**Модуль №5 «Функции агрегирования (Подзапросы)»**

**Слайд №2. Цели занятия:**

* закрепить навыки применения агрегатных функций, ключевых слов GROUP BY и HAVING;
* изучить назначение и принцип работы подзапросов;
* ознакомиться с операторами EXISTS, ANY/SOME, ALL для использования в подзапросах;
* рассмотреть понятие транзакции;
* изучить функции для работы с типом данных «дата-время».

**Слайд №3. План занятия:**

1. Актуализация знаний.

2. Подзапросы:

* необходимость создания и использования подзапросов;
* сравнение подзапросов и многотабличных запросов;
* принцип работы подзапросов.

3. Операторы для использования в подзапросах:

* оператор EXISTS;
* операторы ANY/SOME;
* оператор ALL.

4. Понятие транзакции. Использование транзакций.

5. Функции для работы с типом данных «дата/время».

6. Выполнение лабораторной работы на тему: «Функции агрегирования».

7. Домашнее задание.

Слайд №4.

* ***Подзапрос (вложенный запрос)*** - это оператор выбора, который содержится внутри другого оператора выбора, вставки, обновления или удаления, внутри условного оператора или внутри другого подзапроса.

Вложенный запрос по-другому называют *внутренним запросом* или *внутренней операцией выбора*, в то время как инструкцию, содержащую вложенный запрос, называют *внешним запросом* или *внешней операцией выбора*.

Возможность включения одного оператора выбора внутрь другого является одной из причин, по которой язык SQL называется “структурированным” (Structured Query Language).

*Вложенный во внешнюю инструкцию SELECT запрос*, имеет следующие компоненты:

* обычный запрос SELECT, включающий обычные компоненты списка выборки;
* обычное предложение FROM, включающее одно или более имен таблиц или представлений;
* необязательное предложение WHERE;
* необязательное предложение GROUP BY;
* необязательное предложение HAVING.

Подзапросы могут быть вложенными в конструкциях (предложениях) **where** или **having** внешних операторов выбора (**select**), вставки (**insert**), обновления (**update**) или удаления (**delete**), а также вложенными в другие подзапросы или помещены в список выбора.

Подзапрос может появляться везде, где может использоваться выражение, если он возвращает одно значение.

Если таблица появляется только во вложенном запросе, а не во внешнем запросе, в этом случае столбцы данной таблицы не могут быть включены в выходные данные (список выборки внешнего запроса).

Очень часто встречается необходимость в запросе использовать результат какого-либо другого запроса. По сути, это представляет собой вложенность одного запроса в другой (в одном Select находится другой Select). Лучше всего необходимость этого рассмотреть на примере.

Предположим, что нам необходимо вывести фамилию сотрудника, у которого максимальная заработная плата. Теоретически запрос должен выглядеть так:

ОШИБКА!!!

SELECT max(salary) as Max, Name

FROM Employees

GROUP BY Name

Но результатом будет совсем не то, что нужно.

На первый взгляд - это просто список всех сотрудников. Это не совсем так. На самом деле фамилии сотрудников выведены без повторений, и для каждой из групп указано максимальное значение заработной платы. Т.е. если есть 2 сотрудника с одинаковыми ФИО, но с разной зарплатой, то выведется одна фамилия (учитывая, что группировка производится по ФИО) и максимальная значение зарплаты из этих двух сотрудников. Если же ФИО сотрудника не повторяется, то оно будет выведено на экран, как единственный представитель своей группы.

И тогда возникает вопрос, а что же делать? И нам на помощь приходят ***Подзапросы***. Вначале выполняется вложенный запрос, а только после этого основной.

В MS SQL Server возможно создавать **вложенность до 32-го уровня**, хотя ограничения меняются в зависимости от объема доступной памяти и сложности других выражений в запросе. Отдельные запросы могут не поддерживать вложенность до 32-го уровня.

Запрос SELECT вложенного запроса всегда заключен в скобки. Он может включать предложение ORDER BY только вместе с предложением TOP.

Теперь давайте посмотрим на проблему с максимумом с другой стороны. Сначала нужно найти просто максимальную зарплату. Это 1 запрос. После этого нужно найти ФИО сотрудника, у которого самая высокая зарплата.

