NOT NULL ,
    [ShippingMethod] [varchar] (25) COLLATE SQL_Latin1_General_CI_AS NOT NULL
);

CREATE TABLE DEPT
(
    DEPTNO SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 500, INCREMENT BY 1),
    DEPTNAME VARCHAR (36) NOT NULL,
    MGRNO CHAR(6),
    ADMRDEPT SMALLINT NOT NULL,
    LOCATION CHAR(30)
);

create table Video
(
    VideoID char(5) constraint pk_Video primary key,
    MovID char(5) constraint fk_VideoMovie foreign key references Movie(MovID),
    Format char(4) not null,

```

```
Price money not null  
);
```

Изменение определения базовой таблицы

Оператор изменения определения базовой таблицы ALTER TABLE имеет следующий синтаксис:

```
изменение_базовой_таблицы ::=  
    ALTER TABLE имя_базовой_аблицы {  
        действие_по_изменению_определения_столбца  
        | действие_по_изменению_определения_табличного_ограничения }
```

Как видно из этого синтаксического правила, при выполнении одного оператора ALTER TABLE может быть выполнено либо действие по изменению определения столбца, либо действие по изменению определения табличного ограничения целостности.

Добавление, изменение или удаление определения столбца

Действие по изменению определения столбца специфицируется в следующем синтаксисе:

```
действие_по_изменению_определения_столбца ::=  
    ADD [ COLUMN ] определение_столбца  
    | ALTER [ COLUMN ] имя_столбца { SET значение_по_умолчанию | DROP DEFAULT }  
    | DROP [ COLUMN ] имя_столбца { RESTRICT | CASCADE }
```

Действие DROP COLUMN отменяет *определение* существующего столбца (удаляет его из таблицы). Действие DROP COLUMN отвергается, если:

- (a) указанный столбец является единственным столбцом таблицы;
- (b) или в этом действии присутствует спецификация RESTRICT, и данный столбец используется в определении каких-либо представлений или ограничений целостности.

Если в действии присутствует спецификация CASCADE, то его выполнение порождает неявное выполнение оператора DROP для всех представлений и ограничений целостности, в определении которых используется данный столбец.

Примеры изменения определения столбца

A. Добавление к таблице EMP нового столбца EMP_BONUS:

```
ALTER TABLE EMP  
    ADD EMP_BONUS SALARY DEFAULT NULL  
    CONSTRAINT BONSAL CHECK (VALUE < EMP_SAL);
```

B. При *определении столбца* EMP_SAL таблицы EMP для этого столбца явно не определялось *значение по умолчанию* (оно наследовалось из определения домена). Если в какой-то момент это стало неправильным (например, повысился размер минимальной зарплаты), можно установить новое *значение по умолчанию*:

```
ALTER TABLE EMP ALTER EMP_SAL SET DEFAULT 15000.00
```

C. При *определении столбца* DEPT_TOTAL_SAL таблицы DEPT для него было установлено *значение по умолчанию* 1000000. Можно отменить это *значение по умолчанию*:

```
ALTER TABLE DEPT ALTER DEPT_TOTAL_SAL DROP DEFAULT
```

D. Может оказаться разумным отменить *определение столбца* DEPT_EMP_NO, выполнив следующий оператор ALTER TABLE:

```
ALTER TABLE DEPT DROP DEPT_EMP_NO CASCADE
```

Спецификация CASCADE ведет к тому, что при выполнении оператора будет уничтожено не только *определение* указанного столбца, но и определения всех ограничений целостности и представлений, в которых используется уничтожаемый столбец. В нашем случае единственное связанное с этим столбцом ограничение целостности, определенное вне *определения столбца*, было бы отменено, даже если бы в операторе *отмены определения столбца* DEPT_EMP_NO содержалась спецификация RESTRICT, поскольку это единственное внешнее определение ограничения является ограничением только столбца DEPT_EMP_NO.

Изменение набора табличных ограничений

Действие по изменению набора табличных ограничений специфицируется в следующем синтаксисе:

действие_no_изменению_табличного_ограничения ::=
ADD [CONSTRAINT] *определение_ограничения базовой таблицы* |
DROP CONSTRAINT *имя_ограничения* { RESTRICT | CASCADE }

Действие ADD [CONSTRAINT] позволяет добавить к набору существующих ограничений таблицы новое ограничение целостности. Можно считать, что новое ограничение добавляется через AND к конъюнкции существующих ограничений, как если бы оно определялось в составе оператора *CREATE TABLE*. Но здесь имеется одно существенное отличие. Если внимательно посмотреть на все возможные виды табличных ограничений, можно убедиться, что любое из них удовлетворяется на пустой таблице. Поэтому, какой бы набор табличных ограничений ни был определен при создании таблицы, это определение является допустимым и не препятствует выполнению оператора *CREATE TABLE*. При добавлении нового табличного ограничения с использованием действия ADD [CONSTRAINT] мы имеем другую ситуацию, поскольку таблица, скорее всего, уже содержит некоторый набор строк, для которого условное выражение нового ограничения может принять значение false. В этом случае выполнение оператора *ALTER TABLE*, включающего действие ADD [CONSTRAINT], отвергается.

Выполнение действия DROP CONSTRAINT приводит к отмене *определения* существующего табличного ограничения. Можно отменить определение только именованных табличных ограничений. Спецификации RESTRICT и CASCADE осмыслены только в том случае, если отменяемое ограничение является *ограничением потенциального ключа* (UNIQUE или PRIMARY KEY). При указании RESTRICT действие отвергается, если на данный возможный ключ ссылается хотя бы один внешний ключ. При указании CASCADE действие DROP CONSTRAINT выполняется в любом случае, и все определения таких внешних ключей также отменяются.

Примеры изменения набора табличных ограничений

А. Добавление к набору ограничений таблицы DEPT нового ограничения:

```
ALTER TABLE DEPT ADD CONSTRAINT TOTAL_INCOME  
CHECK (DEPT_TOTAL_SAL >= (SELECT SUM(EMP_SAL + COALESCE(EMP_BONUS, 0))  
FROM EMP WHERE EMP.DEPT_NO = DEPT_NO))
```

Смысл этого ограничения следующий: суммарный доход служащих отдела не должен превышать объем зарплаты отдела.

В. В арифметическом выражении под знаком агрегатной операции SUM используется операция COALESCE. Эта двуместная операция определяется следующим образом:

```
COALESCE (x, y) IF x IS NOT NULL THEN x ELSE y
```

С. При определении таблицы EMP было специфицировано *проверочное табличное ограничение* PRO_EMP_NO, устанавливающее, что над одним проектом не должно работать более 50 служащих. Для отмены ограничения нужно выполнить следующий оператор:

```
ALTER TABLE EMP DROP CONSTRAINT PRO_EMP_NO
```

Отмена определения (уничтожение) базовой таблицы

Для отмены определения (уничтожения) базовой таблицы служит оператор *DROP TABLE*, задаваемый в следующем синтаксисе:

DROP TABLE имя_базовой_таблицы { RESTRICT | CASCADE }

При наличии спецификации RESTRICT выполнение оператора *DROP TABLE* отвергается, если имя таблицы используется в каком-либо определении представления или ограничения целостности. При наличии спецификации CASCADE оператор выполняется в любом случае, и все определения представлений и ограничений целостности, содержащие ссылки на данную таблицу, также отменяются.

Инструкции языка обработки данных (Инструкции DML)

Инструкции DML предназначены для добавления данных, изменения данных, запроса данных и удаления данных из базы данных. К числу основных инструкций языка DML относятся: SELECT, INSERT, UPDATE, DELETE, MERGE, BULK INSERT.

Извлечение данных: инструкция SELECT

Инструкция SELECT извлекает строки из базы данных и позволяет делать выборку одной или нескольких строк или столбцов из одной или нескольких таблиц. Полный синтаксис инструкции SELECT сложен, однако основные предложения можно вкратце описать следующим образом:

```
[ WITH общее_табличное_выражение ]
SELECT [ DISTINCT | ALL ] [ TOP выражение [ PERCENT ] ] { * | список_выбора }
[ INTO новая_таблица ]
[ FROM список_табличных_источников ]
[ WHERE условие_поиска ]
[ GROUP BY group_by_выражение ]
[ HAVING условие_поиска ]
[ ORDER BY order_by_выражение [ ASC | DESC ] ]
```

Порядок предложений в инструкции SELECT имеет значение. Любое из необязательных предложений может быть опущено; но если необязательные предложения используются, они должны следовать в определенном порядке. При обработке инструкции SELECT составляющие ее предложения выполняются в следующем порядке:

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. HAVING
7. SELECT
8. DISTINCT
9. ORDER BY
10. TOP

Предложение FROM указывает таблицу, представление или источник производной таблицы с указанием или без указания псевдонима для использования в инструкции SQL. В инструкции можно использовать до 256 источников таблиц, хотя предел изменяется в зависимости от доступной памяти и сложности других выражений в запросе. Отдельные запросы могут не поддерживать 256 источников таблиц. В качестве источника таблицы может быть указана переменная table.

Примечание. Производительность выполнения запросов может снизиться из-за большого количества таблиц, указанных в запросе. На время компиляции и оптимизации также влияют дополнительные факторы. Они включают в себя наличие индексов и индексированных представлений в каждом табличном источнике и размер списка выбора в инструкции SELECT.

Важным случаем табличного источника является *joined_таблица* - результирующий набор, полученный из двух или более таблиц. Для множественных соединений следует использовать скобки, чтобы изменить естественный порядок соединений.

joined_таблица ::=
левая_таблица CROSS JOIN правая_таблица |
левая_таблица [NATURAL] [INNER | { LEFT | RIGHT | FULL [OUTER] } | UNION] JOIN правая_таблица [ON
условное_выражение |
(joined_таблица)

CROSS JOIN

Указывает произведение двух таблиц. Возвращает те же строки, что и соединение без предложения WHERE в старом варианте SQL-92.

JOIN

Указывает, что данная операция соединения должна произойти между указанными источниками или представлениями таблицы.

INNER

Указывает, что возвращаются все совпадающие пары строк. Несовпадающие строки из обеих таблиц отбрасываются. Если тип соединения не указан, этот тип задается по умолчанию.

FULL [OUTER]

Указывает, что в результирующий набор включаются строки как из левой, так и из правой таблицы, несоответствующие условиям соединения, а выходные столбцы, соответствующие оставшейся таблице, устанавливаются в значение NULL. Этим дополняются все строки, обычно возвращаемые при помощи INNER JOIN.

LEFT [OUTER]

Указывает, что все строки из левой таблицы, не соответствующие условиям соединения, включаются в результирующий набор, а выходные столбцы из оставшейся таблицы устанавливаются в значение NULL в дополнение ко всем строкам, возвращаемым внутренним соединением.

RIGHT [OUTER]

Указывает, что все строки из правой таблицы, не соответствующие условиям соединения, включаются в результирующий набор, а выходные столбцы, соответствующие оставшейся таблице, устанавливаются в значение NULL в дополнение ко всем строкам, возвращаемым внутренним соединением.

ON *условие_поиска*

Задает условие, на котором основывается соединение. Условие может указывать любой предикат, хотя чаще используются столбцы и операторы сравнения, например.

Предложение WHERE определяет условия поиска строк, возвращаемых запросом. Количество предикатов, которое может содержать условие поиска, неограниченно. Дополнительные сведения об условиях поиска и предикатах см. в разделе 5. «Скалярные выражения».

Предложение GROUP BY группирует выбранный набор строк для получения набора сводных строк по значениям одного или нескольких столбцов или выражений. Дополнительные сведения о предложении GROUP BY см. в разделе 14. «GROUP BY запросы».

Предложение HAVING определяет условие поиска для группы. Дополнительные сведения о предложении HAVING см. в разделе 14. «GROUP BY запросы».

Выражение SELECT указывает столбцы, возвращаемые запросом. Выражение SELECT может содержать следующие аргументы:

ALL

Указывает, что в результирующем наборе могут появиться повторяющиеся строки. ALL является значением по умолчанию.

DISTINCT

Указывает, что в результирующем наборе могут появиться только уникальные строки. Значения NULL считаются равными для ключевого слова DISTINCT.

TOP (*выражение*) [PERCENT]

Указывает на то, что только заданное число или процент строк будет возвращен из результирующего набора запроса. Аргумент *выражение* может быть либо числом, либо процентом строк. В целях обратной совместимости использование TOP *выражение* без скобок в инструкциях SELECT поддерживается, но не рекомендуется.

список_выбора

Столбцы, выбираемые для результирующего набора. Список выбора представляет собой серию выражений, отделяемых запятыми. Максимальное число выражений, которое можно задать в списке выбора — 4 096.

элемент_выбора ::=

скалярное_выражение [[AS] имя_столбца] | имя_таблицы . *

В целях избежания неоднозначности ссылок, которые могут возникнуть, если в двух таблицах из предложения FROM содержатся столбцы с одинаковыми именами, следует указывать квалификатор для аргумента *имя_столбца*. Например таблицы SalesOrderHeader и SalesOrderDetail в базе данных AdventureWorks содержат столбцы с именем ModifiedDate. Если в запросе соединяются две таблицы, то данные о дате изменения из таблицы SalesOrderDetail могут быть заданы в списке выбора как SalesOrderDetail.ModifiedDate. Выражение в списке выбора может быть константой, функцией, любым сочетанием имен столбцов, констант и функций, соединенных оператором (операторами) или вложенным запросом.

*

Указывает на то, что все столбцы из всех таблиц и представлений в предложении FROM должны быть возвращены. Столбцы возвращаются таблицей или представлением, как указано в предложении FROM, и в порядке, в котором они находятся в таблице или представлении.

Предложение INTO в инструкции SELECT создает новую таблицу в файловой группе по умолчанию и вставляет в нее результирующие строки из запроса.

Предложение ORDER BY указывает порядок сортировки для столбцов, возвращаемых инструкцией SELECT. Предложение ORDER BY не может применяться в представлениях, встроенных функциях, производных таблицах и вложенных запросах, если не указано предложение TOP. Предложение ORDER BY может содержать следующие аргументы:

order_by_выражение

Указывает столбец, по которому должна выполняться сортировка. Столбец сортировки может быть указан с помощью имени или псевдонима столбца или неотрицательного целого числа, представляющего позицию имени или псевдонима в списке выбора. Имена и псевдонимы столбцов могут быть дополнены именем таблицы или представления. В SQL Server уточненные имена и псевдонимы столбцов связываются со столбцами, перечисленными в предложении FROM. Если в выражении *order_by_выражение* отсутствует квалификатор, то значение должно быть уникальным во всех столбцах, перечисленных в инструкции SELECT. Можно указать несколько столбцов сортировки. Последовательность столбцов сортировки в предложении ORDER BY определяет организацию упорядоченного результирующего набора. В предложение ORDER BY могут входить элементы, которых нет в списке выборки. Однако если указана конструкция SELECT DISTINCT, или инструкция содержит предложение GROUP BY, или если инструкция SELECT содержит оператор UNION, то столбцы сортировки должны присутствовать в списке выборки. Кроме того, если в инструкцию SELECT входит оператор UNION, то имена и псевдонимы столбцов должны быть из числа уточненных в первом списке выборки. Столбцы типа ntext, text, image или xml не могут быть использованы в предложении ORDER BY.

COLLATE {collation_имя}

Указывает, что операция ORDER BY должна выполняться в соответствии с параметрами сортировки, указанными в аргументе *collation_имя*, но не в соответствии с параметрами сортировки столбца, определенных в таблице или представлении. Значение *collation_имя* может быть именем параметров сортировки Windows или именем параметров сортировки SQL. Дополнительные сведения см. в MSDN в разделах «Настройка параметров сортировки в программе установки» и «Использование параметров сортировки SQL Server». Аргумент COLLATE применяется только к столбцам данных типа char, varchar, nchar и nvarchar.

ASC

Указывает, что значения в указанном столбце должны сортироваться по возрастанию, от меньших значений к большим значениям.

DESC

Указывает, что значения в указанном столбце должны сортироваться по убыванию, от больших значений к меньшим.

Примечания.

1. Значения NULL рассматриваются как минимально возможные значения.
2. Число элементов в предложении ORDER BY не ограничивается. Однако существует ограничение в 8 060 байт для размера строки промежуточных рабочих таблиц, необходимых для операций сортировки. Это ограничивает общий размер столбцов, указываемый в предложении ORDER BY.

WITH обобщенное_табличное_выражение задает временно именованный результирующий набор, называемый общим табличным выражением (OTB). Он получается при выполнении простого запроса и определяется в области выполнения одиночной инструкции SELECT, INSERT, UPDATE, MERGE или DELETE. Это предложение может использоваться также в инструкции CREATE VIEW как часть определяющей ее инструкции SELECT. Общее табличное выражение может включать ссылки на само себя. Такое выражение называется рекурсивным обобщенным табличным выражением.

Примеры простейших запросов

```
SELECT 2*2 AS "2x2"
SELECT 'Hello, World!' AS Field
SELECT AVG(Qty) AS [Средний размер поставки] FROM SPJ
```

Примеры запросов для демонстрационной базы данных AdventureWorks

A. Использование простого предложения FROM. В следующем примере извлекаются столбцы TerritoryID и Name из таблицы SalesTerritory в образце базы данных AdventureWorks.

```
SELECT TerritoryID, Name
FROM Sales.SalesTerritory
ORDER BY TerritoryID ;
```

B. Использование синтаксиса SQL-92 для CROSS JOIN. В следующем примере возвращается векторное произведение двух таблиц Employee и Department. Возвращается список всех возможных сочетаний строк EmployeeID и все строки имен Department .

```
SELECT e.EmployeeID, d.Name AS Department
FROM HumanResources.Employee e
CROSS JOIN HumanResources.Department d
ORDER BY e.EmployeeID, d.Name;
```

Г. Использование синтаксиса SQL-92 для FULL OUTER JOIN. В следующем примере возвращается имя продукта и любые соответствующие заказы на продажу в таблице SalesOrderDetail. В примере также возвращаются все заказы на продажу, продукты для которых не представлены в таблице Product, и все продукты с заказом на продажу, отличные от тех, которые представлены в таблице Product.

```
-- The OUTER keyword following the FULL keyword is optional.
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
FULL OUTER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID IS NULL
OR sod.ProductID IS NULL
ORDER BY p.Name ;
```

Д. Использование синтаксиса SQL-92 для LEFT OUTER JOIN. Следующий пример соединяет две таблицы по столбцу ProductID и сохраняет несовпадающие строки из левой таблицы. Таблица Product сопоставляется с таблицей SalesOrderDetail по столбцам ProductID в каждой таблице. В результирующем наборе отображаются все продукты, как входящие, так и не входящие в заказы.

```
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
LEFT OUTER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

Е. Использование синтаксиса SQL-92 для INNER JOIN. Следующий пример возвращает все имена продуктов и все идентификаторы заказов на продажу.

```
-- By default, SQL Server performs an INNER JOIN if only the JOIN
-- keyword is specified.
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
INNER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

Ж. Использование синтаксиса SQL-92 для RIGHT OUTER JOIN. Следующий пример соединяет две таблицы по столбцу TerritoryID и сохраняет несовпадающие строки из правой таблицы. Таблица SalesTerritory сопоставляется с таблицей SalesPerson по столбцу TerritoryID каждой таблицы. В результирующем наборе отображаются все представители отдела продаж независимо от того, назначена им или нет обслуживаемая территория.

```
SELECT st.Name AS Territory, sp.SalesPersonID
FROM Sales.SalesTerritory st
RIGHT OUTER JOIN Sales.SalesPerson sp
ON st.TerritoryID = sp.TerritoryID ;
```

И. Использование производной таблицы. Следующий пример использует производную таблицу, инструкцию SELECT после предложения FROM, для возврата имен и фамилий сотрудников и городов, в которых они проживают.

```
SELECT RTRIM(c.FirstName) + ' ' + LTRIM(c.LastName) AS Name, d.City
FROM Person.Contact c INNER JOIN HumanResources.Employee e
ON c.ContactID = e.ContactID
INNER JOIN
(SELECT ea.AddressID, ea.EmployeeID, a.City
FROM Person.Address a INNER JOIN HumanResources.EmployeeAddress ea
ON a.AddressID = ea.AddressID) AS d
ON e.EmployeeID = d.EmployeeID
ORDER BY c.LastName, c.FirstName;
```

Добавление новых строк: инструкция INSERT

Инструкция INSERT добавляет одну или несколько новых строк в таблицу или представление.

Упрощенный синтаксис

инструкция_insert ::=

```
INSERT [INTO] имя_таблицы_или_представления [ ( список_имен_столбцов ) ] выражение_запроса |  
DEFAULT VALUES }
```

Распространенные случаи инструкции INSERT

Случай 1: INSERT *имя_таблицы_или_представления* (*список_имен_столбцов*)
 VALUES (*список-константных-выражений*)

Случай 2: INSERT *имя_таблицы_или_представления*
 SELECT *список_выбора*
 FROM *список_таблиц*
 WHERE *условное_выражение*

Замечание. Значением константного выражения может быть DEFAULT или NULL.

Примеры (для демонстрационной базы данных AdventureWorks2012)

```
INSERT INTO dbo.demoCustomer (CustomerID, FirstName, MiddleName, LastName)  
VALUES (1, 'Orlando', 'N.', 'Gee');  
  
INSERT INTO dbo.demoCustomer (CustomerID, FirstName, MiddleName, LastName)  
VALUES (12, 'Johnny', 'A.', 'Capino'),  
(16, 'Christopher', 'R.', 'Beck'),  
(18, 'David', 'J.', 'Liu');  
  
INSERT INTO dbo.demoCustomer (CustomerID, FirstName, MiddleName, LastName)  
SELECT BusinessEntityID, FirstName, MiddleName, LastName  
FROM Person.Person  
WHERE BusinessEntityID BETWEEN 19 AND 35;
```

Изменение существующих данных: инструкция UPDATE

Инструкция UPDATE изменяет существующие данные в таблице или представлении.

Упрощенный синтаксис

инструкция_update ::=

```
UPDATE имя_таблицы_или_представления  
SET список_set_фраз  
[ WHERE условное_выражение ]
```

set_фраза ::=
 имя_столбца = { *выражение* | NULL | DEFAULT }

Распространенные случаи инструкции UPDATE

Случай 1: UPDATE *имя_таблицы* SET *имя_столбца* = *выражение*
 WHERE *условное_выражение*

Случай 2: UPDATE *имя_таблицы* SET *имя_столбца* = *выражение*
 FROM *имя_таблицы*
 WHERE *условное_выражение*

Примеры (для демонстрационной базы данных pubs)

```

update publishers set city = 'Atlanta', state = 'GA'
update publishers set pub_name = null
update authors set state = 'PC', city = 'Big Bad Bay City'
where state = 'CA' and city = 'Oakland'

update titleauthor set title_id = titles.title_id
from titleauthor, titles, authors
where titles.title = 'The Psychology of Computer Cooking'
and authors.au_id = titleauthor.au_id
and au_lname = 'Stringer'

update titles set ytd_sales = ytd_sales + qty
from titles, sales
where titles.title_id = sales.title_id
and sales.ord_date in (select max(sales.ord_date) from sales)

```

По поводу этих примеров следует сделать следующее пояснение. FROM-предложение в инструкции UPDATE определяет, что для формулирования критериев операции обновления используется таблица, представление или производный источник таблицы. Если обновляемый объект тот же самый, что и объект в предложении FROM, и в предложении FROM имеется только одна ссылка на этот объект, псевдоним объекта указывать не обязательно. Если обновляемый объект встречается в предложении FROM несколько раз, только одна ссылка на этот объект не должна указывать псевдоним таблицы. Все остальные ссылки на объект в предложении FROM должны включать псевдоним объекта. Представление с триггером INSTEAD OF UPDATE не может быть целью инструкции UPDATE с предложением FROM.

Удаление данных: инструкция DELETE

Инструкция DELETE удаляет строки из таблиц и представлений.

Упрощенный синтаксис

```

инструкция delete ::=

    DELETE [FROM] имя_таблицы_или_представления
    [ FROM список_табличных_источников ]
    [ WHERE условное_выражение ]

```

Примеры (для демонстрационной базы данных pubs)

```

delete publishers where pub_name = 'Jardin, Inc.'

delete titles from authors, titles, titleauthor
where titles.title_id = titleauthor.title_id
and authors.au_id = titleauthor.au_id
and city = 'Big Bad Bay City'

```

Для второй инструкции DELETE будет напечатано следующее сообщение:

Сообщение 547, уровень 16, состояние 0, строка 1
Конфликт инструкции DELETE с ограничением REFERENCE "FK__titleauth__title__0BC6C43E". Конфликт произошел в базе данных "pubs", таблица "dbo.titleauthor", column 'title_id'.

Выполнение данной инструкции было прервано.

Инструкции управления потоком (Инструкции FCL)

Ключевыми словами языка управления потоком Transact-SQL являются следующие:

```
BEGIN...END  
BREAK  
CONTINUE  
GOTO  
IF...ELSE  
RETURN  
TRY...CATCH  
WAITFOR  
WHILE
```

В сочетании с инструкциями языка управления потоком могут использоваться и другие конструкции Transact-SQL:

```
CASE  
Комментарии (в стиле Си /* комментарий */ и в стиле Ада -- комментарий)  
DECLARE  
SET  
EXECUTE  
PRINT  
RAISERROR
```

DECLARE

Переменные объявляются в теле пакета или процедуры при помощи инструкции DECLARE, а значения им присваиваются при помощи инструкций SET или SELECT. Т. о., областью локальной переменной является пакет или процедура, в которых она объявлена. После объявления все переменные инициализируются значением NULL. Переменная не может принадлежать к типу данных text, ntext или image.

Синтаксис

```
DECLARE { @локальная_переменная [AS] тип_данных [ = константное_выражение ] } [ ,...n]
```

Примеры

```
DECLARE @find varchar(30) = 'Man%';
```

SET

Устанавливает локальную переменную, предварительно созданную при помощи инструкции DECLARE @локальная_переменная, в указанное значение.

Синтаксис

```
SET @локальная_переменная = выражение
```

Примеры

```
USE AdventureWorksLT;  
GO  
DECLARE @find varchar(30);  
SET @find = 'Man%';  
SELECT LastName, FirstName, Phone  
FROM SalesLT.Customer  
WHERE LastName LIKE @find;
```

LastName	FirstName	Phone
Manchepalli	Ajay	1 (11) 500 555-0174
Manzanares	Tomas	1 (11) 500 555-0178

```
USE AdventureWorksLT;
GO
DECLARE @rows int;
SET @rows = (SELECT COUNT(*) FROM SalesLT.Customer);
SELECT @rows AS N'Число строк';
```

Число строк
440

EXECUTE

Выполняет командную строку — строку символов, в которой содержится пакет Transact-SQL или один из следующих модулей: системная хранимая процедура, пользовательская хранимая процедура, скалярная пользовательская функция или расширенная хранимая процедура.

Синтаксис

```
[ { EXEC | EXECUTE } ]
{
    [ @код_возврата = ]
    { имя_модуля | @переменная_имени_модуля }
        [ [ @параметр = ] { значение | @параметр [ OUTPUT ] | [ DEFAULT ] }
        ]
    [ ,...n ]
}
[ ; ]
```

Примеры

A. Вызов EXECUTE с передачей единственного аргумента

```
USE AdventureWorks;
GO
EXEC dbo.uspGetEmployeeManagers 6;
GO
```

При выполнении переменная может быть явно поименована

```
USE AdventureWorks;
GO
EXEC dbo.uspGetEmployeeManagers @EmployeeID = 6;
GO
```

Если приведенная инструкция является первой в пакете или сценарии или sqlcmd, то указание EXEC не требуется

```
USE AdventureWorks;
GO
dbo.uspGetEmployeeManagers 6;
GO
-- ИЛИ
dbo.uspGetEmployeeManagers @EmployeeID = 6;
GO
```

Б. Передача нескольких аргументов

```
USE AdventureWorks;
GO
DECLARE @CheckDate datetime;
SET @CheckDate = GETDATE();
EXEC dbo.uspGetWhereUsedProductID 819, @CheckDate;
GO
```

В. Использование в EXECUTE переменной хранимой процедуры

```
DECLARE @proc_name varchar(30);
```

```
SET @proc_name = 'sys.sp_who';
EXEC @proc_name;
```

Г. Выполнение пользовательской функции с помощью EXECUTE

```
USE AdventureWorksLT;
GO
DECLARE @returnstatus nvarchar(15);
SET @returnstatus = NULL;
EXEC @returnstatus = dbo.ufnGetSalesOrderStatusText @Status = 2;
PRINT @returnstatus;
GO
```

PRINT

Возвращает клиенту пользовательское сообщение.

Синтаксис

```
PRINT строковая_константа | @локальная_переменная | строковое_выражение
```

Пример

```
DECLARE @PrintMessage NVARCHAR(50);
SET @PrintMessage = N'Это сообщение было напечатано '
+ RTRIM(CAST(GETDATE() AS NVARCHAR(30)))
+ N'.';
PRINT @PrintMessage;
GO
```

Для возвращения сообщений можно также использовать функцию RAISERROR.

RAISERROR

Создает сообщение об ошибке и запускает обработку ошибок для сеанса. Инструкция RAISERROR может либо ссылаться на определенное пользователем сообщение, находящееся в представлении каталога sys.messages, либо динамически создавать сообщение. Это сообщение возвращается как сообщение об ошибке сервера вызывающему приложению или соответствующему блоку CATCH конструкции TRY...CATCH.

Синтаксис

```
RAISERROR ( { номер ошибки | строковая_константа | @локальная_переменная }
{ , серьезность ошибки , состояние ошибки }
[ , аргумент [ ,...n ] ] )
```

Пример

```
DECLARE @ErrorMessage NVARCHAR(4000);
DECLARE @ErrorSeverity INT;
DECLARE @ErrorState INT;

SELECT @ErrorMessage = ERROR_MESSAGE(),
@ErrorSeverity = ERROR_SEVERITY(),
@ErrorState = ERROR_STATE();

RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
```

Преимущества функции RAISERROR перед функцией PRINT:

- функция RAISERROR поддерживает вставку аргументов в строку сообщения об ошибке с помощью механизма, основанного на функции printf из стандартной библиотеки языка C;
- в дополнение к текстовому сообщению, функция RAISERROR может в сообщении указать уникальный номер ошибки, степень ее серьезности и код состояния;
- функция RAISERROR может быть использована для возвращения пользовательских сообщений, созданных с помощью системной хранимой процедуры sp_addmessage.

TRY...CATCH

Синтаксис

```
BEGIN TRY
    { sql_инструкция | блок_инструкций }
END TRY
BEGIN CATCH
    { sql_инструкция | блок_инструкций }
END CATCH
[ ; ]
```

Примеры

```
BEGIN TRY
    BEGIN TRANSACTION
        DELETE [Order Details] WHERE OrderID IN
            (SELECT OrderID FROM Orders WHERE CustomerID = 'ALFKI')
        DELETE Orders WHERE CustomerID = 'ALFKI'
        DELETE Customers WHERE CustomerID = 'ALFKI'
        PRINT 'committing deletes'
        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION
        RETURN
    END CATCH

USE AdventureWorksLT;
GO

BEGIN TRY
    -- Генерация ошибки деления на ноль.
    SELECT 1/0;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS N'Номер ошибки',
        ERROR_SEVERITY() AS N'Серьезность ошибки',
        ERROR_STATE() AS N'Состояние ошибки',
        ERROR_PROCEDURE() AS N'Имя хранимой процедуры',
        ERROR_LINE() AS N'Номер строки',
        ERROR_MESSAGE() AS N'Текст сообщения об ошибке';
END CATCH;
GO
```

Результаты выполнения этого сценария будут иметь вид

Номер ошибки	Серьезность ошибки	Состояние ошибки	Имя хранимой процедуры	Номер строки	Текст
8134	16	1	NULL	4	Divide by zero error encountered.

WHILE

Ставит условие повторного выполнения SQL-инструкции или блока инструкций. Эти инструкции вызываются в цикле, пока указанное условие истинно. Вызовами инструкций в цикле WHILE можно контролировать из цикла с помощью ключевых слов BREAK и CONTINUE.

Синтаксис

```
WHILE булевское_выражение
    { sql_инструкция | блок_инструкций }
```

```
[ BREAK ]
{ sql_инструкция | блок_инструкций }
[ CONTINUE ]
{ sql_инструкция | блок_инструкций }
```

Пример

```
declare @i int set @i = 1
declare @sum int set @sum = 0
while @i <= 100
begin
    set @sum = @sum + ROUND(RAND(), 0)
    set @i = @i + 1
end
select @sum
```

WAITFOR

Блокирует выполнение пакета, хранимой процедуры или транзакции до наступления указанного времени или интервала времени, либо заданная инструкция изменяет или возвращает, по крайней мере, одну строку.

Синтаксис

```
WAITFOR { DELAY 'период_времени_ожидания' | TIME 'время_завершения_инструкции' }
```

Пример

```
USE msdb;
EXECUTE sp_add_job @job_name = 'TestJob';
BEGIN
    WAITFOR TIME '22:20';
    EXECUTE sp_update_job @job_name = 'TestJob',
        @new_name = 'UpdatedJob';
END;
GO
```

Сценарии и пакеты

Общие сведения о сценариях

В общем случае **сценарии (script)** пишутся для решения от начала и до конца какой-то определенной задачи. С этой целью в один сценарий объединяются несколько команд. Примерами могут служить сценарии, используемые для построения баз данных (они часто применяются при установке систем). Сценарий не является сценарием до тех пор, пока он не сохранен в файле, из которого он может быть извлечен и использован в дальнейшем. SQL-сценарии хранятся в текстовых файлах. Любой сценарий всегда рассматривается как единое целое. Либо выполняется весь сценарий целиком, либо совсем ничего не выполняется. В сценариях могут использоваться системные функции и локальные переменные.

Пример

```
USE Northwind
DECLARE @Ident int
INSERT INTO Orders (CustomerID,OrderDate)
VALUES ('ALFKI', DATEADD(day,-1,GETDATE()))
SELECT @Ident = @@IDENTITY
INSERT INTO [Order Details] (OrderID, ProductID, UnitPrice, Quantity)
VALUES (@Ident, 1, 50, 25)
SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

Здесь для решения задачи целиком объединено шесть команд. Используются системные функции, локальные переменные, операторы USE и INSERT. Оператор SELECT был использован двумя способами - в рамках присваивания и обычным способом (для выборки). Все эти команды выполняют одну задачу - добавляют записи в базу данных.

Замечание. То, что раньше в SQL Server называлось глобальными переменными (да и сейчас довольно часто именно так и называется в документации) на самом деле не является переменными вовсе (в них нельзя хранить все, что угодно), и будет более корректным называть их "системными функциями" или "системными функциями без аргументов". Везде далее будет использоваться термин "системные функции".

Оператор USE

С помощью оператора USE устанавливается текущая база данных. В дальнейшем при выполнении операций над объектами этой базы данных уже не нужно указывать ее имя. После использования оператора USE эта база становится используемой по умолчанию. Это совсем не означает, что оператор USE всегда надо включать в сценарий. Если создается сценарий общего назначения (т. е. предназначенный для использования с произвольной базой), тогда вполне разумно будет не включать в него оператор USE.

Объявление переменных

Для объявления переменных используется оператор DECLARE, который имеет следующий синтаксис:

```
DECLARE список-объявлений-переменных
```

где объявление переменной имеет вид

```
@имя-переменной тип-переменной
```

Можно объявить одну или несколько переменных одновременно. Использование отдельного оператора DECLARE для каждой объявляемой переменной дает тот же результат, что и разделение переменных запятыми в общем операторе. Применимый стиль зависит предпочтений программиста, но независимо от выбранного способа объявления значение переменной будет равно NULL до тех пор, пока ей не будет присвоено значение.

Присвоение значений переменным

Существует два способа присваивания значений переменным. Можно использовать операторы SELECT или SET. Функционально они практически эквивалентны, за исключением того, что прямо внутри оператора SELECT можно использовать в качестве присваиваемых значений содержимое полей.

Оператор SET традиционно используется для присвоения значений переменным подобно тому, как это делается во многих процедурных языках. Типичным примером может служить следующий фрагмент кода:

```
USE Northwind
SET @TotalCost = 10
SET @TotalCost = @UnitCost * 1.1
```

При помощи оператора SET нельзя присвоить переменной значение, полученное непосредственно из запроса – нужно отделить запрос от SET. Например, следующий сценарий:

```
DECLARE @Test minåy
SET @Test = MAX (UnitPrice) FROM [Order Details]
SELECT @Test
```

вызовет ошибку, а сценарий

```
USE Northwind
DECLARE @Test money
SET @Test = (SELECT MAX (UnitPrice) FROM [Order Details])
SELECT @Test
```

будет работать.

SELECT обычно используется в том случае, когда переменной непосредственно нужно присвоить значение, полученное в результате запроса. Такой пример, как предыдущий, чаще всего реализуются с использованием SELECT:

```
USE Northwind
DECLARE @Test minåo
SELECT @Test = MAX (UnitPrice) FROM [Order Details]
SELECT @Test
```

Таким образом, можно сделать следующие выводы:

- SET используется в случае, если переменной просто присваивается известное на момент операции значение (константа) или же одной переменной присваивается значение другой;
- SELECT лучше использовать в том случае, когда переменной необходимо присвоить результат выполнения запроса.

