МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

Институт кибербезопасности и цифровых технологий

Кафедра КБ-2 «Информационно-аналитические системы кибербезопасности»

Кафедра «Информационно-аналитические системы кибербезопасности» (КБ-2)

# Методические рекомендации для практической работы 4 по дисциплине «Безопасность систем баз данных»

**10.03.01** «Информационная безопасность», профиль подготовки: «Организация и технология защиты информации»;

**10.05.03** «Информационная безопасность автоматизированных систем», специализация: «Создание автоматизированных систем в защищенном исполнении

**Москва 2024**

**Практическая работа № 4 Запросы**

- **Инструкция *SELECT*. Ее предложения и функции**
- **Подзапросы**
- **Временные таблицы**
- **Операторы соединения**
- **Связанные подзапросы**
- **Табличные выражения**

**Инструкция *SELECT*. Ее предложения и функции**

В языке Transact-SQL имеется одна основная инструкция для выборки информации из базы данных — инструкция SELECT. Эта инструкция позволяет извлекать информацию из одной или нескольких таблиц базы данных и даже из нескольких баз данных. Результаты выполнения инструкции select помещаются в еще одну таблицу, называемую ***результирующим набором*** (result set).

Самая простая форма инструкции SELECT состоит из списка столбцов выборки и предложения FROM. (Все прочие предложения являются необязательными.) Эта форма инструкции SELECT имеет следующий синтаксис:

SELECT [ALL | DISTINCT] ***column_list***
FROM {***table1*** [***tab_alias1]***} , . . .

В параметре ***tablel*** указывается имя таблицы, из которой извлекается информация, а в параметре ***tab_aliasi*** — псевдоним имени соответствующей таблицы. ***Псевдоним*** — это другое, сокращенное, имя таблицы, посредством которого можно обращаться к этой таблице или к двум логическим экземплярам одной физической таблицы. Если вы испытываете трудности с пониманием этого концепта, не волнуйтесь, поскольку все станет ясно в процессе рассмотрения примеров.

В параметре ***column_list*** указывается один или несколько из следующих спецификаторов:

- символ звездочка (*) указывает все столбцы таблиц, перечисленных в предложении from (или с одной таблицы, если задано квалификатором в виде ***table2***.*);
- явное указание имен столбцов, из которых нужно извлечь значения;
- спецификатор в виде ***column_name*** [as] ***column_heading,*** что позволяет заменить имя столбца или присвоить новое имя выражению;
- выражение;
- системная или агрегатная функция.

*ПРИМЕЧАНИЕ*

Кроме только что перечисленных спецификаторов есть и другие опции, которые рассматриваются далее в этой работе.

Инструкция select может извлекать из таблицы как отдельные столбцы, так и строки. Первая операция называется ***списком выбора*** (или ***проекцией),*** а вторая — ***выборкой.*** Инструкция select также позволяет выполнять комбинацию обеих этих операций.

В примере 1 показана самая простая форма выборки данных посредством инструкции SELECT.

**Пример 1. Извлечение полной информации об отделах**

```
USE sample;
SELECT dept_no, dept_name, location
FROM department;
```

В примере 1 инструкция SELECT извлекает все строки всех столбцов таблицы department. Если список выбора инструкции SELECT содержит все столбцы таблицы (как это показано в примере 1), их можно указать с помощью звездочки (*), но использовать этот способ не рекомендуется. Имена столбцов служат в качестве заголовков столбцов в результирующем выводе.

Только что рассмотренная простейшая форма инструкции SELECT не очень полезна для практических запросов. На практике в запросах с инструкцией SELECT приходится применять намного больше

предложений, чем в запросе, приведенном в примере 1. Далее показан синтаксис инструкции SELECT, содержащий почти все возможные предложения:

SELECT ***select_list*** [INTO ***new_table]***
FROM ***table***
[WHERE ***search_condition]***
[GROUP BY ***group_by_expression]***
[HAVING ***search_condition***]
[ORDER BY ***order_expression*** [ASC | DESC]];

**Предложение *WHERE***

Часто при выборке данных из таблицы нужны данные только из определенных строк, для чего в запросе определяется одно или несколько соответствующих условий. В предложении WHERE определяется логическое выражение (т. е. выражение, возвращающее одно из двух значений: TRUE или FALSE), которое проверяется для каждой из строк, кандидатов на выборку. Если строка удовлетворяет условию выражения, т. е. выражение возвращает значение TRUE, она включается в выборку; в противном случае строка пропускается. Применение предложения WHERE показано в примере 2.