SELECT Name, Salary

FROM Employees

WHERE Salary = (SELECT MAX(Salary) FROM Employees)

Сначала выполнится вложенный запрос (SELECT MAX(Salary) FROM Employees), который вернет, в результате, число 5000, определяющее максимальную зарплату в таблице Employees. После этого внешний запрос находит ФИО сотрудника, у которого зарплата равна 5000. Возможно, что таких сотрудников может быть несколько. Но важнее следить за тем, чтобы вложенный запрос возвращал не более одной записи, при стандартном сравнении (<, >, >=, <=, = и т.д.). Но бывает обратная ситуация.

Существует два типа подзапросов:

* независимые;
* связанные.

В независимых подзапросах вложенный запрос логически выполняется ровно один раз. Связанный запрос отличается от независимого тем, что его значение зависит от переменной, получаемой от внешнего запроса. Таким образом, вложенный запрос связанного подзапроса выполняется каждый раз, когда система получает новую строку от внешнего запроса.

***Независимый подзапрос*** может применяться со следующими операторами:

* операторами сравнения (=, <>, <, <=, >, >=);
* оператором IN;
* операторами ANY и ALL.

**Подзапросы и операторы сравнения**

Рассмотрим использование оператора равенства (=) в независимом подзапросе.

Пример 6.51. Выборка имен и фамилий сотрудников ИТ-отдела

USE test;

SELECT Name

FROM Employees

WHERE DepartmentID = (SELECT ID FROM Departments WHERE Name = 'ИТ')

Сначала выполняется вложенный запрос, возвращая номер ИТ-отдела (d1). После выполнения внутреннего запроса подзапрос в примере 6.51 можно представить следующим эквивалентным запросом:

USE test;

SELECT Name

FROM Employees

WHERE DepartmentID = ‘d1’;

В подзапросах можно также использовать *любые другие операторы сравнения*, при условии, что вложенный запрос возвращает в результате одну строку. Это очевидно, поскольку *невозможно сравнить конкретные значения столбца, возвращаемые внешним запросом, с набором значений, возвращаемым вложенным запросом*.

Далее рассмотрим, как можно решить проблему, когда результат вложенного запроса содержит набор значений.

**Подзапросы и оператор IN**

**Оператор IN** позволяет определить набор выражений (или констант), которые затем можно использовать в поисковом запросе. Этот оператор можно использовать в подзапросах при таких же обстоятельствах, т.е. когда вложенный запрос возвращает набор значений. Использование оператора IN в подзапросе показано в примере 6.52.

Пример 6.52. Получение всей информации о сотрудниках, работающих в Астане:

SELECT \*

FROM Employees

WHERE DepartmentID IN

(SELECT ID

FROM Departments

WHERE city = 'Астана')

Каждый вложенный запрос может содержать свои вложенные запросы. Подзапросы такого типа называются ***подзапросами с многоуровневым вложением***. Максимальная глубина вложения (т. е. количество вложенных запросов) зависит от объема памяти, которым компонент Database Engine располагает для каждой инструкции SELECT. В случае подзапросов с многоуровневым вложением система сначала выполняет самый глубокий вложенный запрос и возвращает полученный результат запросу следующего высшего уровня, который в свою очередь возвращает свой результат запросу следующего уровня над ним и т. д. Конечный результат выдается запросом самого высшего уровня.

Самостоятельно: Реализовать запрос: *-- отделы где есть сотрудники*

**SELECT** \*

**FROM** Departments

**WHERE** **ID** **IN**

(**SELECT** **DISTINCT** DepartmentID

**FROM** Employees

**WHERE** DepartmentID **IS** **NOT** NULL)

Самостоятельно: Реализовать запрос: *-- отделы где нет сотрудников*

**SELECT** \* **FROM** Departments **WHERE** **ID** **NOT** **IN**

(**SELECT** **DISTINCT** DepartmentID

**FROM** Employees

**WHERE** DepartmentID **IS** **NOT** NULL)

Запрос с несколькими уровнями вложенности показан в примере 6.53.

Пример 2. Необходимо показать студентов, которые брали книги тематик 'Базы данных', либо 'Web-дазайн'.