Обзор системных функций

Существует около 30 системных функций без параметров. Некоторые из них являются особо интересными и полезными:

- @@ERROR. Возвращает номер ошибки последнего оператора Transact-SQL, выполненного в текущем подключении. При отсутствии ошибки возвращает 0.
- @@IDENTITY. Возвращает последнее identity-значение, которое было вставлено в результате выполнения последнего оператора INSERT или SELECT INTO.
- @@ROWCOUNT. Возвращает число строк, полученных в результате выполнения последнего оператора.

Общие сведения о пакетах

Пакет (batch) - это несколько объединенных в одну логическую группу операторов Transact-SQL. Все операторы в рамках пакета комбинируются в единый план исполнения (execution plan) таким образом, что пока все операторы не будут успешно проанализированы синтаксическим анализатором, ни один из операторов пакета не будет исполняться. Если на этапе синтаксического анализа какой-либо оператор из пакета оказывается ошибочным, то ни один оператор пакета не выполняется. Если какой-либо оператор пакета вызывает ошибку на этапе выполнения программы, то выполняются все операторы пакета вплоть до ошибочного.

Для того чтобы разделить сценарий на несколько пакетов, необходимо использовать оператор GO. Оператор GO:

- должен писаться в отдельной строке (ничего, кроме комментариев, не должно следовать за ним в этой же строке); есть исключения, которые будут рассматриваться ниже;
- все операторы от начала сценария (или ближайшего предыдущего оператора GO) и до данного оператора GO компилируются в один план исполнения и пересыпаются на сервер отдельно от других пакетов, т. е. ошибка в одном пакете не может помешать выполнению другого пакета;
- это не команда Transact-SQL, а директива, которую распознают различные утилиты с командным интерфейсом SQL Server (OSQL, ISQL и Анализатор Запросов).

Сценарий можно построить и таким образом, чтобы выполнение одного пакета зависело от завершения другого.

Ошибки, встречающиеся в пакетах, можно разделить на две группы:

- синтаксические ошибки. Если синтаксический анализатор находит синтаксическую ошибку в запросе, обработка данного пакета сразу же прекращается. Синтаксический анализ выполняется перед компиляцией пакета и его запуском на выполнение, наличие ошибки при этом означает, что никакая часть пакета не будет запущена на выполнение, пока ошибка не будет устранена.
- ошибки, возникающие при выполнении программы. Эти ошибки имеют несколько другие свойства. Любой оператор, который выполнился до возникновения ошибки, считается уже выполненным и все, что было сделано в результате его выполнения остается действительным, конечно с учетом того, что он является частью незавершенной транзакции.

Пример

```
USE Northwind
DECLARE @MyVarchar varchar(50) -- Только этот DECLARE остается от данного пакета!
SELECT @MyVarchar = 'Привет, я дома ...'
PRINT 'Выполним первый пакет ...'
GO
PRINT @MyVarchar -- Приводит к ошибке, поскольку переменная @MyVarchar
                  -- в этом пакете не объявлена
PRINT 'Выполним второй пакет ...'
GO
PRINT 'Выполним третий пакет ...' -- Этот пакет выполняется
                                    -- несмотря на предшествующие ошибки
GO
```

Когда используются пакеты?

Цели применения пакетов различны, но обычно они используются в ситуациях, когда в сценарии необходимо что-нибудь выполнить либо перед, либо отдельно от всего остального. Чаще всего пакеты используются для явного задания предшествования, – когда необходимо полностью завершить решение одной задачи перед тем, как начнет решаться другая.

Правила использования пакетов

Для использования пакетов применяются следующие правила.

1. Инструкции CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER и CREATE VIEW не могут быть объединены с другими инструкциями в пакете. Инструкция CREATE должна быть первой инструкцией в пакете. Все последующие инструкции в этом пакете рассматриваются как часть определения первой инструкции CREATE.
2. В одном и том же пакете нельзя сначала изменить таблицу и затем обратиться к новым столбцам.
3. Если инструкция EXECUTE — первая инструкция в пакете, ключевое слово EXECUTE не требуется. Ключевое слово EXECUTE требуется, если инструкция EXECUTE не является первой инструкцией в пакете.
4. Если необходимо удалить объект, то следует поместить оператор DROP в отдельный пакет или, по крайней мере, в пакет с другими операторами DROP. Объяснение этому следующее. Если необходимо создать объект с тем именем, которое перед этим удалили, то CREATE вызовет ошибку во время синтаксического анализа, несмотря на то, что оператор DROP уже был выполнен. Это означает, что следует запускать DROP в отдельном пакете, который предшествует CREATE, чтобы удаление старого объекта завершилось к тому моменту, когда начнет выполняться создание нового объекта с таким же именем.

Пример пакетного задания.

```
USE AdventureWorks;
GO
CREATE VIEW dbo.vProduct
AS
SELECT ProductNumber, Name
FROM Production.Product;
GO
SELECT *
FROM dbo.vProduct;
GO
```

В данном примере создается представление. Так как инструкция CREATE VIEW должна быть единственной в пакете, требуются команды GO, чтобы изолировать инструкцию CREATE VIEW от предшествующей и последующей инструкций USE и SELECT.

Массовый импорт и экспорт данных

Массовый экспорт означает копирование данных из таблицы SQL Server в файл данных. Массовый импорт означает загрузку данных из файла данных в таблицу SQL Server. Например, можно экспорттировать данные из приложения Microsoft Excel в файл данных, а затем выполнить массовый импорт данных в таблицу SQL Server. SQL Server поддерживает массовый экспорт данных (массовых данных) из таблиц SQL Server и импорт массовых данных в таблицу SQL Server. Массовый импорт и массовый экспорт имеют большое значение для эффективной передачи данных между SQL Server и разнородными источниками данных.

Массовый импорт и экспорт данных с помощью мастера служб Integration Services

В состав служб MSIS входит несколько мастеров, одним из которых является мастер импорта и экспорта SQL Server. Этот мастер может копировать данные из любого источника в таблицы и представления, если для этого источника существует поставщик данных .NET Framework или собственный поставщик данных OLE DB. Для источников данных SQL Server, Плоские файлы, Microsoft Access, Microsoft Excel есть соответствующие поставщики. Чтобы выполнить импорт, нужно запустить созданный мастером пакет. При желании пакет можно сохранить для повторного использования.

Запуск мастера импорта и экспорта SQL Server в среде SSMS выполняется так:

1. Подключитесь к серверу типа Database Engine,
2. Разверните базы данных,
3. Правой кнопкой мыши щелкните базу данных, выберите пункт Задачи, затем выберите пункт Импорт данных.

Массовый импорт и экспорт данных с помощью программы bcp

Программа **bcp** используется для массового копирования данных между экземпляром Microsoft SQL Server и файлом данных в пользовательском формате. С помощью программы **bcp** можно выполнять импорт большого количества новых строк в таблицы SQL Server или экспорт данных из таблиц в файлы данных. Сведения о синтаксисе команды **bcp** можно получить, выполнив команду без аргументов. В простейших случаях используются следующие аргументы.

{[[*database_name*.][*schema*.]]{*table_name* / *view_name*}

Имя базы данных *database_name* и имя схемы *schema* являются необязательными параметрами.

in *data_file*

Выполняется копирование из файла *data_file* в таблицу базы данных или представление.

out *data_file*

Выполняется копирование из таблицы базы данных или представления в файл *data_file*. Если указать существующий файл, то файл перезаписывается.

queryout

Выполняется копирование из запроса.

format

Создается файл форматирования, основанный на указанных параметрах (-n, -c, -w или -N) и разделителях таблиц или представлений. При выполнении массового копирования данных команда **bcp** может обратиться к файлу форматирования, что позволяет избежать повторного ввода данных о формате в интерактивном режиме. Параметр **format** требует наличия параметра -f. Для создания XML-файла форматирования также необходим параметр -x. В качестве значения параметра **format** необходимо указать **nul**.

-T

Указывает, что программа **bcp** устанавливает доверительное соединение с SQL Server с использованием встроенной безопасности. Не требуются учетные данные безопасности для сетевого пользователя, параметры *login_id* и *password*. Если параметр **-T** не указан, для входа необходимо указать **-U** *login_id* и **-P** *password*.

-c

Выполняет операцию, используя символьный тип данных. При использовании этого параметра не запрашивается тип данных каждого поля. Для хранения данных используется тип char, префиксы отсутствуют, в качестве разделителя полей используется символ табуляции \t, а в качестве признака конца строки — символ новой строки \r\n.

-S *server_name*[\i_{instance_name}]

Указывает экземпляр SQL Server, к которому выполняется подключение. Если сервер не указан, то программа **bcp** выполняет подключение к экземпляру SQL Server по умолчанию на локальном компьютере. Этот

параметр необходим, когда команда **bcp** запускается из удаленного компьютера в сети или из локального именованного экземпляра. Чтобы подключиться к экземпляру по умолчанию SQL Server на сервере, укажите только *server_name*. Чтобы подключиться к именованному экземпляру SQL Server, укажите *server_name\instance_name*.

Пример 1. Массовое копирование символьных данных из файла **S.txt** в таблицу **dbSPJ.dbo.S** (с помощью доверительного соединения):

```
bcp dbSPJ.dbo.S in Currency.dat -T -c -S .\SQLEXPRESS
```

Чтобы убедиться в успешном выполнении команды, отобразите в редакторе запросов содержимое таблицы и введите:

```
SELECT * FROM S
```

Пример 2. Массовое копирование строк таблицы **dbSPJ.dbo.S** в файл данных **S.txt** (с помощью доверительного соединения):

```
bcp dbSPJ.dbo.S out S.txt -T -c -S .\SQLEXPRESS
```

Пример 3. Копирование данных из запроса в файл **S.txt**:

```
bcp "SELECT SName, Status, City FROM dbSPJ.dbo.S ORDER BY SName" queryout S.txt -c -T -S .\SQLEXPRESS
```

Пример 4. Создание XML-файла форматирования **S.xml**:

```
bcp dbSPJ.dbo.S format nul -T -c -x -f S.xml -S .\SQLEXPRESS
```

Пример 5. Применение файла форматирования для выполнения массового импорта данных с помощью программы **bcp**

```
bcp dbSPJ.dbo.tempS in S.txt -T -f S.xml -S .\SQLEXPRESS
```

Массовый импорт данных при помощи инструкции BULK INSERT

Инструкция **BULK INSERT** загружает данные из файла данных в таблицу. Принцип работы здесь тот же, что и при выполнении команды **bcp** с параметром **in**, однако файл данных считывается процессом SQL Server.

```
BULK INSERT dbSPJ.dbo.S FROM 'C:\Documents and Settings\Admin\S.txt'  
WITH (DATAFILETYPE = char', FIELDTERMINATOR='t', ROWTERMINATOR='\n', CODEPAGE='ACP')
```

Массовый импорт данных при помощи OPENROWSET(BULK...)

Чтобы импортировать групповые данные, вызовите функцию **OPENROWSET(BULK...)** из предложения **SELECT...FROM** инструкции **INSERT**. Основной синтаксис массового импорта данных:

```
INSERT ... SELECT * FROM OPENROWSET(BULK...)
```

Массовый импорт XML-данных в виде двоичного байтового потока

Для проверки примера необходимо создать образец таблицы **T**.

```
USE tempdb  
GO  
CREATE TABLE T (IntCol int, XmlCol xml)  
GO
```

Перед запуском примера необходимо создать файл в кодировке UTF-8 (**C:\Documents and Settings\Admin\S2.xml**), содержащий следующий образец, который определяет схему в кодировке UTF-8.

```
SELECT * FROM S FOR XML AUTO, ELEMENTS
```

```
INSERT INTO T(XmlCol)  
SELECT * FROM OPENROWSET(BULK 'C:\Documents and Settings\Admin\S2.xml', SINGLE_BLOB) AS x
```

Представления T-SQL

Представление (View) – это виртуальная таблица, содержимое которой (столбцы и строки) определяется запросом. Представление можно использовать в следующих целях:

- Для направления, упрощения и настройки восприятия информации в базе данных каждым пользователем.
- В качестве механизма безопасности, позволяющего пользователям обращаться к данным через представления, но не дающего им разрешений на непосредственный доступ к базовым таблицам.

Создание представления

Для создания представления данных, содержащихся в одной или более таблицах базы данных, необходимо использовать инструкцию CREATE VIEW, которая должна быть первой в пакетном запросе.

Базовый синтаксис

```
CREATE VIEW [ имя-схемы . ] имя-представления [ (столбец [ ,...n ] ) ]
AS инструкция-select
```

Здесь

столбец – 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 столбец не указан, столбцам представления назначаются такие же имена, которые имеют столбцы в инструкции SELECT.

Пример

```
USE dbSPJ ;
GO
IF OBJECT_ID ('dbo.vwSPJ', 'V') IS NOT NULL
    DROP VIEW dbo.vwSPJ ;
GO
CREATE VIEW dbo.vwSPJ
AS
SELECT S.SName, P.PName, J.JName, SPJ.Qty
FROM SPJ INNER JOIN S ON SPJ.Sno = S.Sno
    INNER JOIN P ON SPJ.Pno = P.Pno
    INNER JOIN J ON SPJ.Jno = J.Jno
WHERE S.City = 'Владимир' ;
GO

USE dbSPJ
GO
SELECT * FROM vwSPJ
GO
```

Обновляемые представления

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. Как правило, 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, используйте триггер INSTEAD OF.

Триггер INSTEAD OF

Чтобы сделать представление обновляемым, 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.

Функции T-SQL

SQL Server содержит набор встроенных функций и предоставляет возможность создавать пользовательские функции (User Defined Function – UDF). Различают *детерминированные* и *недетерминированные* функции.

Функция является детерминированной, если при одном и том же заданном входном значении она всегда возвращает один и тот же результат. Например,строенная функция DATEADD является детерминированной; она возвращает новое значение даты, добавляя интервал к указанной части заданной даты.

Функция является недетерминированной, если она может возвращать различные значения при одном и том же заданном входном значении. Например, встроенная функция GETDATE является недетерминированной; при каждом вызове она возвращает различные значения даты и времени компьютера, на котором запущен экземпляр SQL Server.

Все функции конфигурации, курсора, метаданных, безопасности и системные статистические – недетерминированные. Список этих функций приводится в справочнике. Функции, вызывающие недетерминированные функции и расширенные хранимые процедуры, также считаются недетерминированными.

Пользовательские функции в зависимости от типа данных возвращаемых ими значений могут быть *скалярными* и *табличными*. Табличные пользовательские функции бывают двух типов: *подставляемые* и *многооператорные*.

Скалярные пользовательские функции обычно используются в списке столбцов инструкции SELECT и в предложении WHERE. Табличные пользовательские функции обычно используются в предложении FROM, и их можно соединять с другими таблицами и представлениями.

Создание и вызов скалярной функции

Для создания скалярной функции используется инструкция CREATE FUNCTION, имеющая следующий синтаксис:

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS скалярный-тип-данных
[ WITH список-опций-функций ]
[ AS ]
BEGIN
    тело-функции
    RETURN скалярное-выражение
END [ ; ]
```

где

- Список объявлений параметров является необязательным, но наличие скобок обязательно.
- Объявление параметра в списке объявлений параметров имеет вид:

```
@имя-параметра [ AS ] [ имя-схемы. ] тип-данных [ = значение-по-умолчанию ] [ READONLY ]
```

- Тип данных параметра – любой тип данных, включая определяемые пользователем типы данных CLR и определяемые пользователем табличные типы, за исключением скалярного типа timestamp и нескаларных типов cursor и table.
- Значение по умолчанию -
- Необязательное ключевое слово READONLY указывает, что параметр не может быть обновлен или изменен при определении функции. Если тип параметра является определяемым пользователем табличным типом, то должно быть указано ключевое слово READONLY.
- Возвращаемое значение может быть любого скалярного типа данных, включая определяемые пользователем типы данных CLR, за исключением типа данных timestamp.
- В качестве опций функции используются:
 - ENCRYPTION — SQL Server шифрует определение функции.
 - SCHEMABINDING - привязывает функцию к схеме базовой таблицы или таблиц. Если аргумент SCHEMABINDING указан, нельзя изменить базовую таблицу или таблицы таким способом, который может повлиять на определение функции.

```
RETURNS NULL ON NULL INPUT или| CALLED ON NULL INPUT - ...
```

Предложение EXECUTE AS - указывает контекст безопасности, в котором может быть выполнена функция.

Например, EXECUTE AS CALLER указывает, что функция будет выполнена в контексте пользователя, который ее вызывает. Также могут быть указаны параметры SELF, OWNER и имя-пользователя.

- h) Операторы BEGIN и END, которыми ограничивается тело функции, являются обязательными.
- i) Тело функции представляет собой ряд инструкций T-SQL, которые в совокупности вычисляют скалярное выражение.

Синтаксис вызова скалярных функций схож с синтаксисом, используемым для встроенных функций T-SQL:

имя-схемы.имя-функции ([список-параметров])

где

- a) Имя схемы (имя владельца) для скалярной функции является обязательным.
- b) Нельзя использовать синтаксис с именованными параметрами (*@имя-параметра = значение*).
- c) Нельзя не указывать (опускать) параметры, но можно применять ключевое слово DEFAULT для указания значения по умолчанию.

Для скалярной функции можно использовать инструкцию EXECUTE:

EXECUTE @возвращаемое-значение = имя-функции ([список-параметров])

где

- a) Не нужно указывать имя схемы (имя владельца).
- b) Можно использовать именованные параметры (*@имя-параметра = значение*).
- c) Если используются именованные параметры, они не обязательно должны следовать в том порядке, в котором указаны в объявлении функции, но необходимо указать все параметры; нельзя опускать ссылку на параметр для использования значения по умолчанию.

Примеры создания и вызова скалярных функций

```
USE dbSPJ
GO
IF OBJECT_ID (N'dbo.AveragePrice', N'FN') IS NOT NULL
    DROP FUNCTION dbo.AveragePrice
GO
CREATE FUNCTION dbo.AveragePrice()
RETURNS smallmoney
WITH SCHEMABINDING
AS
BEGIN
    RETURN (SELECT AVG(Price) FROM dbo.R)
END
GO
IF OBJECT_ID (N'dbo.PriceDifference', N'FN') IS NOT NULL
    DROP FUNCTION dbo.PriceDifference
GO
CREATE FUNCTION dbo.PriceDifference(@Price smallmoney)
RETURNS smallmoney
AS
BEGIN
    RETURN @Price - dbo.AveragePrice()
END
GO
SELECT Pname, Price, dbo.AveragePrice() AS Average, dbo.PriceDifference(Price) AS Difference
FROM R
WHERE City='Смоленск'
GO
```

Создание и вызов подставляемой табличной функции

Синтаксис создания подставляемой табличной функции выглядит так:

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS TABLE
[ WITH список-опций-функций ]
[ AS ]
    RETURN [ ( ] выражение-выборки [ ) ]
END [ ; ]
```

Тело подставляемой табличной функции фактически состоит из единственной инструкции SELECT.

Пример создания и вызова подставляемой табличной функции

```
USE dbSPJ
GO
IF OBJECT_ID (N'dbo.FullSPJ', N'FN') IS NOT NULL
    DROP FUNCTION dbo.FullSPJ
GO
CREATE FUNCTION dbo.FullSPJ()
RETURNS TABLE
AS
RETURN (SELECT S.Sname, P.Pname, J.Jname, SPJ.Qty
        FROM S
            INNER JOIN SPJ ON S.Sno=SPJ.Sno
            INNER JOIN P ON P.Pno=SPJ.Pno
            INNER JOIN J ON J.Jno=SPJ.Jno)
GO
SELECT *
FROM dbo.FullSPJ()
GO
```

Создание и вызов многооператорной табличной функции

Синтаксис создания многооператорной табличной функции выглядит так:

```
CREATE FUNCTION [ имя-схемы. ] имя-функции ( [ список-объявлений-параметров ] )
RETURNS @имя-возвращаемой-переменной TABLE определение-таблицы
[ WITH список-опций-функций ]
[ AS ]
BEGIN
    тело-функции
    RETURN
END [ ; ]
```

Подобно скалярным функциям, в многооператорной табличной функции команды T-SQL располагаются внутри блока BEGIN-END. Поскольку блок может содержать несколько инструкций SELECT, в предложении RETURNS необходимо явно определить таблицу, которая будет возвращаться. Поскольку оператор RETURN в многооператорной табличной функции всегда возвращает таблицу, заданную в предложении RETURNS, он должен выполняться без аргументов.

Пример создания и вызова многооператорной табличной функции

```
USE Northwind
GO
CREATE FUNCTION dbo.fnGetReports ( @EmployeeID AS int )
RETURNS @Reports TABLE
(
    EmployeeID     int     NOT NULL,
    ReportsToID   int     NULL
)
```

```

AS
BEGIN

/* Объявляем переменную @Employee для хранения ID текущего служащего (чтобы случайно не
включить его дважды). */

DECLARE @Employee AS int

/* Добавляем текущего служащего и его начальника в рабочую таблицу @Reports. Суть в том, что
необходима некоторая начальная строка из-за рекурсивной природы функции. */

INSERT INTO @Reports
    SELECT EmployeeID, ReportsTo
    FROM Employees
    WHERE EmployeeID = @EmployeeID

/* Подготавливаемся к выполнению рекурсивного вызова функции. */

SELECT @Employee = MIN(EmployeeID)
FROM Employees
WHERE ReportsTo = @EmployeeID

/* Рекурсивный вызов функции! */

WHILE @Employee IS NOT NULL
BEGIN
    INSERT INTO @Reports
        SELECT *
        FROM fnGetReports(@Employee)

    SELECT @Employee = MIN(EmployeeID)
    FROM Employees
    WHERE EmployeeID > @Employee
        AND ReportsTo = @EmployeeID
END
RETURN
END
GO

```

Для исходных данных

1	Davolio	Nancy	2
2	Fuller	Andrew	NULL
3	Leverling	Janet	2
4	Peacock	Margaret	2
5	Buchanan	Steven	2
6	Suyama	Michael	5
7	King	Robert	5
8	Callahan	Laura	2
9	Dodsworth	Anne	5

вызов функции

```

SELECT * FROM dbo.fnGetReports(5)
GO

```

даст следующие результаты

```

5      2

```

6 5
7 5
9 5

Какие результаты будут получены при таком вызове функции?

```
DECLARE @EmployeeID int

SELECT @EmployeeID = EmployeeID
      FROM Employees
     WHERE LastName = 'Fuller' AND FirstName = 'Andrew'

SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Mgr.LastName AS ReportsTo
  FROM Employees AS Emp
  JOIN dbo.fnGetReports(@EmployeeID) AS gr ON gr.EmployeeID = Emp.EmployeeID
  JOIN Employees AS Mgr ON Mgr.EmployeeID = gr.ReportsToID
```

Триггеры T-SQL

Триггеры DDL

Триггер DDL - это хранимая процедура особого типа, которая выполняет одну или несколько инструкций T-SQL в ответ на событие из области действия сервера или базы данных. Например, триггер DDL может активироваться, если выполняется такая инструкция, как ALTER SERVER CONFIGURATION, или если происходит удаление таблицы с использованием команды DROP TABLE. По адресу <http://msdn.microsoft.com/ru-ru/library/bb510452.aspx> приведен список DDL-событий, при помощи которых, можно вызвать запуск триггеров DDL. Все события (кроме ALTER_SERVER_CONFIGURATION) разделены на две группы: события уровня базы данных (DDL_DATABASE_LEVEL_EVENTS) (180 событий) и события уровня сервера (DDL_SERVER_LEVEL_EVENTS) (80 событий). Следует обратить внимание на иерархическую природу групп событий.

В отличие от триггеров DML, триггеры DDL не ограничены областью схемы. Поэтому для запроса метаданных о триггерах DDL нельзя воспользоваться такими функциями как OBJECT_ID, OBJECT_NAME, OBJECTPROPERTY и OBJECTPROPERTYEX. Используйте вместо них представления каталога.

Как получить сведения о DDL триггерах?

Сведения о событиях, при возникновении которых может сработать триггер DDL - sys.trigger_event_types.

Зависимости триггера - sys.sql_expression_dependencies, sys.dm_sql_referenced_entities, sys.dm_sql_referencing_entities

Триггеры DDL уровня базы данных:

- Сведения о триггерах с областью действия на уровне базы данных - sys.triggers (Transact-SQL)
- Сведения о событиях базы данных, вызывающих срабатывание триггеров - sys.trigger_events (Transact-SQL)
- Определения триггеров уровня базы данных - sys.sql_modules (Transact-SQL)
- Сведения о триггерах среды CLR уровня базы данных - sys.assembly_references (Transact-SQL)

Триггеры DDL уровня сервера:

- Сведения о триггерах с областью действия на уровне сервера - sys.server_triggers (Transact-SQL)
- Сведения о событиях сервера, вызывающих срабатывание триггеров - sys.server_trigger_events (Transact-SQL)
- Определения триггеров с областью действия на уровне сервера - sys.server_sql_modules (Transact-SQL)
- Сведения о триггерах CLR с областью действия на уровне сервера - sys.server_assembly_modules (Transact-SQL)

Пример триггера DDL

```
CREATE TRIGGER safety ON DATABASE FOR DROP_TABLE, ALTER_TABLE
AS
    PRINT 'You must disable Trigger "safety" to drop or alter tables!'
    ROLLBACK;
```

Триггеры DML

Для поддержания согласованности и точности данных используются *декларативные и процедурные* методы. Триггеры представляют собой особый вид хранимых процедур, привязанных к таблицам и представлениям. Они позволяют реализовать в базе данных сложные **процедурные** методы поддержания целостности данных. События при модификации данных вызывают автоматическое срабатывание триггеров. Как и в случае хранимых процедур, глубина вложенности триггеров достигает 32 уровней, также возможно рекурсивное срабатывание триггеров.

Прежде чем реализовывать триггер, следует выяснить, нельзя ли получить аналогичные результаты с использованием ограничений или правил. Для уникальной идентификации строк табличных данных используют целостность сущностей (ограничения **primary key** и **unique**). Доменная целостность служит для определения значений по умолчанию (определения **default**) и ограничения диапазона значений, разрешенных для ввода в данное поле (ограничения **check** и **ссылочные ограничения**). Ссылочная целостность используется для реализации логических связей между таблицами (ограничения

foreign key и **check**). Если значение обязательного поля не задано в операторе INSERT, то оно определяется с помощью определения **default**. Лучше применять ограничения, чем триггеры и правила.

Триггеры применяются в следующих случаях:

- если использование методов декларативной целостности данных не отвечает функциональным потребностям приложения. Например, для изменения числового значения в таблице при удалении записи из этой же таблицы следует создать триггер;
- если необходимо каскадное изменение через связанные таблицы в базе данных. Чтобы обновить или удалить данные в столбцах с ограничением **foreign key**, вместо пользовательского триггера следует применять ограничения каскадной ссылочной целостности;
- если база данных денормализована и требуется способ автоматизированного обновления избыточных данных в нескольких таблицах;
- если необходимо сверить значение в одной таблице с неидентичным значением в другой таблице;
- если требуется вывод пользовательских сообщений и сложная обработка ошибок.

События, вызывающие срабатывание триггеров (типы триггеров)

Автоматическое срабатывание триггера вызывают три события: INSERT, UPDATE и DELETE, которые происходят в таблице или представлении. Триггеры нельзя запустить вручную. В синтаксисе триггеров перед фрагментом программы, уникально определяющим выполняемую триггером задачу, всегда определено одно или несколько таких событий. Один триггер разрешается запрограммировать для реакции на несколько событий, поэтому несложно создать процедуру, которая одновременно является триггером на обновление и добавление. Порядок этих событий в определении триггера является несущественным.

Классы триггеров

В SQL Server существуют два класса триггеров:

- INSTEAD OF. Триггеры этого класса выполняются в обход действий, вызывавших их срабатывание, заменяя эти действия. Например, обновление таблицы, в которой есть триггер INSTEAD OF, вызовет срабатывание этого триггера. В результате вместо оператора обновления выполняется код триггера. Это позволяет размещать в триггере сложные операторы обработки, которые дополняют действия оператора, модифицирующего таблицу.
- AFTER. Триггеры этого класса исполняются после действия, вызвавшего срабатывание триггера. Они считаются классом триггеров по умолчанию.

Между триггерами этих двух классов существует ряд важных отличий, показанных в таблице.

Характеристика	INSTEAD OF-триггер	AFTER-триггер
Объект, к которому можно привязать триггер	Таблица или представление. Триггеры, привязанные к представлению, расширяют список типов обновления, которые может поддерживать представление	Таблица. AFTER-триггеры срабатывают при модификации данных в таблице в ответ на модификацию представления
Допустимое число триггеров	В таблице или представлении допускается не больше одного триггера в расчете на одно действие. Можно определять представления для других представлений, каждое со своим собственным INSTEAD OF-триггером	К таблице можно привязать несколько AFTER- триггеров
Порядок исполнения	Поскольку в таблице или представлении допускается не больше одного такого триггера в расчете на одно событие, порядок не имеет смысла	Можно определять триггеры, срабатывающие первым и последним. Для этого служит системная хранимая процедура sp_settriggerorder. Порядок срабатывания других триггеров, привязанных к таблице, случаен

К таблице разрешено привязывать триггеры обоих классов. Если в таблице определены ограничения и триггеры обоих классов, то первым из них срабатывает триггер INSTEAD OF, затем обрабатываются ограничения и последним срабатывает AFTER-триггер. При нарушении ограничения выполняется откат действий INSTEAD OF-триггера. Если нарушаются

ограничения или происходят какие-либо другие события, не позволяющие модифицировать таблицу, AFTER-триггер не исполняется.

Создание триггеров DML

```
CREATE TRIGGER имя_триггера
ON имя_таблицы_или_представления
[ WITH ENCRYPTION ]
[ class_триггера тип(ы)_триггера ]
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS sql_инструкции
```

В квадратных скобках указаны опции триггера, которые в данной теме не рассматриваются.

Пояснения к инструкции CREATE TRIGGER:

- a) Триггеры не допускают указания имени базы данных в виде префикса имени объекта. Поэтому перед созданием триггера необходимо выбрать нужную базу данных с помощью конструкции USE *имя_базы_данных* и ключевого слова GO. Ключевое слово GO требуется, поскольку оператор CREATE TRIGGER должен быть первым в пакете. Право на создание триггеров по умолчанию принадлежит владельцу таблицы.
- a) Триггер необходимо привязать к таблице или представлению. Любой триггер можно привязать только к одной таблице или представлению. Чтобы привязать к другой таблице триггер, выполняющий ту же самую задачу, следует создать новый триггер с другим именем, но с той же самой функциональностью и привязать его к другой таблице. AFTER-триггеры (этот класс задан по умолчанию) разрешено привязывать только к таблицам, а триггеры класса INSTEAD OF – как к таблицам, так и к представлениям.
- b) При создании триггера следует задать тип события, вызывающего его срабатывание: INSERT, UPDATE и DELETE. Один и тот же триггер может сработать на одно, два или все три события. Если необходимо, чтобы он срабатывал на все события, то после конструкций FOR, AFTER или INSTEAD OF следует поместить все три ключевых слова: INSERT, UPDATE и DELETE в любом порядке.
- c) Конструкция AS и следующие за ней команды языка T-SQL определяют задачу, которую будет выполнять триггер.

Пример простого триггера AFTER на событие UPDATE

```
USE dbSPJ
GO
IF OBJECT_ID (N'AfterUpdateSPJ', 'TR') IS NOT NULL
    DROP TRIGGER AfterUpdateSPJ
GO
CREATE TRIGGER AfterUpdateSPJ
ON SPJ
AFTER UPDATE
AS
BEGIN
    RAISERROR(N'Произошло обновление в таблице поставок', 1, 1)
END
GO

USE dbSPJ
GO
UPDATE SPJ
SET StartDate = Null
WHERE Sno = 5
GO

Произошло обновление в таблице поставок
Msg 50000, Level 1, State 1

(10 row(s) affected)
```

Пример упрощен, чтобы более ясно продемонстрировать создание задачи в триггере.

Управление триггерами

Для управления триггерами предназначен ряд команд и инструментов баз данных. Триггеры разрешается:

- модифицировать с помощью оператора ALTER TRIGGER;
- переименовать средствами системной хранимой процедуры sp_rename;
- просмотреть путем запроса системных таблиц или с использованием системных хранимых процедур sp_helptrigger или sp_helptext;
- удалить с помощью оператора DROP TRIGGER;
- включить или выключить при помощи конструкций DISABLE TRIGGER и ENABLE TRIGGER оператора ALTER TABLE.

Программирование триггеров

A. Псевдотаблицы Inserted и Deleted

При срабатывании триггера на события INSERT, UPDATE или DELETE создается одна или несколько псевдотаблиц (также известных как логические таблицы). Можно рассматривать логические таблицы как журналы транзакций для события. Существует два типа логических таблиц: **Inserted** и **Deleted**. **Inserted** создается в результате события добавления или обновления данных. В ней находится набор добавленных или измененных записей. UPDATE-триггер создает также логическую таблицу **Deleted**. В ней находится исходный набор записей в том состоянии, в каком он был до операции обновления. Следующий пример создает триггер, который выводит содержимое **Inserted** и **Deleted** после события UPDATE в таблице S:

```
USE dbSPJ
GO
IF OBJECT_ID ('dbo.UpdateTables', 'TR') IS NOT NULL
    DROP TRIGGER dbo.UpdateTables
GO
CREATE TRIGGER dbo.UpdateTables
ON S
AFTER UPDATE
AS
SELECT * FROM inserted
SELECT * FROM deleted
GO
```

После исполнения простой инструкции UPDATE, изменяющей имя поставщика с 'Алмаз' на 'Графит'

```
USE dbSPJ
GO
UPDATE S set Sname='Графит'
WHERE Sname='Алмаз'
GO
```

срабатывает триггер, который выводит следующие результаты:

```
1      Графит      20      Смоленск
1      Алмаз        20      Смоленск
```

После исполнения триггера обновления в таблице S содержится обновленная запись. При срабатывании триггера возможен откат модификаций таблицы S, если в триггере запрограммированы соответствующие действия. Откат транзакции также предусмотрен для триггеров INSERT и DELETE. При срабатывании триггера на удаление набор удаленных записей помещается в логическую таблицу **Deleted**. Таблица **Inserted** не участвует в событии DELETE.

B. Конструкции UPDATE (имя-столбца) и COLUMNS_UPDATED()

Важными компонентами операторов CREATE TRIGGER и ALTER TRIGGER являются две конструкции – UPDATE(*имя-столбца*) и (COLUMN_UPDATED()). Эти конструкции разрешается включать в UPDATE- и INSERT-триггеры, и они могут располагаться в любом месте оператора CREATE TRIGGER или ALTER TRIGGER.

Конструкция IF UPDATE (*имя-столбца*) определяет, произошло ли в столбце *имя-столбца* событие INSERT или UPDATE. Если нужно задать несколько столбцов, следует разделить их конструкциями UPDATE (*имя-столбца*). Например, следующий фрагмент кода проверяет, выполнено ли добавление или обновление в столбцах First_Name и Last_Name, и выполняет некоторые действия над этими столбцами в результате событий INSERT или UPDATE:

```
USE Northwind
GO
CREATE TABLE Orders_Audit (
    OrderID INT ,
    CustomerID NCHAR (5) NULL ,
    EmployeeID INT NULL ,
    OrderDate DATETIME NULL ,
    RequiredDate DATETIME NULL ,
    ShippedDate DATETIME NULL ,
    ShipVia INT NULL ,
    Freight MONEY NULL ,
    ShipName NVARCHAR (40) NULL ,
    ShipAddress NVARCHAR (60) NULL ,
    ShipCity NVARCHAR (15) NULL ,
    ShipRegion NVARCHAR (15) NULL ,
    ShipPostalCode NVARCHAR (10) NULL ,
    ShipCountry NVARCHAR (15) NULL ,
    DateAdded DATETIME NOT NULL DEFAULT GETDATE()
)
CREATE TRIGGER tr_iud_Orders
ON Orders
AFTER INSERT,UPDATE,DELETE
AS
BEGIN
-- If either of these two dates altered, then create the audit record
-- If either are not altered, then no audit record
    IF UPDATE( ShippedDate ) OR UPDATE( RequiredDate ) THEN
        INSERT INTO Orders_Audit ( OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate,
ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode,
ShipCountry )
            SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate,
ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
            FROM deleted
END
```

Если подставить значение вместо *имя_столбца*, конструкция UPDATE вернет **true**. Конструкция COLUMN_UPDATED() также проверяет, произошло ли обновление столбцов. Вместо **true** или **false** конструкция COLUMN_UPDATED() возвращает битовую маску типа **varbinary**, описывающую столбцы, в которых выполнено добавление или обновление.

Написать конструкцию (COLUMN_UPDATED()) сложнее, чем UPDATE (*имя-столбца*), зато она позволяет точно определить, в каких из проверенных столбцов добавлены или обновлены данные. Подлежащие проверке столбцы задают с помощью маски, представляющей номер (порядковый) каждого столбца в таблице. В расположенной далее таблице показаны первые восемь столбцов и назначенные им маски.