**Пример 2. Выборка имен и номеров отделов, расположенных в городе Даллас**

Кроме знака равенства, в предложении WHERE могут применяться другие операторы сравнения, включая следующие:

| | |
|---|---|
| <> (или !=) | не равно |
| < | меньше чем |
| > | больше чем |
| >= | больше чем или равно |
| <= | меньше чем или равно |
| !> | не больше чем |
| !< | не меньше чем |

**Пример 3. Выборка имен и фамилий всех сотрудников, чей табельный номер больше или равен 15000**

**Пример 4. Выборка проектов с бюджетом свыше 60000 фунтов стерлингов.**
**Валютный курс: 0.51 фунтов стерлингов за $1**

Сравнение строк (т. е. значений с типами данных CHAR, VARCHAR, NCHAR и NVARCAR) выполняется в действующем порядке сортировки, а именно в порядке сортировки, указанном при установке компонента Database Engine. При сравнении строк в кодировке ASCI (или в любой другой кодировке) сравниваются соответствующие символы каждой строки (т. е. первый символ первой строки с первым символом второй строки, второй символ первой строки со вторым символом второй строки и т. д.). Старшинство символа определяется его позицией в кодовой таблице: символ, чей код стоит в таблице перед кодом другого, считается меньше этого символа. При сравнении строк разной длины, более короткая строка дополняется в конце пробелами до длины более длинной строки. Числа сравниваются алгебраически. Значения временных типов данных (таких как DATE, TIME и DATETIME) сравниваются в хронологическом порядке.

**Логические операторы**

Условия предложения where могут быть простыми или составными, т. е. содержащими несколько простых условий. Множественные условия можно создавать посредством логических операторов and, or и not. Действия этих операторов изложены в таблицах истинности, приведенных в ***работе 2.***
При соединении условий оператором AND возвращаются только те строки, которые удовлетворяют обоим условиям. При соединении двух условий оператором OR возвращаются все строки таблицы, которые удовлетворяют одному или обоим этим условиям, как показано в примере 5.

**Пример 5. Выборка номеров сотрудников, которые работают над проектом p1 или p2 (или над обоими)**

Результаты выполнения примера 5 содержат дубликаты значений столбца emp_no. Эту избыточную информацию можно устранить с помощью ключевого слова DISTINCT, как показано в следующем примере:

```
USE sample;
SELECT DISTINCT emp_no
FROM works_on
WHERE project_no = 'p1' OR project_no = 'p2';
```

**Пример 6. Неправильный запрос**

```
USE sample;
SELECT emp_fname, DISTINCT emp_no FROM employee
    WHERE emp_lname = 'Moser';
```

Код в примере 6 ошибочен. Почему? Объяснить.

**Пример 7. Неправильное размещение логических операторов**

```
USE sample;
SELECT emp_no, emp_fname, emp_lname FROM employee
    WHERE emp_no = 25348 AND emp_lname = 'Smith'
    OR emp_fname = 'Matthew' AND dept_no = 'd1';

SELECT emp_no, emp_fname, emp_lname FROM employee
    WHERE ((emp_no = 25348 AND emp_lname = 'Smith')
    OR emp_fname ='Matthew') AND dept_no = 'd1';
```

Объяснить почему, эти два кажущиеся одинаковыми запроса SELECT выдают два разных результирующих набора данных.

Наличие логических операторов в предложении WHERE усложняет содержащую его инструкцию SELECT и способствует появлению в ней ошибок. В таких случаях настоятельно рекомендуется применять скобки, даже если они не являются необходимыми. Применение скобок значительно улучшает читаемость инструкции SELECT и уменьшает возможность появления в ней ошибок.

Логический оператор not изменяет логическое значение, к которому он применяется, на противоположное. В таблице истинности в *работе 3* показано, что отрицание истинного значения (true) дает ложь (false) и наоборот. Отрицание значения null также дает null.

Пример 8 выполните с использованием оператора отрицания NOT.

**Пример 8. Выборка табельных номеров и имен сотрудников, не принадлежащих к отделу d2**

Как можно выполнить это запрос без использования оператора NOT.

**Операторы IN *и BETWEEN***

Оператор IN позволяет указать одно или несколько выражений, по которым следует выполнять поиск в запросе. Результатом выражения будет истина (TRUE), если значение соответствующего столбца равно одному из условий, указанных в предикате IN.

**Пример 9. Выборка всех столбцов сотрудников, чей табельный номер равен 29346, 28559 или 25348**

Оператор IN равнозначен последовательности условий, соединенных операторами OR. (Число операторов OR на один меньше, чем количество выражений в списке оператора IN.)
Выполните пример 10 с использованием совместно с логическим оператором NOT.

