К основным объектам БД относятся следующие

* Базы данных
* Схемы
* Таблицы
* Представления
* Хранимые процедуры
* Триггеры
* Функции
* Индексы
* Определяемые пользователем типы данных
* Ключи
* Роли
* Правила БД
* Умолчания и стандартные установки базы данных

**DATABASE**

Создается вызовом **CREATE DATEBASE** *db\_name*.

При создании и изменении базы данных можно указать имя файла, который будет для нее создан (а также его размер, путь). Можно указывать те же характеристики и для *журнала транзакий* (файл, в котором хранится инфа о целостности данных, состояниях базы данных до и после выполнения транзакций).

Для изменения **ALTER TABLE,** для удаления **DROP TABLE**.

**SCHEMA**

**Схема –** это именованный контейнер для объектов базы данных, который позволяет группировать объекты в отдельные пространства имен. Таким образом, разные схемы могут содержать объекты с одинаковыми именами.

Обращаться к объекту в схеме можно следующим образом: **schema\_name.object**

В рамках подключения к серверу можно обращаться только к данным одной базы данных. Схемы не ограничивают доступ к данным, пользователь может обращаться к любой схеме в текущей бд (если имеет соответсвтующие права.)

**Преимущества схем:**

* К схеме можно применить правила безопасности, которые наследуются всеми объектами схемы.
* Схема позволяет более гибко управлять правами доступа. Можно даже создать отдельную схему для каждого пользователя.
* Не нужно назначать права доступа каждому юзеру. При этом пользователи и схемы независимы. Можно удалять пользователя без изменения схем.
* Чтобы одну базу данных могли использовать несколько пользователей независимо друг от друга.
* Чтобы объединить объекты БД в логические группы для облегчения управления ими.
* Чтобы в одной базе данных не возникало конфликтов имен.

Каждая БД имеет схему по умолчанию, которая используется для определения имен объектов, ссылки на которые делаются без указания их полных уточненных имен (В postgres – public, ms server - dbo).

Создать схему можно командой **CREATE SCHEMA** *name* **AUTHORIZATION** *user\_name;*

User\_name будет являться владельцем схемы. Схемой может владеть любой пользователь, и ее право собственности можно передавать.

Для того, чтобы переместить объект из одной схемы в другую используется команда:

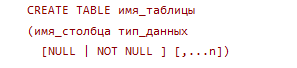
**ALTER SCHEMA** *schema\_name* **TRANSFER** *object\_name*

Для удаления схемы (если она пустая) используется **DROP SCHEMA** *schema\_name*.

**TABLE**

**Таблица –** основной объект для хранения данных. В них данные логически организованы в виде строк и столбцов. Пересечение строки и столбца называется полем.

Базовый синтаксис создания таблиц:



Изменить таблицу можно с помощью команды **ALTER TABLE**

Для удаления таблицы используется



**ИНДЕКСЫ**

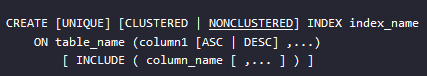
**Индекс –** объект базы данных, создаваемый с целью улучшения поиска записей в бд. Набор ссылок, упорядоченных по определенному столбцу таблицы.

Физически индекс – упорядоченный набор значений из индексированного столбца с указателями на места физического размещения исходных строк в структуре базы данных.

Обычно ссылки хранятся в виде какой-либо структуры данных, оптимизированной для поиска.

Поскольку индексы должны обновляться системой при каждом внесении изменений в таблицу, то они создают дополнительную нагрузку на систему.

Для создания индекса используется следующая команда:



**Кластерные индексы -** При кластеризованном индексе строки физически хранятся на диске в том же порядке, что и индекс.Таблица может иметь только 1 кластерный индекс.

Кластерный индекс автоматически создается при создании первичного ключа.

Он всегда должен быть уникальным. Если столбец содержит дублируещиеся значения, то СУБД принудительно обеспечивает уникальность, добавляя идентификатор к строкам, содержащим дубликаты значений.