Столбец	1	2	3	4	5	6	7	8
Битовая маска	1	2	4	8	16	32	64	128

Следующий код позволят проверить, были ли добавлены или обновлены данные в столбцах 4 или 6:

```
IF (COLUMN_UPDATED() & 40) > 0
```

Значение 40 – это результат суммирования маски 8 для столбца 4 и маски 32 для столбца 6. Выражение проверяет, действительно ли значение COLUMN_UPDATED() больше 0. Другими словами, условие выполняется, если хотя бы один из двух или оба столбца обновлены. Если задать условие

```
IF (COLUMN_UPDATED() & 40) = 40
```

то проверка обновления выполняется в обоих столбцах. Если обновление произошло только в одном из столбцов, условие не выполнится.

Чтобы проверить девять и больше столбцов, следует использовать функцию SUBSTRING, которая позволяет указать триггеру маску, подлежащую проверке. Например, следующий код позволяет проверить, обновлен ли девятый столбец:

```
IF ((SUBSTRING(COLUMN_UPDATED(), 2, 1)=1))
```

Функция SUBSTRING заставляет конструкцию COLUMN_UPDATED() перейти ко второму октету столбцов и проверить, обновлен ли первый столбец второго октета (его реальный порядковый номер равен 9). Для этого столбца возвращается значение типа varbinary, равное 1. В показанной далее таблице иллюстрируется принцип действия функции SUBSTRING, необходимой для проверки столбцов с 9 по 16.

```
IF ((SUBSTRING(COLUMN_UPDATED(), 2, y)=z))
```

Столбец	9	10	11	12	13	14	15	16
Значения у и z	1	2	4	8	16	32	64	128

Чтобы проверить несколько столбцов на предмет модификации, следует просто сложить значения битовой маски для каждого из них. Например, чтобы проверить столбцы 14 и 16, нужно задать для z выражение 160 (32 + 128).

Функции и системные команды

Для реализации бизнес-логики в триггерах предназначены различные функции и системные команды.

1. В триггерах часто используется функция @@ROWCOUNT. Она возвращает число строк, на которое повлияло исполнение предыдущего оператора T-SQL.
2. Триггер срабатывает на событие INSERT, UPDATE или DELETE, даже если при этом не изменяется ни одна строка. Поэтому для выхода из триггера при отсутствии модификаций таблицы используется системная команда RETURN.
3. В случае возникновения ошибки иногда требуется вывести сообщение с описанием ее причины. Для вывода сообщений об ошибках используется системная команда RAISERROR.
4. Пользовательские сообщения об ошибках создают с помощью системной хранимой процедуры sp_addmessage или выводят встроенные сообщения при вызове системной команды RAISERROR. За дополнительной информацией о системной хранимой процедуре sp_addmessage обращайтесь в SQL Server Books Online.
5. В триггерах, написанных на языке Transact-SQL, также иногда применяют системную команду ROLLBACK TRANSACTION. Она вызывает откат всего пакета триггера. При фатальной ошибке также происходит откат, но в неявном виде. Если задачей триггера является завершение транзакции в любом случае (кроме тех, когда во время транзакции возникает фатальная ошибка), то в код триггера можно не включать системную команду ROLLBACK TRANSACTION.

Использование триггера для поддержки простой ссылочной целостности

Следует заметить, что использовать в этих целях триггеры — не всегда лучшее решение. Предположим, что в демонстрационной базе данных Northwind не действуют никакие декларативные правила по поддержке ссылочной целостности (DRI). Для этого необходимо сформировать копию базы данных Northwind, в которой будут отсутствовать DRI. Пусть это будет база данных NorthwindTriggers. В этом случае появляется пространство для экспериментов. Займемся отношениями, которые в оригинальной базе данных Northwind обеспечивались при помощи DRI. Восстановим отношение между таблицами Customers и Orders. При помощи данного отношения грантируется, что нельзя добавить заказ в таблицу Orders, если он ссылается на CustomerID, отсутствующий в таблице Customers. Чтобы реализовать это ограничение, создадим следующий триггер:

```
CREATE TRIGGER OrderHasCustomer  
ON Orders
```

```
FOR INSERT, UPDATE
AS
IF EXISTS
(
    SELECT 'True'
    FROM Inserted i LEFT JOIN Customers c ON i.CustomerID = c.CustomerID
    WHERE c.CustomerID IS NULL
)
BEGIN
    RAISERROR('Счет должен иметь валидный CustomerID',16,1)
    ROLLBACK TRAN
END
```

Теперь, если попытаться добавить в таблицу `Orders` данные, не удовлетворяющие условиям, заданным в триггере:

```
INSERT Orders
DEFAULT VALUES
```

то будет получено сообщение об ошибке:

```
Сообщение 50000, уровень 16, состояние 1, процедура OrderHasCustomer, строка 15
Счет должен иметь валидный CustomerID
Сообщение 3609, уровень 16, состояние 1, строка 1
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Данный триггер можно усовершенствовать, чтобы повысить его информативность.

Хранимые процедуры T-SQL

Хранимая процедура – именованный объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры похожи на обычные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные. В хранимых процедурах могут выполняться операторы DDL, DML, TCL, FCL. Процедуры можно создавать для постоянного использования, для временного использования в одном сеансе (локальная временная процедура), для временного использования во всех сеансах (глобальная временная процедура). Хранимые процедуры могут выполняться автоматически при запуске экземпляра SQL Server.

Создание хранимых процедур

Для создания хранимой процедуры используется инструкция CREATE PROCEDURE, имеющая следующий синтаксис:

```
CREATE PROCEDURE [ имя-схемы. ] имя-процедуры [ список-объявлений-параметров ]
[ WITH список-опций-процедуры ]
[ FOR REPLICATION ]
AS
тело-процедуры
[ ; ]
```

где

- a) Если имя схемы не указано при создании процедуры, то автоматически назначается схема по умолчанию для пользователя, который создает процедуру.
- b) Список объявлений параметров является необязательным.
- c) Объявление параметра в списке объявлений параметров имеет вид:

```
@имя-параметра тип-данных [ VARYING ] [ = значение-по-умолчанию ] [ OUT | OUTPUT ] [READONLY]
```

- d) Параметрами процедуры могут быть любые типы данных, за исключением **table**.
- e) Для создания параметров, возвращающих табличное значение, можно использовать определяемый пользователем табличный тип. Возвращающие табличное значение параметры могут быть только входными и должны сопровождаться ключевым словом **READONLY**.
- f) Тип данных **cursor** может быть использован только в качестве выходного параметра.
- g) **VARYING** применяется только к аргументам типа **cursor**.
- h) **OUT** или **OUTPUT** показывает, что параметр процедуры является выходным. Параметры типов **text**, **ntext** и **image** не могут быть выходными.
- i) **READONLY** указывает, что параметр не может быть обновлен или изменен в тексте процедуры.
- j) Опциями функции могут быть:

ENCRYPTION - SQL Server шифрует определение процедуры.

RECOMPILE - SQL Server перекомпилирует процедуру при каждом ее выполнении.

Предложение EXECUTE AS - определяет контекст безопасности, в котором должна быть выполнена процедура.

- k) Тело процедуры - одна или несколько инструкций T-SQL. Инструкции можно заключить в необязательные ключевые слова **BEGIN** и **END**.
- l) **FOR REPLICATION** указывает, что процедура создается для репликации.
- m) Тело процедуры может содержать оператор **RETURN**, возвращающий целочисленное значение вызывающей процедуре или приложению.

Примечания.

1. Локальную временную процедуру можно создать, указав один символ номера (#) перед именем процедуры.
2. Глобальную временную процедуру можно создать, указав два символа номера (##) перед именем процедуры.
3. Временные процедуры создаются в базе данных **tempdb**.
4. Рекомендуется начинать текст процедуры с инструкции **SET NOCOUNT ON** (она должна следовать сразу за ключевым словом **AS**). В этом случае отключаются сообщения, отправляемые SQL Server клиенту после выполнения любых инструкций **SELECT**, **INSERT**, **UPDATE**, **MERGE** и **DELETE**.

Выполнение хранимых процедур

При выполнении процедуры в первый раз она компилируется, при этом определяется оптимальный план получения данных. При последующих вызовах процедуры может быть повторно использован уже созданный план, если он еще находится в кэше планов компонента Database Engine.

Одна процедура может вызывать другую. Уровень вложенности увеличивается на 1, когда начинается выполнение вызванной процедуры, и уменьшается на 1, когда вызванная процедура завершается. Уровень вложенности процедур может достигать 32. Текущий уровень вложенности процедур можно получить при помощи функции @@NESTLEVEL.

Чтобы выполнить процедуру, надо использовать инструкцию EXECUTE. Также можно выполнить процедуру без использования ключевого слова EXECUTE, если процедура является первой инструкцией в пакете.

При выполнении процедуры (в пакете или внутри хранимой процедуры или функции) **настоятельно рекомендуется уточнять имя хранимой процедуры указанием, по крайней мере, имени схемы.**

Примеры создания и выполнения хранимых процедур

Пример процедуры без параметров

```
IF OBJECT_ID ( N'dbo.TestDataGenerator', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.TestDataGenerator
GO
CREATE PROCEDURE TestDataGenerator
AS
    DECLARE @count INT
    IF OBJECT_ID('dbo.TestDataTable') IS NULL
        CREATE TABLE dbo.TestDataTable
        (
            ID INT PRIMARY KEY,
            Name VARCHAR(255)
        )
    SELECT @count=1
    WHILE @count<100
    BEGIN
        INSERT TestDataTable(ID, Name)
        SELECT @count, REPLICATE(CHAR((@count%26)+65), @count%255)
        SELECT @count=@count+1
    END
GO
```

Пример процедуры с входными и выходными параметрами

```
USE dbSPJ
GO
IF OBJECT_ID ( N'dbo.Factorial', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.Factorial
GO
CREATE PROCEDURE dbo.Factorial @ValIn bigint, @ValOut bigint output
AS
BEGIN
    IF @ValIn > 20
    BEGIN
        PRINT N'Входной параметр должен быть <= 20'
        RETURN -99
    END
    DECLARE @WorkValIn bigint, @WorkValOut bigint
    IF @ValIn != 1
    BEGIN
        SET @WorkValIn = @ValIn - 1
        PRINT @@NESTLEVEL
        EXEC dbo.Factorial @WorkValIn, @WorkValOut OUTPUT
        SET @ValOut = @WorkValOut * @ValIn
    END
END
```

```

    END
    ELSE
        SET @ValOut = 1
END
GO
DECLARE @FactIn int, @FactOut int
SET @FactIn = 8
EXEC dbo.Factorial @FactIn, @FactOut OUTPUT
PRINT N'Факториал ' + CONVERT(varchar(3),@FactIn) + N' равен ' +
    CONVERT(varchar(20),@FactOut)

```

Пример процедуры без параметров, но возвращающая значение

```

USE dbSPJ
GO
IF OBJECT_ID ( N'dbo.SelectShipments', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.SelectShipments
GO
CREATE PROCEDURE dbo.SelectShipments
AS
BEGIN
    DECLARE @Rc INT
    SELECT * FROM SPJ
    SET @Rc = @@ROWCOUNT
    RETURN @Rc
END
GO
DECLARE @RcRet INT
EXEC @RcRet = dbo.SelectShipments
SELECT @RcRet "Количество строк"
GO

```

Пример процедуры с выходным параметром и возвращающая значение

```

USE dbSPJ
GO
IF OBJECT_ID ( N'dbo.SelectShipmentsWithOutput', 'P' ) IS NOT NULL
    DROP PROCEDURE dbo.SelectShipmentsWithOutput
GO
CREATE PROCEDURE dbo.SelectShipmentsWithOutput @Rcnt INT OUTPUT
AS
BEGIN
    DECLARE @Rc INT
    SELECT * FROM SPJ
    SET @Rcnt = @@ROWCOUNT
    RETURN 1
END
GO
DECLARE @OutParm INT, @RetVal INT
EXEC @RetVal = dbo.SelectShipmentsWithOutput @OutParm OUTPUT
SELECT @OutParm "Выходной параметр", @RetVal "Возвращаемое значение"
GO

```

Изменение хранимых процедур

Если нужно изменить инструкции или параметры хранимой процедуры, можно или удалить (DROP PROCEDURE) и создать ее заново (CREATE PROCEDURE), или изменить ее за один шаг (ALTER PROCEDURE). При удалении и повторном создании хранимой процедуры все разрешения, связанные с ней, будут потеряны при восстановлении. При изменении хранимой процедуры ее определение или определение ее параметров меняются, но разрешения, связанные с ней, остаются и все зависящие от нее процедуры или триггеры не затрагиваются. Хранимую процедуру можно также

изменить таким образом, чтобы ее определение было зашифровано или чтобы она компилировалась заново при каждом запуске.

Курсоры T-SQL

Операции в реляционной базе данных выполняются над множеством строк. Набор строк, возвращаемый инструкцией SELECT, содержит все строки, которые удовлетворяют условиям, указанным в предложении WHERE. Такой полный набор строк, возвращаемых инструкцией, называется результирующим набором. Приложения, особенно интерактивные, не всегда эффективно работают с результирующим набором как с единым целым. Им нужен механизм, позволяющий обрабатывать одну строку или небольшое их число за один раз. Курсоры предоставляют такой механизм.

Microsoft SQL Server поддерживает три способа реализации курсоров:

1. **Курсыры T-SQL.** Основываются на синтаксисе DECLARE CURSOR и используются главным образом в сценариях T-SQL, хранимых процедурах и триггерах. Курсоры T-SQL реализуются на сервере и управляются инструкциями T-SQL, направляемыми клиентом серверу. Они также могут содержаться в пакетах, хранимых процедурах или триггерах.
2. **Серверные курсоры интерфейса прикладного программирования (API).** Поддерживают функции курсоров API в OLE DB и ODBC. Курсоры API реализуются на сервере. Всякий раз, когда клиентское приложение вызывает функцию курсора API, поставщик OLE DB или драйвер ODBC для собственного клиента SQL передает требование на сервер для выполнения действия в отношении серверного курсора API.
3. **Клиентские курсоры.** Реализуются внутренне драйвером ODBC для собственного клиента SQL и DLL, реализующим ADO API. Клиентские курсоры реализуются посредством кэширования всех строк результирующего набора на клиенте. Всякий раз, когда клиентское приложение вызывает функцию курсора API, драйвер ODBC для собственного клиента SQL или ADO DLL выполняет операцию курсора на строках результирующего набора, кэшированных на клиенте.

Курсыры T-SQL и серверные курсоры API собирательно называются серверными курсорами.

Методика использования курсора языка Transact-SQL

Методика использования курсора языка T-SQL такова:

1. С помощью инструкции DECLARE необходимо объявить переменные T-SQL, которые будут содержать данные, возвращенные курсором. Для каждого столбца результирующего набора надо объявить по одной переменной. Переменные должны быть достаточно большого размера для хранения значений, возвращаемых в столбце и имеющих тип данных, к которому могут быть неявно преобразованы данные столбца.
2. С помощью инструкции DECLARE CURSOR необходимо связать курсор T-SQL с инструкцией SELECT. Инструкция DECLARE CURSOR определяет также характеристики курсора, например имя курсора и тип курсора (read-only или forward-only).
3. С помощью инструкции OPEN выполнить инструкцию SELECT и заполнить курсор.
4. С помощью инструкции FETCH INTO организовать перемещение по курсору для выборки отдельных строк; значение каждого столбца отдельной строки заносится в указанную переменную. После этого другие инструкции T-SQL могут ссылаться на эти переменные для доступа к выбранным значениям данных. Курсоры T-SQL не поддерживают выборку группы строк.
5. После завершения работы с курсором необходимо применить инструкцию CLOSE. Закрытие курсора освобождает некоторые ресурсы, например результирующий набор курсора и его блокировки на текущей строке, однако структура курсора будет доступна для обработки, если снова выполнить инструкцию OPEN. Поскольку курсор все еще существует на этом этапе, повторно использовать его имя нельзя.
6. Наконец инструкция DEALLOCATE полностью освобождает все ресурсы, выделенные курсору, в том числе имя курсора. После освобождения курсора его необходимо строить заново с помощью инструкции DECLARE CURSOR.

Пример

```
USE dbSPJ
GO
-- Объявляем переменную
DECLARE @TableName varchar(255)
-- Объявляем курсор
DECLARE TableCursor CURSOR FOR
    SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_TYPE = 'BASE TABLE'
-- Открываем курсор и выполняем извлечение первой записи
OPEN TableCursor
FETCH NEXT FROM TableCursor INTO @TableName
-- Проходим в цикле все записи из множества
WHILE @@FETCH_STATUS = 0
```

```

BEGIN
    PRINT @TableName
    FETCH NEXT FROM TableCursor INTO @TableName
END
-- Убираем за собой «хвосты»
CLOSE TableCursor
DEALLOCATE TableCursor

```

Рекомендация. Примените этот курсор к своей базе данных и проанализируйте результаты его исполнения.

В SQL Server поддерживаются четыре типа серверных курсоров:

- *Статические курсоры (STATIC)*. Создается временная копия данных для использования курсором. Все запросы к курсору обращаются к указанной временной таблице в базе данных tempdb, поэтому изменения базовых таблиц не влияют на данные, возвращаемые выборками для данного курсора, а сам курсор не позволяет производить изменения.
- *Динамические курсоры (DYNAMIC)*. Отображают все изменения данных, сделанные в строках результирующего набора при просмотре этого курсора. Значения данных, порядок, а также членство строк в каждой выборке могут меняться. Параметр выборки ABSOLUTE динамическими курсорами не поддерживается.
- *Курсоры, управляемые набором ключей (KEYSET)*. Членство или порядок строк в курсоре не изменяются после его открытия. Набор ключей, однозначно определяющих строки, встроен в таблицу в базе данных tempdb с именем keyset.
- *Быстрые последовательные курсоры (FAST_FORWARD)*. Параметр FAST_FORWARD указывает курсор FORWARD_ONLY, READ_ONLY, для которого включена оптимизация производительности. Параметр FAST_FORWARD не может указываться вместе с параметрами SCROLL или FOR_UPDATE.

Статическими курсорами обнаруживаются лишь некоторые изменения или не обнаруживаются вовсе, но при этом в процессе прокрутки такие курсоры потребляют сравнительно мало ресурсов. Динамические курсоры обнаруживают все изменения, но потребляют больше ресурсов при прокрутке. Управляемые набором ключей курсоры имеют промежуточные свойства, обнаруживая большинство изменений, но потребляя меньше ресурсов, чем динамические курсоры.

По области видимости имени курсора различают:

- *Локальные курсоры (LOCAL)*. Область курсора локальна по отношению к пакету, хранимой процедуре или триггеру, в которых этот курсор был создан. Курсор неявно освобождается после завершения выполнения пакета, хранимой процедуры или триггера, за исключением случая, когда курсор был передан параметру OUTPUT. В этом случае курсор освобождается при освобождении всех ссылающихся на него переменных или при выходе из области видимости.
- *Глобальные курсоры (GLOBAL)*. Область курсора является глобальной по отношению к соединению. Имя курсора может использоваться любой хранимой процедурой или пакетом, которые выполняются соединением. Курсор неявно освобождается только в случае разрыва соединения. Глобальность курсора позволяет создавать его в рамках одной хранимой процедуры, а вызывать его из совершенно другой процедуры, причем передавать его в эту процедуру необязательно.

По способу перемещения по курсору различают:

- *Последовательные курсоры (FORWARD_ONLY)*. Курсор может просматриваться только от первой строки к последней. Поддерживается только параметр выборки FETCH NEXT. Если параметр FORWARD_ONLY указан без ключевых слов STATIC, KEYSET или DYNAMIC, то курсор работает как DYNAMIC. Если не указан ни один из параметров FORWARD_ONLY или SCROLL, а также не указано ни одно из ключевых слов STATIC, KEYSET или DYNAMIC, то по умолчанию задается параметр FORWARD_ONLY. Курсоры STATIC, KEYSET и DYNAMIC имеют значение по умолчанию SCROLL.
- *Курсы прокрутки (SCROLL)*. Перемещение осуществляется по группе записей как вперед, так и назад. В этом случае доступны все параметры выборки (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Параметр SCROLL не может указываться вместе с параметром для FAST_FORWARD.

По способу распараллеливания курсоров различают:

- READ_ONLY. Содержимое курсора можно только считывать.
- SCROLL_LOCKS. При редактировании данной записи вами никто другой вносить в нее изменения не может. Такую блокировку прокрутки иногда еще называют «пессимистической» блокировкой. При блокировке прокрутки важным фактором является продолжительность действия блокировки. Если курсор не входит в состав некоторой транзакции, то блокировка распространяется только на текущую запись в курсоре. В противном случае все зависит от того, какой используется уровень изоляции транзакций.
- OPTIMISTIC. Означает отсутствие каких бы то ни было блокировок. «Оптимистическая» блокировка предполагает, что даже во время выполнения вами редактирования данных, другие пользователи смогут к ним обращаться. Если

кто-то все-таки попытается выполнить модификацию данных одновременно с вами, генерируется ошибка 16394. В этом случае вам придется повторно выполнить доставку данных из курсора или же осуществить полный откат транзакции, а затем попытаться выполнить ее еще раз.

Для выявления ситуаций с преобразованием типа курсора используется опция TYPE_WARNING, которая указывает, что клиенту посыпается предупреждающее сообщение при неявном преобразовании типа курсора из запрошенного типа в другой тип.

По умолчанию курсор, разрешающий изменение данных, позволяет выполнять модификацию любых столбцов в своем составе. Опция FOR UPDATE указывает столбцы курсора, которые можно изменять. Все остальные столбцы становятся доступными только для чтения. Если опция FOR UPDATE используется без списка столбцов, то обновление возможно для всех столбцов, за исключением случая, когда был указан параметр параллелизма READ_ONLY.

На содержательном уровне синтаксис инструкции DECLARE CURSOR будет иметь вид:

```
DECLARE имя-курсора CURSOR  
[ область-видимости-имени-курсора ]  
[ возможность-перемещения-по-курсопу ]  
[ типы-курсоров ]  
[ опции-распараллеливания-курсопов ]  
[ выявление-ситуаций-с-преобразованием-типа-курсопа ]  
FOR инструкция_select  
[ опция-FOR-UPDATE ]
```

Формальный синтаксис инструкции DECLARE CURSOR имеет вид:

```
DECLARE имя-курсора CURSOR  
[ LOCAL | GLOBAL ]  
[ FORWARD_ONLY | SCROLL ]  
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]  
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
[ TYPE_WARNING ]  
FOR инструкция_select  
[ FOR UPDATE [ OF список-имен_столбцов ] ]
```

Использование переменной типа cursor

Microsoft SQL Server поддерживает переменные с типом данных CURSOR. Курсор может быть связан с переменной типа CURSOR двумя способами, например:

1.

```
DECLARE @MyVariable CURSOR  
DECLARE MyCursor CURSOR FOR  
SELECT LastName FROM AdventureWorks.Person.Contact  
SET @MyVariable = MyCursor;
```
2.

```
DECLARE @MyVariable CURSOR  
SET @MyVariable = CURSOR SCROLL KEYSET FOR  
SELECT LastName FROM AdventureWorks.Person.Contact;
```

После связи курсора с переменной типа CURSOR эта переменная может использоваться в инструкциях курсора языка T-SQL вместо имени курсора. Кроме того, выходным параметрам хранимой процедуры можно назначить тип данных CURSOR и связать их с курсором. Это позволяет управлять локальными курсорами из хранимых процедур.

Перемещение внутри курсора (прокрутка курсора)

Основой всех операций прокрутки курсора является ключевое слово FETCH. Оно может использоваться для перемещения по курсору в обоих направлениях, в том числе и для перехода к заданной позиции. В качестве аргументов оператора FETCH могут выступать:

- NEXT – возвращает строку результата сразу же за текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH NEXT выполняет первую выборку в отношении курсора, она возвращает первую строку в результирующем наборе. NEXT является параметром по умолчанию выборки из курсора.

- PRIOR – возвращает строку результата, находящуюся непосредственно перед текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция FETCH PRIOR выполняет первую выборку из курсора, не возвращается никакая строка и положение курсора остается перед первой строкой.
- FIRST – возвращает первую строку в курсоре и делает ее текущей.
- LAST – возвращает последнюю строку в курсоре, и делает ее текущей.
- ABSOLUTE { n | @nvar }. Если n или @nvar имеют положительное значение, возвращает строку, стоящую дальше на n строк от передней границы курсора, и делает возвращенную строку новой текущей строкой. Если n или @nvar имеют отрицательное значение, возвращает строку, отстоящую на n строк от задней границы курсора, и делает возвращенную строку новой текущей строкой. Если n или @nvar равны 0, не возвращается никакая строка. n должно быть целым числом, а @nvar должна иметь тип данных smallint, tinyint или int.
- RELATIVE { n | @nvar }. Если n или @nvar имеют положительное значение, возвращает строку, отстоящую на n строк вперед от текущей строки, и делает возвращенную строку новой текущей строкой. Если n или @nvar имеют отрицательное значение, возвращает строку, отстоящую на n строк назад от текущей строки, и делает возвращенную строку новой текущей строкой. Если n или @nvar равны 0, возвращает текущую строку. Если при первой выборке из курсора инструкция FETCH RELATIVE указывается с отрицательными или равными нулю параметрами n или @nvar, то никакая строка не возвращается. n должно быть целым числом, а @nvar должна иметь тип данных smallint, tinyint или int.

Опциями оператора FETCH являются:

- GLOBAL – указывает, что параметр *имя-курсора* ссылается на глобальный курсор.
- INTO – позволяет поместить данные из столбцов выборки в локальные переменные. Каждая переменная из списка, слева направо, связывается с соответствующим столбцом в результирующем наборе курсора. Тип данных каждой переменной должен соответствовать типу данных соответствующего столбца результирующего набора, или должна обеспечиваться поддержка неявного преобразования в тип данных этого столбца. Количество переменных должно совпадать с количеством столбцов в списке выбора курсора.

Синтаксис инструкции FETCH имеет вид:

```
FETCH
  [ [ NEXT | PRIOR | FIRST | LAST
    | ABSOLUTE { n | @nvar }
    | RELATIVE { n | @nvar }
  ]
  FROM
  ]
{ { [ GLOBAL ] имя-объявленного-курсора } | @имя-переменной-курсора }
[ INTO список-имен-переменных ]
```

После выполнения каждой инструкции FETCH функция @@FETCH_STATUS обновляется, отражая состояние последней выборки. Функция @@FETCH_STATUS показывает такие состояния, как выборка за пределами первой или последней строк курсора. Функция @@FETCH_STATUS глобальна для соединения и обновляется после любой выборки в любом курсоре, открытом во время соединения. Если состояние нужно посмотреть позже, то перед выполнением следующей инструкции в пределах соединения необходимо сохранить значение функции @@FETCH_STATUS в пользовательской переменной. Даже если следующей инструкцией будет не FETCH, а INSERT, UPDATE или DELETE, сработает триггер, содержащий инструкции FETCH, что приведет к обновлению значения функции @@FETCH_STATUS.

Примеры, демонстрирующие различие между STATIC и DYNAMIC курсорами

```
USE Northwind
-- Создаем таблицу специально для данного примера
SELECT OrderID, CustomerID
INTO CursorTable
FROM Orders
WHERE OrderID BETWEEN 10701 AND 10705
-- Объявляем курсор
DECLARE CursorTest CURSOR
GLOBAL
SCROLL
STATIC
FOR
SELECT OrderID, CustomerID
FROM CursorTable
```

```

-- Объявляем переменные для хранения
DECLARE @OrderID      int
DECLARE @CustomerID   varchar(5)

-- Откроем курсор и запросим первую запись
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
-- Обрабатаем в цикле все записи курсора
WHILE @@FETCH_STATUS=0
BEGIN
    PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Внесем изменения. Далее мы увидим, что эти изменения не будут отражены в курсоре
UPDATE CursorTable
    SET CustomerID = 'XXXXX'
    WHERE OrderID = 10703
-- Проверяем, что изменения были в действительности внесены в таблицу
SELECT OrderID, CustomerID
FROM CursorTable
-- Вернемся к началу списка
FETCH FIRST FROM CursorTest INTO @OrderID, @CustomerID
-- Еще раз пройдемся по списку записей
WHILE @@FETCH_STATUS=0
BEGIN
    PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Теперь уберем за собой «хвосты»
CLOSE CursorTest
DEALLOCATE CursorTest
DROP TABLE CursorTable

```

Сообщения
(строк обработано: 5)

10701	HUNGO
10702	ALFKI
10703	FOLKO
10704	QUEEN
10705	HILAA

(строк обработано: 1)

10701	HUNGO
10702	ALFKI
10703	FOLKO
10704	QUEEN
10705	HILAA

Результаты

10701	HUNGO
10702	ALFKI
10703	XXXXX
10704	QUEEN
10705	HILAA

```

USE Northwind
-- Создадим таблицу специально для данного примера
SELECT OrderID, CustomerID
INTO CursorTable
FROM Orders
WHERE OrderID BETWEEN 10701 AND 10705

```

```

-- Создадим уникальный индекс в виде первичного ключа
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
        PRIMARY KEY (OrderID)
/* Свойство IDENTITY автоматически устанавливается при выполнении инструкции SELECT INTO, но
поскольку в качестве уникального значения мы хотим использовать OrderID, нужно явно включить
опцию IDENTITY_INSERT, для того чтобы переопределить назначаемое по умолчанию identity-
значение. */
SET IDENTITY_INSERT CursorTable ON
-- Объявим курсор
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see if the changes are there
DYNAMIC         -- This is what we're testing this time
FOR
SELECT OrderID, CustomerID
FROM CursorTable
-- Объявим переменные для хранения
DECLARE @OrderID      int
DECLARE @CustomerID   varchar(5)
-- Откроем курсор и запросим первую запись
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
-- Обработаем в цикле все записи курсора
WHILE @@FETCH_STATUS=0
BEGIN
    PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Внесем в таблицу изменения. We'll see that it does affect the cursor this time.
UPDATE CursorTable
    SET CustomerID = 'XXXXX'
    WHERE OrderID = 10703
-- Удалим запись, чтобы проверить, отразится ли это на курсоре
DELETE CursorTable
    WHERE OrderID = 10704
-- Вставим новую запись. Но теперь курсор ее увидит
INSERT INTO CursorTable
    (OrderID, CustomerID)
VALUES
    (99999, 'IIIII')
-- Проверим, что изменения были в действительности внесены в таблицу
SELECT OrderID, CustomerID
FROM CursorTable
-- Вернемся к началу списка
FETCH FIRST FROM CursorTest INTO @OrderID, @CustomerID
-- Еще раз пройдемся по списку записей.
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.'
    END
    ELSE
    BEGIN
        PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    END
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Теперь уберем за собой «хвосты»
CLOSE CursorTest
DEALLOCATE CursorTest
DROP TABLE CursorTable

```

Сообщения

(строк обработано: 5)

```
10701 HUNGO
10702 ALFKI
10703 FOLKO
10704 QUEEN
10705 HILAA
```

(строк обработано: 1)

(строк обработано: 1)

(строк обработано: 1)

(строк обработано: 5)

```
10701 HUNGO
10702 ALFKI
10703 XXXXX
10705 HILAA
99999 IIIEEE
```

Результаты

```
10701 HUNGO
10702 ALFKI
10703 XXXXX
10705 HILAA
99999 IIIEEE
```

Наблюдение за активностью курсора T-SQL

Для получения списка курсоров, видимых в текущем соединении, используется системная хранимая процедура `sp_cursor_list`, а для определения характеристик курсора используются процедуры `sp_describe_cursor`, `sp_describe_cursor_columns` и `sp_describe_cursor_tables`.

После открытия курсора столбец `cursor_rows`, возвращенный процедурами `sp_cursor_list` или `sp_describe_cursor`, показывают количество строк в курсоре.

После выполнения каждой инструкции `FETCH` сведения о состоянии последней выборки имеются в столбце `fetch_status`, возвращенном процедурой `sp_describe_cursor`. Столбец `fetch_status`, возвращенный процедурой `sp_describe_cursor`, относится только к указанному курсору и не зависит от инструкций `FETCH`, ссылающихся на другие курсоры. Однако процедура `sp_describe_cursor` зависит от инструкций `FETCH`, ссылающихся на тот же курсор, поэтому при ее использовании следует соблюдать осторожность.

Изменения данных непосредственно в курсоре

Для того чтобы выполнить обновление, необходимо воспользоваться специальным синтаксисом инструкций `UPDATE` и `DELETE`, в которых предложение `WHERE` имеет вид:

`WHERE CURRENT OF имя_курсора`

Пример

```
USE Northwind
-- Создадим таблицу специально для данного примера
SELECT OrderID, CustomerID
INTO CursorTable
FROM Orders
WHERE OrderID BETWEEN 10701 AND 10705
-- Создадим уникальный индекс в виде первичного ключа
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
    PRIMARY KEY (OrderID)
```

```

/* Свойство IDENTITY автоматически устанавливается при выполнении инструкции SELECT INTO, но
поскольку в качестве уникального значения мы хотим использовать OrderID, нужно явно включить
опцию IDENTITY_INSERT, для того чтобы переопределить назначаемое по умолчанию identity-
значение. */
SET IDENTITY_INSERT CursorTable ON
-- Объявим курсор
DECLARE CursorTest CURSOR
SCROLL          -- Чтобы перемещаться по курсору и видеть изменения
KEYSET
FOR
SELECT OrderID, CustomerID
FROM CursorTable
-- Объявим переменные для хранения
DECLARE @OrderID      int
DECLARE @CustomerID   varchar(5)
-- Откроем курсор и запросим первую запись
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
-- Обрабатываем в цикле все записи курсора
WHILE @@FETCH_STATUS=0
BEGIN
    IF (@OrderID % 2 = 0)      -- Обновим четные строки
    BEGIN
        -- Внесем изменения при помощи курсора
        UPDATE CursorTable
            SET CustomerID = 'EVEN'
            WHERE CURRENT OF CursorTest
    END
    ELSE                      -- Удалим нечетные строки
    BEGIN
        -- Удалим текущую строку курсора
        DELETE CursorTable
            WHERE CURRENT OF CursorTest
    END
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Поскольку в этом курсоре разрешена прокрутка в любом направлении,
-- то можно опять перейти к началу результирующего множества
FETCH FIRST FROM CursorTest INTO @OrderID, @CustomerID
-- И вновь в цикле обрабатываем все строки
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.'
    END
    ELSE
    BEGIN
        PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    END
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END
-- Самое время удалить за собой все «хвосты»
CLOSE CursorTest
DEALLOCATE CursorTest
DROP TABLE CursorTable

```

Сообщения
(строк обработано: 5)
(строк обработано: 1)
(строк обработано: 1)
(строк обработано: 1)
(строк обработано: 1)

(строк обработано: 1)
MISSING! It probably was deleted.
10702 EVEN
MISSING! It probably was deleted.
10704 EVEN
MISSING! It probably was deleted.

Метаданные SQL Server

[В основном справочный материал]

Метаданные, в общем случае, это данные о данных, информация об информации, описание контента. Каждая СУБД сохраняет метаданные обо всех сущностях базы данных. Так в SQL Server с помощью инструкции CREATE можно создать 52 сущности:

- CREATE AGGREGATE
- CREATE FULLTEXT INDEX
- CREATE SEARCH PROPERTY LIST (Transact-SQL)
- CREATE APPLICATION ROLE
- CREATE FULLTEXT STOPLIST
- CREATE SEQUENCE (Transact-SQL)
- CREATE ASSEMBLY
- CREATE FUNCTION
- CREATE SERVER AUDIT
- CREATE ASYMMETRIC KEY
- CREATE INDEX
- CREATE SERVER AUDIT SPECIFICATION
- CREATE BROKER PRIORITY
- CREATE LOGIN
- CREATE SERVICE
- CREATE CERTIFICATE
- CREATE MASTER KEY
- CREATE SPATIAL INDEX
- CREATE COLUMNSTORE INDEX
- CREATE MESSAGE TYPE
- CREATE STATISTICS
- CREATE CONTRACT
- CREATE PARTITION FUNCTION
- CREATE SYMMETRIC KEY
- CREATE CREDENTIAL
- CREATE PARTITION SCHEME
- CREATE SYNONYM
- CREATE CRYPTOGRAPHIC PROVIDER
- CREATE PROCEDURE
- CREATE TABLE
- CREATE DATABASE
- CREATE QUEUE
- CREATE TRIGGER
- CREATE DATABASE AUDIT SPECIFICATION
- CREATE REMOTE SERVICE BINDING
- CREATE TYPE
- CREATE DATABASE ENCRYPTION KEY
- CREATE RESOURCE POOL
- CREATE USER
- CREATE DEFAULT
- CREATE ROLE
- CREATE VIEW
- CREATE ENDPOINT
- CREATE ROUTE
- CREATE WORKLOAD GROUP
- CREATE EVENT NOTIFICATION
- CREATE RULE
- CREATE XML INDEX
- CREATE EVENT SESSION
- CREATE SCHEMA
- CREATE XML SCHEMA COLLECTION
- CREATE FULLTEXT CATALOG

В разных СУБД применяются разные названия для метаданных - системный каталог, словарь данных и др. Однако общим свойством всех современных реляционных СУБД является то, что каталог/словарь сам состоит из таблиц, а точнее - системных таблиц. В результате пользователь может обращаться к метаданным так же, как и к прикладным данным, используя инструкцию SELECT. Изменения же в каталоге/словаре производятся автоматически при выполнении пользователем инструкций, изменяющих состояние объектов базы данных. Системные таблицы не должны изменяться непосредственно ни одним пользователем. Например, не стоит изменять системные таблицы с помощью инструкций DELETE, UPDATE или INSERT либо с помощью пользовательских триггеров. Обращение к документированным столбцам системных таблиц разрешено. Однако многие столбцы системных таблиц не документированы. В приложениях непосредственные запросы к недокументированным столбцам применять не следует. Чтобы исключить прямой доступ к системным таблицам, пользователь «видит» не сами таблицы, а созданные на их базе представления, которые он, конечно же, не может изменять. Состав и структура каталога/словаря очень различны для различных СУБД. Ограничимся рассмотрением метаданных и способов доступа к ним на примере СУБД MS SQL Server.

Microsoft SQL Server предоставляет следующие коллекции системных представлений, содержащие метаданные:

- Представления информационной схемы
- Представления каталога
- Представления совместимости
- Представления репликации
- Динамические административные представления и функции
- Представления приложения уровня данных (DAC)

Представления информационной схемы определяются в особой схеме с именем INFORMATION_SCHEMA. Эта схема содержится в любой базе данных и состоит из 20 представлений:

- CHECK_CONSTRAINTS
- REFERENTIAL_CONSTRAINTS
- COLUMN_DOMAIN_USAGE
- ROUTINES
- COLUMN_PRIVILEGES
- ROUTINE_COLUMNS
- COLUMNS
- SCHEMATA
- CONSTRAINT_COLUMN_USAGE
- TABLE_CONSTRAINTS
- CONSTRAINT_TABLE_USAGE
- TABLE_PRIVILEGES
- DOMAIN_CONSTRAINTS
- TABLES
- DOMAINS
- VIEW_COLUMN_USAGE
- KEY_COLUMN_USAGE
- VIEW_TABLE_USAGE
- PARAMETERS
- VIEWS

Каждое представление информационной схемы содержит метаданные для объектов, хранящихся в этой конкретной базе данных. Представления информационной схемы, включенные в SQL Server, соответствуют стандартному определению ISO для INFORMATION_SCHEMA. При ссылке на представления информационной схемы необходимо использовать полное имя, включающее имя схемы INFORMATION_SCHEMA. Например:

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, COLUMN_DEFAULT
FROM Northwind.INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = N'Products';
GO
```

Представления каталога имеют наиболее универсальный интерфейс к метаданным каталога и предоставляют наиболее эффективный способ для получения этих данных. Некоторые представления каталога наследуют строки других представлений каталога. Например, представление каталога [sys.tables](#) наследует строки из представления каталога [sys.objects](#). Представление каталога sys.objects называется базовым представлением, а представление sys.tables называется производным представлением. Представление каталога sys.tables возвращает столбцы, определенные для

таблиц, а также все столбцы, которые возвращает представление каталога sys.objects. Представление каталога sys.objects возвращает строки для объектов, отличных от таблиц, например для хранимых процедур или представлений.