ОШИБКА!!!

Select students.firstname

from students,s\_cards

where s\_cards.id\_book =

(

select id from books where id\_themes =

(

select id from themes where name in

('Базы данных','Web-дизайн')

)

)

and students.id=s\_cards.id\_student

Дело все в том, что вложенный запрос, который вычитывает id тематик, возвращает более одной записи и поэтому проверять его результат на равенство приводит к ошибке, как и запрос на уровень выше о id книг, этих тематик. Чтобы можно было сделать такой запрос необходимо знак " = " заменить на IN (проверка на членство в множестве).

Правильно!!!

Select students.firstname

from students,s\_cards

where s\_cards.id\_book in

(

select id from books where id\_themes in

(

select id from themes where name in

('Базы данных','Web-дизайн')

)

)

and students.id=s\_cards.id\_student

В принципе, тот же самый результат можно было добиться, прописав связи между таблицами и указав дополнительное условие.

Существует ряд ограничений, которые накладываются на вложенные запросы:

1. Вложенный запрос необходимо всегда заключать в круглые скобки.
2. Вложенный запрос не может содержать Select into, Order by (может включать предложение ORDER BY только вместе с предложением TOP).
3. Вложенный запрос не может возвращать более одного столбца, если он включен в директиву In.
4. Вложенный запрос не может указываться в Order by.

**Подзапросы и операторы ANY и ALL**

Операторы **ANY и ALL** всегда используются в комбинации с одним из операторов сравнения. Оба оператора имеют одинаковый синтаксис:

1. <выражение> <оператор сравнения> SOME | ANY (<подзапрос>)

**SOME**и **ANY**являются синонимами, то есть может использоваться любое из них. Результатом подзапроса является один столбец величин. Если хотя бы для одного значения V, получаемого из подзапроса, результат операции *"<значение выражения> <*[*оператор сравнения*](http://www.sql-tutorial.ru/ru/book_comparison_predicates.html)*> V*" равняется **TRUE**, то предикат **ANY**также равняется **TRUE**.

Предикат в программировании — [функция](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), принимающая один или более аргументов и возвращающая значения [булева типа](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D0%B5%D0%B2_%D1%82%D0%B8%D0%BF).

1. <выражение> <оператор сравнения> ALL (<подзапрос>)

Исполняется так же, как и **ANY**, однако значение предиката **ALL** будет истинным, если для всех значений V, получаемых из подзапроса, предикат "*<значение выражения> <оператор сравнения> V*" дает **TRUE**.

Пусть имеются две таблицы: T1, содержащая как минимум столбец A, и T2, содержащая, по крайней мере, один столбец B. Тогда запрос с квантором ALL можно сформулировать следующим образом:

**SELECT A FROM T1**

**WHERE A оператор\_сравнения ALL (SELECT B FROM T2)**

Здесь оператор\_сравнения обозначает любой допустимый оператор сравнения. Данный запрос должен вернуть список всех тех значений столбца A, для которых оператор сравнения истинен для всех значений столбца B.

Запрос с квантором SOME, очевидно, имеет аналогичную структуру. Он должен вернуть список всех тех значений столбца A, для которых оператор сравнения истинен хотя бы для какого-нибудь одного значения столбца B.

Для примера работы оператора ALL из таблицы сотрудников отдела с кодом 3 выбираются все сотрудники, чей оклад не меньше оклада любого сотрудника, у которого номер отдела не указан. Для этой цели применим ALL:

SELECT Name, salary

FROM Employees

WHERE DepartmentID =3 and Salary >= ALL (SELECT salary FROM Employees where DepartmentID IS NULL)

Результатом будет список, состоящий из одной записи. Здесь более естественно было бы использовать функцию MAX:

SELECT Name, salary

FROM Employees

WHERE DepartmentID =3 and Salary >= (SELECT MAX(salary) FROM Employees where DepartmentID IS NULL)

**Функции ANY и SOME**

Это два названия одной и той же функции, т. е. синонимы.

Функция возвращает значение "**истина**", если операция сравнения истинна хотя бы для одного значения, возвращаемого подзапросом.