**Пример 10. Выборка всех столбцов для сотрудников, чей табельный номер не равен ни 10102, ни 9031**

В отличие от оператора IN, для которого указываются отдельные значения, для оператора BETWEEN указывается диапазон значений, чьи границы определяются нижним и верхним значениями.

**Пример 11. Выборка наименований проектов и бюджетов всех проектов с бюджетом, находящимся в диапазоне от $95 ООО и до $120 ООО включительно**

Оператор between возвращает все значения в указанном диапазоне, включая значения для границ; т. е. приемлемые значения могут быть между значениями указанных границ диапазона или быть *равными* значениям этих границ.

Оператор BETWEEN логически эквивалентен двум отдельным сравнениям, соединенным логическим оператором AND. Поэтому запрос, приведенный в примере 12, эквивалентен запросу примера 11.

**Пример 12. Замена оператора BETWEEN сравнениями, соединенными оператором AND**

Подобно оператору BETWEEN, оператор NOT BETWEEN можно использовать для выборки значений, находящихся за пределами указанного диапазона значений. Оператор BETWEEN также можно применять со значениями, которые имеют символьный или временной тип данных.

В примере 13 покажите две разные формы запроса SELECT, которые дают одинаковые результаты.

**Пример 13. Выборка всех проектов с бюджетом меньшим, чем $100 000 и большим, чем $150 000**

**Запросы, связанные со значением *NULL***

Параметр NULL в инструкции CREATE TABLE указывает, что соответствующий столбец может содержать специальное значение NULL (которое обычно представляет неизвестное или неприменимое значение). Значения NULL отличаются от всех других значений базы данных. Предложение WHERE инструкции SELECT обычно возвращает строки, удовлетворяющие указанным в нем условиям сравнения. Но здесь возникает вопрос, как будут оцениваться в этих сравнениях значения NULL?

Все сравнения со значением NULL возвращают FALSE, даже если им предшествует оператор NOT. Для выборки строк, содержащих значения NULL, в языке Transact-SQL применяется оператор IS NULL. Указание в предложении WHERE строк, содержащих (или не содержащих) значение NULL, имеет следующую общую форму:

```
column IS [NOT] NULL
```

Использование оператора IS NULL демонстрируется в примере 14.

**Пример 14. Выборка табельных номеров служащих и соответствующих номеров проектов для служащих, чья должность неизвестна и которые работают над проектом p2**

А в примере 15 демонстрируется синтаксически правильное, но логически неправильное использование сравнения с NULL. Объясните причину ошибки.

**Пример 15. Неправильное построение проверки значения на NULL**

```
USE sample;
SELECT project_no, job FROM works_on WHERE job <> NULL;
```

Условие "column IS NOT NULL" эквивалентно условию "NOT (column IS NULL)".

Системная функция ISNULL позволяет отображать указанное значение вместо значения NULL (пример 16).

**Пример 16. Использование системной функции ISNULL**

```
USE sample;
SELECT emp_no, ISNULL(job, 'Job unknown') AS task
FROM works_on
```

```
WHERE project_no = ’pi’;
```

В примере 16 для столбца должностей job в результате запроса используется заголовок task.

**Оператор *LIKE***

Оператор LIKE используется для сопоставления с образцом, т. е. он сравнивает значения столбца с указанным шаблоном. Столбец может быть любого символьного типа данных или типа дата. Общая форма оператора LIKE выглядит таким образом:
column [NOT] LIKE ’pattern’

Параметр ’pattern’ может быть строковой константой, или константой даты, или выражением (включая столбцы таблицы), и должен быть совместимым с типом данных соответствующего столбца. Для указанного столбца сравнение значения строки и шаблона возвращает TRUE, если значение совпадает с выражением шаблона.
Определенные применяемые в шаблоне символы, называющиеся *подстановочными символами* (wildcard characters), имеют специальное значение. Рассмотрим два из этих символов:

- % (знак процента) — обозначает последовательность любых символов любой длины;
- _ (символ подчеркивания) — обозначает любой один символ.

Использование подстановочных символов % и _ показано в примере 17.

**Пример 17. Выборка имен, фамилий и табельных номеров сотрудников, у которых второй**

Кроме знака процентов и символа подчеркивания, поддерживает другие специальные символы, применяемые с оператором LIKE. Использование этих символов ([, ], ^) демонстрируется в примерах 18 и 19.

**Пример 18. Выборка всех столбцов отделов, для которых наименование места расположения** (location) **начинается с символа в диапазоне от "C" до "F"**

```
USE sample;
SELECT dept_nt, dept_name, location FROM department WHERE location LIKE ’[C-F]%’;
```

Как можно видеть по результатам примера 18, квадратные скобки [] ограничивают диапазон или список символов. Порядок отображения символов диапазона определяется порядком сортировки, указанным при установке системы.