**Некластерные индексы** – они не перестраивают физическую структуру таблицы, а лишь организуют ссылки на соответствующие строки.

Если в бд есть кластеризованный индекс, то некластерный обычно ссылается на соответсвующий элемент кластерного. Это позволяет не перестраивать структуру некластерного индекса всякий раз, когда кластерный индекс меняет физический порядок строк в таблице.

Если кластеризованного индекса нет, то некластеризованный ссылается непосредственно на строку с данными. Этот указатель содержит в себе:

* информацию об идентификационном номере файла, в котором хранится строка ;
* идентификационный номер страницы соответствующих данных;
* номер искомой строки на соответствующей странице;

**Покрывающий индекс –** индекс, в который включены все столбцы таблицы. Оптимизатор запросов может определить местонахождение всех столбцов по страницам индекса, не обращаясь к данным в таблице.

**ПРЕДСТАВЛЕНИЯ (VIEW)**

**Представление** – виртуальная таблица, которая содержит данные, полученные запросом SELECT из обычных таблиц.

К нему можно обращаться как и к обычной таблице для получения хранимых данных.

Представление может содержать данные из нескольких таблиц.

**Преимущества**:

* Q Сложный SQL запрос можно вынести в представление, и затем обращаться к нему как к простой таблице.
* Могут использоваться как механизм безопасности, открывая доступ только к определенной части таблицы.

Для создания используется выражение

**CREATE VIEW** *название\_представления* [(столбец\_1, столбец\_2, ....)]

**AS** *выражение*\_*SELECT*

Обращаемся как к обычной таблице

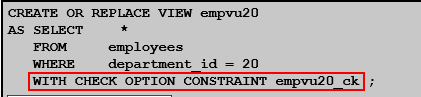
**SELECT** \* **FROM** *название\_представления* **WHERE** *условия*

**Модифицируемое представление –** при изменении данных в самом представлении (UPDATE/INSERT/DELETE), эти данные будут изменятся и в таблицах, которые эти данные хранят.

Чтобы представление было модифицируемым, оно не должно содержать в теле SELECT:

* TOP
* DISTINCT
* UNION
* JOIN
* Агрегатные функции
* GROUP BY и HAVING
* Подзапросы
* Производные столбцы и столбцы, которые вычисляются на основании нескольких значений.
* Обращения одновременно к нескольким таблицам

Для того, чтобы через представление нельзя было изменять данные, отличные от тех, что возвращаются представлением, используется предложение **WITH CHECK OPTION.**



**ХРАНИМЫЕ ПРОЦЕДУРЫ**

**Хранимая процедура**  - это набор SQL-команд. Хранимая процедура позволяет:

* Обрабатывать входные параметры и возвращать значение в виде выходных параметров.
* Содержит в себе любые инструкции SQL, включая вызовы других процедур.
* Возвращать значение состояния, предавая таким образом сведения об успешном или не успешном выполнении, а так же причины неудачи.

**Преимущества**

* **Снижение сетевого трафика** между клиентом и сервером. По сети выполняется только вызов на выполнение процедуры.
* **Большая безопасность**. Процедура проверяет, какие из процессов и действий могут выполняться, и защищает базовые объекты базы данных. Это устраняет необходимость предоставлять разрешения на уровне индивидуальных объектов и упрощает формирование уровней безопасности.  
  При вызове процедуры через сеть, виден только сам вызов. Злоумышленники не смогут видеть структуру бд.  
  Защита от иньекций. Параметры обрабатываются как литералы, и даже если передать sql-код он не будет выполнен.
* **Повторное использование кода.**
* **Повышение производительности**. Процедуры компилируется один раз, при первом запуске и сохраняется в скомпилированной форме.
* **Поддержание целостности данных**. В хранимой процедуре можно проверять правила целостности и генерировать ошибку, если часть логики нарушена, либо выполнять какие-то действия, по поддержанию целостности. (Например есть таблица отделы и сотрудники. В таблице отделы мы храним количество сотрудников. При вставке нового сотрудника, надо увеличивать и количество сотрудников).
* **Поддержка актуальности бизнес**-**приложений** – при обновлении процедур, изменения автоматически отражаются во всех приложениях, которые используют эту бд.