SELECT Name, salary

FROM Employees

WHERE DepartmentID =3 and Salary <= ANY (SELECT salary FROM Employees where DepartmentID IS NULL)

В список должны попасть те записи, для которых найдется такая запись, возвращаемая подзапросом, в которой оклад будет больше или равен, чем оклад отыскиваемой записи. Запрос вернет три строки.

Вместо функции SOME можно применить в данном случае функцию MAX, и запрос будет более понятным:

SELECT Name, salary

FROM Employees

WHERE DepartmentID =3 and Salary <= (SELECT MAX(salary) FROM Employees where DepartmentID IS NULL)

Приведем формальные правила оценки истинности предикатов, использующих параметры ANY|SOME и ALL.

* Если определен параметр ALL или SOME и все результаты сравнения значения выражения и каждого значения, полученного из подзапроса, являются TRUE, истинностное значение равно TRUE.
* Если результат выполнения подзапроса не содержит строк и определен параметр ALL, результат равен TRUE. Если же определен параметр SOME, результат равен FALSE.
* Если определен параметр ALL и результат сравнения значения выражения хотя бы с одним значением, полученным из подзапроса, является FALSE, истинностное значение равно FALSE.
* Если определен параметр SOME и хотя бы один результат сравнения значения выражения и значения, полученного из подзапроса, является TRUE, истинностное значение равно TRUE.
* Если определен параметр SOME и каждое сравнение значения выражения и значений, полученных из подзапроса, равно FALSE, истинностное значение тоже равно FALSE.
* В любом другом случае результат будет равен UNKNOWN.

**ПРИМЕЧАНИЕ**

Настоятельно рекомендуется избегать использования операторов ANY и ALL. Любой запрос с применением этих операторов можно сформулировать лучшим образом посредством функции EXISTS. Кроме этого, семантическое значение оператора ANY можно легко принять за семантическое значение оператора ALL и наоборот.

**Связанные (коррелированные) подзапросы**

Все приведенные до сих пор запросы не зависели от своих содержащих выражений, т. е. могли выполняться самостоятельно и представлять свои результаты для проверки.

Подзапрос называется **связанным (correlated)**, если любые значения вложенного запроса зависят от внешнего запроса.

Связанный подзапрос (коррелированный), напротив, зависит от содержащего выражения, из которого он ссылается на один или более столбцов. В отличие от несвязанного подзапроса, который выполняется непосредственно перед выполнением содержащего выражения, связанный подзапрос выполняется по разу для каждой строки-кандидата (это строки, которые предположительно могут быть включены в окончательные результаты).

Для примера получим отделы, в которых числится хотя бы 1 сотрудник:

SELECT \*

FROM Departments d

WHERE 0 < (SELECT COUNT(\*) FROM Employees e WHERE e.DepartmentID = d.ID)

Ссылка на d.ID в самом конце подзапроса - это то, что делает этот подзапрос связанным. Чтобы подзапрос мог выполняться, основной запрос должен поставлять значения для d.ID. В данном случае основной запрос извлекает из таблицы Departments все строки и выполняет по одному подзапросу для всех отделов, передавая в него соответствующий Id отдела при каждом выполнении. Если подзапрос возвращает значение большее 0, условие фильтрации выполняется и строка добавляется в результирующий набор.

Связанные подзапросы часто используются с условиями сравнения (в предыдущем примере <) и вхождения в диапазон, но самый распространенный оператор, применяемый в условиях со связанными подзапросами, - это **оператор EXISTS (существует)**.

**Подзапросы и функция EXISTS**

Функция EXISTS принимает вложенный запрос в качестве аргумента и возвращает значение FALSE, если вложенный запрос не возвращает строк и значение TRUE в противном случае.

Эта функция возвращает значение "истина", если в списке выбора существует, как минимум, одно возвращаемое значение.

Оператор EXISTS применяется, если требуется показать, что связь есть, а количество связей при этом не имеет значения. Например, следующий запрос возвращает список всех товаров, которые когда-либо заказывали.