Представления каталога в SQL Server организованы в следующие 25 категории:

- Представления каталога групп доступности AlwaysOn
- Представления каталога связанных серверов
- Представления каталога отслеживания изменений
- Представления каталога сообщений (для ошибок)
- Представления каталога сборки среды CLR
- Представления каталога объектов
- Представления каталога баз данных и файлов
- Представления каталога функции секционирования
- Представления компонента Database Mail
- Представления управления на основе политик
- Представления каталога зеркального отображения базы данных
- Представления каталога регулятора ресурсов
- Представления сборщика данных
- Представления каталога скалярных типов
- Пространства данных
- Представления каталога схем
- Представления каталога конечных точек
- Представления каталога безопасности
- Представления каталога расширенных событий
- Представления каталога компонента Service Broker
- Представления каталога расширенных свойств
- Представления каталога конфигурации уровня сервера
- Представления каталога FileTable
- Представления каталога схем XML (система типов XML)
- Представления каталога полнотекстового и семантического поиска

Представления каталогов баз данных и файлов содержит следующие 6 представления:

- sys.backup_devices
- sys.database_files
- sys.database_mirroring
- sys.database_recovery_status
- sys.databases
- sys.master_files

sys.databases - содержит одну строку для каждой базы данных в экземпляре Microsoft SQL Server. Чтобы проверить существование базы данных можно воспользоваться предикатом EXISTS и запросом на выборку из представления sys.databases. Следующий пример проверяет существование указанной базы данных. Если база данных существует, то она удаляется. Если базы данных не существует, то инструкция DROP DATABASE не выполняется.

```
USE [master]
GO
IF EXISTS (SELECT name FROM sys.databases WHERE name = N'FolktaleDB')
    DROP DATABASE [FolktaleDB]
GO
```

Представления каталога объектов содержит следующие 36 представления:

- sys.allocation_units
- sys.parameters
- sys.assembly_modules
- sys.partitions
- sys.check_constraints
- sys.procedures
- sys.columns
- sys.sequences
- sys.computed_columns
- sys.service_queues
- sys.default_constraints

- sys.sql_dependencies
- sys.events
- sys.sql_expression_dependencies
- sys.event_notifications
- sys.sql_modules
- sys.extended_procedures
- sys.stats
- sys.foreign_key_columns
- sys.stats_columns
- sys.foreign_keys
- sys.synonyms
- sys.function_order_columns
- sys.table_types
- sys.identity_columns
- sys.tables
- sys.index_columns
- sys.trigger_event_types
- sys.indexes
- sys.trigger_events
- sys.key_constraints
- sys.triggers
- sys.numbered_procedure_parameters
- sys.views
- sys.numbered_procedures
- sys.objects

sys.objects содержит одну строку для каждого определенного пользователем объекта в области схемы, который создан в базе данных. Представление sys.objects не показывает триггеры DDL, так как они не принадлежат области схемы. Все триггеры (как DML, так и DDL) размещены в представлении sys.triggers. Чтобы проверить существование объекта можно воспользоваться предикатом EXISTS и запросом на выборку из представления sys.objects. Аналогичный результат можно получить, если воспользоваться системной функцией OBJECT_ID. Возвращает идентификационный номер объекта базы данных для объекта области схемы. Следующий пример проверяет существование указанной таблицы, проверяя наличие у таблицы идентификатора объекта. Если таблица существует, то она удаляется. Если таблица не существует, то инструкция DROP TABLE не выполняется.

```
USE Northwind;
GO
IF OBJECT_ID ('dbo.Products', 'U') IS NOT NULL
    DROP TABLE dbo.Products;
GO
```

Замечание. В справочнике приводятся сведения о скалярных функциях, которые возвращают информацию о базе данных и объектах баз данных. Пример:

```
-- Как найти все ограничения определенной таблицы?
SELECT OBJECT_NAME(object_id) AS [constraint_name]
    ,SCHEMA_NAME(schema_id) AS [schema_name]
    ,OBJECT_NAME(parent_object_id) AS [table_name]
    ,type_desc
FROM sys.objects
WHERE type_desc LIKE '%CONSTRAINT' AND parent_object_id = OBJECT_ID('dbo.Products');
GO
```

Замечание. В электронной документации по SQL Server приводится список часто задаваемых вопросов (33 шт.) и ответы на эти вопросы — запросы, основанные на представлениях каталога.

В SQL Server для получения информации о состоянии сервера применяются **динамические административные представления и функции**, расположенные в схеме sys. Их имена следуют такому соглашению по именованию: dm_*

- (от англ. dynamic_management). Есть два типа динамических административных представлений и функций:
- a) **области сервера** - для них необходимо разрешение VIEW SERVER STATE на сервере;
 - b) **области базы данных** - для них необходимо разрешение VIEW DATABASE STATE на базе данных.

Динамические административные представления и функции организованы в виде 20 категорий:

- Динамические административные представления управления и функции для групп доступности AlwaysOn
- Динамические административные представления и функции, связанные с индексами
- Динамические административные представления, относящиеся к системе отслеживания измененных данных
- Динамические административные представления и функции, связанные с вводом-выводом
- Динамические административные представления, относящиеся к отслеживанию изменений
- Динамические административные представления и соответствующие функции, связанные с объектами
- Динамические административные представления, связанные со средой CLR
- Динамические административные представления, связанные с уведомлениями запроса
- Динамические административные представления, связанные с зеркальным отображением базы данных
- Динамические административные представления, относящиеся к репликации
- Динамические административные представления базы данных
- Динамические административные представления для регулятора ресурсов
- Динамические административные представления и соответствующие функции, связанные с выполнением
- Динамические административные представления, связанные с безопасностью
- Динамические административные представления расширенных событий
- Динамические административные представления компонента Service Broker
- Динамические административные представления Filestream and FileTable (Transact-SQL)
- Динамические административные представления, относящиеся к операционной системе SQL Server
- Динамические административные представления полнотекстового и семантического поиска
- Динамические административные представления и функции, связанные с транзакциями

Например, категория «Динамические административные представления, относящиеся к операционной системе SQL Server» содержит 31 представление и функции. Работа с этими представлениями и функциями предполагает соответствующий уровень профессиональной квалификации. Только два примера.

`sys.dm_os_wait_stats` - возвращает данные обо всех случаях ожидания, обнаруженных выполняющимися потоками. Это представление можно использовать для диагностики проблем производительности как всего SQL Server, так и конкретных запросов и пакетов. Ожидание ресурсов имеет место, когда исполнитель запрашивает доступ к ресурсу, который недоступен по причине его использования другим исполнителем или пока он еще недоступен. Несмотря на небольшое количество атрибутов представления (5), интерпретация полученных результатов может вызвать определенные затруднения, т. к. атрибут `wait_type` (имя типа ожидания) может принимать ориентировочно 370 различных значений. Если тип ожидания `WAITFOR` имеет очевидный смысл, т. к. является результатом выполнения инструкции `WAITFOR T-SQL`, в которой длительность ожидания определяется параметрами инструкции. То тип ожидания `PREEMPTIVE_CLOSEBACKUPVIDEVICE` возникает, когда планировщик SQL Server переключается в режим с вытеснением, чтобы закрыть виртуальное устройство резервного копирования. (Реплика. Звучит красиво, но не совсем понятно).

`sys.dm_os_threads` - возвращает список всех потоков SQL Server в операционной системе, запущенных процессом SQL Server. Содержит 26 атрибутов.

Для получения данных, хранящихся в системных таблицах, в приложениях можно использовать один из следующих компонентов:

- системные хранимые процедуры;
- инструкции и функции языка Transact-SQL;
- Управляющие объекты SQL Server (SMO)
- объекты RMO;
- функции каталога Database API

Системные хранимые процедуры

Системные хранимые процедуры объединяются в категории (20 категорий)

ЛР: `sp_binefault`, `sp_bindrule`, [sp_configure](#).

Для Ш.: [sp_lock](#), [sp_who](#), [sp_who2](#)

Функции

Функции метаданных (41 штука)

Возвращают информацию о базе данных и объектах баз данных, например, `DB_ID` и `OBJECT_ID`.

Для Ш.: [APPLOCK_MODE](#) и [APPLOCK_TEST](#).

Функции конфигурации (15 штук)

Возвращают сведения о текущих значениях параметров конфигурации, например, [@@NESTLEVEL](#).

Для И.: [@@LOCK_TIMEOUT](#).

Функции безопасности (26 штук)

Возвращают сведения, полезные для управления безопасностью, например, PERMISSIONS. Но есть еще Хранимые процедуры безопасности (50 штук).

Системные функции (29 штук)

Возвращают сведения об объектах и настройках SQL Server, например, [@@ERROR](#) и [@@IDENTITY](#).

Для И.: [@@TRANCOUNT](#) и [XACT_STATE](#).

DBCC команда (34 штуки)

Для И.: DBCC ROWLOCK

AF = агрегатная функция (среда CLR)

C = ограничение CHECK

D = значение по умолчанию (DEFAULT), в ограничении или независимо заданное

F = ограничение FOREIGN KEY

FN = скалярная функция SQL

FS = скалярная функция сборки (среда CLR)

FT = возвращающая табличное значение функция сборки (среда CLR)

IF = встроенная возвращающая табличное значение функция SQL

IT = внутренняя таблица

P = хранимая процедура SQL

PC = хранимая процедура сборки (среда CLR)

PG = структура плана

PK = ограничение PRIMARY KEY

R = правило (старый стиль, изолированный)

RF = процедура фильтра репликации

S = системная базовая таблица

SN = синоним

SQ = очередь обслуживания

TA = триггер DML сборки (среда CLR)

TF = возвращающая табличное значение функция SQL

TR = триггер DML SQL

TT = табличный тип

U = таблица (пользовательская)

UQ = ограничение UNIQUE

V = представление

X = расширенная хранимая процедура

Следующие скалярные функции возвращают информацию о базе данных и объектах баз данных:

@@PROCID	fn_listextendedproperty
ASSEMBLYPROPERTY	FULLTEXTCATALOGPROPERTY
COL_LENGTH	FULLTEXTSERVICEPROPERTY
COL_NAME	INDEX_COL
COLUMNPROPERTY	INDEXKEY_PROPERTY
DATABASEPROPERTY	INDEXPROPERTY
DATABASEPROPERTYEX	OBJECT_ID
DB_ID	OBJECT_NAME
DB_NAME	OBJECTPROPERTY
FILE_ID	OBJECTPROPERTYEX
FILE_INDEX	SCHEMA_ID
FILE_NAME	SCHEMA_NAME
FILEGROUP_ID	SQL_VARIANT_PROPERTY
FILEGROUP_NAME	TYPE_ID
FILEGROUPPROPERTY	TYPE_NAME
FILEPROPERTY	TYPEPROPERTY

Хранимые процедуры каталога

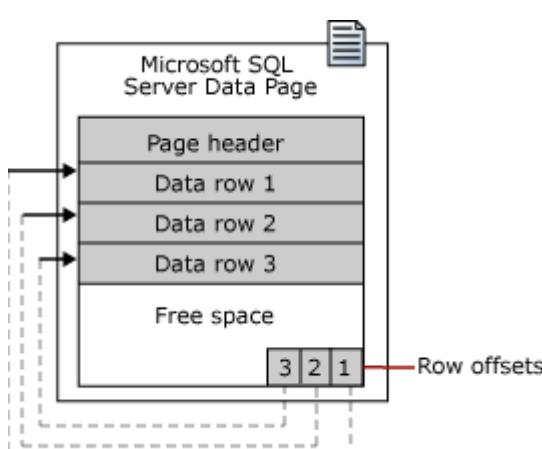
SQL Server поддерживает следующие системные хранимые процедуры, реализующие функции словаря данных ODBC и изоляцию приложений ODBC от изменений базовых системных таблиц.

```
sp_column_privileges  
sp_special_columns  
sp_columns  
sp_sproc_columns  
sp_databases  
sp_statistics  
sp_fkeys  
sp_stored_procedures  
sp_pkeys  
sp_table_privileges  
sp_server_info  
sp_tables
```

Методы физического хранения данных на диске

Хранение данных и в прошлых, и в новых версиях организовано в виде иерархических структур:

- **База данных.** Рассматривается как наивысший уровень абстракции для хранилища данных (конкретного сервера). Является высшим уровнем, на который может быть наложена блокировка (lock), хотя явно задать блокировку на уровне базы данных нельзя.
- **Файл.** Рассматривать как ближайшее подобие устройства (в более ранних версиях). Для хранения единственной базы данных могут быть выделены несколько файлов. Но в один файл нельзя поместить более одной базы данных. По умолчанию база данных использует два файла.
 - 1) В первом физическом файле базы данных хранится реальная информация. Файл должен иметь расширение .mdf (это рекомендация, а не требование).
 - 2) Второй файл является вспомогательным файлом базы данных – ее журналом. Журнал постоянно хранится в файле с расширением .ldf, и без него база данных работать не будет.
- **Экстент.** Является основной единицей пространства, выделяемую под таблицу или индекс. Экстент состоит из восьми смежных страниц данных. В SQL Server при создании таблицы изначально не выделяется ни одной страницы. Они добавляются только при вставке в таблицу новых строк. По мере добавления в таблицу записей, для хранения информации выделяются все новые страницы до тех пор, пока их количество не достигнет восьми. С этого момента при необходимости дополнительного пространства для хранения данных SQL Server будет выделяться как минимум экстент. Экстенты бывают двух типов:
 - 1) **Разделяемые экстенты.** Разделяемые экстенты могут совместно использоваться восемью различными объектами (т. е. каждая страница экстента может принадлежать другому объекту). Все создаваемые таблицы и индексы помещаются в разделяемые экстенты. Как только количество страниц, выделенных под объект, достигает восьми, для объекта будет выделен новый однородный экстент.
 - 2) **Однородные экстенты.** Структура однородных экстентов понятна из названия. Каждая страница такого экстента принадлежит одному и тому же объекту.
- Об экстентах должно быть известно следующее:
 - 1) После заполнения экстента при вставке новой записи для ее хранения выделяется целый новый экстент, а не только пространство, соответствующее размеру добавляемой записи.
 - 2) Благодаря предварительному выделению пространства, SQL Server экономит время, которое потребовалось бы в случае выделения пространства во время добавления каждой новой строки.
 - 3) Экстенты могут блокироваться. Однако блокировка может накладываться на экстент только во время выделения нового либо высвобождения старого экстента.
- **Страница.** Является элементарной единицей пространства выделяемого внутри отдельного экстента. В SQL Server размер страницы составляет 8 КБ. Страница может рассматриваться как контейнер для хранения и строк таблиц и индексов. Одна строка не может быть разделена между двумя страницами. В состав страницы входят:
 - 1) 96-байтный заголовок страницы (page header); заголовок включает: номер страницы, тип страницы, количество свободного пространства на странице, идентификатор единицы распределения объекта, которому принадлежит страница.
 - 2) сами данных и
 - 3) указатели смещения строк (row offset).



Следует обратить внимание на размещаемые в конце страницы указатели, необходимые для определения в странице позиции, с которой начинаются данные конкретной строки.

В SQL Server существуют следующие типы страниц:

- страницы данных (Data Pages);
- страницы индекса (Index Pages);

- BLOB-страницы (Text/Image Pages) - для хранения данных типа text, ntext, image, nvarchar(max), varchar(max), varbinary(max) и xml, а также varchar, nvarchar, varbinary, и sql_variant, если их длина > 8 Кб;
- карты размещения страниц (Global Allocation Map, Shared Global Allocation Map – GAM, SGAM);
- страницы свободного пространства (Page Free Space - PFS);
- карты размещения индекса (Index Allocation Map - IAM);
- схема массовых изменений (Bulk Changed Map);
- схема разностных изменений (Differential Changed Map).

Страницы данных (Data Pages)

- В страницах данных хранятся все реальные данные таблицы, кроме BLOB-данных.
- Если в строке имеется столбец, в котором содержатся BLOB-данные, тогда все обычные данные помещаются в страницу данных, в которой также находится 16-байтовый указатель на страницу, хранящую BLOB-информацию.
- При использовании опции TEXT IN ROW, BLOB-данные помещаются непосредственно внутри текста таблицы, если при этом строка целиком помещается в странице, в противном случае, опять будет использоваться 16-байтовый указатель.
- Страницы файлов данных SQL Server нумеруются последовательно: первая страница файла получает нулевой номер (0).
- Каждый файл базы данных имеет уникальный цифровой идентификатор. Чтобы уникальным образом определить страницу базы данных, необходимо использовать как идентификатор файла, так и номер этой страницы.
- Первая страница каждого файла — это страница заголовка файла; она содержит сведения об атрибутах данного файла. Некоторые другие страницы, расположенные в начале файла, тоже содержат системные сведения, например карты размещения. Одна из системных страниц, хранимых как в первичном файле данных, представляет собой загрузочную страницу базы данных, которая содержит сведения об атрибутах этой базы данных.

Страницы индекса (Index Pages)

Страницы индекса содержат страницы разных уровней дерева некластерного индекса, а также страницы, не являющиеся узлами дерева кластерного индекса.

BLOB-страницы

- BLOB-страницы хранятся в собственном специальном формате.
- В BLOB -страницах отсутствуют отдельные строки как таковые.
- SQL Server выделяет для хранения BLOB столько страниц, сколько нужно, однако он не гарантирует, что выделенные страницы будут являться смежными - они могут быть размещены где угодно внутри файла базы данных.
- BLOB -страницы организовываются в структуру сбалансированного дерева (balanced tree).

Страницы типа GAM, SGAM, PFS используются для определения свободных и используемых страниц и экстентов. Фактически в этих страницах содержится информация о доступном пространстве. Рассмотрение данных типов страниц не является обязательным для высококачественной разработки приложений или системного администрирования. Подробную информацию по этому вопросу (если вы страдаете бессонницей) можно найти в Books Online.

Индексы и проблемы производительности

Общие сведения об индексах

Индекс – это объект базы данных, обеспечивающий дополнительные способы быстрого поиска и извлечения данных. Индекс может создаваться на одном или нескольких столбцах. Это означает, что индексы бывают *простыми* и *составными*. Если в таблице нет индекса, то поиск нужных строк выполняется простым сканированием по всей таблице. При наличии индекса время поиска нужных строк можно существенно уменьшить. К недостаткам индексов следует отнести:

- a) дополнительное место на диске и в оперативной памяти,
- b) замедляются операции вставки, обновления и удаления записей.

В SQL Server индексы хранятся в виде сбалансированных деревьев. Представление индекса в виде сбалансированного дерева означает, что стоимость поиска любой строки остается относительно постоянной, независимо от того, где находится эта строка. Сбалансированное дерево состоит из:

- a) *корневого узла* (root node), содержащего одну страницу,
- b) нескольких *промежуточных уровней* (intermediate levels), содержащих дополнительные страницы, и
- c) *листового уровня* (leaf level).

На страницах листового уровня находятся отсортированные элементы, соответствующие индексируемым данным. По мере добавления данных в таблицу индекс будет разрастаться, но по-прежнему оставаться в форме сбалансированного дерева.

Рассмотрим пример. Известно, что страница данных SQL Server имеет размер 8192 байта и может содержать до 8060 байт пользовательских данных. Определим таблицу CREATE TABLE Tab (Col char(60)); и построим индекс для столбца Col. Если в таблице будет 100 строк с данными, то для их хранения потребуется 6000 байт памяти. Все эти строки разместятся на одной странице данных, поэтому в индексе будет всего одна страница – одновременно на уровне корня и на листовом уровне. На самом деле в таблицу можно занести 134 строки и при этом выделить только одну страницу для индекса.

При добавлении в таблицу 135-й строки SQL Server создаст две дополнительные страницы. Индекс будет состоять из корневой страницы и двух страниц листового уровня. На первой странице листового уровня будет размещаться первая половина таблицы, на второй странице листового уровня – вторая половина таблицы, а на корневой странице – две строки данных. Для этого индекса не нужен промежуточный уровень, потому что корневая страница будет содержать все значения, с которых начинаются страницы листового уровня. Для поиска нужной строки в таблице придется просмотреть ровно две страницы индекса.

Для таблицы, содержащей 17957 строк, индекс будет состоять из 134 страниц листового уровня и корневой страницы. На каждой странице будет размещаться по 134 строки. При добавлении в таблицу 17957-й строки SQL Server должен создать еще одну страницу индекса на листовом уровне, но на корневой странице нельзя разместить 135 строк. Поэтому SQL Server добавит промежуточный уровень, содержащий две страницы. На первой странице будут находиться начальные строки первой половины страниц листового уровня, а на второй – начальные строки второй половины страниц листового уровня. Теперь корневая страница будет содержать две строки, соответствующие начальным значениям двух страниц промежуточного уровня.

Когда в таблицу будет добавлена 2406105-я строка, SQL Server создаст еще один промежуточный уровень и т. д. Очевидно, что такой тип структуры позволяет SQL Server достаточно быстро находить строки, удовлетворяющие запросам, даже в очень больших таблицах. Так, чтобы найти строку в таблице, содержащей около 2500000 строк, SQL Server должен просмотреть всего три страницы индекса. **Конец примера.**

В SQL Server существует два типа индексов:

- a) *кластерные* индексы;
- b) *некластерные* индексы, которые включают:
 - i. некластерные индексы на основе кучи;
 - ii. некластерные индексы на основе кластерных индексов.

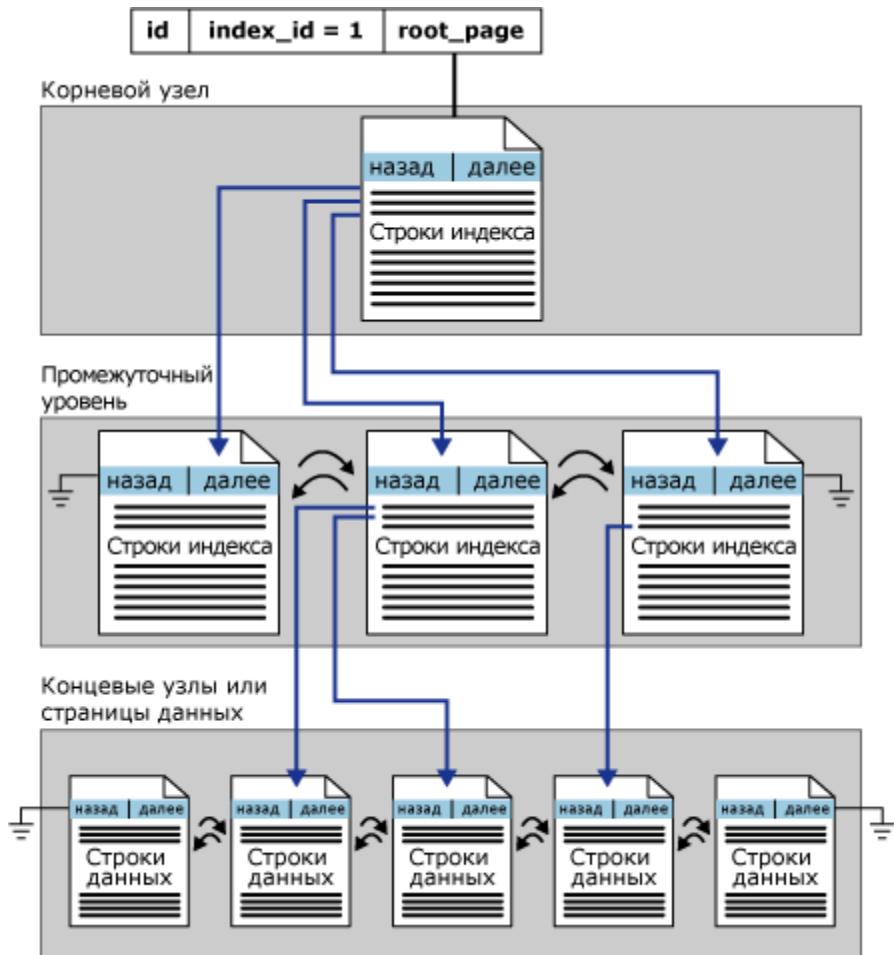
Кластерные индексы

В кластерном индексе таблица представляет собой часть индекса, или индекс представляет собой часть таблицы в зависимости от вашей точки зрения. Листовой узел кластерного индекса – это страница таблицы с данными. Поскольку сами данные таблицы являются частью индекса, то очевидно, что для таблицы может быть создан только один кластерный индекс.

В SQL Server кластерный индекс является уникальным индексом по определению. Это означает, что все ключи записей должны быть уникальные. Если существуют записи с одинаковыми значениями, SQL Server делает их уникальными, добавляя номера из внутреннего (невидимого снаружи) 4-х байтного счетчика. Кластерный индекс использовать необязательно. Тем не менее, кластерные индексы часто используют в качестве первичного ключа. На уровне листьев кластерного индекса информация упорядочивается и физически сохраняется на диске в соответствии с заданными критериями сортировки. На каждом уровне индекса страницы организованы в виде двунаправленного связного списка. Вход в корень кластерного индекса дает поле **root_page** системного представления **sys.system_internals_allocation_units**.

При добавлении каждой новой записи хранимые данные пересортируются, чтобы сохранить физическую упорядоченность данных. Если требуется создать новую запись в середине индексной структуры, то происходит обычное разбиение страницы. Половина записей из старой страницы переносится в новую, а новая запись добавляется либо в новую страницу, либо в старую страницу (для сохранения равновесия). Если же новая запись логически попадает в конец индексной структуры, тогда в новую создаваемую страницу помещается только новая строка (информация не делится поровну между двумя рассматриваемыми страницами).

На следующем рисунке изображена структура кластерного индекса.



Некластерные индексы на основе кучи

В листьях некластерного индекса на основе кучи хранятся указатели на строки данных. Указатель строится на основе идентификатора файла (ID), номера страницы и номера строки на странице. Весь указатель целиком называется идентификатором строки (RID). С точки зрения физического размещения данных – они хранятся в полном беспорядке. С точки зрения технической реализации нужные записи приходится искать по всему файлу. Вполне может статься, что в этом случае SQL Server придется несколько раз возвращаться к одной и той же странице для чтения различных строк, поскольку определить заранее, откуда придется физически считывать следующую строку нет никакой возможности.

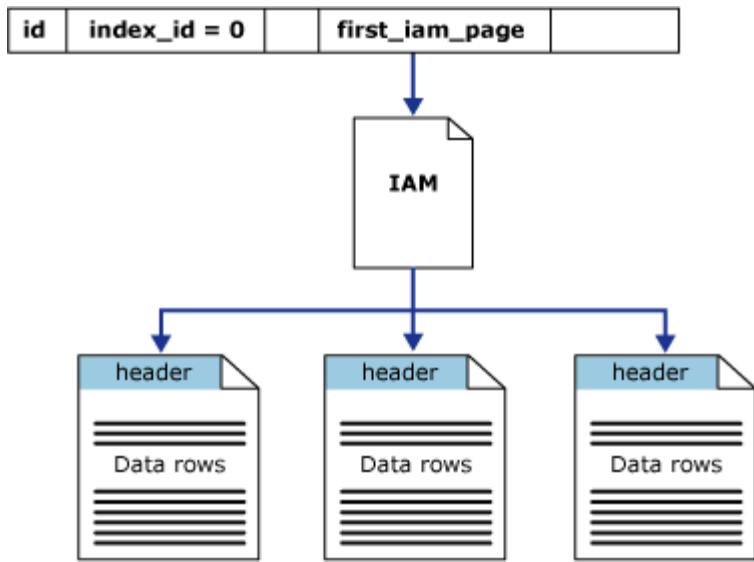
Некластерные индексы, основанные на кластерных таблицах

В листьях некластерного индекса, основанного на кластерных таблицах, хранятся указатели на корневые узлы кластерных индексов. Поиск в таком индексе состоит из двух этапов:

- поиск в некластерном индексе и
- поиск в кластерном индексе.

Замечание. Столбец **first_iam_page** в системном представлении **sys.system_internals_allocation_units** указывает на первую IAM-страницу в цепи IAM-страниц, управляющей выделением пространства в куче. 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.

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



Индексы могут создаваться двумя способами:

- явно при помощи инструкции CREATE INDEX;
- неявно в качестве объекта при создании ограничения.

Создание индекса с помощью инструкции CREATE INDEX

Упрощенный синтаксис инструкции следующий:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX имя-индекса
ON имя-таблицы-или-представления ( список-столбцов )
[ INCLUDE (список-столбцов) ]
[WITH список-опций]
[ ON файловая-группа ]
```

Замечания.

- Для каждого столбца в списке столбцов могут указываться две взаимоисключающие опции ASC/DESC, которые позволяют выбрать способ сортировки индекса. По умолчанию используется вариант ASC, что означает сортировку по возрастанию.
- Предложение INCLUDE указывает неключевые столбцы, добавляемые к ключевым столбцам некластеризованного индекса.
- Следующие за ключевым словом WITH опции являются необязательными. Их можно использовать в любом сочетании. Одни (как например PAD_INDEX, IGNORE_DUP_KEY, DROP_EXISTING, STATISTICS_NORECOMPUTE, SORT_IN_TEMPDB) используются довольно редко, другие (как например FILLFACTOR) существенно влияют на быстродействие и алгоритм работы системы. Описания опций индекса находятся по адресу <http://msdn.microsoft.com/ru-ru/library/ms188783.aspx>.

- d) С помощью предложения ON можно задать место хранения индекса. По умолчанию индекс помещается в ту же файловую группу, где находится таблица или представление, для которых он создан.

Пример создания индекса.

```
CREATE UNIQUE NONCLUSTERED INDEX  
[IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode] ON  
[Person].[Address]  
( [AddressLine1] ASC, [AddressLine2] ASC, [City] ASC, [StateProvinceID] ASC, [PostalCode]  
ASC)  
WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY  
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)  
ON [PRIMARY]
```

Индексы, создаваемые вместе с ограничениям

Такой тип индексов часто называют «связанными индексами». Связанные индексы создаются при добавлении одного из следующих двух типов ограничений:

- ограничения первичного ключа (PRIMARY KEY);
- ограничения уникальности (UNIQUE).

Пример создания индекса.

```
CREATE TABLE [Person].[Address] (  
[AddressID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,  
...  
CONSTRAINT [PK_Address_AddressID] PRIMARY KEY CLUSTERED  
( [AddressID] ASC)  
WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,  
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)  
ON [PRIMARY]  
) ON [PRIMARY]
```

Анализ запросов

Как индексация влияет на производительность?