Команды:

* **CREATE PROCEDURE** *name* **AS**  - создать процедуру.
* **CALL, EXEC (EXECUTE)** *name [params] –* выполнить процедуру  
  Параметры предовать либо списком *param1, param2*; либо по имени *@param1=value1, @param2=value2*
* **DROP PROCEDURE** *name* – удалить процедуру
* **@***parameter\_name* ***TYPE* –** объявление параметров.   
  Параметру можно присвоить значение по умолчанию  
  Параметр можно объявить со словом **OUTPUT**. Такой параметр используется для возвращения значения из процедуры. Их может быть несколько. Возвратить единственное значение, называемое *кодом возврата*, из процедуры можно также с помощью **RETURN (**только INT**).**
* -- Создать процедуру, которая обновит таблицу student\_exam - запишет, сдал ли студент зачет и вернет его имя  
  CREATE PROCEDURE *UpdateStudentExam* @student\_id INT,  
   @student\_name VARCHAR(50) OUTPUT  
   @has\_pased VARCHAR(10) = 'зачет'  
  AS  
  UPDATE student\_exam SET result = @has\_pased WHERE id = @student\_id  
  SELECT @student\_name=student\_name FROM student\_exam WHERE id = @student\_id  
    
  -- выполнение процедуры (последний параметр будет присвоен по умолчанию)  
  -- в результате в переменной @stud\_name будет имя студента  
  DECLARE @stud\_name VARCHAR(50)  
  EXEC UpdateStudentExam 22, @stud\_name OUTPUT

Если процедура содержит обычный SELECT, то выбранный ResultSet будет выходным значением процедуры. Если выполняется несколько селектов, то клиенту возвращается несколько результирующих наборов.

**Недостатки хранимых процедур**

* Сложность написания
* Невозможность переноса между разными серверами баз данных.

**ТРИГГЕРЫ**

**Триггер** – это особая разновидность хранимых процедур, которая автоматически выполняется при возникновении события на сервере бд.

Триггеры предназначены для проверки и изменения данных при выполнении инструкций модификации или определения данных, и помогают обеспечивать целостность данных.

**Триггеры DML** выполняются, когда пользователь пытается изменить данные с помощью команд *INSERT*, *UPDATE* или *DELETE*.

**Триггеры DDL** активируются в ответ на события *CREATE*, *ALTER*, *DROP, GRANT, DENY, REVOKE и UPDATE STATISTICS* и на некоторые системные процедуры.

**Триггеры входа** могут срабатывать в ответ на событие LOGON, которое возникает при создании пользовательского сеанса.

Синтаксис создания DML триггера

CREATE TRIGGER [schema\_name.]trigger\_name  
 ON {table\_name | view\_name}  
 [WITH dml\_trigger\_option [,…]]  
 {FOR | AFTER | INSTEAD OF} { [INSERT] [,] [UPDATE] [,] [DELETE]}  
 [WITH APPEND]  
 {AS sql\_statement | EXTERNAL NAME method\_name}

**FOR, AFTER** – указывает, что триггер срабатывает только после успешного запуска всех операций в инструкции SQL. Можно использовать только для таблиц.

**BEFORE –** триггер будет срабатывать до выполнения основного sql-запроса

**INSTEAD OF** – указывает, что триггер выполняется вместо указанного действия. Можно использовать как для таблиц, так и для представлений. Нельзя использовать для триггеров DDL и входа.

**FOR EACH ROW** – означает, что триггер будет вызываться для каждой строки, обновляемой основным SQL-оператором (т.е., если вставляем 5 строк, триггер вызовется 5 раз).