Позволяют проверить есть ли соответствующие условию записи в подзапросе:

*-- отделы в которых есть хотя бы один сотрудник*

**SELECT** \*

**FROM** Departments dep

**WHERE** **EXISTS**(**SELECT** \* **FROM** Employees emp **WHERE** emp.DepartmentID=dep.**ID**)

*-- отделы в которых нет ни одного сотрудника*

**SELECT** \*

**FROM** Departments dep

**WHERE** **NOT** **EXISTS**(**SELECT** \* **FROM** Employees emp **WHERE** emp.DepartmentID=dep.**ID**)

Здесь все просто – EXISTS возвращает True, если подзапрос возвращает хотя бы одну строку, и False, если подзапрос не возвращает строк. NOT EXISTS – инверсия результата.

Вложенный запрос функции EXISTS почти всегда зависит от переменной с внешнего запроса. Поэтому функция EXISTS обычно определяет связанный подзапрос.

Инструкции, включающие вложенные запросы, обычно имеют один из следующих форматов:

WHERE expression [NOT] IN (subquery)

WHERE expression comparison\_operator [ANY | ALL] (subquery)

WHERE [NOT] EXISTS (subquery)

Для примера в каждом отделе выберем сотрудника, у которого ЗП больше ЗП всех сотрудников работающих в этом же отделе. Для этой цели применим ALL:

**SELECT** **ID**,**Name**,DepartmentID,Salary

**FROM** Employees e1

**WHERE** e1.Salary>ALL(

**SELECT** e2.Salary

**FROM** Employees e2

**WHERE** e2.DepartmentID=e1.DepartmentID *-- учесть только сотрудников этого же отдела*

**AND** e2.**ID**<>e1.**ID** *-- чтобы исключить сравнение со своей же ЗП*

**AND** e2.Salary **IS** **NOT** NULL *-- исключить NULL значения*

*)*

Здесь происходит проверка на то, что e1.Salary больше значений e2.Salary, которые вернул подзапрос.

Как думаете, почему здесь вернулись даже те сотрудники, для которых подзапрос не вернул ни одной строки? А потому что логика такая – нет записей, не с чем проверять, а значит я и так больше всех. ))) Вот такая хитрость здесь скрыта.

Для большего понимания, давайте посмотрим, как можно здесь оператор ALL заменить оператором NOT EXISTS:

**SELECT** **ID**,**Name**,DepartmentID,Salary

**FROM** Employees e1

**WHERE** **NOT** **EXISTS**(

**SELECT** \*

**FROM** Employees e2

**WHERE** e2.DepartmentID=e1.DepartmentID *-- учесть только сотрудников этого же отдела*

**AND** e2.Salary>e1.Salary *-- выбираем только ЗП больше ЗП этого сотрудника*

)

Т.е. мы тут выразили то же самое только другими словами «Верни сотрудников для которых нет сотрудников из того же отдела с большей ЗП чем у него».

Здесь становится понятно почему ALL возвращает истинное значение в том случае если подзапрос не возвращает данных.

Так же обратите внимание, что для ALL важно исключить NULL-значения из подзапроса, иначе результат проверки на каждое значение может оказаться неопределенным. Сравнивайте в этом случае логика ALL логикой при использовании AND, т.е. выражение (Salary>1000 AND Salary>1500 AND Salary>NULL) вернет NULL.

А вот с ANY (он же SOME) будет по-другому:

**SELECT** **ID**,**Name**,DepartmentID,Salary

**FROM** Employees e1

**WHERE** e1.Salary>**ANY**( *-- ANY = SOME*

**SELECT** e2.Salary

**FROM** Employees e2

**WHERE** e2.DepartmentID=e1.DepartmentID *-- учесть только сотрудников этого же отдела*

**AND** e2.**ID**<>e1.**ID** *-- чтобы исключить сравнение со своей же ЗП*

)

C оператором ANY важно, чтобы подзапрос вернул записи, с которыми можно сравнить на любое выполнение условия. Т.к. во всех отделах сидят только по одному сотруднику, кроме ИТ-отдела, то вернулся только Андреев А.А. и Николаев Н.Н., чьи ЗП удалось сравнить с ЗП третьего сотрудника этого же отдела. Т.е. мы вытащили здесь тех, чья ЗП больше любой ЗП сотрудника из этого же отдела.