Символ ^ обозначает отрицание диапазона или списка символов. Но такое значение этот символ имеет только тогда, когда находится внутри квадратных скобок, как показано в примере 19. В данном примере запроса осуществляется выборка имен и фамилий тех сотрудников, чьи фамилии начинаются с буквы отличной от J до O и чьи имена начинаются не с буквы E или Z.

**Пример 19. Использование символа отрицания**

```
USE sample;
SELECT emp_no, emp_fname, emp_lname FROM employee
    WHERE emp_lname LIKE ’[^J-0]%1’
    AND emp_fname LIKE ’[^EZ]%’;
```

Условие "column NOT LIKE ’pattern’" эквивалентно условию "NOT (column LIKE ’pattern’)".

В примере 20 демонстрируется использование оператора LIKE совместно с отрицанием NOT.

**Пример 20. Выборка всех столбцов сотрудников, чьи имена оканчиваются на букву, отличную от буквы "n"**

```
USE sample;
SELECT emp_no, emp_fname, emp_lname FROM employee
    WHERE emp_fname NOT LIKE ’%n’;
```

Любой подстановочный символ (%, _, [, ] или ^), заключенный в квадратные скобки, остается

обычным символом и представляет сам себя. Такая же возможность существует при использовании параметра ESCAPE. Поэтому оба варианта применения инструкции SELECT, показанные в примере 21, эквивалентны.

**Пример 21. Представление подстановочных символов, как обычных символов**

```
USE sample;
SELECT project_no, project_name FROM project
    WHERE project_name LIKE '%[_]%';

SELECT project_no, project_name FROM project
    WHERE project_name LIKE '%!_%' ESCAPE '!';
```

В примере 21 обе инструкции SELECT рассматривают символ подчеркивания в значениях столбца project_name, как таковой, а не как подстановочный. В первой инструкции SELECT это достигается заключением символа подчеркивания в квадратные скобки. А во второй инструкции SELECT этот же эффект достигается за счет применения символа перехода (escape character), каковым в данном случае является символ восклицательного знака. Символ перехода переопределяет значение символа подчеркивания, делая его из подстановочного символа обычным. (Результат выполнения этих инструкций содержит ноль строк, потому что ни одно имя проекта не содержит символов подчеркивания.)

*ПРИМЕЧАНИЕ*

Стандарт SQL поддерживает только подстановочные символы %, _ и оператор escape. Поэтому если требуется представить подстановочный символ как обычный символ, то вместо квадратных скобок рекомендуется применять оператор escape.

**Предложение *GROUP BY***

Предложение GROUP BY группирует выбранный набор строк для получения набора сводных строк по значениям одного или нескольких столбцов или выражений. Простой случай применения предложения GROUP BY показано в примере 22.

**Пример 22. Выбрать все должности сотрудников. Используйте 2 варианта: путем использования предложения GROUP BY и путем использования конструкции DISTINCT.**

В примере 22 предложение GROUP BY создает отдельную группу для всех возможных значений (включая значение NULL) столбца job.

***ПРИМЕЧАНИЕ***

Использование столбцов в предложении group by должно отвечать определенным условиям. В частности, каждый столбец в списке выборки запроса также должен присутствовать в предложении group by. Это требование не распространяется на константы и столбцы, являющиеся частью агрегатной функции. (Агрегатные функции рассматриваются в следующем подразделе.) Это имеет смысл, т. к. только для столбцов в предложении group by гарантируется одно значение для каждой группы.

Таблицу можно сгруппировать по любой комбинации ее столбцов. В примере 23 демонстрируется группирование строк таблицы works_on по двум столбцам.

**Пример 23. Группирование сотрудников по номеру проекта и должности**

Последовательность имен столбцов в предложении GROUP BY не обязательно должна быть такой же, как и в списке столбцов выборки SELECT.

**Агрегатные функции**

Агрегатные функции используются для получения суммарных значений. Все агрегатные функции можно разделить на следующие категории:

- обычные агрегатные функции;
- статистические агрегатные функции;

- агрегатные функции, определяемые пользователем;
- аналитические агрегатные функции.

Первые три типа агрегатных функций рассматриваются в последующих разделах.

**Обычные агрегатные функции**

Язык Transact-SQL поддерживает следующие шесть агрегатных функций:

- MIN
- MAX
- SUM
- AVG
- COUNT
- COUNT_BIG

Все агрегатные функции выполняют вычисления над одним аргументом, который может быть или столбцом, или выражением. (Единственным исключением является вторая форма двух функций: COUNT и COUNT_BIG, а именно COUNT(*) и COUNT_BIG(*) соответственно. Результатом вычислений любой агрегатной функции является константное значение, отображаемое в отдельном столбце результата.