**FOR EACH STATEMENT** (FOR EACH ROW не указан явно) – триггер сработает один раз для всего запроса (вставляем 5 строк, но триггер отработает 1 раз).

Внутри триггеров имеется возможность использовать псевдозаписи NEW и OLD.

**OLD** хранит начальные значение строк, те что были до выполнения инициирующего sql-запроса.

**NEW** хранит новые значения, которые будут записаны в таблицу после выполнения запроса.

Обе записи имеют такую же структуру, как таблица, для который создан триггер.

Для триггеров INSERT структура OLD пустая. Для триггеров UPDATE заполняются обе структуры (OLD – до апдейта, NEW – после аптдейта). Для триггеров DELETE структура NEW – пустая.

Изменять данные OLD нельзя. Изменение данных new – допускается.

Внутри триггеров не разрешаются следующие операции

* ALTER DATABASE
* CREATE DATABASE
* DROP DATABASE
* RESTORE DATABASE
* RESTORE LOG
* RECONFIGURE

**Недостатки**

* Триггеры работают в транзакциях и блокируют ресурсы. Чем дольше выполняется триггер, тем больше вероятность блокировки другого процесса.
* Работа триггеров неявная.
* Триггеры могут вызывать другие триггеры, а те в свою очередь другие триггеры. Неправильное определение триггеров может привести к зацикливанию.

**ФУНКЦИИ**

**Определяемые пользователем функции** являются подпрограммами, которые принимают параметры, выполняют действие, и возвращают результат в виде единичного скалярного значения либо результирующего набора.

**Преимущества**

* Функцию можно создать однажды и использовать многократно.
* Оптимизированное выполнение, компиляция и оптимизация выполняются один раз, как и в хранимых процедурах.
* Могут быть использованы внутри SQL-запросов (В то время как процедуры вызываются через EXEC).

Создание пользовательской функции

CREATE FUNCTION [schema\_name.]function\_name  
 [( {@param } type [= default]) {,...}  
 RETURNS {scalar\_type | [@variable] TABLE}  
 [WITH {ENCRYPTION | SCHEMABINDING}  
 [AS] {block | RETURN (select\_statement)}

**RETURNS** – определяет тип возвращаемого из функции значения. TABLE означает что будет возвращен набор строк.

**SCHEMABINDING** – позволяет привязать функцию к схеме каких-либо объектов, на которые она ссылается. Попытка изменения или удаления любого объекта, к которому обращается привязанная к схеме функция, приводит к ошибке.

**Параметры** могут иметь **значение по умолчанию**. Если нужно использовать значение по умолчанию, при вызове функции на месте параметра нужно писать **DEFAULT**.

Внутри функции **разрешаются** только **следующие инструкции**:

* инструкции присвоения, такие как SET;
* инструкции для управления ходом выполнения, такие как WHILE и IF;
* инструкции DECLARE, объявляющие локальные переменные;
* инструкции SELECT, содержащие списки столбцов выборки с выражениями, значения которых присваиваются переменным, являющимися локальными для данной функции;
* инструкции INSERT, UPDATE и DELETE, которые изменяют переменные с типом данных TABLE, являющиеся локальными для данной функции.

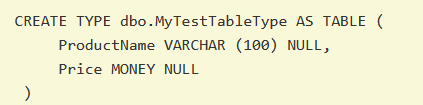
**ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ**

Пользовательские типы данных могут использоваться при определении какого-либо специфического или часто употребляемого формата.

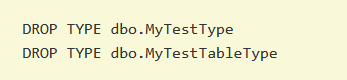
Создается с помощь команды **CREATE TYPE**



Можно также создать табличный тип, т.е. переменная сможет хранить табличные данные, например результирующие наборы.



Удаление типа



**КЛЮЧИ**

**Первичный ключ** – набор столбцов уникально идентифицирующих каждую строку таблицы.

При создании первичного ключа автоматически создается уникальный индекс (кластеризованный, если такого еще нет в таблице). На первичный ключ накладывается ограничение NOT NULL.