Давайте для большего понимания, попробуем выразить здесь ANY при помощи EXISTS:

**SELECT** **ID**,**Name**,DepartmentID,Salary

**FROM** Employees e1

**WHERE** **EXISTS**(

**SELECT** \*

**FROM** Employees e2

**WHERE** e2.DepartmentID=e1.DepartmentID *-- учесть только сотрудников этого же отдела*

**AND** e2.Salary<e1.Salary *-- проверяем есть ли сотрудники с меньшей ЗП чем у данного сотрудника*

)

Смысл здесь стал «есть ли хоть какой-то сотрудник из этого отделу у которого ЗП ниже ЗП данного сотрудника».

В таком виде становится понятно, почему ANY возвращает ложное значение, если подзапрос не возвращает данных.

Наличие NULL-значений в подзапросе здесь не так опасно, т.к. мы сравниваем на любое значение. Сравнивайте в этом случае логика ANY логикой при использовании OR, т.е. выражение (Salary>1000 OR Salary>1500 OR Salary>NULL) может вернуть истинное значение если выполнится хотя бы одно условие.

**Что использовать, соединения или подзапросы?**

Почти все инструкции SELECT для соединения таблицы посредством оператора соединения можно заменить инструкциями подзапроса и наоборот. Конструкция инструкции SELECT с использованием оператора соединения часто более удобно читаемая и легче понимаемая, а также может помочь компоненту Database Engine найти более эффективную стратегию для выборки требуемых данных. Но некоторые задачи легче поддаются решению посредством подзапросов, а другие при помощи соединений.

**Преимущества подзапросов**

*Подзапросы будет более выгодно использовать в таких случаях, когда требуется вычислить агрегатное значение "на лету" и использовать его в другом запросе для сравнения.* Это показано в примере 6.73.

Пример 6.73. Выборка табельных номеров сотрудников и дат начала их работы над проектом (enter\_date) для всех сотрудников, у которых дата начала работы равна самой ранней дате

USE sample;

SELECT emp\_no, enter\_date

FROM works\_on

WHERE enter\_date = (SELECT min(enter\_date)

FROM works\_on);

Решить эту задачу с помощью соединения будет нелегко, поскольку для этого нужно поместить агрегатную функцию в предложении WHERE, а это не разрешается. (Эту задачу можно решить, используя два отдельных запроса по отношении к таблице works\_on.)

**Преимущества многотабличных запросов**

*Использовать соединения вместо подзапросов выгоднее в тех случаях, когда список выбора инструкции SELECT в запросе содержит столбцы более чем из одной таблицы*. Это показано в примере 6.74.

Пример 6.74. Выборка информации о всех сотрудниках (табельный номер, фамилия и должность), которые начали участвовать в работе над проектом 15 октября 2007 г.

USE sample;

SELECT employee,emp\_no, emp\_lname, job

FROM employee, works\_on

WHERE employee.emp\_no = works\_on.emp\_no

AND enter\_date = '10.15.2007';

Список выбора инструкции SELECT в запросе примера 6.74 содержит столбцы emp\_no и emp\_lname из таблицы employee и столбец job из таблицы works\_on. По этой причине решение с применением подзапроса возвратило бы ошибку, поскольку подзапросы могут отображать информацию только из внешней таблицы.

## Понятие транзакции. Использование транзакций.

Прежде чем давать определение, выяснить у студентов их понимание транзакции и на примерах продемонстрировать применение транзакционных процессов, которые они уже наблюдали в обыденной жизни (например, сообщение в банкоматах о выполненной или не выполненной успешно транзакции).

Привести примеры процессов, которые можно назвать транзакционными: банковские операции двух операторов одновременно, выдача хвостовки и т.д.

Подчеркнуть, что существуют такие ситуации, для разрешения которых нужно предусмотреть способ, который может обеспечить возможность построения группы действий , которые будут выполняться по-принципу «все или ничего», т.е. будут выполнены все действия группы либо они будут отменены.