```
USE AdventureWorks;  
GO  
SELECT name AS index_name, STATS_DATE(object_id, index_id) AS statistics_update_date  
FROM sys.indexes  
WHERE object_id = OBJECT_ID('Person.Address');  
GO
```

```
PK_Address_AddressID 2012-02-27 22:22:03.257  
AK_Address_rowguid 2012-02-27 22:22:04.910  
IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode 2012-02-27 22:22:04.987  
IX_Address_StateProvinceID 2012-02-27 22:22:05.033
```

Использование статистики для повышения производительности запросов

[http://msdn.microsoft.com/ru-ru/library/ms190397\(v=sql.105\).aspx#StatisticsOptions](http://msdn.microsoft.com/ru-ru/library/ms190397(v=sql.105).aspx#StatisticsOptions)

Анализ запроса

[http://msdn.microsoft.com/ru-ru/library/ms191227\(v=sql.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms191227(v=sql.105).aspx)

В справочнике «Значки графических планов выполнения» по адресу

[http://msdn.microsoft.com/ru-ru/library/ms175913\(v=sql.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms175913(v=sql.105).aspx)

приведены изображения значков, ранжированных по категориям:

- 1) Основные операторы (60, цвет синий)
- 2) Операторы курсора (7, цвет желтый)
- 3) Операторы параллелизма (3, цвет синий)
- 4) Операторы T-SQL (8, цвет зеленый)

В справочнике «Справочник по логическим и физическим операторам» по адресу

[http://msdn.microsoft.com/ru-ru/library/ms191158\(v=sql.105\).aspx](http://msdn.microsoft.com/ru-ru/library/ms191158(v=sql.105).aspx)

приведен список из 107 операторов, которые классифицируются как логические и физические.

Логические операторы описывают операции реляционной алгебры на концептуальном уровне. Физические операторы фактически реализуют операцию, определенную логическим оператором, используя конкретный метод или алгоритм. План запроса — это дерево физических операторов. Каждый физический оператор является объектом или процедурой, выполняющей операцию. Например, "Inner Join" является логическим оператором внутреннего соединения, "Nested Loops" является физическим оператором, который реализует различные логические операторы соединения (внутреннее соединение, левое внешнее соединение, левое полусоединение и антилевое полусоединение), "Table-valued Function" является одновременно логическим и физическим оператором, который вычисляет функцию, возвращающую табличное значение, а "UDX" - это группа из 8-ми логических и физических операторов, реализующих часть операций XQuery и XPath. Каждый физический оператор является объектом или процедурой, выполняющей операцию.

Замечание. Над операторами первой группы, выполняющимися параллельно, отображается значок параллельного процесса (черный на желтом).

Чтобы создать план выполнения запроса необходимо:

- a) Иметь разрешения на выполнение запросов, для которых создается план выполнения, и разрешение SHOWPLAN для всех баз данных, на которые ссылается запрос.
- b) Открыть файл с существующим запросом или создать его заново.
- c) В контекстном меню редактора запросов выбрать **Включить действительный план выполнения** (Include Actual Execution Plan) или нажать соответствующую кнопку на панели инструментов.
- d) Выполнить запрос. План запроса отображается на вкладке **План выполнения** в области результатов.

Останавливая указатель мыши над логическими и физическими операторами, можно просмотреть описание и свойства операторов во всплывающих подсказках. Также можно просмотреть свойства оператора в окне **Свойства**.

Графически план выполнения запроса представляется в виде дерева (корень слева, листья справа), которое читается справа налево и сверху вниз. Интерпретация графического отображения плана выполнения запроса:

- 1) Каждый узел дерева представлен в виде значка, указывающего логический и физический оператор, используемый для выполнения этой части запроса.
- 2) Каждый узел связан со своим родительским узлом. Дочерние узлы одного родительского узла отображаются в одном столбце. Однако все узлы в одном столбце не обязательно имеют общий родительский узел. Правила со стрелками на конце соединяют каждый узел с его родителем.

Как оптимизатор запросов использует статистику для создания планов запросов

Статистика для оптимизации запросов — это объекты, содержащие статистические сведения о распределении значений в одном или нескольких столбцах. Оптимизатор запросов использует эти сведения для оценки числа строк в результатах запроса. Например, оптимизатор запросов, используя статистику, может выбрать оператор *index seek* вместо оператора *index scan*.

Каждый объект статистики создается по индексу или списку из одного или нескольких столбцов и содержит:

- a) заголовок, содержащий метаданные о статистике,
- b) гистограмму, содержащую распределение значений в первом ключевом столбце объекта статистики, и
- c) вектор плотностей для измерения корреляции с охватом нескольких столбцов.

Для отображения статистики можно использовать:

- a) Properties в контекстном меню Statistics в окне Object Explorer в среде SSMS, например, последовательно раскрывая узлы Databases|AdventureWorks|Tables|Person.Address|Statistics и click правой кнопкой по IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode и Properties
- b) инструкцию DBCC SHOW_STATISTICS, например,

```
DBCC SHOW_STATISTICS ("Person.Address",
IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode)
WITH STAT_HEADER, HISTOGRAM
```

GO

AddressID	AddressLine1	AddressLine2	City	StateProvinceID
PostalCode				
29076	Attaché de Presse	NULL	Saint-Denis	179
93400				
RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	
AVG_RANGE_ROWS				
Attaché de Presse	121	17	117	1,034188

Для большинства запросов оптимизатор создает статистику самостоятельно, но в некоторых случаях для повышения производительности запросов нужно создать дополнительные статистические данные с помощью инструкции CREATE STATISTICS. В этой инструкции с помощью предиката фильтра в предложении WHERE можно создавать отфильтрованную статистику.

Описания столбцов, возвращаемых в результирующий набор, если указан параметр HISTOGRAM.

Имя столбца	Описание
RANGE_HI_KEY	Верхнее граничное значение столбца для шага гистограммы. Это значение столбца называется также ключевым значением.
RANGE_ROWS	Предполагаемое количество строк, значение столбцов которых находится в пределах шага гистограммы, исключая верхнюю границу.
EQ_ROWS	Предполагаемое количество строк, значение столбцов которых равно верхней границе шага гистограммы.
DISTINCT_RANGE_ROWS	Предполагаемое количество строк с различающимся значением столбца в пределах шага гистограммы, исключая верхнюю границу.
AVG_RANGE_ROWS	Среднее количество строк с повторяющимися значениями столбцов в пределах шага гистограммы, исключая верхнюю границу ($RANGE_ROWS / DISTINCT_RANGE_ROWS$ для $DISTINCT_RANGE_ROWS > 0$).

Когда создается и обновляется статистика

- Явно с помощью инструкций CREATE STATISTICS и UPDATE STATISTICS и с помощью системной хранимой процедуры sp_updatestats.
- Неявно с помощью параметров статистики AUTO_CREATE_STATISTICS, AUTO_UPDATE_STATISTICS и AUTO_UPDATE_STATISTICS_ASYNC, действующих на уровне базы данных и устанавливаемых в инструкции ALTER DATABASE.

```
AUTO_CREATE_STATISTICS { ON | OFF }
AUTO_UPDATE_STATISTICS { ON | OFF }
AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }
```

Для определения времени последнего обновления статистики пользуйтесь функцией STATS_DATE.

Статистика и выбор индексов

Возникает вопрос – если таблица имеет один кластерный индекс и четыре не кластерных, как SQL Server узнает какой из индексов использовать. Распределение статистики дает query optimiser'у возможность выбрать определенный индекс.

Распределение статистики

Начиная с SQL Server 2000, статистика хранится в поле данных для хранения имиджей. То есть – растет пропорционально размеру индекса, это позволяет не терять точность при увеличении размера индексов.

Чтобы понять что из себя представляет статистика, рассмотрим следующий пример. Возьмем таблицу Orders из базы данных Northwind. Если мы выполним следующий запрос:

```
SELECT TOP 24 OrderID, convert(char(11), OrderDate) as "Order Date"
FROM Orders ORDER BY OrderDate
```

OrderID	Order Date
---------	------------

10248	Jul 4 1996
10249	Jul 5 1996
10250	Jul 8 1996
10251	Jul 8 1996
10252	Jul 9 1996
10253	Jul 10 1996
10254	Jul 11 1996
10255	Jul 12 1996
10256	Jul 15 1996
10257	Jul 16 1996
10258	Jul 17 1996
10259	Jul 18 1996
10260	Jul 19 1996
10261	Jul 19 1996
10262	Jul 22 1996
10263	Jul 23 1996
10264	Jul 24 1996
10265	Jul 25 1996
10266	Jul 26 1996
10267	Jul 29 1996
10268	Jul 30 1996
10269	Jul 31 1996
10270	Aug 1 1996
10271	Aug 1 1996

(24 rows affected)

В этом примере выбраны записи с 4 июля 1996 по 1 августа 1996. Теперь посчитаем, сколько раз встречаются те или иные значения:

```
SELECT CONVERT(char(11), A.OrderDate) as "Order Date" , count(*) as "# of OrderDate"
FROM(SELECT TOP 24 OrderID, OrderDate FROM Orders ORDER BY OrderDate) as A GROUP BY
A.OrderDate
```

Order Date	# of OrderDate
------------	----------------

Jul 4 1996	1
Jul 5 1996	1
Jul 8 1996	2
Jul 9 1996	1
Jul 10 1996	1
Jul 11 1996	1
Jul 12 1996	1
Jul 15 1996	1
Jul 16 1996	1
Jul 17 1996	1
Jul 18 1996	1
Jul 19 1996	2
Jul 22 1996	1
Jul 23 1996	1
Jul 24 1996	1
Jul 25 1996	1
Jul 26 1996	1
Jul 29 1996	1
Jul 30 1996	1

```
Jul 31 1996      1  
Aug  1 1996      2
```

```
(21 rows affected)
```

Когда SQL Server считает или сортирует данные, он заранее знает как много тех или иных значений он найдет в указанном запросе. Например, выполним следующий запрос:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate  
FROM Orders WHERE OrderDate BETWEEN '1996-07-15' AND '1996-07-20'
```

OrderID	CustomerID	EmployeeID	OrderDate
10256	WELLI	3	1996-07-15 00:00:00.000
10257	HILAA	4	1996-07-16 00:00:00.000
10258	ERNSH	1	1996-07-17 00:00:00.000
10259	CENTC	4	1996-07-18 00:00:00.000
10260	OTTIK	4	1996-07-19 00:00:00.000
10261	QUEDE	4	1996-07-19 00:00:00.000

```
(6 rows affected)
```

SQL Server знает что запрос вернет только 6 записей еще до того как будет произведен доступ к таблице. Назначение статистики базируется на простом алгоритме: для выбора стратегии доступа к данным, SQL Server должен знать, как много записей вернет запрос.

В SQL Server статистика не просто учитывает количество записей для того или иного значения. Первое, статистика более точно описывает распределение данных по значениям. Второе, зона описания статистики может не содержать всех значений индексируемого поля. Например, в таблице Orders 830 записей, но статистика хранится только для 186 значений. Как и почему именно 187 – этот алгоритм не рассматривается в данном издании. Выполним команду

```
dbcc show_statistics (Orders, OrderDate)
```

Этот результат показывает следующее

```
статистика была просчитана в последний раз 23 декабря 2002  
таблица содержит 830 записей  
все записи были проанализированы для получения статистики  
в статистике информация сохранена в дискретности на 187 записей  
средняя плотность распределения примерно 0.17%
```

Пожалуй самым интересным здесь будет значение плотности распределения. Если каждое значение в таблице уникальное, то плотность будет 1/830, то есть 0.12%. Но в нашем примере мы имеем 0.17% показывает что некоторые значения встречаются 2 и более раз. Например мы видим что для 8 июля 1996 встречается дважды. Теория говорит, что чем меньше плотность, тем лучше – это увеличивает избирательность, а следовательно и ценность построенного индекса.

Например если колонка содержит только 3 значения, плотность распределения будет равна 33.3%, что показывает бесполезность построения индекса по данному полю. Индексы занимают место на диске и в оперативной памяти и отнимают быстродействие. В идеале самый лучший индекс имеет плотность распределения равную единице, деленной на количество записей в таблице - все записи уникальны. При построении индексов, обращайте внимание на плотность распределения – если она превышает 10%, то индекс можно считать бесполезным. Сканирование по таблице в таком случае будет более эффективным.

Второй результат, возвращаемый командой dbcc show_statistics

Это дает очень интересный результат потому что плотность для одного поля OrderDate 0.2%, а для пары OrderDate, OrderID уже 0.12. Поскольку OrderID является primary key для таблице, то понижение плотности вполне очевидно.

Последний результат может быть наиболее интересен для полного понимания важности статистики. Во-первых, заметим, что только пять значений в зоне распределения вместо 24 в самой таблице. Тем не менее учитывая значения по всем колонками, система знает сколько записей будет в выборке. Колонка RANGE_HI_KEY дает высшее значение для значения, сохраненного в статистике. Мы знаем что 1996-07-04 является первым значением и следующим за ним идет 1996-07-15. Между этими двумя значениями находится 7 записей. Колонка RANGE_ROWS дает нам эту информацию. Только три

значения есть между 1996-07-15 и 1996-07-19, и так далее. Колонка DISTINCT_RANGE_ROWS содержит информацию о том сколько определенных (неповторяющихся) значений в интервале. Например в интервале с 1996-07-04 по 1996-07-15 есть 6 определенных значений из 7 записей в интервале. Это говорит что одно значение в указанном интервале повторяется дважды. Мы не указали еще одну колонку в результате команды dbcc show_statistics - AVG_RANGE_ROWS, которое является результатом простой арифметической операции RANGE_ROWS /DISTINCT_RANGE_ROWS.

Анализируя эту информацию, можно определить, как распределены данные в индексе, а так же как много записей вернет запрос.

Выбор индексов

Как мы видели в предыдущих примерах, SQL Server достаточно точно знает, как данные распределены по таблице при наличии индекса. Каждый раз при выполнении запроса, SQL Server первым делом оценивает статистику. Давайте рассмотрим несколько примеров для более полного понимания данного процесса.

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-19' AND '1996-07-25'
```

SQL Server сначала проверяет существования индекса по полю OrderDate или составного индекса, начинающегося с поля OrderDate. В таком случае SQL Server знает, как много записей вернется в результате запроса. Query Analyser дает следующую картинку:

Для того что бы посмотреть детали каждого этапа выполнения запроса, просто выберете нужный этап и кликните правой клавишей мыши. Мы видим, что оценочное количество записей – 6, что и реально соответствует действительности.

Теперь выполним этот запрос:

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-19' AND '1996-07-25' AND CustomerID = 'FOLKO'
```

SQL Server найдет один индекс OrderDate и один CustomerID. Он будет использовать их обоих и результатом будет пересечение по этим двум условиям. Оценка дает 6 значений по OrderDate и только одно для CustomerID. В этом конкретном случае SQL Server вернет только одно значение.

Если заменить условие операции AND на OR, система может сделать объединение индексов или может предпочесть полное сканирование таблицы, если сочтет что такой путь будет более быстрым и имеет меньшую стоимость чем работа с индексами. В нашем конкретном случае в результате запроса:

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-19' AND '1996-07-25' OR CustomerID = 'FOLKO'
```

Было произведено полное сканирование таблицы, так как стоимость операции OR оказалась выше. Это объясняется тем, что размеры таблицы сравнительно малы.

Что же произойдет, если для таблицы нет индексов, следовательно, нет статистики? Ответ прост: SQL Server не может работать без статистики! И он создаст статистику без каких либо индексов. Если выполнить запрос:

```
SELECT * FROM Orders WHERE ShipCity='Grass'
```

SQL Server автоматически создаст статистику для этого поля, поскольку нет индекса по нему. Для того что бы просмотреть все индексы, созданные для определенной таблицы, выполним запрос:

```
SELECT name, first, root FROM sysindexes WHERE id=OBJECT_ID('Orders')
```

Мы получили результат:

Name	first	root
PK_Orders	0xC90000	0xCC0000
CustomerID	0x380100	0x3B0100
CustomersOrders	0x3D0100	0x400100
EmployeeID	0x420100	0x450100
EmployeesOrders	0x460100	0x490100

```
OrderDate          0x4A0100 0x4D0100
ShippedDate        0x4E0100 0x510100
ShippersOrders     0x520100 0x550100
ShipPostalCode     0x560100 0x590100
_WA_Sys_0000000B_08EA5793    NULL      NULL
```

(10 rows affected)

Заметьте имя “индекса” _WA_Sys_0000000B_08EA5793 с root адресом NULL – потому что это не индекс. Это статистика для поля ShipCity. Вам не стоит беспокоится о засорении вашей базы данных ненужной информацией – автоматически созданная статистика будет уничтожена так же автоматически когда SQL Server посчитает что она более не нужна. Просто доверьтесь SQL Server. (Хорош оборот, неправда ли!?)

Вы можете так же изучить наличие статистики при помощи команды:

```
sp_helpstats 'Orders'

statistics_name           statistics_keys
-----
_WA_Sys_0000000B_08EA5793    ShipCity
```

Автоматическое обновление статистики является огромной помощью для DBA. Порой крайне сложно определить, какая же статистика должна быть создана и эту функцию на себя взял SQL Server. Существует только одна проблема - поддерживать корректную на каждый момент времени статистику.

Обслуживание статистики

Что произойдет если статистика устареет и не будет отражать реальную картину с данными в таблице. Ответ очень прост – выбор индекса может быть неверен. Представим, что статистика была собрана, когда в таблице было только 1 000 записей, а теперь ее размер 100 000. Что бы быть реально полезной, статистика должна содержать текущие реальные данные.

SQL Server позволяет обновлять статистику автоматически без привлечения дополнительных усилий со стороны DBA. Проверить, установлен ли флаг автоматического обновления статистики, можно командой:

```
SELECT DATABASEPROPERTYEX('Northwind', 'IsAutoUpdateStatistics')

(No column name)
-----
1
```

Если результат равен 1, это значит, что опция автоматического обновления статистики включена. Установить опцию в этот режим можно командой

```
ALTER DATABASE Northwind SET AUTO_UPDATE_STATISTICS ON
```

а выключить

```
ALTER DATABASE Northwind SET AUTO_UPDATE_STATISTICS OFF
```

Так же эту операцию можно установить используя хранимую процедуру sp_dboption, но в SQL Server 2000 она оставлена только для обратной совместимости и может исчезнуть в будущих версиях.

Рекомендуется оставлять эту функцию включенной. В этом случае SQL Server будет сам обновлять статистику когда посчитает ее устаревшей. Алгоритм обновления полностью определяется SQL Server и зависит от количества обновлений, удалений и добавлений записей в таблицу. Если таблица имеет один миллион записей и только 100 из них были изменены (0.01%), вряд ли имеет смысл обновлять статистику поскольку такие изменения в таблице вряд ли драматически поменяли общую картину данных.

Кроме того если размер таблицы более 8МБ (1 000 страниц), SQL Server не будет использовать все данные для вычисления статистики. Все эти ограничения разработаны для того что бы работа со обновлением статистики наносила как можно меньший удар на быстродействие сервера.

Так же управлять автоматическим обновлением статистики можно при помощи хранимых процедур типа sp_autostats:
Команда

```
sp_autostats 'Orders'

Global statistics settings for [Northwind]:
    Automatic update statistics: ON
    Automatic create statistics: ON

settings for table [Orders]

Index Name           AUTOSTATS Last Updated
----- -----
[PK_Orders]          ON      2012-06-20 21:35:35.680
[CustomerID]         ON      2012-06-20 21:35:35.693
[CustomersOrders]    ON      2012-06-20 21:35:35.700
[EmployeeID]          ON      2012-06-20 21:35:35.700
[EmployeesOrders]    ON      2012-06-20 21:35:35.703
[OrderDate]           ON      2012-06-20 21:35:35.707
[ShippedDate]         ON      2012-06-20 21:35:35.710
[ShippersOrders]     ON      2012-06-20 21:35:35.717
[ShipPostalCode]      ON      2012-06-20 21:35:35.720
[_WA_Sys_0000000B_08EA5793] ON      2013-03-21 23:28:18.767

(10 rows affected)
```

показывает включена ли опция автоматического обновления статистики для конкретных индексов и когда обновление было сделано в последний раз. Так же эта команда позволяет включить или отключить опцию обновление для всех индексов таблицы или для какого-то конкретного индекса. Итого что бы отключить автоматическое обновление статистики у вас есть несколько вариантов:

- c) ALTER DATABASE dbname SET AUTO_UPDATE_STATISTICS
- d) sp_autostats
- e) используйте STATISTICS_NORECOMPUTE в команде CREATE INDEX
- f) используйте NORECOMPUTE в STATISTICS UPDATES или CREATE STATISTICS

Как только автоматическое обновление будет отключено, вы будете вынуждены обновлять статистику вручную. Эта операция может быть выполнена при сопровождении индексов или операцией UPDATE STATISTICS.
Полное описание UPDATE STATISTICS стоит изучить по BOL.

Создание статистики

Создание индексов и статистики достаточно простой процесс и выполняется командами CREATE INDEX и CREATE STATISTICS. Давайте посмотрим как же индексы создаются.

Статистика

Мы видели что система может сама создавать статистику для полей таблицы когда индекс не существует. Это работает, но отнимает ресурсы при выполнении запроса. Более того – автоматически создаваемая статистика создается только для одного поля, а для оптимизатора может быть необходима статистика по нескольким полям сразу. Для изучения синтаксиса CREATE STATISTICS обратитесь к BOL. Помните что статистика оказывает воздействие на решения принимаемые оптимизатором при построении плана запроса, но не оказывает воздействие на скорость выполнения запроса. Эта функция лежит на индексах.

Транзакции

Общие сведения о транзакциях

Транзакция — это последовательность операций, выполняемая как единое целое. В составе транзакций можно исполнять почти все операторы языка Transact-SQL. Если при выполнении транзакции не возникает никаких ошибок, то все модификации базы данных, сделанные во время выполнения транзакции, становятся постоянными. Транзакция выполняется по принципу «все или ничего». Транзакция не оставляет данные в промежуточном состоянии, в котором база данных не согласована. Транзакция переводит базу данных из одного целостного состояния в другое.

В качестве примера транзакции рассмотрим последовательность операций по приему заказа в коммерческой компании. Для приема заказа от клиента приложение ввода заказов должно:

- a) выполнить запрос к таблице *товаров* и проверить наличие товара на складе;
- b) добавить заказ к таблице *счетов*;
- c) обновить таблицу *товаров*, вычитя заказанное количество товаров из количества товара, имеющегося в наличии;
- d) обновить таблицу *продаж*, добавив стоимость заказа к объему продаж служащего, принявшего заказ;
- e) обновить таблицу *офисов*, добавив стоимость заказа к объему продаж офиса, в котором работает данный служащий.

Для поддержания целостности транзакция должна обладать четырьмя свойствами АСИД: *атомарность, согласованность, изоляция и долговечность*. Эти свойства называются также ACID-свойствами (от англ., atomicity, consistency, isolation, durability).

- **Атомарность (Atomicity).** Транзакция должна представлять собой атомарную (неделимую) единицу работы (исполняются либо все модификации, из которых состоит транзакция, либо ни одна).
- **Согласованность (или Непротиворечивость) (Consistency).** По завершении транзакции все данные должны остаться в согласованном состоянии. Чтобы сохранить целостность всех данных, необходимо выполнение модификации транзакций по всем правилам, определенным в реляционных СУБД.
- **Изоляция (Isolation).** Модификации, выполняемые одними транзакциями, следует изолировать от модификаций, выполняемых другими транзакциями параллельно. Уровни изоляции транзакции могут изменяться в широких пределах. На каждом уровне изоляции достигается определенный компромисс между степенью распараллеливания и степенью непротиворечивости. Чем выше уровень изоляции, тем выше степень непротиворечивости данных. Но чем выше степень непротиворечивости, тем ниже степень распараллеливания и тем ниже степень доступности данных.
- **Долговечность (или Устойчивость) (Durability).** По завершении транзакции ее результат должен сохраняться в системе, несмотря на сбой системы, либо (что касается незафиксированных транзакций) может быть полностью отменен вслед за сбоем системы.

Способы определения границ транзакций

Замечания.

- 1) В данной теме не будут учитываться особенности режима MARS (Multiple Active Result Sets). Режим MARS — это функция, которая появилась в SQL Server 2005 и ADO.NET 2.0 для выполнения нескольких пакетов по одному соединению. По умолчанию режим MARS отключен. Включить его можно, добавив в строку соединения параметр «MultipleActiveResultSets=True».
- 2) В данной теме не будут рассматриваться распределенные транзакции.

По признаку определения границ различают *автоматические, неявные и явные* транзакции.

Автоматические транзакции. Режим автоматической фиксации транзакций является режимом управления транзакциями SQL Server по умолчанию. В этом режиме каждая инструкция T-SQL выполняется как отдельная транзакция. Если выполнение инструкции завершается успешно, происходит фиксация; в противном случае происходит откат. Если возникает ошибка компиляции, то план выполнения пакета не строится и пакет не выполняется.

В следующем примере ни одна из инструкций INSERT во втором пакете не выполнится из-за ошибки компиляции. При этом произойдет откат первых двух инструкций INSERT, и они не будут выполняться:

```
CREATE TABLE Tab1 (
    Col1 int NOT NULL PRIMARY KEY,
    Col2 char(3)
);
GO
INSERT INTO Tab1 VALUES (1, 'aaa');
INSERT INTO Tab1 VALUES (2, 'bbb');
INSERT INTO Tab1 VALUESE (3, 'ccc') -- Синтаксическая ошибка.
```

```
GO
SELECT * FROM Tab1; -- Результат пустой.
GO
```

В следующем примере третья инструкция INSERT вызывает ошибку дублирования первичного ключа во время выполнения. Поэтому первые две инструкции INSERT выполняются успешно и фиксируются, а третья инструкция INSERT вызывает ошибку времени выполнения и не выполняется.

```
CREATE TABLE Tab1 (
    Col1 int NOT NULL PRIMARY KEY,
    Col2 char(3)
);
GO
INSERT INTO Tab1 VALUES (1, 'aaa');
INSERT INTO Tab1 VALUES (2, 'bbb');
INSERT INTO Tab1 VALUES (1, 'ccc'); -- Ошибка времени исполнения.
GO
SELECT * FROM Tab1; -- Возвращаются две строки.
GO
```

Неявные транзакции. Если соединение работает в режиме неявных транзакций, то после фиксации или отката текущей транзакции SQL Server автоматически начинает новую транзакцию. В этом режиме явно указывается только граница окончания транзакции с помощью инструкций COMMIT TRANSACTION и ROLLBACK TRANSACTION. Для ввода в действие поддержки неявных транзакций применяется инструкция SET IMPLICIT_TRANSACTION ON. В конце каждого пакета необходимо отключать этот режим. По умолчанию режим неявных транзакций в SQL Server отключен.

В следующем примере сначала создается таблица Tab1, затем включается режим неявных транзакций, после чего начинаются две транзакции. После их исполнения режим неявных транзакций отключается:

```
CREATE TABLE Tab1 (
    Col1 int NOT NULL PRIMARY KEY,
    Col2 char(3) NOT NULL
)
GO
SET IMPLICIT_TRANSACTIONS ON
-- Первая неявная транзакция, начатая оператором INSERT
INSERT INTO Tab1 VALUES (1, 'aaa')
INSERT INTO Tab1 VALUES (2, 'bbb')
COMMIT TRANSACTION -- Фиксация первой транзакции
-- Вторая неявная транзакция, начатая оператором INSERT
INSERT INTO Tab1 VALUES (3, 'ccc')
SELECT * FROM Tab1
COMMIT TRANSACTION -- Фиксация второй транзакции
SET IMPLICIT_TRANSACTIONS OFF
GO
```

Явные транзакции. Для определения явных транзакций используются следующие инструкции:

- BEGIN TRANSACTION – задает начальную точку явной транзакции для соединения;
- COMMIT TRANSACTION или COMMIT WORK – используется для успешного завершения транзакции, если не возникла ошибка;
- ROLLBACK TRANSACTION или ROLLBACK WORK – используется для отмены транзакции, во время которой возникла ошибка.
- SAVE TRANSACTION – используется для установки точки сохранения или маркера внутри транзакции. Точка сохранения определяет место, к которому может возвратиться транзакция, если часть транзакции условно отменена. Если транзакция откатывается к точке сохранения, то ее выполнение должно быть продолжено до завершения с обработкой дополнительных инструкций языка T-SQL, если необходимо, и инструкции COMMIT TRANSACTION, либо транзакция должна быть полностью отменена откатом к началу. Для отмены всей транзакции следует использовать инструкцию ROLLBACK TRANSACTION; в этом случае отменяются все инструкции транзакции.

В следующем примере демонстрируется использование точки сохранения транзакции для отката изменений:

```
USE pubs
GO
BEGIN TRANSACTION royaltychange
    UPDATE titleauthor SET royaltypcr = 65
```

```

FROM titleauthor, titles
WHERE royaltyper = 75 AND titleauthor.title_id = titles.title_id
    AND title = 'The Gourmet Microwave'

UPDATE titleauthor SET royaltyper = 35
FROM titleauthor, titles
WHERE royaltyper = 25 AND titleauthor.title_id = titles.title_id
    AND title = 'The Gourmet Microwave'

SAVE TRANSACTION percentchanged
/* После того, как обновлено royaltyper для двух авторов, вставляется точка сохранения
percentchanged, а затем определяется, насколько изменится заработка авторов после
увеличения на 10 процентов цены книги */
UPDATE titles SET price = price * 1.1
WHERE title = 'The Gourmet Microwave'

SELECT (price * royalty * ytd_sales) * royaltyper
FROM titles, titleauthor
WHERE title = 'The Gourmet Microwave' AND titles.title_id = titleauthor.title_id
/* Откат транзакции до точки сохранения и фиксация транзакции в целом */
ROLLBACK TRANSACTION percentchanged
COMMIT TRANSACTION

```

Режим явных транзакций действует только на протяжении данной транзакции. После завершения явной транзакции соединение возвращается в режим, заданный до запуска этого режима, то есть в неявный или автоматический.

Функции для обработки транзакций

- @@TRANCOUNT возвращает число активных транзакций для текущего соединения. Инструкция BEGIN TRANSACTION увеличивает значение @@TRANCOUNT на 1, а инструкция ROLLBACK TRANSACTION уменьшает его до 0 (исключение — инструкция ROLLBACK TRANSACTION *имя_точки_сохранения*, которая не влияет на значение @@TRANCOUNT). Инструкции COMMIT TRANSACTION уменьшают значение @@TRANCOUNT на 1.
- XACT_STATE () сообщает о состоянии пользовательской транзакции текущего выполняемого запроса в соответствии с данными, представленными в таблице.

Возвращаемое значение	Пояснение
1	Текущий запрос содержит активную пользовательскую транзакцию и может выполнять любые действия, включая запись данных и фиксацию транзакции.
0	У текущего запроса нет активной пользовательской транзакции.
-1	В текущем запросе есть активная транзакция, однако произошла ошибка, из-за которой транзакция классифицируется как не фиксируемая. Запросу не удается зафиксировать транзакцию или выполнить откат до точки сохранения; можно только запросить полный откат транзакции.

Ограничения.

- 1) Функция @@TRANCOUNT не может использоваться для определения фиксируемости транзакции.
- 2) Функция XACT_STATE не может использоваться для определения наличия вложенных транзакций.

Ошибки, возникающие в процессе обработки транзакций

Если ошибка делает невозможным успешное выполнение транзакции, SQL Server автоматически выполняет ее откат и освобождает ресурсы, удерживаемые транзакцией. Если сетевое соединение клиента с SQL Server разорвано, то после того, как SQL Server получит уведомление от сети о разрыве соединения, выполняется откат всех необработанных транзакций для этого соединения. В случае сбоя клиентского приложения, выключения либо перезапуска клиентского компьютера соединение также будет разорвано, а SQL Server выполнит откат всех необработанных транзакций после получения уведомления о разрыве от сети. Если клиент выйдет из приложения, выполняется откат всех необработанных транзакций.

В случае ошибки (нарушения ограничения целостности) во время выполнения инструкции в пакете по умолчанию SQL Server выполнит откат только той инструкции, которая привела к ошибке. Это поведение можно изменить с помощью инструкции SET XACT_ABORT. После выполнения инструкции SET XACT_ABORT ON любая ошибка во время выполнения инструкции приведет к автоматическому откату текущей транзакции.

На случай возникновения ошибок код приложения должен содержать исправляющее действие: COMMIT или ROLLBACK. Эффективным средством для обработки ошибок, включая ошибки транзакций, является конструкция языка Transact-SQL TRY...CATCH.

Пример

```
USE MyDB;
GO
IF OBJECT_ID ( N'dbo.SaveTranExample', N'P' ) IS NOT NULL
DROP PROCEDURE dbo.SaveTranExample;
GO
CREATE PROCEDURE dbo.SaveTranExample ... -- Список параметров
AS
-- Надо проверить, была ли процедура вызвана из активной транзакции, и если да,
-- то установить точку сохранения для последующего использования.
-- @TranCounter = 0 означает, что активной транзакции нет и процедура явно начинает
-- локальную транзакцию.
-- @TranCounter > 0 означает, что активная транзакция началась еще до вызова процедуры.
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT;
IF @TranCounter > 0
    SAVE TRANSACTION ProcedureSave;
ELSE
    BEGIN TRANSACTION;
-- Пытаемся модифицировать базу данных.
BEGIN TRY
-- INSERT, UPDATE, DELETE
...
-- Здесь мы окажемся, если не произойдет никакой ошибки.
-- Если транзакция началась внутри процедуры, то выполнить COMMIT TRANSACTION.
-- Если транзакция началась до вызова процедуры, то выполнять COMMIT TRANSACTION
нельзя.
    IF @TranCounter = 0
-- @TranCounter = 0 означает, что никакая транзакция до вызова процедуры не начиналась.
-- Процедура должна завершить начатую в ней транзакцию.
        COMMIT TRANSACTION;
END TRY
BEGIN CATCH
-- Здесь мы окажемся, если произошла ошибка.
-- Надо определить, какой тип отката транзакции применять.
    IF @TranCounter = 0
-- Транзакция началась в теле процедуры.
-- Полный откат транзакции.
        ROLLBACK TRANSACTION;
    ELSE
-- Транзакция началась еще до вызова процедуры.
-- Нельзя отменять изменения, сделанные до вызова процедуры.
        IF XACT_STATE() <> -1
-- Если транзакция активна, то выполнить откат до точки сохранения.
        ROLLBACK TRANSACTION ProcedureSave;
-- Если транзакция актива, однако произошла ошибка, из-за которой транзакция
-- классифицируется как нефиксированная,
-- то вернуться в точку вызова процедуры, где должен произойти откат внешней
транзакции.
-- В точку вызова процедуры передается информация об ошибках.
        DECLARE @ErrorMessage NVARCHAR(4000);
        DECLARE @ErrorSeverity INT;
        DECLARE @ErrorState INT;

        SELECT @ErrorMessage = ERROR_MESSAGE();
        SELECT @ErrorSeverity = ERROR_SEVERITY();
        SELECT @ErrorState = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH
```

Пояснения к примеру

Для получения сведений об ошибках в области блока CATCH конструкции TRY...CATCH можно использовать следующие системные функции:

- ERROR_LINE() - возвращает номер строки, в которой произошла ошибка.
- ERROR_MESSAGE() - возвращает текст сообщения, которое будет возвращено приложению. Текст содержит значения таких подставляемых параметров, как длина, имена объектов или время.
- ERROR_NUMBER() - возвращает номер ошибки.
- ERROR_PROCEDURE() - возвращает имя хранимой процедуры или триггера, в котором произошла ошибка. Эта функция возвращает значение NULL, если данная ошибка не была совершена внутри хранимой процедуры или триггера.
- ERROR_SEVERITY() - возвращает уровень серьезности ошибки.
- ERROR_STATE() - возвращает состояние.

Инструкция RAISERROR позволяет вернуть приложению сообщение в формате системных ошибок или предупреждений.

Управлением параллельным выполнением транзакций

Когда множество пользователей одновременно пытаются модифицировать данные в базе данных, необходимо создать систему управления, которая защитила бы модификации, выполненные одним пользователем, от негативного воздействия модификаций, сделанных другими. Выделяют два типа управления параллельным выполнением:

- **Пессимистическое управление параллельным выполнением.**
- **Оптимистическое управление параллельным выполнением.**

Пессимистическое управление реализуется с помощью технологии блокировок, оптимистическое управления реализуется с помощью технологии версии строк.

Система блокировок не разрешает пользователям выполнить модификации, влияющие на других пользователей. Если пользователь выполнил какое-либо действие, в результате которого установлена блокировка, то другим пользователям не удастся выполнять действия, конфликтующие с установленной блокировкой, пока владелец не освободит ее. Пессимистическое управление используется главным образом в средах, где высока конкуренция за данные.

В случае управления на основе версий строки пользователи не блокируют данные при чтении. Во время обновления система следит, не изменил ли другой пользователь данные после их прочтения. Если другой пользователь модифицировал данные, генерируется ошибка. Как правило, получивший ошибку пользователь откатывает транзакцию и повторяет операцию снова. Этот способ управления в основном используется в средах с низкой конкуренцией за данные.

SQL Server поддерживает различные механизмы оптимистического и пессимистического управления параллельным выполнением. Пользователю предоставляется право определить тип управления параллельным выполнением, установив уровень изоляции транзакции для соединения и параметры параллельного выполнения для курсоров.

Если в случае пессимистического управления в СУБД не реализованы механизмы блокирования, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть следующие проблемы одновременного доступа:

проблема последнего изменения возникает, когда несколько пользователей изменяют одну и ту же строку, основываясь на ее начальном значении; тогда часть данных будет потеряна, т.к. каждая последующая транзакция перезапишет изменения, сделанные предыдущей. Выход из этой ситуации заключается в последовательном внесении изменений;

проблема "грязного" чтения (Dirty Read) возможна в том случае, если пользователь выполняет сложные операции обработки данных, требующие множественного изменения данных перед тем, как они обретут логически верное состояние. Если во время изменения данных другой пользователь будет считывать их, то может оказаться, что он получит логически неверную информацию. Для исключения подобных проблем необходимо производить считывание данных после окончания всех изменений;

проблема неповторяемого чтения (Non-repeatable or Fuzzy Read) является следствием неоднократного считывания транзакцией одних и тех же данных. Во время выполнения первой транзакции другая может внести в данные изменения, поэтому при повторном чтении первая транзакция получит уже иной набор данных, что приводит к нарушению их целостности или логической несогласованности;

проблема чтения фантомов (Phantom) появляется после того, как одна транзакция выбирает данные из таблицы, а

другая вставляет или удаляет строки до завершения первой. Выбранные из таблицы значения будут некорректны.

Для решения перечисленных проблем в стандарте ANSI SQL определены четыре уровня блокирования. Уровень изоляции транзакции определяет, могут ли другие (конкурирующие) транзакции вносить изменения в данные, измененные текущей транзакцией, а также может ли текущая транзакция видеть изменения, произведенные конкурирующими транзакциями, и наоборот. Каждый последующий уровень поддерживает требования предыдущего и налагает дополнительные ограничения:

уровень 0 – запрещение «загрязнения» данных. Этот уровень требует, чтобы изменять данные могла только одна транзакция; если другой транзакции необходимо изменить те же данные, она должна ожидать завершения первой транзакции;

уровень 1 – запрещение «грязного» чтения. Если транзакция начала изменение данных, то никакая другая транзакция не сможет прочитать их до завершения первой;

уровень 2 – запрещение неповторяющегося чтения. Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. Таким образом, при повторном чтении они будут находиться в первоначальном состоянии;

уровень 3 – запрещение фантомов. Если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющиеся строки, которые могут быть считаны при выполнении транзакции. Реализация этого уровня блокирования выполняется путем использования блокировок диапазона ключей. Подобная блокировка накладывается не на конкретные строки таблицы, а на строки, удовлетворяющие определенному логическому условию.

Проблемы, связанные с параллельным выполнением транзакций, разрешают, используя уровни изоляции транзакции. От уровня изоляции зависит то, в какой степени транзакция влияет на другие транзакции и испытывает влияние других транзакций. Более низкий уровень изоляции увеличивает возможность параллельного выполнения, но за это приходится расплачиваться согласованностью данных. Напротив, более высокий уровень изоляции гарантирует согласованность данных, но при этом страдает параллельное выполнение.

Стандарт ISO определяет следующие уровни изоляции:

- **read uncommitted** (самый низкий уровень, при котором транзакции изолируются до такой степени, чтобы только уберечь от считывания физически поврежденных данных);
- **read committed** (уровень по умолчанию);
- изоляция повторяющегося чтения **repeatable read**;
- изоляция упорядочиваемых транзакций **serializable** (самый высокий уровень, при котором транзакции полностью изолированы друг от друга).

SQL Server также поддерживает еще два уровня изоляции транзакций, использующих управление версиями строк.

- **read committed** с использованием управления версиями строк;
- уровень изоляции моментальных снимков **snapshot**.

Следующая таблица показывает побочные эффекты параллелизма, допускаемые различными уровнями изоляции.

Уровень изоляции	«Грязное» чтение	Неповторяющееся чтение	Фантомное чтение
read uncommitted	Да	Да	Да
read committed	Нет	Да	Да
repeatable read	Нет	Нет	Да
snapshot	Нет	Нет	Нет
serializable	Нет	Нет	Нет

Для установки уровня изоляции используется следующая инструкция SET TRANSACTION ISOLATION LEVEL, имеющая следующий синтаксис:

```
SET TRANSACTION ISOLATION LEVEL  
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SNAPSHOT | SERIALIZABLE }  
[ ; ]
```

Описание аргументов

READ UNCOMMITTED - указывает, что инструкции могут считывать строки, которые были изменены другими транзакциями, но еще не были зафиксированы.

READ COMMITTED - указывает, что инструкции не могут считывать данные, которые были изменены другими транзакциями, но еще не были зафиксированы. Это предотвращает чтение «грязных» данных. Данные могут быть изменены другими транзакциями между отдельными инструкциями в текущей транзакции, результатом чего будет неповторяемое чтение или недействительные данные.

Напоминание. Поведение **READ COMMITTED** зависит от настройки аргумента базы данных **READ_COMMITTED_SNAPSHOT** (находится в состоянии OFF по умолчанию). **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.

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. Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить параметр базы данных **ALLOW_SNAPSHOT_ISOLATION** в ON. Если транзакция с уровнем изоляции моментального снимка обращается к данным из нескольких баз данных, аргумент **ALLOW_SNAPSHOT_ISOLATION** должен быть включен в каждой базе данных.

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.

Замечания.

1. Одновременно может быть установлен только один параметр уровня изоляции, который продолжает действовать для текущего соединения до тех пор, пока не будет явно изменен.
2. Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания.
3. 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.
4. Когда для транзакции изменяется уровень изоляции, ресурсы, которыечитываются после изменения, защищаются в соответствии с правилами нового уровня. Ресурсы, которыечитываются до изменения, остаются защищенными в 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.
5. 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**.
6. Определяемые пользователем функции и типы данных среды CLR не могут выполнять инструкцию **SET TRANSACTION ISOLATION LEVEL**. Однако уровень изоляции можно переопределить с помощью табличной подсказки.

Общие сведения о блокировках

Блокировка — это механизм, с помощью которого компонент Database Engine синхронизирует одновременный доступ нескольких пользователей к одному фрагменту данных. Прежде чем транзакция сможет распорядиться 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 транзакцией. 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). Приложения обычно не запрашивают блокировку напрямую. 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. Использование блокировки как механизма управления транзакциями может разрешить проблемы параллелизма. Блокировка позволяет запускать все транзакции в полной изоляции друг от друга, что позволяет одновременно выполнять несколько транзакций. Уровень, на котором транзакция готова к принятию противоречивых данных, называется уровнем изоляции. Чем выше уровень изоляции, тем ниже вероятность непоследовательности данных, однако при этом появляется такой недостаток, как сокращение параллелизма.

Каждая блокировка обладает тремя свойствами: *гранулярностью* (или размером блокировки), *режимом* (или типом блокировки) и *продолжительностью*.

Гранулярность блокировок и иерархии блокировок

SQL Server поддерживает мультигранулярную блокировку, позволяющую транзакции блокировать различные типы ресурсов. Чтобы уменьшить издержки применения блокировок, компонент Database Engine автоматически блокирует ресурсы на соответствующем уровне. Блокировка при меньшей гранулярности, например на уровне строк, увеличивает параллелизм, но в то же время увеличивает и накладные расходы на обработку, поскольку при большом количестве блокируемых строк требуется больше блокировок. Блокировки на большем уровне гранулярности, например на уровне таблиц, обходится дорого в отношении параллелизма, поскольку блокировка целой таблицы ограничивает доступ ко всем частям таблицы других транзакций. Однако накладные расходы в этом случае ниже, поскольку меньше количество поддерживаемых блокировок.

Компонент Database Engine часто получает блокировки на нескольких уровнях гранулярности одновременно, чтобы полностью защитить ресурс. Такая группа блокировок на нескольких уровнях гранулярности называется иерархией блокировки. Например, чтобы полностью защитить операцию чтения индекса, экземпляру компоненту Database Engine может потребоваться получить разделяемые блокировки на строки и намеренные разделяемые блокировки на страницы и таблицу.

Следующая таблица содержит перечень ресурсов, которые могут блокироваться компонентом Database Engine.

Ресурс	Описание
RID	Идентификатор строки, используемый для блокировки одной строки в куче
KEY	Блокировка строки в индексе, используемая для защиты диапазонов значений ключа в сериализуемых транзакциях.
PAGE	8-килобайтовая (КБ) страница в базе данных, например страница данных или индекса.
EXTENT	Упорядоченная группа из восьми страниц, например страниц данных или индекса.
НОВТ	Куча или сбалансированное дерево. Блокировка, защищающая индекс или кучу страниц данных в таблице, не имеющей кластеризованного индекса.
TABLE	Таблица полностью, включая все данные и индексы.
FILE	Файл базы данных.
APPLICATION	Определяемый приложением ресурс.
METADATA	Блокировки метаданных.
ALLOCATION_UNIT	Единица размещения.
DATABASE	База данных, полностью.

По умолчанию блокировки укрупняются с уровня отдельных строк до целых страниц даже до уровня таблицы из соображений повышения производительности. Хотя в общем случае укрупнение считается хорошей штукой, оно может создавать проблемы, например, когда один SPID блокирует всю таблицу, препятствуя другому SPID работатьней. Можно настроить параметры блокировки на уровне строк и страниц. Такие блокировки по умолчанию разрешены для индексов.

Режимы блокировки

SQL Server блокирует ресурсы с помощью различных режимов блокировки, которые определяют доступ одновременных транзакций к ресурсам. В следующей таблице показаны режимы блокировки ресурсов, применяемые компонентом Database Engine.

Режим блокировки	Описание
Совмещаемая	Используется для операций считывания, которые не меняют и не обновляют данные, такие как

блокировка (S)	инструкция SELECT.
Блокировка обновления (U)	Применяется к тем ресурсам, которые могут быть обновлены. Предотвращает возникновение распространенной формы взаимоблокировки, возникающей тогда, когда несколько сеансов считывают, блокируют и затем, возможно, обновляют ресурс.
Монопольная блокировка (X)	Используется для операций модификации данных, таких как инструкции INSERT, UPDATE или DELETE. Гарантирует, что несколько обновлений не будет выполнено одновременно для одного ресурса.
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. Типы 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.
Диапазон ключей	Защищает диапазон строк, считываемый запросом при использовании уровня изоляции сериализуемой транзакции. Запрещает другим транзакциям вставлять строки, что помогает запросам сериализуемой транзакции уточнять, были ли запросы запущены повторно.

Совмещаемые блокировки. Совмещаемые (S) блокировки позволяют одновременным транзакциям считывать (SELECT) ресурс под контролем пессимистичного параллелизма. Пока для ресурса существуют совмещаемые (S) блокировки, другие транзакции не могут изменять данные. 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 указания блокировки.

Блокировки обновления. Блокировки обновления (U) предотвращают возникновение распространенной формы взаимоблокировки. В сериализуемой транзакции или транзакции операцией чтения с возможностью повторения транзакция считывает данные, запрашивает совмещаемую (S) блокировку на ресурс (страницу или строку), затем выполняет изменение данных, что требует преобразование блокировки в монопольную (X). 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.

Чтобы избежать этой потенциальной взаимоблокировки, применяются блокировки обновления (U). Блокировку обновления (U) может устанавливать для ресурса одновременно только одна транзакция. Если транзакция изменяет ресурс, то блокировка обновления (U) преобразуется в монопольную (X) блокировку.

Монопольные блокировки. Монопольная (X) блокировка запрещает транзакциям одновременный доступ к ресурсу. Если ресурс удерживается монопольной (X) блокировкой, то другие транзакции не могут изменять данные. Операции считывания будут допускаться только при наличии указания NOLOCK или уровня изоляции незафиксированной операции чтения. 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 получает данные с помощью операций считывания. Поэтому, как правило, инструкции изменения данных запрашивают как совмещаемые, так и монопольные блокировки. 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 Компонент Database Engine блокировки с намерением применяются для защиты размещения совмещаемой (S) или монопольной (X) блокировки ресурса на более низком уровне иерархии. Блокировки с намерением называются так потому, что их получают 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 транзакциями таким образом, что это сделает недействительной блокировку более низкого уровня;
- повышает эффективность компонента Компонент 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.

Блокировка с намерением повышает производительность, поскольку компонент Database Engine проверяет наличие таких блокировок только на уровне таблицы, чтобы определить, может ли транзакция безопасно получить для этой таблицы совместную блокировку. Благодаря этому нет необходимости проверять блокировки в каждой строке и на каждой странице, чтобы убедиться, что транзакция может заблокировать всю таблицу.

Блокировки схем. В компоненте Компонент Database Engine блокировка изменения схемы (Sch-M) применяется с операциями языка DDL для таблиц, например при добавлении столбца или очистке таблицы. Пока удерживается блокировка изменения схемы (Sch-M), одновременный доступ к таблице запрещен. Это означает, что любые операции вне блокировки изменения схемы (Sch-M) будут запрещены до снятия блокировки.

- 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 Компонент 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), не могут выполняться над таблицей.

Блокировки массового обновления. Блокировка массового обновления (BU) 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. Компонент Database Engine использует блокировки массового обновления (BU), если выполняются два следующих условия. Используется инструкция Transact-SQL BULK INSERT, функция OPENROWSET(BULK) или одна из таких команд массовой вставки API, как .NET SqlBulkCopy, OLEDB Fast Load APIs или ODBC Bulk Copy APIs, для массового копирования данных в таблицу. Выделено указание TABLOCK или установлен параметр таблицы table lock on bulk load с помощью хранимой процедуры sp_tableoption.

Блокировки диапазона ключа. Блокировки диапазона ключей защищают диапазон строк, неявно включенный в набор записей, считываемый инструкцией Transact-SQL при использовании уровня изоляции сериализуемых транзакций. Блокировка диапазона ключей предотвращает фантомные чтения. Кроме того, защита диапазона ключей между строк предотвращает фантомную вставку или удаление из набора записи, к которому получает доступ транзакция.

Совместимость блокировок

Совместимость блокировок определяет, могут ли несколько транзакций одновременно получить блокировку одного и того же ресурса. Если ресурс уже блокирован другой транзакцией, новая блокировка может быть предоставлена только в том случае, если режим запрошенной блокировки совместим с режимом существующей. В противном случае транзакция, запросившая новую блокировку, ожидает освобождения ресурса, пока не истечет время ожидания существующей блокировки.

Например, с монопольными блокировками не совместим ни один из режимов блокировки. Пока удерживается монопольная (X) блокировка, больше ни одна из транзакций не может получить блокировку ни одного из типов (разделяемую, обновления или монопольную) на этот ресурс, пока не будет освобождена монопольная (X) блокировка. И наоборот, если к ресурсу применяется разделяемая (S) блокировка, другие транзакции могут получать разделяемую блокировку или блокировку обновления (U) на этот элемент, даже если не завершилась первая транзакция. Тем не менее, другие транзакции не могут получить монопольную блокировку до освобождения разделяемой. Полная матрица совместимости блокировок приводится в справочниках.

Продолжительность блокировки

Продолжительность блокировок также определяется типами запросов. Когда в рамках транзакции запрос не выполняется, и не используются подсказки блокировки, блокировки для выполнения инструкций SELECT выполняются только на время чтения ресурса, но не во время запроса. Блокировки для инструкций INSERT, UPDATE и DELETE сохраняются на все время выполнения запроса. Это помогает гарантировать согласованность данных и позволяет SQL Server откатывать запросы в случае необходимости.

Когда запрос выполняется в рамках транзакции, продолжительность блокировки определяется тремя факторами:

- типом запроса;
- уровнем изоляции транзакции;
- наличием или отсутствием подсказок блокировки.

Кратковременные (locking) и обычные (blocking) блокировки — нормальное явление в реляционных базах данных, но они могут ухудшать производительность, если блокировки ресурсов сохраняются на протяжении длительного времени. Производительность также страдает, когда, заблокировав ресурс, SPID не в состоянии освободить его.

В первом случае проблема обычно разрешается через какое-то время, так как SPID, в конечном счете, освобождает блокировку, но угроза деградации производительности остается вполне реальной. Проблемы с блокировкой второго типа

могут вызвать серьезное падение производительности, но, к счастью, они легко обнаруживаются при мониторинге SQL Server на предмет кратковременных и обычных блокировок.

Эскалация блокировок и их влияние на работу системы

Эскалация (lock escalation) связана с тем, что по мере увеличения количества отдельных малых заблокированных объектов накладные расходы, связанные с их поддержкой, начинают значительно сказываться на производительности. Блокировки делятся дольше, что приводит к спорным ситуациям – чем дольше существует блокировка, тем выше вероятность обращения к заблокированному объекту со стороны другой транзакции. Очевидно, что на некотором этапе потребуется выполнить объединение (увеличение масштаба) блокировок, чем собственно и занимается диспетчер блокировок. В инструкции ALTER TABLE предусмотрена опция вида SET (LOCK_ESCALATION = { AUTO | TABLE | DISABLE }), которая указывает разрешенные методы укрупнения блокировки для таблицы.

Задание определенного типа блокировки в запросе

Для повышения эффективности исполнения запросов и получения дополнительного контроля над блокировками в запросе можно давать подсказки (hints или хинты), указывая их непосредственно за именем таблицы, которая нуждается в том или ином типе блокировки. Существуют следующие параметры оптимизатора запросов:

SERIALIZABLE/HOLDLOCK
READUNCOMMITTED/NOLOCK
READCOMMITTED
REPEATABLEREAD
READPAST
ROWLOCK
PAGLOCK
TABLOCK
TABLOCKX
UPDLOCK
XLOCK

Пример задания эксклюзивной табличной блокировки для таблицы Orders (вместо блокировки на уровне ключей или строк, которую может предложить оптимизатор) может быть таким:

```
SELECT *
FROM Orders AS o WITH (TABLOCKX) JOIN [Order Details] AS od
ON o.OrderID = od.OrderID
```

Поскольку оптимизатор запросов обычно выбирает лучший план выполнения запроса, использовать подсказки рекомендуется только опытным разработчикам и администраторам баз данных в самом крайнем случае. Сведения об активных блокировках на текущий момент времени можно получить с помощью системной хранимой процедуры sp_lock.

Взаимоблокировки транзакций

Взаимоблокировки или **тупиковые ситуации (deadlocks)** возникают тогда, когда одна из транзакций не может завершить свои действия, поскольку вторая транзакция заблокировала нужные ей ресурсы, а вторая в то же время ожидает освобождения ресурсов первой транзакцией. На рисунке транзакция T1 зависит от транзакции T2 для ресурса блокировки таблицы Детали. Аналогично транзакция T2 зависит от транзакции T1 для ресурса блокировки таблицы Поставщик. Так как эти зависимости из одного цикла, возникает взаимоблокировка транзакций T1 и T2.



Транзакция T1 не может завершиться до того, как завершится транзакция T2, а транзакция T2 заблокирована транзакцией T1. Такое условие также называется циклической зависимостью: транзакция T1 зависит от транзакции T2, а транзакция T2

зависит от транзакции T1 и этим замыкает цикл. Обе транзакции находятся в состоянии взаимоблокировки и будут всегда находиться в состоянии ожидания, если взаимоблокировка не будет разрушена внешним процессом.

Взаимоблокировки часто путают с обычными блокировками. Если транзакция запрашивает блокировку на ресурс, заблокированный другой транзакцией, то запрашивающая транзакция ожидает до тех пор, пока блокировка не освобождается. По умолчанию время ожидания транзакций SQL Server не ограничено, если только не установлен параметр LOCK_TIMEOUT.

Значение -1 (по умолчанию) указывает на отсутствие времени ожидания (то есть инструкция будет ждать всегда). Когда ожидание блокировки превышает значение времени ожидания, возвращается ошибка. Значение «0» означает, что ожидание отсутствует, а сообщение возвращается, как только встречается блокировка. Функция @@LOCK_TIMEOUT возвращает значение времени ожидания блокировки в миллисекундах для текущего сеанса.

Запрашивающая транзакция блокируется, но не устанавливается в состояние взаимоблокировки, потому что запрашивающая транзакция ничего не сделала, чтобы заблокировать транзакцию, владеющую блокировкой. Наконец, владеющая транзакция завершится и освободит блокировку, и затем запрашивающая транзакция получит блокировку и продолжится.

Как SQL Server обнаруживает взаимоблокировки?

Обнаружение взаимоблокировки выполняется потоком диспетчера блокировок, который периодически производит поиск по всем задачам в экземпляре компонента Database Engine. Следующие пункты описывают процесс поиска:

- Значение интервала поиска по умолчанию составляет 5 секунд.
- Если диспетчер блокировок находит взаимоблокировки, интервал обнаружения взаимоблокировок снижается с 5 секунд до 100 миллисекунд в зависимости от частоты взаимоблокировок.
- Если поток диспетчера блокировки прекращает поиск взаимоблокировок, компонент Database Engine увеличивает интервал до 5 секунд.
- Если взаимоблокировка была только что найдена, предполагается, что следующие потоки, которые должны ожидать блокировки, входят в цикл взаимоблокировки. Первая пара элементов, ожидающих блокировки, после того как взаимоблокировка была обнаружена, запускает поиск взаимоблокировок вместо того, чтобы ожидать следующий интервал обнаружения взаимоблокировки. Например, если текущее значение интервала равно 5 секунд и была обнаружена взаимоблокировка, следующий ожидающий блокировки элемент немедленно приводит в действие детектор взаимоблокировок. Если этот ожидающий блокировки элемент является частью взаимоблокировки, она будет обнаружена немедленно, а не во время следующего поиска взаимоблокировок.
- Компонент Database Engine обычно выполняет только периодическое обнаружение взаимоблокировок. Так как число взаимоблокировок, произошедших в системе, обычно мало, периодическое обнаружение взаимоблокировок помогает сократить издержки от взаимоблокировок в системе.
- Если монитор блокировок запускает поиск взаимоблокировок для определенного потока, он идентифицирует ресурс, ожидаемый потоком. После этого монитор блокировок находит владельцев определенного ресурса и рекурсивно продолжает поиск взаимоблокировок для этих потоков до тех пор, пока не найдет цикл. Цикл, определенный таким способом, формирует взаимоблокировку.
- После обнаружения взаимоблокировки компонент Database Engine завершает взаимоблокировку, выбрав один из потоков в качестве жертвы взаимоблокировки. Компонент Database Engine прерывает выполняемый в данный момент пакет потока, производит откат транзакции жертвы взаимоблокировки и возвращает приложению ошибку 1205. Откат транзакции жертвы взаимоблокировки снимает все блокировки, удерживаемые транзакцией. Это позволяет транзакциям потоков разблокироваться, и продолжить выполнение. Ошибка 1205 жертвы взаимоблокировки записывается в журнал ошибок сведениями обо всех потоках и ресурсах, затронутых взаимоблокировкой. Сведения об ошибках см. в

C:\Program Files\Microsoft SQL Server\MSSQL10_50.SQLEXPRESS\MSSQL\Log\ERRORLOG или ERRORLOG.n

[

Каждые пять секунд SQL Server проверяет состояние текущих транзакций на предмет наличия блокировок, которые ожидают своей очереди. В случае наличия подобных блокировок, SQL Server берет их на заметку. Через следующие пять секунд производится повторная проверка всех открытых блокировок, и если одна из отмеченных блокировок по-прежнему находится в состоянии ожидания, выполняется рекурсивная проверка всех открытых транзакций на предмет существования в очереди замкнутых циклов. Если такой цикл обнаружен, в нем выбираются одна или несколько «жертв» взаимоблокировки.

]

Как выбирается жертва взаимоблокировки?

По умолчанию в качестве жертвы взаимоблокировки выбирается сеанс, выполняющий ту транзакцию, откат которой потребует меньше всего затрат. В качестве альтернативы пользователь может указать приоритет сеансов, используя инструкцию SET DEADLOCK_PRIORITY. DEADLOCK_PRIORITY может принимать значения LOW, NORMAL или HIGH или в качестве альтернативы может принять любое целочисленное значение на отрезке [-10..10]. По умолчанию DEADLOCK_PRIORITY устанавливается на значение NORMAL. Если у двух сеансов имеются различные приоритеты, то в качестве жертвы взаимоблокировки будет выбран сеанс с более низким приоритетом. Если у обоих сеансов установлен одинаковый приоритет, то в качестве жертвы взаимоблокировки будет выбран сеанс, откат которого потребует наименьших затрат. Если сеансы, вовлеченные в цикл взаимоблокировки, имеют один и тот же приоритет и одинаковую стоимость, то жертва взаимоблокировки выбирается случайным образом.

Мониторинг транзакций и блокировок

В SQL Server для получения информации о состоянии сервера применяются динамические административные представления и функции, расположенные в схеме sys. Их имена следуют такому соглашению по именованию: dm_* (от англ. dynamic_management). Есть два типа динамических административных представлений и функций:

- **области сервера** - для них необходимо разрешение VIEW SERVER STATE на сервере;
- **области базы данных** - для них необходимо разрешение VIEW DATABASE STATE на базе данных.

Динамические административные представления и функции разбиты на категории: 20 в SQL Server 2012. Одна из категорий связана с транзакциями. В эту категорию входят 10 представлений и функций:

sys.dm_tran_active_snapshot_database_transactions - возвращает виртуальную таблицу всех активных транзакций, формирующих или потенциально получающих доступ к версиям строк

sys.dm_tran_active_transactions - возвращает данные о транзакциях для экземпляра SQL Server.

sys.dm_tran_current_snapshot - возвращает виртуальную таблицу, которая отображает все активные транзакции в момент запуска текущей транзакции моментального снимка.

sys.dm_tran_current_transaction - возвращает строку, которая отображает сведения о состоянии транзакции в текущей сессии.

sys.dm_tran_database_transactions - возвращает сведения о транзакциях на уровне базы данных.

sys.dm_tran_locks - возвращает строки, представляющие текущий активный запрос диспетчеру блокировок на блокировку, которая была наложена или находится в ожидании получения.

sys.dm_tran_session_transactions - возвращает информацию о том, как выполняется каждая транзакция во всех сеансах.

sys.dm_tran_top_version_generators - возвращает информацию о том, какие версии строк созданы в базах данных.

sys.dm_tran_transactions_snapshot - возвращает виртуальную таблицу номеров транзакций, активных при запуске каждой транзакции моментальных снимков.

sys.dm_tran_version_store - возвращает виртуальную таблицу, в которой отображаются все записи версий в хранилище версий.

Рекомендации по предотвращению взаимоблокировок

Невозможно полностью исключить возникновение взаимоблокировок в сложных программных системах, но с практической точки зрения можно сделать их вероятность настолько малой, что данный вопрос перестанет быть актуальным. Для того чтобы избежать появления взаимоблокировок или, по крайней мере, свести их количество к минимуму, надо следовать простым правилам:

- обращайтесь к объектам в одном и том же порядке;
- делайте транзакции как можно более короткими и размещайте их в одном пакете;
- используйте наименьший возможный уровень изоляции транзакций;
- не допускайте разрывов внутри транзакции (со стороны пользователя либо в результате разделения пакета);
- в контролируемой среде используйте связанные подключения.

Общие сведения об управлении версиями строк

Моментальный снимок 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 параметр базы данных ALLOW_SNAPSHOT_ISOLATION в ON. Если транзакция с уровнем изоляции моментального снимка обращается к данным из нескольких баз данных, аргумент ALLOW_SNAPSHOT_ISOLATION должен быть включен в каждой базе данных.

Невозможно изменить на уровень изоляции моментального снимка 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 моментального снимка, 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.

Моментальный снимок с уровнем изоляции READ COMMITTED. 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.

Дополнительный материал

Обработка ошибок в T-SQL

@@ERROR - Возвращает номер ошибки для последней выполненной инструкции Transact-SQL. Возвращает 0, если в предыдущей инструкции Transact-SQL не возникли ошибки. Просмотреть текст, связанный с номером ошибки @@ERROR можно в sys.messages.

@@ROWCOUNT - Возвращает число строк, затронутых при выполнении последней инструкции Transact-SQL.

@@TRANCOUNT - Возвращает число инструкций BEGIN TRANSACTION, выполненных в текущем соединении.

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Реализация обработчика ошибок на языке Transact-SQL. В области блока CATCH для получения сведений об ошибке, приведшей к выполнению данного блока CATCH, можно использовать следующие системные функции:

ERROR_NUMBER() возвращает номер ошибки.

ERROR_SEVERITY() возвращает степень серьезности ошибки.

ERROR_STATE() возвращает код состояния ошибки.

ERROR_PROCEDURE() возвращает имя хранимой процедуры или триггера, в котором произошла ошибка.

ERROR_LINE() возвращает номер строки, которая вызвала ошибку, внутри подпрограммы.

ERROR_MESSAGE() возвращает полный текст сообщения об ошибке. Текст содержит значения подставляемых параметров, таких как длина, имена объектов или время.

XACT_STATE() - Возвращает состояние транзакции текущего выполняемого запроса.

1 Текущий запрос содержит активную пользовательскую транзакцию. Запрос может выполнять любые действия, включая запись данных и фиксирование транзакции.

0 У текущего запроса нет активной пользовательской транзакции.

-1 В текущем запросе есть активная транзакция, однако произошла ошибка, из-за которой транзакция классифицируется как нефиксированная. Запрос не удается зафиксировать транзакцию или выполнить откат до точки сохранения; можно только запросить полный откат транзакции. Запрос не может выполнить никакие операции записи, пока не будет проведен откат транзакции. До отката транзакции запрос может выполнять только операции считывания. После отката транзакции запросу будут доступны как операции считывания, так и операции записи, а также запуск новых транзакций. После завершения работы пакета компонент Database Engine автоматически выполнит откат любых активных нефиксированных транзакций. Если при переходе транзакции в нефиксированное состояние не было отправлено сообщение об ошибке, после завершения выполнения пакета сообщение об ошибке будет отправлено клиентскому приложению. Сообщение показывает, что обнаружены нефиксированные транзакции и выполнен откат.

SET XACT_ABORT { ON | OFF } - Указывает, выполняет ли SQL Server автоматический откат текущей транзакции, если инструкция языка Transact-SQL вызывает ошибку выполнения. Если выполнена инструкция SET XACT_ABORT ON и инструкция языка Transact-SQL вызывает ошибку, вся транзакция завершается и выполняется ее откат. Если выполнена инструкция SET XACT_ABORT OFF, в некоторых случаях выполняется откат только вызвавшей ошибку инструкции языка Transact-SQL, а обработка транзакции продолжается.

Обработка ошибок в транзакциях (на примере)

```
USE tempdb
GO

CREATE TABLE dbo.Employees
(
    empid INT NOT NULL,
    empname VARCHAR(25) NOT NULL,
    mgrid INT NULL,
    CONSTRAINT PK_Employees PRIMARY KEY(empid),
    CONSTRAINT CHK_Employees_empid CHECK(empid > 0),
    CONSTRAINT FK_Employees_Employees FOREIGN KEY(mgrid) REFERENCES Employees(empid)
);

-- Запускаем следующий пример дважды:
BEGIN TRY
    INSERT INTO dbo.Employees(empid, empname, mgrid)
    VALUES(1, 'Emp1', NULL);
    PRINT 'INSERT succeeded.';
END TRY
BEGIN CATCH
    PRINT 'INSERT failed.';
    /* handle error here */
END CATCH

-- 1
-- (1 row(s) affected)
-- INSERT succeeded.

-- 2
-- INSERT failed.

SELECT * FROM dbo.Employees
GO

CREATE PROC dbo.AddEmp @empid AS INT, @empname AS VARCHAR(25), @mgrid AS INT
AS
DECLARE @tc AS INT = @@TRANCOUNT; -- Сохраняем количество внешних транзакций
IF @tc > 0 -- Если уже есть активная транзакция, то создаем точку сохранения S1
    SAVE TRAN S1;
ELSE
BEGIN TRAN -- Если активных транзакций нет, то открываем новую транзакцию
BEGIN TRY;
    -- Модифицируем данные
    INSERT INTO dbo.Employees(empid, empname, mgrid) VALUES(@empid, @empname, @mgrid);
    IF @tc = 0 -- Если процедура открыла транзакцию, то она ее и фиксирует
        COMMIT TRAN;
END TRY
BEGIN CATCH
    IF @tc = 0 -- Если процедура открыла транзакцию, то ...
    BEGIN
        IF XACT_STATE() <> 0 -- Если в текущем запросе есть активная транзакция, однако произошла
        ошибка, то
        BEGIN
            PRINT 'Откат Транзакции, открытой Процедурой.';
        END
    END
END CATCH
```

```

        ROLLBACK TRAN
    END
END
ELSE -- Если процедура не открывала транзакцию (т.е. существовали внешние транзакции), то
BEGIN
    IF XACT_STATE() = 1 -- Если текущий запрос содержит активную пользовательскую
транзакцию, то
        BEGIN
            PRINT 'Процедура была вызвана в открытой Транзакции. Откат до точки сохранения。';
            ROLLBACK TRAN S1
        END
    ELSE IF XACT_STATE() = -1 -- Если в текущем запросе есть активная транзакция, однако
произошла ошибка, из-за которой транзакция классифицируется как нефиксированная, то
        PRINT 'Процедура была вызвана в открытой Транзакции, но не зафиксирована. Передать
обработку ошибки в точку вызова процедуры。'
        END
    -- Создаем сообщение об ошибке, чтобы 'caller' решал, что делать с отказом в процедуре
    DECLARE @ErrorMessage NVARCHAR(400) = ERROR_MESSAGE(), @ErrorSeverity INT =
ERROR_SEVERITY(), @ErrorState INT = ERROR_STATE();
    RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH
GO

-- Чтобы проверить процедуру, сначала очистить таблицу 'Employees':
TRUNCATE TABLE dbo.Employees;
GO

-- Затем выполняем следующий код дважды, но не в явной транзакции:
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;

-- 1)
-- (1 row(s) affected)

-- 2)
-- (0 row(s) affected)
-- Откат Транзакции, открытой Процедурой.
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

-- Теперь запустите процедуру снова, но на этот раз в рамках явной транзакции:
BEGIN TRAN
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;
ROLLBACK

-- (0 row(s) affected)
-- Процедура была вызвана в открытой Транзакции. Откат до точки сохранения。
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

-- Последнее испытание
SET XACT_ABORT ON;
BEGIN TRAN
EXEC AddEmp @empid = 1, @empname = 'Emp1', @mgrid = NULL;
-- Обработка ошибки ...
ROLLBACK
SET XACT_ABORT OFF;

-- (0 row(s) affected)
-- Процедура была вызвана в открытой Транзакции, но не зафиксирована. Передать обработку ошибки в точку вызова
процедуры。
-- Msg 50000, Level 14, State 1, Procedure AddEmp, Line 35
-- Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

```

Транзакции и точки сохранения

```

USE tempdb;
GO

IF OBJECT_ID('dbo.Sequence', 'U') IS NOT NULL DROP TABLE dbo.Sequence;
GO

CREATE TABLE dbo.Sequence(val INT IDENTITY);
GO

IF OBJECT_ID('dbo.GetSequence', 'P') IS NOT NULL DROP PROC dbo.GetSequence;
GO

CREATE PROC dbo.GetSequence @val AS INT OUTPUT
AS
    BEGIN TRAN
        SAVE TRAN S1;
        INSERT INTO dbo.Sequence DEFAULT VALUES;
        SET @val = SCOPE_IDENTITY()
        ROLLBACK TRAN S1;
    COMMIT TRAN
GO

SELECT * FROM dbo.Sequence
GO
-- Пусто

DECLARE @key AS INT;
EXEC dbo.GetSequence @val = @key OUTPUT;
SELECT @key;
GO

SELECT * FROM dbo.Sequence
-- Снова пусто,
-- -- @key = 1, 2, 3, и т.д.
GO

```

Функции SCOPE_IDENTITY, IDENT_CURRENT и @@IDENTITY идентичны друг другу, поскольку возвращают значения, вставленные в столбцы идентификаторов.

Взаимоблокировки

```

SET NOCOUNT ON;

USE tempdb
GO

IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.T2', 'U') IS NOT NULL DROP TABLE dbo.T2;
GO

CREATE TABLE dbo.T1
(
keycol INT NOT NULL PRIMARY KEY,
col1 INT NOT NULL,
col2 VARCHAR(50) NOT NULL
);
INSERT INTO dbo.T1(keycol, col1, col2) VALUES
(1, 101, 'A'),
(2, 102, 'B'),
(3, 103, 'C');

CREATE TABLE dbo.T2
(

```

```

keycol INT NOT NULL PRIMARY KEY,
col1 INT NOT NULL,
col2 VARCHAR(50) NOT NULL
);
INSERT INTO dbo.T2(keycol, col1, col2) VALUES
(1, 201, 'X'),
(2, 202, 'Y'),
(3, 203, 'Z');
GO

SELECT * FROM dbo.T1
SELECT * FROM dbo.T2
GO

BEGIN TRAN
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');
INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN
GO

SELECT * FROM dbo.T1
SELECT * FROM dbo.T2
GO

-- connection 1:

SET NOCOUNT ON;
USE tempdb;
GO
BEGIN TRAN
UPDATE dbo.T1 SET col1 = col1 + 1 WHERE keycol = 2;

-- connection 2:

SET NOCOUNT ON;
USE tempdb;
GO
BEGIN TRAN
UPDATE dbo.T2 SET col1 = col1 + 1 WHERE keycol = 2;

-- connection 1, which attempts to read data from T2:

SELECT col1 FROM dbo.T2 WHERE keycol = 2;
COMMIT TRAN

-- Ждем

-- connection 2, attempting to query the data from T1:

SELECT col1 FROM dbo.T1 WHERE keycol = 2;
COMMIT TRAN

Msg 1205, Level 13, State 51, Server HOMEPC-ПК\SQLEXPRESS, Line 1
Transaction (Process ID 58) was deadlocked on lock resources with another process and has been chosen as the deadlock victim.
Rerun the transaction.

-- Только после этого в connection 1 имеем:

col1
-----
202

```

Тупики вызванные отсутствием индексов

```
BEGIN TRAN  
UPDATE dbo.T1 SET col2 = col2 + 'A' WHERE col1 = 101;
```

```
BEGIN TRAN  
UPDATE dbo.T2 SET col2 = col2 + 'B' WHERE col1 = 203;
```

```
SELECT col2 FROM dbo.T2 WHERE col1 = 201;  
COMMIT TRAN
```

-- Ждем

```
SELECT col2 FROM dbo.T1 WHERE col1 = 103;  
COMMIT TRAN
```

Конечно, возникает взаимоблокировка, и один из процессов "выбран", как жертва тупика (подключение 2 в данном случае), и вы получите следующее сообщение об ошибке:

```
Msg 1205, Level 13, State 51, Server HOMEPC-ПК\SQLEXPRESS, Line 1  
Transaction (Process ID 58) was deadlocked on lock resources with another process and has been chosen as the deadlock victim.  
Rerun the transaction.
```

Для предотвращения подобных тупиков в будущем, создаем следующие индексы:

```
CREATE INDEX idx_col1 ON dbo.T1(col1);  
CREATE INDEX idx_col1 ON dbo.T2(col1);
```

```
BEGIN TRAN  
UPDATE dbo.T1 SET col2 = col2 + 'A' WHERE col1 = 101;
```

```
BEGIN TRAN  
UPDATE dbo.T2 SET col2 = col2 + 'B' WHERE col1 = 203;
```

```
SELECT col2 FROM dbo.T2 WITH (index = idx_col1) WHERE col1 = 201;  
COMMIT TRAN
```

```
SELECT col2 FROM dbo.T1 WITH (index = idx_col1) WHERE col1 = 103;  
COMMIT TRAN
```

Запрос выполняется успешно, и транзакция фиксируется.