**Внешний ключ –** столбец или сочетание столбцов, которые применяются для установления связи между данными в двух таблицах. Столбцы, которые ссылаются на UNIQUE или PRIMARY KEY другой таблицы (Таблица может ссылаться и на саму себя).

Внешний ключ используется при

* **Вставке.** При этом проверяется, чтобы внешний ключ имелся в родительской таблице.
* **Изменении.** Если меняется поле в родительской таблице, на которое ссылается FOREIGN KEY, то оно меняется и в подчиненной.
* **Удалении.** При удалении из родительской таблицы, проверяется есть ли связанное значение в подчиненной.

**РОЛИ И ПРАВА ДОСТУПА**

**Права доступа (Permissions) –** это права, дающие возможность доступа к объекту базы данных.

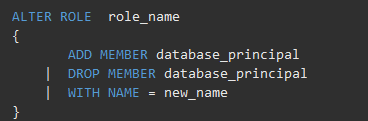
Права доступа, а также роли хранятся в системных таблицах. В них хранится информация о том кому какие права и на какие действия выделены.

**Роль** – именованный набор правил в рамках сервера или конкретной базы данных. Позволяет определять права доступа для всех членов этой роли.

Для создания роли базы данных используется команда



Для того чтобы добавить пользователя к роли, удалить его, либо изменить название роли



Для удаления роли используется DROP ROLE.

Ниже приведен список и описание прав доступа к табли­цам:

* **SELECT** — дает возможность пользователю вы­брать или прочесть данные из таблицы или представления. Сле­дует заметить, что право доступа **SELECT** может быть предос­тавлено к индивидуальным столбцам, а не только ко всей таблице или представлению.
* **INSERT** — дает возможность пользователю доба­вить новые данные в таблицу или представление.
* **UPDATE** — дает возможность пользователю изме­нить данные в таблице или представлении.
* **DELETE** — дает возможность пользователю уда­лить данные из таблицы или представления.
* **EXECUTE** — дает возможность пользователю вы­полнить хранимую про­цедуру.
* **DRI/REFERENCES** — дает возможность пользова­телю добавить к табли­це условие на значение.

Для назначения и отмены прав на выполнение действий применяются команды **GRANT**, **DENY** и **REVOKE.**

С точки зрения системы раз­граничения доступа каждый пользователь может находиться в одном из трех состояний:

* Пользователь имеет право выполнять определенное действие.
* Пользователю запрещено выполнять определенное действие.
* Пользователь не имеет четко определенных разреше­ний и запретов.

Инструкция **GRANT** используется для предоставления прав доступа пользователям или ролям

GRANT permission ON object\_name TO user\_or\_role\_name

Инструкция **DENY** применяется для удаления прав доступа, то есть пользователю или роли запрещается выполнять указанное действие. Кроме того, сервер отслеживает, чтобы пользователь или роль не унаследовали эти права от другой роли.

DENY SELECT ON Orders TO USER1;

**REVOKE** удаляет разрешение как из **GRANT** так и из **DENY**. Но это разрешение может быть унаследовано из других ролей.

REVOKE INSERT ON Orders FROM USER1;

**SEQUENCE**

**SEQUENCE** – это объект SQL Server, который генерирует числовые значения в определенной последовательности в соответствии с заданной спецификацией.

В основном используется для формирования столбца идентификаторов.

Последовательность создается как и другие объекты бд. Последовательность можно настроить чтобы она формировалась в возрастающем или убывающем порядке с заданным интервалом, настроить перезапуск если она исчерпана, задать минимальное и максимальное значение.

Последовательность кэшируется. Несколько значений последовательности генерируются заранее, и мы получаем быстрый доступ к ним.

Для получения следующего элемента используется конструкция

**NEXT VALUE FOR** *sequence\_name* в MSServer

*Sequence\_name.****nextval*** в Oracle

***Nextval(****sequence\_name****)*** в Postgres.