Рассмотрим классическую задачу – банковскую проводку. Допустим, что у нас есть таблица из двух полей – номер счета в банке и сумма денег на этом счету. Нам необходимо перевести деньги с одного счета на другой. Для этого нужно выполнить запрос UPDATE, чтобы уменьшить сумму первого счета на нужную сумму. После этого выполняем UPDATE, чтобы увеличить значение второго счета. Все вроде бы нормально. А что, если после уменьшения первого счета выключат свет и сервер не успеет пополнить другой счет? Деньги уже сняты, но никуда не записаны, а значит, они пропали.

Если сначала пополнять счет, а потом снимать деньги, то если снятие не успеет произойти, то банк может оказаться банкротом, ведь появляется лишняя сумма, снятия не происходит.

Некоторые могут сказать, что сбои в электроэнергии и работе компьютера происходят редко, но помимо этого бывают блокировки записей. Один пользователь может заблокировать запись для изменения и это не позволит запросу изменить ее и снятие или увеличение счета не произойдет.

Проблему решает транзакция. Перед выполнением операций обновления необходимо явно начать транзакцию. После этого выполняем две операции UPDATE и по их окончанию завершаем транзакцию. Если в момент выполнения одного из запросов происходит сбой, то все изменения, происшедшие после начала транзакции отменяются.

**Транзакция** - это механизм, позволяющий объединить набор операций в один логический блок, определяющий либо подтверждение полного выполнения каждой из операций, либо отмена всех из них.

Любая транзакция обладает четырьмя важными свойствами, определенными в английском языке, как свойства ACID (Atomicity, Consistency, Isolation, Durability). Обо всех по порядку.

1. Atomicity - атомарность. Это свойство определяет неделимость транзакции, т.е. не может выполниться только какая либо ее часть, она выполняется либо полностью, либо не выполняется вообще.
2. Consistency - связность. Определяет, что, как в случае успешного, так и аварийного завершения транзакции, это не может повлиять на целостность данных.
3. Isolation - изолированность. Определяет, что одна транзакция не может взаимодействовать с другими транзакциями.
4. Durability - надежность. Определяет, что выполнение транзакции не зависит от внешних факторов. Т.е. даже если вдруг выключили свет и сервер отключился в момент выполнения транзакции, то если она не закончила свое выполнение, после перезагрузки сервера, вся информация останется в первоначальном виде.

Транзакция может быть выполнена в одном из трех режимов:

- автофиксация (autocommit),

- явный режим (explicit)

- неявный режим (implicit).

По умолчанию для SQL Server принят режим автофиксации. Перейти к рассмотрению каждого из режимов.

**Режим автофиксации**

В этом режиме каждый оператор будет зафиксирован после того как будет завершен. Никаких дополнительных операторов в этом режиме не используется.

Т.е., как было сказано раннее, один оператор всегда представляет собой минимальную единицу транзакции и будет фиксироваться сразу после его завершения.

Данный режим по умолчанию будет актуален до тех пор , пока вы не активируете другие режимы.

### Явные транзакции.

Явные - это транзакции, заданные явно. Такие транзакции начинаются с ключевых слов

* Begin Tran, либо Begin Transaction

, а заканчиваться должны либо словами

* Commit Tran, либо Commit Word

для успешного завершения работы транзакции, а

* Rollback Tran, Rollback Transaction, либо Rollback Work

для полного отката транзакции.

Например.

Begin Transaction

Insert into books

values(771,8899,0,'Проверка работы транзакции',520,'Test',120,'20x20/15',

2004-05-03,1,'Использование транзакций','транзакции')

Select \* from books

Where N=771

Commit Transaction

Иногда требуется проверить, успешно ли завершился какой-либо запрос и относительно этого определить, начинать ли выполнение следующего запроса. Для отслеживания успешности операций служит специальная системная переменная @@ERROR. Если Запрос прошел успешно, то в эту переменную записывается ноль, если же произошла ошибка, то в нее записывается код ошибки. Это значение хранится в переменной до начала выполнения следующей транзакции.