```
DROP INDEX idx_col1 ON dbo.T1(col1);  
DROP INDEX idx_col1 ON dbo.T2(col1);
```

Приложение

Инструкции SET и управление транзакциями

Язык T-SQL имеет несколько инструкций SET, которые изменяют параметры текущего сеанса:

1. Инструкции блокировки:
2. Инструкции управления транзакциями:

SET DEADLOCK_PRIORITY определяет относительную важность продолжения текущего сеанса, если произошла взаимоблокировка с другим сеансом. Синтаксис:

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | числовой_приоритет | @символьная_переменная |  
@числовая_переменная },
```

где *числовая-переменная* принимает значение из диапазона -10..10. NORMAL - приоритет по умолчанию. LOW соответствует -5, NORMAL - 0, HIGH - 5. Например, SET DEADLOCK_PRIORITY LOW означает, что текущий сеанс будет выбран в качестве жертвы в случае взаимоблокировки с другим сеансом, если другой сеанс входит в цепочку взаимного блокирования с приоритетом NORMAL, HIGH или равным целочисленному значению не менее -5. Текущий сеанс может быть выбран в качестве жертвы, если другому сеансу назначен приоритет LOW или равный целочисленному значению -5.

SET LOCK_TIMEOUT указывает количество миллисекунд, в течение которых инструкция ожидает снятия блокировки.
Синтаксис:

SET LOCK_TIMEOUT *количество-миллисекунд*

По умолчанию -1 (то есть инструкция будет ждать всегда). Например, SET LOCK_TIMEOUT 1800 устанавливает время ожидания снятия блокировки равным 1800 миллисекунд. Функция @@LOCK_TIMEOUT возвращает значение времени ожидания блокировки в миллисекундах для текущего сеанса.

SET IMPLICIT_TRANSACTIONS устанавливает для соединения режим неявных транзакций. Синтаксис:

SET IMPLICIT_TRANSACTIONS { ON | OFF }.

SET REMOTE_PROC_TRANSACTIONS указывает, что в момент, когда активна локальная транзакция, выполнение удаленной хранимой процедуры запускает распределенную транзакцию, управляемую координатором распределенных транзакций (MS DTC). Синтаксис:

SET REMOTE_PROC_TRANSACTIONS { ON | OFF }.

Рекомендуется вместо вызова удаленных хранимых процедур использовать распределенные запросы, ссылающиеся на связанные серверы, которые определяются с помощью хранимой процедуры sp_addlinkedserver.

SET XACT_ABORT указывает, выполняет ли SQL Server автоматический откат текущей транзакции, если инструкция T-SQL вызывает ошибку выполнения. Синтаксис:

SET XACT_ABORT { ON | OFF }.

OFF – установка по умолчанию. В случае ON вся транзакция завершается и выполняется ее откат. В случае OFF в зависимости от серьезности ошибки возможен откат всей транзакции или откат только вызвавшей ошибку инструкции.

SET TRANSACTION ISOLATION LEVEL управляет работой блокировки и версиями строк инструкций T-SQL в текущем сеансе. Свойства инструкции:

- 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 подсказка в предложении FROM инструкции не указывает на другое поведение блокировки или управления версиями строк для таблицы.
- Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания. Совмещаемые 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 времени выполнения транзакции можно переключиться с одного уровня изоляции на другой, однако есть одно исключение. 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.

Синтаксис:

```
SET TRANSACTION ISOLATION LEVEL {  
    READ UNCOMMITTED |  
    READ COMMITTED |  
    REPEATABLE READ |  
    SNAPSHOT |  
    SERIALIZABLE }
```

READ COMMITTED — установка по умолчанию.

READ UNCOMMITTED указывает, что транзакция может считывать строки, которые были изменены другими транзакциями, но еще не были зафиксированы.

READ COMMITTED указывает, что транзакция не может считывать строки, которые были изменены другими

транзакциями, но еще не были зафиксированы. Это предотвращает чтение «грязных» данных. Данные могут быть изменены другими транзакциями между отдельными инструкциями в текущей транзакции, результатом чего будет неповторяемое чтение или недействительные данные. Поведение READ COMMITTED зависит от настройки аргумента базы данных READ_COMMITTED_SNAPSHOT.

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.

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. Перед запуском транзакции, использующей уровень изоляции моментальных снимков, необходимо установить параметр базы данных ALLOW_SNAPSHOT_ISOLATION в ON.

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.

Журнализация

Базы данных SQL Server содержат файлы трех типов:

- **Первичные файлы данных.** Первичный файл данных является отправной точкой базы данных. Он указывает на остальные файлы базы данных. В каждой базе данных имеется один первичный файл данных. Для имени первичного файла данных рекомендуется использовать расширение MDF.
- **Вторичные файлы данных.** Ко вторичным файлам данных относятся все файлы данных, за исключением первичного файла данных. Некоторые базы данных могут вообще не содержать вторичных файлов данных, тогда как другие содержат несколько вторичных файлов данных. Для имени вторичного файла данных рекомендуется использовать расширение NDF.
- **Файлы журналов.** Файлы журналов содержат все сведения журналов, используемые для восстановления базы данных. В каждой базе данных должен быть, по меньшей мере, один файл журнала, но их может быть и больше. Для имен файлов журналов рекомендуется использовать расширение LDF. Журнал транзакций нельзя ни удалять, ни изменять, если только не известны возможные последствия. Кэш журнала управляется отдельно от буферного кэша для страниц данных.

Журнал транзакций поддерживает следующие операции:

- **Восстановление отдельных транзакций.** Если приложение выполняет инструкцию ROLLBACK или если компонент Database Engine обнаруживает ошибку, такую как потеря связи с клиентом, записи журнала используются для отката изменений, сделанных незавершенной транзакцией.
- **Восстановление всех незавершенных транзакций при запуске SQL Server.** Если на сервере, где работает SQL Server, происходит сбой, базы данных могут оставаться в таком состоянии, в котором часть изменений не была переписана из буферного кэша в файлы данных, и могут быть изменения в файлах данных, совершенные незаконченными транзакциями. Когда экземпляр SQL Server будет запущен, он выполнит восстановление каждой базы данных. Будет выполнен накат каждого записанного в журнал изменения, которое, возможно, не было переписано в файл данных. Чтобы сохранить целостность базы данных, будет также произведен откат каждой незавершенной транзакции, найденной в журнале транзакций.
- **Накат восстановленной базы данных, файла, файловой группы или страницы до момента сбоя.** После потери оборудования или сбоя диска, затрагивающего файлы базы данных, можно восстановить базу данных на момент, предшествующий сбою. Сначала восстановите последнюю полную резервную копию и последнюю дифференциальную резервную копию базы данных, затем восстановите последующую серию резервных копий журнала транзакций до момента возникновения сбоя. Поскольку восстанавливается каждая резервная копия журнала, компонент Database Engine повторно применяет все модификации, записанные в журнале, для наката всех транзакций. Когда последняя резервная копия журнала будет восстановлена, тогда компонент Database Engine начнет использовать данные журнала для отката всех транзакций, которые не были завершены на момент сбоя.
- **Поддержка репликации транзакций.** Агент чтения журнала следит за журналами транзакций всех баз данных, которые настроены на репликацию транзакций, и копирует отмеченные для репликации транзакции из журнала транзакций в базу данных распространителя. Дополнительные сведения см. в разделе Как работает репликация транзакций.
- **Поддержка решений с резервными серверами.** Решения резервного сервера, зеркальное отображение базы данных и доставка журналов в значительной степени полагаются на журнал транзакций. В сценарии доставки журналов основной сервер отправляет активный журнал транзакций основной базы данных одному или более адресатам. Каждый сервер-получатель восстанавливает журнал в свою локальную базу данных-получатель.

Операции резервного копирования и восстановления выполняются в контексте модели восстановления. **Модель восстановления** — это свойство базы данных, которое управляет процессом регистрации транзакций, определяет, требуется ли для журнала транзакций резервное копирование, а также определяет, какие типы операций восстановления доступны. Есть три модели восстановления:

- **Модель полного восстановления (FULL).** Обеспечивает модель обслуживания для баз данных, в которых необходима поддержка длительных транзакций. Требуются резервные копии журналов. При использовании этой модели выполняется полное протоколирование всех транзакций, и сохраняются записи журнала транзакций до момента их резервного копирования. Модель полного восстановления позволяет восстановить базу данных до точки сбоя при условии, что после сбоя возможно создание резервной копии заключительного фрагмента журнала. Кроме того, модель полного восстановления поддерживает восстановление отдельных страниц данных.
- **Модель восстановления с неполным протоколированием (BULK_LOGGED).** Эта модель восстановления обеспечивает неполное протоколирование большинства массовых операций. Она предназначена для работы только в качестве дополнения к полной модели полного восстановления. Для ряда масштабных массовых операций (массовый импорт, создание индекса и т. п.) временное переключение на модель восстановления с неполным протоколированием повышает производительность и уменьшает место, необходимое для журналов. Тем не менее, для работы этой модели требуются резервные копии журналов. Как и в модели полного восстановления, в модели восстановления с неполным протоколированием сохраняются записи журнала транзакций после его резервного копирования. Это увеличивает объем резервных копий журналов и повышает риск потери результатов работы,

поскольку модель восстановления с неполным протоколированием не поддерживает восстановление до заданного момента времени.

- **Простая модель восстановления (SIMPLE).** Сводит к минимуму административные затраты, связанные с журналом транзакций, поскольку его резервная копия не создается. При использовании простой модели восстановления в случае повреждения базы данных возникает риск потери значительной части результатов работы. Данные могут быть восстановлены только до момента последнего резервного копирования. Поэтому при использовании простой модели восстановления интервалы между резервным копированием должны быть достаточно короткими, чтобы предотвратить потерю значительного объема данных. В то же время они должны быть велики настолько, чтобы затраты на резервное копирование не влияли на производительность. Снизить затраты поможет использование разностного резервного копирования.

Обычно в базе данных используется модель полного восстановления или простая модель восстановления. Модели восстановления оказывают влияние на поведение журнала транзакций и способ регистрации операций, либо на 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. Однако, если вы предполагаете использовать зеркальное отображение базы данных, тогда у вас нет выбора, поскольку оно поддерживает только модель восстановления FULL.

Модель восстановления BULK_LOGGED обладает такой же семантикой усечения журнала транзакций, как и модель восстановления FULL, но допускает частичную регистрацию некоторых операций, что называется минимальной регистрацией. Примерами являются повторное создание индекса и некоторые операции массовой загрузки — в модели восстановления FULL регистрируется вся операция.

Но в модели восстановления BULK_LOGGED регистрируются только изменения распределения, что радикально сокращает число создаваемых записей журнала и, в свою очередь, сокращает потенциал 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.

SQL Server поддерживает восстановление данных на следующих уровнях:

- **База данных** (*полное восстановление базы данных*). Вся база данных возвращается в прежнее состояние и восстанавливается, при этом база данных находится в автономном режиме во время операций возврата и восстановления.
- **Файл данных** (*восстановление файла*). Файл данных или набор файлов данных возвращается в исходное состояние и восстанавливается. Во время восстановления файлов файловые группы, содержащие обрабатываемые файлы, автоматически переводятся в автономный режим на время восстановления. Любые попытки подключения и работы с недоступной файловой группой приведут к ошибке.
- **Страница данных** (*восстановление страницы*). При использовании модели полного восстановления или модели восстановления с неполным протоколированием можно восстановить отдельные базы данных. Восстановление страниц может применяться для любой базы данных вне зависимости от числа файловых групп.

Логическая архитектура журнала транзакций

На логическом уровне журнал транзакций состоит из последовательности записей. Каждая запись содержит:

- *Log Sequence Number*: регистрационный номер транзакции (*LSN*). Каждая новая запись добавляется в логический конец журнала с номером *LSN*, который больше номера *LSN* предыдущей записи.
- *Prev LSN*: обратный указатель, который предназначен для ускорения отката транзакции.
- *Transaction ID number*: идентификатор транзакции.
- *Type*: тип записи *Log*-файла.
- *Other information*: Прочая информация.

Двумя основными типами записи *Log*-файла являются:

- *Transaction Log Operation Code*: код выполненной логической операции, либо
- *Update Log Record*: исходный и результирующий образ измененных данных. Исходный образ записи — это копия данных до выполнения операции, а результирующий образ — копия данных после ее выполнения.

Действия, которые необходимо выполнить для восстановления операции, зависят от типа *Log*-записи:

- Зарегистрирована логическая операция.
 - Для наката логической операции выполняется эта операция.
 - Для отката логической операции выполняется логическая операция, обратная зарегистрированной.
- Зарегистрированы исходный и результирующий образы записи.
 - Для наката операции применяется результирующий образ.
 - Для отката операции применяется исходный образ.

В журнал транзакций записываются различные типы операций:

```
0 BEGINXACT: Begin Transaction
4 Insert
5 Delete
6 Indirect Insert
7 Index Insert
8 Index Delete
9 MODIFY: Modify the record on the page
11 Deferred Insert (NO-OP)
12 Deferred Delete (NO-OP)
13 Page Allocation
15 Extent Allocation
16 Page Split
17 Checkpoint
20 DEXTENT
30 End Transaction (Either a commit or rollback)
38 CHGSYSINDSTAT — A change to the statistics page in sysindexes
39 CHGSYSINDPG — Change to a page in the sysindexes table
```

Каждая транзакция резервирует в журнале транзакций место, чтобы при выполнении инструкции отката или возникновения ошибки в журнале было достаточно места для регистрации отката. Объем резервируемого пространства зависит от выполняемых в транзакции операций, но обычно он равен объему, необходимому для регистрации каждой из операций. Все это пространство после завершения транзакции освобождается.

Раздел журнального файла, который начинается от первой записи, необходимой для успешного отката на уровне базы данных, до последней зарегистрированной записи называется *активной частью журнала*, или *активным журналом*. Именно этот раздел необходим для выполнения полного восстановления базы данных. Активный журнал не может быть усечен.

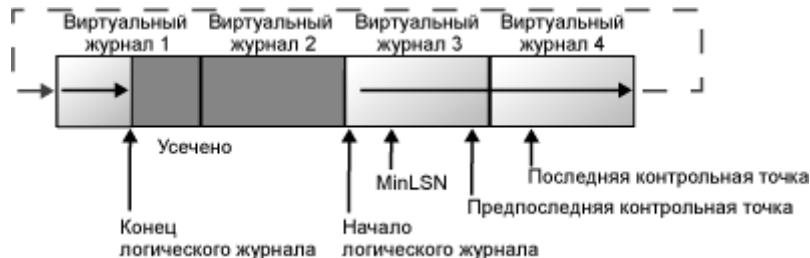
Физическая архитектура журнала транзакций

На физическом уровне журнал транзакций состоит из одного или нескольких физических файлов. Каждый физический файл журнала разбивается на несколько виртуальных файлов журнала (*VLF*). *VLF* не имеет фиксированного размера. Не существует также и определенного числа *VLF*, приходящихся на один физический файл журнала. Компонент Database Engine динамически определяет размер *VLF* при создании или расширении файлов журнала. Компонент Database Engine стремится обслуживать небольшое число *VLF*. Администраторы не могут настраивать или устанавливать размеры и число *VLF*.

Журнал транзакций является *оборачиваемым файлом*. Рассмотрим 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 регистрационным номером самой старой записи, которая необходима для успешного отката на уровне всей базы данных. Журнал транзакций рассматриваемой в данном примере базы данных будет выглядеть примерно так же, как на следующей иллюстрации.



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**, и новые записи журнала добавляются к этому расширению. Дополнительные сведения о настройке FILEGROWTH см. в разделе ALTER DATABASE (Transact-SQL).
- Если установка FILEGROWTH не применяется или диск, на котором размещается файл журнала, имеет меньше свободного места, чем это указано в **growth_increment**, формируется ошибка 9002.
- Если в журнале содержится несколько физических файлов журнала, логический журнал будет продвигаться по всем физическим файлам журнала до тех пор, пока он не вернется на начало первого физического файла журнала.

Просмотр журнала транзакций

Вариант А. Недокументированная команда DBCC LOG(*имя_базы_данных, тип_вывода*)

где *тип_вывода* принимает значение из диапазона 0..4. По умолчанию принят 0 (*Current LSN, Operation, Context, Transaction ID*). 3 выдает полный информационный дамп каждой операции.

Вариант В. Недокументированная функция fn_dblog().

Пример.

```
create table t1 (id int, name varchar(20));
insert into t1 values (1, 'test1');
checkpoint;
insert into t1 values (2, 'test2');
checkpoint;
select * from t1;
drop table t1;
-- DBCC LOG(N'dbtest', 3)
-- Выводится таблица 45*102
-- select top 1 * from fn_dblog(null, null)
-- Выводится таблица 1*97
```

Контрольные точки и активная часть журнала

Так как все изменения страниц данных происходят в страничных буферах, то изменения данных в памяти не обязательно отражаются в этих страницах на диске. Процесс кэширования происходит по алгоритму последней использованной страницы, поэтому страница, подверженная постоянным изменениям, помечается как последняя использованная, и она не записывается его на диск. Чтобы эти страницы были записаны на диск применяется контрольная точка. Все грязные страницы должны быть сохранены на диске в обязательном порядке.

Контрольная точка выполняет в базе данных следующее:

- Записывает в файл журнала запись, отмечающую начало контрольной точки.
- Сохраняет данные, записанные для контрольной точки в цепи записей журнала контрольной точки. Одним из элементов данных, регистрируемых в записях контрольной точки, является *номер LSN* первой записи журнала, при отсутствии которой успешный откат в масштабе всей базы данных невозможен. Такой *номер LSN* называется минимальным *номером LSN восстановления (MinLSN)*. Номер *MinLSN* является наименьшим значением из:
 - *номера LSN* начала контрольной точки;
 - *номера LSN* начала старейшей активной транзакции;
 - *номера LSN* начала старейшей транзакции репликации, которая еще не была доставлена базе данных распространителя.
 - Записи контрольной точки содержат также список активных транзакций, изменивших базу данных.
- Если база данных использует простую модель восстановления, помечает для повторного использования пространство, предшествующее номеру *MinLSN*.
- Записывает все измененные страницы журналов и данных на диск.
- Записывает в файл журнала запись, отмечающую конец контрольной точки.
- Записывает в страницу загрузки базы данных *номер LSN* начала соответствующей цепи.

Действия, приводящие к срабатыванию контрольных точек

Контрольные точки срабатывают в следующих ситуациях:

- При явном выполнении инструкции **CHECKPOINT**. Контрольная точка срабатывает в текущей базе данных соединения.
- При выполнении в базе данных операции с минимальной регистрацией, например при выполнении операции массового копирования для базы данных, на которую распространяется модель восстановления с неполным протоколированием.
- При добавлении или удалении файлов баз данных с использованием инструкции **ALTER DATABASE**.
- При остановке экземпляра SQL Server с помощью инструкции **SHUTDOWN** или при остановке службы SQL Server (**MSSQLSERVER**). И в том, и в другом случае будет создана контрольная точка каждой базы данных в экземпляре SQL Server.
- Если экземпляр SQL Server периодически создает в каждой базе данных автоматические контрольные точки для сокращения времени восстановления базы данных.
- При создании резервной копии базы данных.
- При выполнении действия, требующего отключения базы данных. Примерами могут служить присвоение параметру **AUTO_CLOSE** значения **ON** и закрытие последнего соединения пользователя с базой данных или изменение параметра базы данных, требующее перезапуска базы данных.

Автоматические контрольные точки

Компонент Database Engine создает контрольные точки автоматически. Интервал между автоматическими контрольными точками определяется на основе использованного места в журнале и времени, прошедшего с момента создания последней контрольной точки. Интервал между автоматическими контрольными точками колеблется в широких пределах и может быть довольно длительным, если база данных изменяется редко. При крупномасштабных изменениях данных частота автоматических контрольных точек может быть гораздо выше.

Можно использовать параметр конфигурации сервера **recovery interval** для вычисления интервала между автоматическими контрольными точками для всех баз данных на экземпляре сервера. Значение этого параметра определяет максимальное время, отводимое компоненту Database Engine на восстановление базы данных при перезапуске системы. Компонент Database Engine оценивает количество записей журнала, которые он может обработать за время **recovery interval** при выполнении операции восстановления.

Если используется простая модель восстановления базы данных, автоматическая контрольная точка создается каждый раз, когда число записей в журнале достигает меньшего из двух предельных условий:

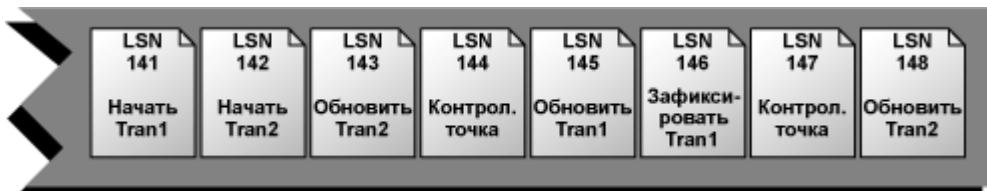
- журнал заполняется на 70 процентов;
- число записей в журнале достигает значения, определенного компонентом Database Engine в качестве количества записей, которое он может обработать за время, заданное параметром **recovery interval**.

Примечание. Сведения об установке интервала восстановления см. в разделе Как установить интервал восстановления (среда SQL Server Management Studio).

Активный журнал

Часть журнала, начинающаяся с номера MinLSN и заканчивающаяся последней записью, называется активной частью журнала, или активным журналом. Этот раздел журнала необходим для выполнения полного восстановления базы данных. Ни одна часть активного журнала не может быть усечена. Все записи журнала до номера MinLSN должны быть удалены из частей журнала.

На следующем рисунке изображена упрощенная схема журнала завершения транзакций, содержащего две активные транзакции. Записи контрольных точек были сжаты в одну запись.



Последней записью в журнале транзакций является запись с *номером LSN*, равным 148. На момент обработки записанной контрольной точки с *номером LSN 147* транзакция 1 уже зафиксирована и единственной активной транзакцией является транзакция 2. В результате первая запись журнала, созданная для транзакции 2, становится старейшей записью активной транзакции на момент последней контрольной точки. Таким образом, *номером MinLSN* становится *номер LSN*, равный 142 и соответствующий записи начала транзакции 2.

Длительные транзакции

Активный журнал должен включать в себя все элементы всех незафиксированных транзакций. Приложение, инициирующее транзакцию и не выполняющее ее фиксацию или откат, не позволяет компоненту Database Engine повышать *MinLSN*. Это может привести к проблемам двух типов.

- Если система будет выключена после того, как транзакцией было выполнено много незафиксированных изменений, этап восстановления при последующем перезапуске может занять гораздо больше времени, чем указано параметром *recovery interval*.
- Журнал может достичь очень большого объема, потому что после *номера MinLSN* усечь его нельзя. Это справедливо даже в том случае, если используется простая модель восстановления, когда журнал транзакций обычно усекается при каждой автоматической контрольной точке.

Журнал транзакций с упреждающей записью

SQL Server использует журнал с упреждающей записью, который гарантирует, что до занесения на диск записи, связанной с журналом, никакие изменения данных записаны не будут. Таким образом обеспечиваются свойства ACID для транзакции. Для понимания принципов работы журнала с упреждающей записью важно знать принципы записи измененных данных на диск. 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. Изменение не записывается на диск, пока в базе данных не возникает контрольная точка, или же изменение должно быть записано на диск таким образом, чтобы для хранения новой страницы мог использоваться буфер. 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.

Во время изменения страницы в буфере запись журнала строится в кэше журнала, который записывает изменение. Данная запись журнала должна быть перенесена на диск до того, как соответствующая «грязная» страница будет записана из буферного кэша на диск. Если «грязная» страница переносится на диск до записи журнала, эта страница создает изменение на диске, которое не может быть откачено, если сервер выйдет из строя до переноса записи журнала на диск. SQL Server обладает алгоритмом, который защищает «грязную» страницу от записи на диск до переноса на него соответствующей записи журнала. Содержимое журнала запишется на диск после того, как будут зафиксированы транзакции.

Управление журналом транзакций

Чтобы логический журнал не увеличивался до размера физических файлов журнала, следует периодически выполнять его усечение. Процесс усечения журнала уменьшает размер файла логического журнала, помечая виртуальные файлы журнала, которые не содержат частей логического журнала, как неактивные. В некоторых случаях может оказаться полезным физическое сжатие или расширение размера реального файла журнала.

Время усечения журнала зависит от модели восстановления базы данных. Есть три модели восстановления: простая модель восстановления, модель полного восстановления и модель восстановления с неполным протоколированием. Обычно в базе данных используется полная модель восстановления или простая модель восстановления. В качестве примера рассмотрим усечение журнала в простой модели восстановления.

При использовании простой модели восстановления усечение журналов выполняется автоматически. Если все записи в виртуальном файле журнала неактивны, то этот логический журнал усекается обычно после контрольной точки. При этом освобождается место для повторного использования. Это относится и к контрольным точкам инструкции **CHECKPOINT**, и к неявным контрольным точкам, сформированным системой. Однако усечение журнала может быть отложено, если виртуальные файлы журнала остаются активными вследствие выполнения долгой транзакции или резервного копирования.

Эта модель регистрирует только минимальные сведения, необходимые для обеспечения согласованности базы данных после сбоя системы или для восстановления данных из резервной копии. Это сводит к минимуму расход места на диске под журнал транзакций по сравнению с другими моделями восстановления. Чтобы предотвратить переполнение журнала, базе данных требуется достаточно места для записи в случае задержки его усечения.

Как работает усечение журнала

Кроме прочих данных, в контрольной точке записывается *номер LSN* первой записи журнала, которую необходимо сохранить для успешного отката на уровне базы данных. Этот *номер LSN* называется *минимальным номером LSN восстановления (MinLSN)*. Начало активной части журнала занято *VLF*, содержащим *MinLSN*. При усечении журнала транзакций освобождаются только те записи, которые находятся перед этим *VLF*.

На следующем рисунке показан журнал транзакций до усечения и после. На первом рисунке показан журнал транзакций, который никогда не усекался. В настоящий момент логический журнал состоит из четырех виртуальных файлов. Логический журнал начинается с начала первого файла виртуального журнала и заканчивается виртуальным файлом журнала 4. Запись *MinLSN* находится в виртуальном журнале 3. Виртуальные журналы 1 и 2 содержат только неактивные записи журнала. Эти записи можно усечь. Виртуальный журнал 5 пока не используется и не является частью текущего логического журнала.



На втором рисунке показан журнал после усечения. Виртуальные журналы 1 и 2 усечены и могут использоваться повторно. Логический журнал теперь начинается с начала виртуального журнала 3. Виртуальный журнал 5 все еще не используется и не является частью текущего логического журнала.



Управление размером файла журнала транзакций

Контролировать используемое пространство журнала можно с помощью процедуры DBCC SQLPERF (LOGSPACE). Она возвращает сведения об объеме пространства, используемого журналом в данный момент, и указывает, если необходимо провести усечение журнала транзакций. Для получения сведений о текущем размере файла журнала, его максимальном размере и параметре автоматического увеличения файла можно использовать столбцы **size**, **max_size** и **growth** для данного файла журнала в представлении **sys.database_files**.

Сжатие файла журнала

Усечение журнала освобождает место на диске для повторного использования, но не уменьшает размер физического файла журнала. Для уменьшения физического размера файла журнала должен быть сжат с целью удаления одного или более неактивных *VLF*. *VLF*, хранящий какие-либо активные записи журнала, удалить нельзя. При сжатии файла журнала транзакций в конце файла журнала удаляется достаточное количество неактивных виртуальных файлов журнала, чтобы журнал уменьшился до приблизительного целевого размера.

Примечание. Если журнал транзакций не усекался в последнее время, его сжатие может быть невозможным до тех пор, пока не выполнится усечение.

Система безопасности SQL Server

SQL Server обеспечивает защиту данных от неавторизированного доступа и от фальсификации. Основными функциями безопасности SQL Server являются:

- **проверка подлинности (аутентификация)** – процедура проверки соответствия некоего лица и его учетной записи в компьютерной системе. Один из способов аутентификации состоит в задании пользовательского идентификатора, в просторечии называемого «логином» (login – регистрационное имя пользователя) и пароля – некой конфиденциальной информации, знание которой обеспечивает владение определенным [ресурсом](#). Аутентификацию не следует путать с [идентификацией](#). Идентификация – это установление личности самого физического лица (а не его виртуальной учетной записи, коих может быть много).
- **авторизация** – это предоставление лицу прав на какие-то действия в системе.

Дополнительными функциями безопасности SQL Server являются:

- **шифрование,**
- **контекстное переключение,**
- **олицетворение,**
- **встроенные средства управления ключами.**

В основе системы безопасности SQL Server лежат три понятия:

- 1) **участники системы безопасности** (Principals);
- 2) **защищаемые объекты** (Securables) и
- 3) **система разрешений** (Permissions).

Участники системы безопасности

Участники системы безопасности или принципалы – это сущности, которые могут запрашивать ресурсы SQL Server. Принципалы могут быть иерархически упорядочены. Область влияния принципала зависит

- от области его определения: Windows, SQL Server, база данных,
- от того, коллективный это участник или индивидуальный. Имя входа Windows является примером индивидуального (неделимого) участника, а группа Windows — коллективного.

При создании учетной записи SQL Server ей назначается идентификатор и идентификатор безопасности. В представлении каталога sys.server_principals они отображаются в столбцах principal_id и SID.

Участники уровня Windows – это а) Имя входа домена Windows, б) Локальное имя входа Windows. Участники уровня SQL Server – это а) Имя входа SQL Server, б) Роль сервера.

Участники уровня базы данных – это а) Пользователь базы данных, б) Роль базы данных, с) Роль приложения.

Замечания.

1. **Имя входа «sa» SQL Server.** Имя входа sa SQL Server является участником уровня сервера. Оно создается по умолчанию при установке экземпляра. В SQL Server для имени входа sa базой данных по умолчанию будет **master**.
2. **Роль базы данных public.** Каждый пользователь базы данных является членом роли базы данных **public**. Если пользователю не были предоставлены или запрещены особые разрешения на защищаемый объект, то он наследует на него разрешения роли **public**.
3. **Пользователи INFORMATION_SCHEMA и sys.** Каждая база данных включает в себя две сущности, которые отображены в представлениях каталога в виде пользователей: **INFORMATION_SCHEMA** и **sys**. Они необходимы для работы SQL Server; эти пользователи не являются участниками и не могут быть изменены или удалены.

Защищаемые объекты

Защищаемые объекты – это ресурсы, доступ к которым регулируется системой авторизации. Некоторые защищаемые объекты могут храниться внутри других, создавая иерархию «областей», которые сами могут защищаться. К областям защищаемых объектов относятся (а) **сервер**, (б) **база данных** и (с) **схема**. Далее введем следующие ограничения:

- 1) для области «сервер» ограничимся рассмотрением вопросами управления учетными записями подключения (login),
- 2) для области «база данных» ограничимся рассмотрением вопросами управления

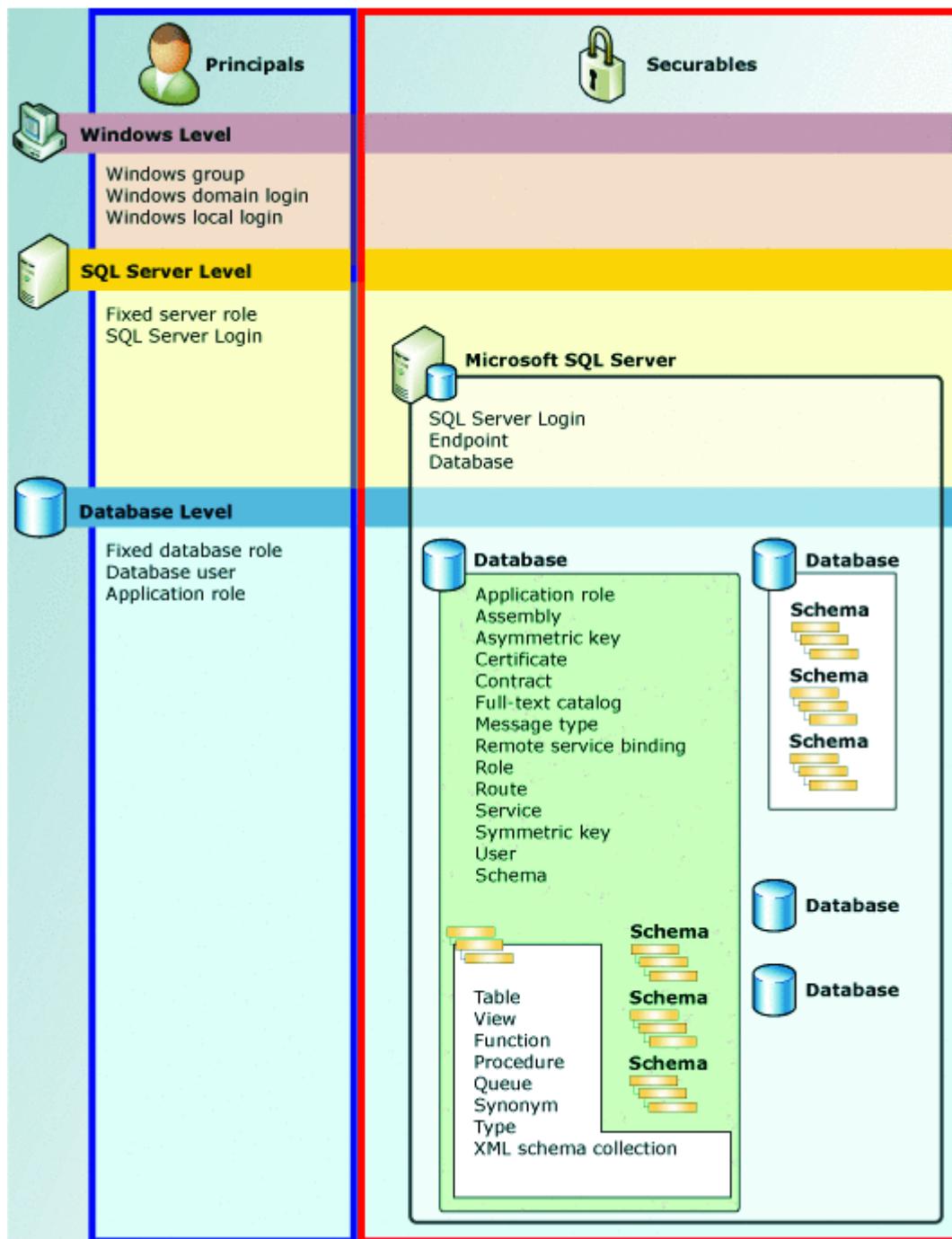
 - учетными записями пользователей (user),
 - ролями (role),
 - ролями приложений (application role).

- 3) для области «схема» ограничимся рассмотрением вопросами управления разрешениями на

 - функции (function),
 - процедуры (stored procedure),
 - таблицы (table),
 - представления (view).

Разрешения

У субъекта системы есть только один путь получения доступа к объектам - иметь назначенные непосредственно или опосредовано разрешения. При непосредственном управлении разрешениями они назначаются субъекту явно, а при опосредованном разрешения назначаются через членство в группах, ролях или наследуются от объектов, лежащих выше по цепочке иерархии. Управление разрешениями производится путем выполнения инструкций языка DCL (Data Control Language): GRANT (разрешить), DENY (запретить) и REVOKE (отменить).



Управление учетными данными сервера (credential)

Поддерживаются два вида учетных записей подключения к серверу

- учетные записи сервера, создаваемые на основании учетных записей операционной системы, и
- учетные записи сервера, создаваемые для прямого подключения к серверу.

Проверка подлинности Windows означает, что для подключения к SQL Server проверка подлинности полностью выполняется операционной системой Microsoft Windows. В этом случае клиент идентифицируется на основании учетной записи Windows. Проверка подлинности SQL Server означает, что для подключения к SQL Server проверка подлинности выполняется путем сравнения имени пользователя и пароля с хранящимся на сервере SQL Server списком действительных имен пользователей и паролей). Учетные данные сервера создаются при помощи инструкции **CREATE CREDENTIAL**. После создания учетных данных можно сопоставить их имени входа SQL Server, используя инструкцию **CREATE LOGIN** или **ALTER LOGIN**.

Управление именами входа SQL Server (login)

Сами имена входа не имеют доступа ни к одной конкретной базе данных на сервере, они позволяют только подключиться к SQL Server. Имена входа – это объекты, которым может быть дано разрешение в масштабе сервера на выполнение определенных действий. Эти действия собираются в фиксированные серверные роли: **bulkadmin**, **dbcreator**, **diskadmin**, **processadmin**, **public**, **securityadmin**, **serveradmin**, **setupadmin**, **sysadmin**. Все имена входа SQL Server являются обладателями роли **public**. Добавление имени входа в качестве члена предопределенной роли сервера выполняется при помощи хранимой процедуры **sp_addsrvrolemember**. Имя входа сервера может быть создано в SSMS, при помощи инструкции **CREATE LOGIN** или при помощи хранимой процедуры **sp_addlogin**.

Замечание. В SQL Server 2012 при помощи инструкции **CREATE SERVER ROLE** можно создать новую, определяемую пользователем роль сервера, а для изменения членства в роли сервера можно использовать инструкцию **ALTER SERVER ROLE**.