Begin Transaction

Insert into books

values(771,8899,0,'Проверка работы транзакции',520,'Test',120,'20x20/15',

2004-05-03,1,'Использование транзакций','транзакции')

IF @@ERROR<>0

BEGIN

PRINT 'Error occured'

rollback transaction

END

ELSE

Commit Transaction

Если вы поле N таблицы books сделаете первичным ключем (при помощи изменения настроек таблицы) и выполните указанную транзакцию, то на экране появится стандартное сообщение о том, что в поле, которое является первичным, нельзя записать два одинаковых значения, и, учитывая, что в переменной @@ERROR будет не ноль, то высветится еще настроенное нами сообщение Error occured и транзакция будет откатана.

Команда Rollback transaction откатывает всю выполняемую транзакцию. Но иногда требуется сохранить какую-либо уже успешно законченную операцию и ее не откатывать, даже если произошла ошибка в операции, указанной ниже. Для этого служит установка ***точки сохранения***.

***Точка сохранения*** - устанавливается при помощи команды

Save Transaction имя\_точки\_сохранения

Тогда, для отката того, что находится после этой точки сохранения необходимо указывать команду:

Rollback transaction имя\_точки\_сохранения

commit transaction

Если указать Rollback transaction, то будет произведен откат всей транзакции.

Если не указать commit transaction после Rollback transaction имя\_точки\_сохранения, то транзакция зависнет и доступа к базе данных, к которой производился запрос, не будет, вплоть до вызова этого же запроса с коммандой commit transaction.

Пример.

Begin Transaction

insert into books (N,name,new,price) values (770,'Test1',1,12.53)

Save transaction savepoint1

insert into books (N,name,new,price) values (771,'Test1',1,12.53)

insert into books (N,name,new,price) values (770,'Test1',1,12.53)

if @@ERROR<>0

begin

print 'my error!!!'

Rollback Transaction savepoint1

Commit Transaction

end

else

Commit Transaction

Давайте рассмотрим данный пример подробнее.

* Определяем начало транзакции.
* Добавляем в таблицу books значения 770 - первичный ключ, Test1 - название книги, 1 - это книга новинка (это поле обязательное для заполнения), 12.53 - цена этой воображаемой книги.
* Устанавливаем точку сохранения.
* Добавляем в базу данных еще одну книгу, но номер ей указываем 771. Такого номера нет, потому этот запрос тоже выполнится.
* ***ВНИМАНИЕ*** Добавляем в таблицу снова книгу с номером 770. Если N - первичный ключ, то происходит ошибка добавления записи, т.к. номера повторяются.
* Учитывая, что произошла ошибка, в переменной @@ERROR будет не ноль и команда PRINT 'строка' выведет на экран сообщение об ошибке.
* Производим откат всех операций, произведенных после точки сохранения. **Книга с номером 770 в базе должна остаться, а с номером 771, хотя запрос на добавление был выполнен успешно, сохранена не будет, за счет отката транзакции.**
* Завершаем транзакцию.

На этом примере очень четко можно рассмотреть работу транзакции. Мы определяем, что должен выполнится весь блок из трех запросов на добавление информации. Но мы также ограничиваем откат транзакции, используя точку сохранения. Т.е. если вызвать Rollback Transaction, то произойдет откат двух удачно выполненных запросов, но, учитывая точку сохранения, откатывается все, кроме тех операций, которые указаны до точки сохранения.

### Неявные транзакции.

Работая с транзакциями, не всегда удобно для каждой из них указывать Begin Transaction. Если включить механизм неявных транзакций, то Begin Transaction дописывается автоматически. Но, учитывая, что транзакцию можно как закончить, так и отменить, ни Commit transaction, ни Rollback Transaction автоматически дописываться не будут. Их все равно необходимо в конце транзакции указывать явно.

Для включения механизма неявных транзакций необходимо указать:

Set Implicit\_transactions ON

Чтобы выключить:

Set Implicit\_transactions OFF

### Автоматические транзакции.

Все дело в том, что даже если не указывать начало и конец транзакции, они все равно присутствуют. Например, запрос на вставку данных можно рассматривать как отдельную автоматическую транзакцию.

Пример.

insert into books

values(772,8882,1,'Test2',125.23,'test\_izd',550,'test\_fromat',

2005-08-30,12501,'test\_themes','test\_category')

В результате этой транзакции будет либо добавлена информация во все поля, либо не будет сохранена ни в одном из них. Так любой запрос сам по себе является транзакцией.