Примеры.

```
--Creating a new SQL login
CREATE LOGIN Carol
WITH PASSWORD = 'Th1sI$|\\|yP@ssw0rd';
GO

--Creating a credential based on a Windows user
CREATE CREDENTIAL StreetCred
WITH IDENTITY = 'AughtFive\CarolStreet',
SECRET = 'P@ssw0rd';

--Associating a login with a credential
ALTER LOGIN Carol WITH CREDENTIAL = StreetCred;
GO

--Creating a login, and adding the user to a fixed server role
CREATE LOGIN Ted WITH PASSWORD = 'P@ssw0rd';
GO

EXEC sp_addsrvrolemember 'Ted', 'securityadmin';
GO
```

Управление учетными записями пользователей базы данных (user)

После создания имени входа в SQL Server можно предоставить этому имени доступ к конкретной базе данных. Для этого сначала надо создать пользователя базы данных для определенного ранее имени входа. Это можно сделать в SSMS, при помощи инструкции **CREATE USER** или при помощи хранимой процедуры **sp_adduser**. При входе в SQL Server под именем входа база данных запросит имя и идентификатор созданного пользователя базы данных. Если при создании пользователя базы данных не будет указано имя входа SQL Server, то новый пользователь базы данных будет сопоставлен с именем входа SQL Server, имеющим такое же имя. После создания пользователя базы данных можно (но это необязательно) предоставить ему одну из фиксированных ролей базы данных: **db_accessadmin**, **db_backupoperator**, **db_datareader**, **db_datawriter**, **db_ddladmin**, **db_denydatareader**, **db_denydatawriter**, **db_owner**, **db_securityadmin**. При желании можно определить дополнительные роли базы данных при помощи инструкции **CREATE ROLE**. Для добавления пользователя базы данных к роли текущей базы данных используется инструкция **ALTER ROLE** или хранимая процедура **sp_addrolemember**.

Примеры

```
-- Managing Server RoleMembers
CREATE LOGIN Veronica
WITH PASSWORD = 'PalmTree1'
GO

EXEC master..sp_addsrvrolemember 'Veronica', 'sysadmin'
GO

USE TestDB
```

```
GO
```

```
CREATE USER Veronica
GO
-- Т. к. предложение FOR LOGIN не указано, то новый пользователь базы данных будет ----
сопоставлен с именем входа SQL Server, имеющим такое же имя.

CREATE ROLE HR_ReportSpecialist AUTHORIZATION db_owner
GO

EXEC sp_addrolemember 'HR_ReportSpecialist', 'Veronica'
GO
```

Замечание. В SQL Server 2012 многие системные хранимые процедуры, используемые для управления безопасностью, считаются устаревшими, но остаются доступными для обратной совместимости. Для устаревших процедур рекомендуется использовать инструкции Transact-SQL, например, ALTER LOGIN, ALTER SERVER ROLE, ALTER USER, ALTER ROLE и др.

Управление разрешениями

После добавления пользователя следует определить разрешения, управляющие действиями, которые пользователь может выполнять, с помощью инструкций **GRANT**, **DENY** и **REVOKE**. Необходимо, чтобы пользователь был владельцем базы данных.

A. Предоставление разрешений на объекты (инструкция GRANT)

Инструкция GRANT предоставляет разрешения на таблицу, представление, функцию, хранимую процедуру, очередь обслуживания, синоним. Синтаксис инструкции GRANT:

```
GRANT { ALL [ PRIVILEGES ] | список_разрешений }
ON список_объектов
TO список_принципалов
[ WITH GRANT OPTION ]
[ AS принципал ]
```

Пояснения к инструкции GRANT

1. Ключевое слово ALL с необязательным словом RIVILEGES не включает все возможные разрешения, оно эквивалентно предоставлению всех разрешений ANSI-92, применимых к указанному объекту. Значение ALL различается для разных типов объектов. Ниже перечислены главные классы разрешений и защищаемых объектов, к которым эти разрешения могут применяться:
 - a) разрешения на скалярные функции: EXECUTE, REFERENCES;
 - b) разрешения на функции, возвращающие табличное значение: DELETE, INSERT, REFERENCES, SELECT, UPDATE;
 - c) разрешения на хранимые процедуры: EXECUTE;
 - d) разрешения на таблицы: DELETE, INSERT, REFERENCES, SELECT, UPDATE;
 - e) разрешения на представления: DELETE, INSERT, REFERENCES, SELECT, UPDATE.Полный список разрешений содержит 195 пунктов.
2. Если разрешение предоставляется на таблицу, представление или функцию, возвращающую табличное значение, то справа от разрешения в круглых скобках могут указываться имена столбцов. На столбец могут быть предоставлены только разрешения SELECT, REFERENCES и UPDATE.
3. Объект, на который предоставляется разрешение, имеет следующее описание: [ОБЪЕКТ ::] [имя_схемы].*имя_объекта*. Фраза ОБЪЕКТ необязательна, если указан аргумент имя_схемы. Если же она указана, указание квалификатора области (::) обязательно. Если не указан аргумент имя_схемы, подразумевается схема по умолчанию. Если указан аргумент имя_схемы, обязательно указание квалификатора области схемы (.).
4. Принципалом может быть:
 - a) пользователь базы данных,
 - b) роль базы данных,
 - c) роль приложения.
5. Необязательная фраза WITH GRANT OPTION указывает, что принципалу такжедается возможность предоставлять указанное разрешение другим принципалам.
6. Необязательная фраза AS *принципал* определяет принципала, у которого другой принципал, выполняющий данный запрос, наследует право предоставлять данное разрешение.

Пример. Предоставление разрешения EXECUTE на хранимую процедуру HumanResources.uspUpdateEmployeeHireInfo роли приложения Role03.

```
USE AdventureWorks;
GRANT EXECUTE ON OBJECT::HumanResources.uspUpdateEmployeeHireInfo TO Role03;
GO
```

Пример. Предоставление разрешения REFERENCES на столбец EmployeeID в представлении HumanResources.vEmployee пользователю User02 с параметром GRANT OPTION.

```
USE AdventureWorks;
GRANT REFERENCES (EmployeeID) ON OBJECT::HumanResources.vEmployee TO User02
WITH GRANT OPTION;
GO
```

B. Отмена разрешений на объекты (инструкция REVOKE)

Инструкция REVOKE отменяет разрешения, ранее предоставленные инструкцией GRANT. Синтаксис инструкции REVOKE:

```
REVOKE [ GRANT OPTION FOR ] список_разрешений
ON список_объектов
{ FROM | TO } список_принципалов
[ CASCADE ]
[ AS принципал ]
```

Пояснения к инструкции REVOKE.

1. Необязательная фраза GRANT OPTION FOR показывает, что право на предоставление заданного разрешения другим принципалам будет отменено. Само разрешение отменено не будет.
2. Необязательное ключевое слово CASCADE показывает, что отменяемое разрешение также отменяется для других принципалов, для которых оно было предоставлено или запрещено этим принципалом. Каскадная отмена разрешения, предоставленного с помощью параметра WITH GRANT OPTION, приведет к отмене прав GRANT и DENY для этого разрешения.
3. Необязательная фраза AS принципал указывает принципала, от которого принципал, выполняющий данный запрос, получает право на отмену разрешения.

Пример. Отмена разрешения EXECUTE для хранимой процедуры

```
USE AdventureWorks;
REVOKE EXECUTE ON OBJECT::HumanResources.uspUpdateEmployeeHireInfo
FROM Role03;
GO
```

C. Запрет разрешений на объекты (инструкция DENY)

Инструкция DENY запрещает разрешения на члены класса OBJECT защищаемых объектов. Синтаксис инструкции DENY:

```
DENY список_разрешений
ON список_объектов
TO список_принципалов
[ CASCADE ]
[ AS принципал ]
```

Пример. Запрет разрешения REFERENCES на представление с CASCADE

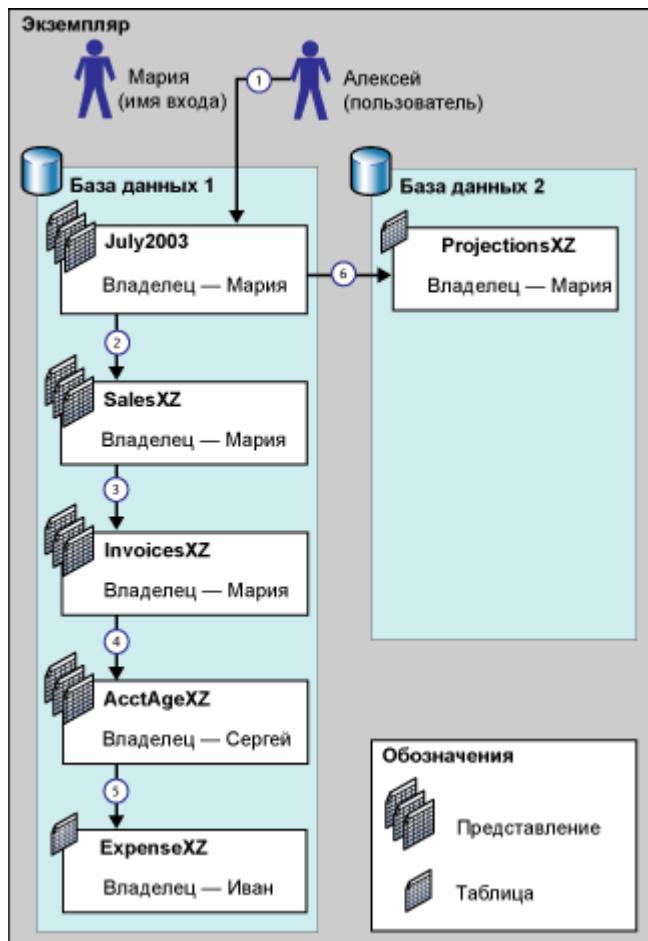
```
USE AdventureWorks;
DENY REFERENCES (EmployeeID) ON OBJECT::HumanResources.vEmployee TO User02 CASCADE;
GO
```

Цепочки владения

Если несколько объектов базы данных последовательно обращаются друг к другу, то такая последовательность известна как *цепочка*. Такие цепочки не могут существовать независимо, но когда SQL Server проходит по звеньям цепи, то SQL Server проверяет разрешения составляющих объектов иначе, нежели при раздельном доступе к объектам. Эти различия имеют важные последствия для обеспечения безопасности. Цепочки владения позволяют управлять доступом к нескольким объектам, таким как таблицы, назначая разрешения одному объекту, например представлению. Цепочки владения также обеспечивают небольшое повышение производительности в случаях, когда позволено пропускать проверку наличия разрешений. Проверка разрешений в цепи выполняется так:

- 1) SQL Server сначала сравнивает владельца вызываемого объекта с владельцем вызывающего объекта.
- 2) Если оба объекта имеют одного владельца, то разрешения для ссылаемого объекта не проверяются.

Пример цепочки владения



1. Алексей выполняет инструкцию `SELECT *` в представлении **July2003**. SQL Server проверяет разрешения в представлении и подтверждает, что Алексей имеет разрешение выбирать.
2. Представление **July2003** требует данные из представления **SalesXZ**. SQL Server проверяет владение **SalesXZ**. Владелец этого представления (**Мария**) такой же, как у вызывающего представления, поэтому разрешения для **SalesXZ** не проверяются. Возвращаются необходимые данные.
3. Представление **SalesXZ** требует данные из представления **InvoicesXZ**. SQL Server проверяет владение представления **SalesXZ**. Владелец этого представления такой же, как у предшествующего объекта, поэтому разрешения для **InvoicesXZ** не проверяются. Возвращаются необходимые данные. До этого этапа все элементы последовательности имели одного владельца (**Мария**). Это известно как *неразрывная цепочка владения*.
4. Представление **InvoicesXZ** требует данные из представления **AcctAgeXZ**. SQL Server проверяет владение представления **AcctAgeXZ**. Владелец этого представления иной, чем у предшествующего объекта (**Сергей**, а не **Мария**), поэтому должны быть получены полные сведения о разрешениях на это представление. Если на представление **AcctAgeXZ** имеются разрешения, которые обеспечивают доступ со стороны пользователя **Алексей**, сведения будут возвращены.
5. Представление **AcctAgeXZ** требует данные из представления **ExpenseXZ**. SQL Server проверяет владение таблицы **ExpenseXZ**. Владелец этой таблицы иной, чем у предшествующего объекта (**Иван**, а не **Сергей**), поэтому должны быть получены полные сведения о разрешениях на эту таблицу. Если таблица **ExpenseXZ** имеет разрешения, которые обеспечивают доступ со стороны пользователя **Алексей**, сведения возвращаются.

- Если представление **July2003** пытается получить данные из таблицы **ProjectionsXZ**, то сервер сначала проверяет наличие цепочечных связей между базами данных **Database 1** и **Database 2**. Если цепочечные связи между базами данных активны, то сервер проверяет владение для таблицы **ProjectionsXZ**. Владелец этой таблицы такой же, как у вызывающего представления (**Мэрия**), поэтому разрешения для этой таблицы не проверяются. Возвращаются необходимые данные.

По умолчанию межбазовые цепочки владения отключены. SQL Server можно настроить так, чтобы разрешить цепочку владения между конкретными базами данных или между всеми базами данных внутри одного экземпляра SQL Server.

Пример потенциальных опасностей при использовании цепочек владения

Включены межбазовые цепочки владения между базой данных **A** и базой данных **B**. В этом случае член предопределенной роли базы данных **db_owner** любой из этих баз данных может незаконно проникнуть в другую базу данных. Процедура выглядит так:

- Член предопределенной роли базы данных **db_owner** в базе данных **A** по имени **X** создает пользователя **Y** в базе данных **A**, который уже существует как пользователь в базе данных **B**.
- Затем **X** создает в базе данных **A** объект, владельцем которого является **Y**.
- Пользователь **Y** из базы данных **A** вызывает любой объект, принадлежащий пользователю **Y** в базе данных **B**.
- Вызывающий и вызываемый объекты имеют общего владельца, поэтому разрешения для объекта в базе данных **B** не будут проверяться, когда **X** обратится к ним через созданный ею объект.

Контекст выполнения и переключение контекста

Контекст выполнения определяется подключенным к сеансу именем входа, пользователем или выполняющимся модулем. Он устанавливает идентификатор пользователя или имени входа, чьи разрешения на выполнение инструкций или совершение действий проверяются. В SQL Server контекст выполнения может переключаться на другое имя входа или пользователя при помощи выполнения инструкции EXECUTE AS или предложения EXECUTE AS в модуле. После переключения контекста SQL Server проверяет разрешения у имени входа и пользователя этой учетной записи, а не у того, кто вызвал инструкцию EXECUTE AS, или модуля. Имя входа или пользователь олицетворяется на оставшееся время выполнения сеанса или модуля либо до того момента, когда происходит явное восстановление переключения контекста (сеанс удаляется, контекст переключен на другое имя входа или на другого пользователя с помощью новой инструкции EXECUTE AS, контекст восстановлен до контекста предыдущего выполнения с помощью инструкции REVERT).

Синтаксис инструкции EXECUTE AS имеет вид:

```
EXECUTE AS {LOGIN | USER} = 'имя' [WITH {NO REVERT | COOKIE INTO @varbinary_переменная}]  
| CALLER
```

- LOGIN** указывает, что область олицетворения — это уровень сервера.
- USER** указывает, область олицетворения ограничена текущей базой данных.
- '**имя**' - допустимое имя входа или имя пользователя.
- NO REVERT** указывает, что переключение контекста нельзя вернуть к предыдущему контексту.
- COOKIE INTO @ varbinary_переменная** указывает, что контекст выполнения можно переключить к предыдущему контексту, если при вызове инструкция REVERT WITH COOKIE содержит правильное значение @ varbinary_переменная. SQL Server передает куки-файл в @ varbinary_переменная.
- CALLER** указывает, что инструкции модуля выполняются в контексте вызывающей стороны. Вне модуля инструкция не действует.

Синтаксис предложения EXECUTE AS имеет вид:

```
EXECUTE AS { CALLER | SELF | OWNER | 'имя_пользователя' }
```

- EXECUTE AS CALLER** указывает, что инструкции, содержащиеся в модуле, выполняются в контексте пользователя, вызывающего этот модуль.
- EXECUTE AS SELF** указывает, что модуль выполняется от имени того пользователя, который последним модифицировал модуль.
- EXECUTE AS OWNER** указывает, что инструкции, содержащиеся в модуле, выполняются в контексте текущего владельца этого модуля.
- EXECUTE AS 'имя_пользователя'** указывает, что инструкции, содержащиеся в модуле, выполняются в контексте пользователя, указанного аргументом 'имя_пользователя'.

Пример контекста выполнения

```

USE master
GO
-- Создаем имена входов
CREATE LOGIN User1 WITH PASSWORD='^*ahfn2@^(K'
GO
CREATE LOGIN User2 WITH PASSWORD='*HABA7s7aas'
GO
CREATE LOGIN User3 WITH PASSWORD='zxd837&^gqF'
GO
CREATE DATABASE MyDB
GO
USE MyDB
GO
-- Создаем пользователей и схемы
CREATE USER User3 WITH DEFAULT_SCHEMA=User3
GO
CREATE SCHEMA User3 AUTHORIZATION User3
GO
CREATE USER User2 WITH DEFAULT_SCHEMA=User2
GO
CREATE SCHEMA User2 AUTHORIZATION User2
GO
CREATE USER User1 WITH DEFAULT_SCHEMA=User1
GO
CREATE SCHEMA User1 AUTHORIZATION User1
GO
-- Пользователь User3 имеет право создавать таблицу
GRANT CREATE TABLE TO User3
GO
-- Пользователь User2 имеет право создавать процедуру
GRANT CREATE PROC TO User2
GO
EXECUTE AS LOGIN='User3'
GO
CREATE TABLE User3.CustomerInformation
(
    CustomerName nvarchar(50)
)
GO
INSERT INTO CustomerInformation VALUES('Bryan''s Bowling Alley')
INSERT INTO CustomerInformation VALUES('Tammie''s Tavern')
INSERT INTO CustomerInformation VALUES('Frank''s Fresh Produce')
GO
GRANT SELECT ON CustomerInformation TO User2
GO
REVERT
GO
EXECUTE AS LOGIN='User2'
GO
--create a stored proc that will return the rows in our table
CREATE PROC ViewCustomerNames
AS
BEGIN
    SELECT * FROM User3.CustomerInformation
END
GO
GRANT EXECUTE ON ViewCustomerNames TO User1
GO
REVERT
GO
EXECUTE AS LOGIN='User1'
-- User1 cannot access table directly
SELECT * FROM User3.CustomerInformation

```

```

Msg 229, Level 14, State 5, Line 3
The SELECT permission was denied on the object 'CustomerInformation', database 'myDB',
schema 'User3'.

--User1 can execute the procedure but does not have permissions on the underlying table
EXEC User2.ViewCustomerNames

Msg 229, Level 14, State 5, Procedure ViewCustomerNames, Line 5
The SELECT permission was denied on the object 'CustomerInformation', database 'myDB',
schema 'User3'.


GO
REVERT
GO
EXECUTE AS LOGIN='User2'
GO
ALTER PROCEDURE ViewCustomerNames WITH EXECUTE AS OWNER
AS
BEGIN
    SELECT * FROM User3.CustomerInformation
END
GO
REVERT
GO
EXECUTE AS LOGIN='User1'
--User1 still cannot access table directly
SELECT * from User3.CustomerInformation

Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'CustomerInformation', database 'myDB',
schema 'User3'

--User1 can execute a procedure that uses the CustomerInformation table
EXEC User2.ViewCustomerNames
GO
REVERT
GO

```

Аудит SQL Server

Начиная с версии SQL Server 2008 в редакции Enterprise вводится аудит SQL Server Audit – функциональность системы безопасности, которая может отслеживать практически любое действие с сервером или базой данных (выполняемое пользователями) и записывать эти действия в файловую систему или системный журнал Windows.

Система управления на основе политик

Одна из новых возможностей, появившаяся в SQL Server 2008, – система управления на основе политик (Policy-Based Management), которая позволяет создавать политики для обеспечения соответствия нормативам управления базой данных.

Система управления на основе политик позволяет администратору баз данных (АБД) создавать политики для управления объектами и экземплярами базы данных. Эти политики дают АБД возможность устанавливать правила создания и изменения объектов и их свойств. С помощью новой системы можно, например, создать политику уровня БД, запрещающую использование для БД свойства AutoShrink. Другой пример – политика, в соответствии с которой все имена табличных триггеров в таблице БД начинаются с tr_.

Система управления на основе политик предусматривает использование новых терминов и понятий. Основными из них являются:

- 1) **Политика (Policy)** – набор условий, определенных аспектами цели управления. Другими словами, политика — это набор правил для свойств БД или серверных объектов.
- 2) **Цель управления (Target)** – объект, управляемый данной системой. Сюда относятся такие объекты, как экземпляр БД, база данных, таблица, хранимая процедура, триггер, индекс.
- 3) **Аспект (Facet)** – свойство объекта (цели управления), которое используется системой управления на основе политик. Например, имя триггера или свойство базы данных AutoShrink.

- 4) **Условие (Condition)** – критерий для аспектов цели управления. Например, можно создать для факта условие, по которому все имена хранимых процедур в схеме «Banking» должны начинаться с bnk_.

Кроме того, политику можно связать с определенной категорией, что позволяет осуществлять управление набором политик, привязанных к той же самой категории. Политика может принадлежать только к одной категории.

Режим оценки политик

Существует несколько режимов оценки политик:

- 1) **По запросу (On demand)** – оценку политики запускает непосредственно администратор.
- 2) **При изменении: запретить (On change: prevent)** – для предотвращения нарушения политики используются триггеры DDL.
- 3) **При изменении: только внесение в журнал (On change: log only)** – для проверки политики при изменении используются уведомления о событии.
- 4) **По расписанию (On schedule)** – для проверки политики на нарушения используется задание агента SQL (SQL Agent).

Преимущества системы управления на основе политик

Система управления на основе политик позволяет АБД в полной мере контролировать процессы, происходящие в базе данных. Администратор получает возможность реализовать принятые в компании политики на уровне БД. Политики, принятые только на бумаге, помогают определить основные принципы управления базой данных и могут служить прекрасным руководством к действию, но воплощать их в жизнь очень нелегко. Для обеспечения строгого соответствия БД принятым нормативом АБД приходится пристально следить за ее повседневным использованием и функционированием. Система управления на основе политик позволяет раз и навсегда выработать политики управления и быть уверенными в том, что они будут применяться постоянно и в полном объеме.

Дополнительные функции безопасности SQL Server

Шифрование

SQL Server поддерживает три механизма шифрования:

1. на основе сертификатов (стандарт X. 509v3),
2. на основе асимметричных ключей (алгоритм RSA),
3. на основе симметричных ключей (алгоритмы шифрования RC4, RC2, DES, AES).

Для работы с этими сущностями используются следующие операторы T-SQL:

CREATE/ALTER/DROP/BACKUP CERTIFICATE
CREATE/ALTER/DROP/OPEN/CLOSE ASYMMETRIC KEY
CREATE/ALTER/DROP/OPEN/CLOSE SYMMETRIC KEY

Детальное описание иерархической схемы шифрования данных в SQL Server 2005 см. в разделе «Encryption Hierarchy» MSDN.

Контекстное переключение

SQL Server позволяет задать контекст выполнения хранимых процедур и пользовательских функций с помощью выражения EXECUTE AS, помещенного в заголовок определения модуля. Данный механизм позволяет одному пользователю выполнять действия внутри модуля так, будто он аутентифицирован как другой пользователь. Параметр EXECUTE AS имеет четыре возможных значения:

- CALLER. Если программист указывает EXECUTE AS CALLER, операторы внутри модуля выполняются в контексте пользователя, вызвавшего процедуру. Поэтому пользователь, выполняющий процедуру, должен иметь соответствующие разрешения не только на запуск процедуры, но и на любые объекты базы данных, на которые она ссылается.
- USER. Если используется значение EXECUTE AS USER *имя_пользователя*, процедура выполняется в контексте того пользователя, чье имя указано в параметре. При выполнении процедуры SQL Server сначала проверяет, имеет ли пользователь разрешение EXECUTE на данную процедуру, затем проверяет разрешения на операторы внутри процедуры для пользователя, указанного в параметре EXECUTE AS USER. Для того чтобы иметь возможность указать AS *имя_пользователя*, необходимо иметь специальные разрешения (например, IMPERSONATE) или быть членом специальной роли (sysadmin или db_owner).

- SELF. EXECUTE AS SELF, аналогично EXECUTE AS USER *имя_пользователя*, где *имя_пользователя* является именем человека, создающего или изменяющего модуль. Система сохраняет идентификатор пользователя (UID) вместо самого значения SELF. Пользователь, указанный в параметре SELF, не обязательно должен быть владельцем объекта. На самом деле объекты в SQL Server 2005 не имеют владельцев, но можно думать о владельцах схем так, как будто они владеют всеми объектами в схемах.
- OWNER. Применение EXECUTE AS OWNER хорошо подходит в ситуациях, если пользователь создает хранимые процедуры и таблицы в ходе своей работы, тогда, переходя на другое место, он сможет передать их в собственность другому пользователю, будучи уверенными, что они всегда будут выполняться в контексте безопасности собственника.

Олицетворение

Можно настроить SQL Server и Windows таким образом, чтобы экземпляр SQL Server мог подключаться к другому экземпляру SQL Server в контексте пользователя Windows, прошедшего проверку подлинности. Это называется олицетворением или делегированием. При использовании делегирования экземпляр SQL Server, к которому подключается пользователь Windows с проверкой подлинности Windows, выполняет олицетворение этого пользователя при обмене данными с другим экземпляром SQL Server или поставщиком SQL Server. Этот второй экземпляр или поставщик могут располагаться на том же компьютере или на удаленном компьютере в том же домене Windows, что и первый экземпляр.

Встроенная поддержка прозрачного шифрования

Механизм прозрачного шифрования основан на использовании мастер-ключа, сертификата, защищенного мастер-ключом, и ключа для шифрования базы данных. Для работы с мастер-ключом используются следующие операторы T-SQL:

CREATE/ALTER/DROP/OPEN/CLOSE MASTER KEY

Прозрачное шифрование позволяет создавать индексы и выполнять поиск по зашифрованным данным без использования каких-либо дополнительных функций.

Постреляционные базы данных

В классической теории БД, модель данных - это формальная теория представления и обработки данных в СУБД, которая включает, по меньшей мере, три аспекта:

- 1) аспект структуры: методы описания типов и логических структур данных в БД;
- 2) аспект манипуляции: методы манипулирования данными;
- 3) аспект целостности: методы описания и поддержки целостности БД.

Аспект структуры определяет, что из себя логически представляет БД, аспект манипуляции определяет способы перехода между состояниями БД (то есть способы модификации данных) и способы извлечения данных из БД, аспект целостности определяет средства писаний корректных состояний БД.

Существует большое количество разновидностей БД, отличающихся по различным критериям. По признаку «модель данных» различают дюреационные, реляционные и постреляционные БД и соответствующие им СУБД. Постреляционные БД часто ассоциируют с БД NoSQL.

Англоязычная Wikipedia в статье «Database» не дает точного определения термина «постреляционная» (post-relational). Вместо этого заявляется следующее: «Жесткость реляционной модели, в которой все данные хранятся в таблицах с фиксированной структурой строк и столбцов, все чаще рассматривается как ограничение при работе с информацией, более богатой и более разнообразной по структуре, чем традиционные «книги книг» данных корпоративных информационных систем.» Проблема ограниченности реляционных баз данных решается двумя подходами:

- 1) Разрабатываются новые типы баз данных, которые позиционируются как постреляционные или NoSQL базы данных.
- 2) Поставщики реляционных баз данных расширяют возможности своих продуктов, например путем поддержки более широкого круга типов данных.

Wikipedia в статье «NoSQL» также не дает точного определения терминов постреляционные и NoSQL базы данных. Вместо этого заявляется следующее: «NoSQL - это термин, обозначающий ряд подходов, проектов, направленных на реализацию моделей баз данных, имеющих существенные отличия от используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL.» Сторонниками концепции NoSQL подчёркивается, что она не является полным отрицанием языка SQL и реляционной модели, проект исходит из того, что SQL - это важный и весьма полезный инструмент, но при этом он не может считаться универсальным.

В учебнике «Базы данных: языки и модели» автор учебника С.Д. Кузнецов пишет: «При всех достоинствах РМД, частично унаследованных в языке SQL, традиционный подход часто подвергался справедливой критике на основании того, что он породил кардинальные различия между прикладной программой и данными, хранимыми в базе данных (для обозначения этой ситуации обычно используется термин impedance mismatch - несоответствие).» Для решения этой проблемы было предложено три подхода, отраженных в трех документах, получивших название манифестов.

1. Манифест систем объектно-ориентированных баз данных (1989). Подход был развит консорциумом ODMG, последовательно опубликовавшим три версии спецификации объектной модели данных. В 2001 году ODMG завершил свою работу и был расформирован. Последний стандарт ODMG – «The Object Data Standard ODMG 3.0».
2. Манифест систем баз данных следующего поколения (1990). Необходимо сохранить все свойства SQL-ориентированных систем, но усовершенствовать их систему типов. Подход был поддержан в стандарте SQL:1999 и в последующих редакциях стандарта SQL (SQL:2003, SQL:2006, SQL:2008). Примеры реализации объектно-реляционных СУБД: Informix, Oracle, DB2.
3. Третий манифест (1995). Вернуться к первоначальной РМ, переосмыслив ее таким образом, чтобы допустить хранение в БД данных, типы которых определяются пользователем. Пример реализации: Dataphor компании Alphora.

NoSQL базы данных

Термин NoSQL (Not Only SQL - Не только SQL) введен в употребление в 2009 году для описания различных технологий баз данных, возникших для удовлетворения требований, известных как «Web-scale» или «Internet-scale». Описание схемы данных в случае использования NoSQL-решений может осуществляться через использование различных структур данных: хеш-таблиц, деревьев и т.д.

К Web-scale приложениям предъявляются следующие три требования:

- 1) много данных: (Facebook - 50 терабайт для поиска по входящим сообщениям; eBay - 2 петабайта);
- 2) «огромное» количество пользователей: исчисляются миллионами, доступ к системам выполняется одновременно и постоянно;
- 3) сложные данные: как правило, это приложения не простой обработки табличных данных, которые можно найти во многих коммерческих и бизнес-приложениях.

Технологии реляционных баз данных показывают свои слабые места при переходе к веб-масштабам именно в этих трех аспектах. Апологеты NoSQL поясняют, что проект NoSQL не занимается полным отрицанием языка SQL и реляционной модели, проект исходит из того, что SQL — это важный и весьма полезный инструмент, но при этом он не может считаться универсальным. Одной из проблем, которую указывают для классических реляционных БД, являются проблемы при работе с данными очень большого объема и в проектах с высокой нагрузкой. Основная цель проекта — расширить возможности БД там, где SQL недостаточно гибок, и не вытеснять его там, где он справляется со своими задачами.

В качестве одного из методологических обоснований подхода NoSQL используется эвристический принцип, известный как теорема CAP, утверждающий, что в распределённой системе невозможно одновременно обеспечить согласованность данных, доступность (англ. availability, в смысле наличия отклика по любому запросу) и устойчивость к расщеплению распределённой системы на изолированные части. Таким образом, при необходимости достижения высокой доступности и устойчивости к разделению предполагается не фокусироваться на средствах обеспечения согласованности данных, обеспечиваемых традиционными SQL-ориентированными СУБД с транзакционными механизмами на принципах ACID.

В июле 2011 компания Couchbase, разработчик CouchDB, Memcached и Membase, анонсировала создание нового SQL-подобного языка запросов – UnQL (Unstructured Data Query Language). Работы по созданию нового языка выполнили создатель SQLite Ричард Гипп (англ. Richard Hipp) и основатель проекта CouchDB Дэмиен Кац (англ. Damien Katz). Разработка передана сообществу на правах общественного достояния.

Разнообразие NoSQL баз данных велико и постоянно растет, но все они имеют общие черты:

- 1) обрабатываются огромные объемы данных, разделяемые между серверами;
- 2) поддерживается огромное количество пользователей;
- 3) используется упрощенная, более гибкая, не ограниченная схемой структура базы данных;

В основе NoSQL лежат следующие идеи:

- 1) нереляционность,
- 2) распределенность,
- 3) открытый исходный код,
- 4) горизонтальная масштабируемость.

Пояснение. Горизонтальная масштабируемость — это увеличение количества серверов для равномерного распределения клиентских запросов между ними и гарантии сохранения работоспособности всей системы в случае выхода из строя одного или даже нескольких узлов (технические характеристики отдельных серверов могут при этом быть намного ниже, чем при «вертикальном» масштабировании).

Хорошее общее введение в архитектуру NoSQL можно найти в статье Р. Твида и Дж. Джеймса «Универсальное NoSQL - введение в теорию»: <http://blogerator.ru/page/nosql-vvedenie-v-teoriju-bd>. На сайте <http://nosql-database.org/> приводится список из 122 NoSQL СУБД, которые могут быть условно разделены на следующие категории:

- Поколоночные СУБД.
- Документо-ориентированные СУБД.
- Хранилища «ключ-значение», кортежные хранилища.
- Базы данных на основе графов.
- Другие модели данных.

Ярким представителем документо-ориентированных СУБД является MongoDB, краткое знакомство с которой можно выполнить с помощью статьи «Куда идет NoSQL с MongoDB» Тэда Ньюарда, которую можно найти по адресу <https://msdn.microsoft.com/ru-ru/magazine/ee310029.aspx>. **Замечание.** У этой статьи есть продолжения.

Не реляционные элементы в реляционной СУБД SQL Server

Ранее отмечалось, что многие поставщики реляционных баз данных расширяют возможности своих продуктов, например, путем поддержки более широкого круга типов данных. По такому пути пошла компания Microsoft.

xml

Тип данных, в котором хранятся XML-данные. Можно хранить экземпляры xml в столбце либо в переменной типа xml.

SQL Server поддерживает набор методов для типа данных xml:

- query() - выполняет запрос к экземпляру XML.

- `value()` - получает значения типа SQL из экземпляра XML.
- `exist()` - определяет, вернул ли запрос непустой результат.
- `modify()` - выполняет обновления.
- `nodes()` - разделяет XML на несколько строк для распространения XML-документов по наборам строк.

Для привязки реляционных данных с типом, отличным от XML, внутри XML используются следующие псевдофункции:

- `sql:column()` - позволяет использовать значения реляционного столбца в выражении XQuery или XML DML.
- `sql:variable()` - позволяет использовать значения переменной SQL в выражении XQuery или XML DML.

SQL Server поддерживает поднабор языка XQuery, который используется для выполнения запросов к типу данных xml. Эта реализация XQuery совпадает с рабочим эскизом XQuery на июль 2004 г. Язык разрабатывается консорциумом W3C с участием всех основных поставщиков баз данных, а также корпорации Майкрософт. Дополнительные сведения см. в разделе Спецификация языка W3C XQuery 1.0.

geography

Географический пространственный тип данных `geography` в SQL Server реализуется как тип данных среды .NET CLR. Этот тип представляет данные в системе координат круглой земли и сохраняет эллиптические данные, такие как координаты широты и долготы в системе GPS.

SQL Server поддерживает набор методов для пространственного типа данных `geography`:

- методы для работы с типом `geography`, как они определены в спецификации консорциума OGC,
- набор расширений Майкрософт для этого стандарта.

geometry

Плоский пространственный тип данных `geometry` в SQL Server реализуется как тип данных среды CLR. Этот тип представляет данные в евклидовом пространстве (плоской системе координат).

SQL Server поддерживает набор методов для пространственного типа данных `geometry`:

- методы для работы с типом `geometry`, определенные спецификацией консорциума OGC,
- набор расширений Майкрософт для этого стандарта.

hierarchyid

Тип данных `hierarchyid` является системным типом данных переменной длины. Тип данных `hierarchyid` используется для представления положения в 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. Значение типа данных `hierarchyid` представляет позицию в древовидной иерархии.

FILESTREAM

`FILESTREAM` позволяет приложениям на основе SQL Server хранить в файловой системе неструктурированные данные, например документы и изображения. Приложения могут одновременно использовать многопоточные API-интерфейсы и производительность файловой системы, тем самым обеспечивая транзакционную согласованность между неструктуризованными и соответствующими им структуризованными данными.

Хранилище `FILESTREAM` объединяет компонент Компонент SQL Server Database Engine с файловой системой NTFS, размещая данные больших двоичных объектов (BLOB) типа `varbinary(max)` в файловой системе в виде файлов. С помощью инструкций Transact-SQL можно вставлять, обновлять, запрашивать, искать и создавать резервные копии данных `FILESTREAM`. Интерфейсы файловой системы Win32 предоставляют потоковый доступ к этим данным.

Для кэширования данных файлов в хранилище `FILESTREAM` используется системный кэш NT. Это позволяет снизить возможное влияние данных `FILESTREAM` на производительность компонента Database Engine. Буферный пул SQL Server не используется, поэтому эта память доступна для обработки запросов.

`FILESTREAM` не включается автоматически при установке или обновлении SQL Server. `FILESTREAM` необходимо включить с помощью диспетчера конфигурации SQL Server и среды Среда SQL Server Management Studio. Для использования `FILESTREAM` нужно создать или изменить базу данных, которая будет содержать заданный тип файловой группы. После этого следует создать или изменить таблицу, чтобы она содержала столбец `varbinary(max)` с атрибутом `FILESTREAM`. После завершения выполнения этих задач можно будет пользоваться Transact-SQL и Win32 для управления данными `FILESTREAM`.