Агрегатные функции указываются в списке столбцов инструкции SELECT, который также может содержать предложение GROUP BY. Если в инструкции SELECT отсутствует предложение GROUP BY, а список столбцов выборки содержит, по крайней мере, одну агрегатную функцию, тогда он не должен содержать простых столбцов (кроме как столбцов, служащих аргументами агрегатной функции)..

**Пример 24. Неправильный синтаксис использования агрегатной функции**

```
USE sample;
SELECT emp_lname, MIN(emp_no)
    FROM employee;
```

Объясните почему, код в примере 24 неправильный

Аргументу агрегатной функции может предшествовать одно из двух возможных ключевых слов:

- ALL — указывает, что вычисления выполняются над всеми значениями столбца. Значение по умолчанию;
- DISTINCT — указывает, что для вычислений применяются только уникальные значения столбца.

**Агрегатные функции MIN *и MAX***

Агрегатные функции MIN и MAX вычисляют наименьшее и наибольшее значение столбца соответственно. Если запрос содержит предложение WHERE, функции MIN и MAX возвращают наименьшее и наибольшее значение строк, отвечающих указанным условиям.

**Пример 25. Определение наименьшего значения табельного номера (в этом и дальнейших примерах используйте конструкцию AS для именования агрегатных столбцов.**

Возвращенный в примере 25 результат не очень информативный. Например, неизвестно имя сотрудника, которому принадлежит этот номер. Но получить это имя обычным способом невозможно, потому что, как упоминалось ранее, явно указать столбец emp_name не разрешается. Для того чтобы вместе с наименьшим табельным номером сотрудника также получить и имя этого сотрудника, используется подзапрос. В примере 26 показано использование такого подзапроса, где вложенный запрос содержит инструкцию SELECT из предыдущего примера.

**Пример 26. Выборка наименьшего табельного номера сотрудника и соответствующего имени сотрудника**

```
USE sample;
SELECT emp_no, emp_lname
FROM employee
WHERE emp_no = SELECT MIN(emp_no)
                    FROM employee);
```

**Пример 27. Выборка табельного номера менеджера, введенного последним (дата) в таблицу** works_on. **Также используйте подзапрос.**

В качестве аргумента функции MIN и MAX также могут принимать строки и даты. В случае строкового аргумента значения сравниваются, используя фактический порядок сортировки. Для всех аргументов временных данных типа "дата" наименьшим значением столбца будет наиболее ранняя дата, а наибольшим — наиболее поздняя.

С функциями MIN и MAX можно применять ключевое слово DISTINCT. Перед применением агрегатных функций MIN и MAX из столбцов их аргументов исключаются все значения NULL.

**Агрегатная функция SUM**

Агрегатная функция SUM вычисляет общую сумму значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных.

**Пример 28. Вычисление общей суммы бюджетов всех проектов**

В примере 28 агрегатная функция группирует все значения бюджетов проектов и определяет их общую сумму. По этой причине запрос в примере 28 (как и все аналогичные запросы) содержит неявную функцию группирования.

Неявную функцию группирования из примера 28 можно указать явно, как это показано в примере 29.

**Пример 29. Явное указание группирующей функции в запросе с агрегатной функцией SUM**

```
SELECT SUM(budget) sum_of_budgets FROM project GROUP BY();
```

Рекомендуется использовать этот синтаксис в предложении group by, поскольку таким образом группирование определяется явно.

Использование параметра DISTINCT устраняет все повторяющиеся значения в столбце перед применением функции SUM. Аналогично удаляются все значения NULL перед применением этой агрегатной функции.

**Агрегатная функция AVG**

Агрегатная функция AVG возвращает среднее значение для всех значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных. Перед применением функции SUM все значения NULL удаляются из ее аргумента.

**Пример 30. Вычисление среднего значения бюджета для всех бюджетов, превышающих $100 ООО**

**Агрегатные функции *COUNT и COUNT_BIG***

Агрегатная функция COUNT имеет две разные формы:
COUNT([DISTINCT] *col_name)*
COUNT(*)

Первая форма функции подсчитывает количество значений в столбце *col_name*. Если в запросе используется ключевое слово distinct, перед применением функции count удаляются все повторяющиеся значения столбца. При подсчете количества значений столбца эта форма функции count не принимает во внимание значения NULL.
Использование первой формы агрегатной функции COUNT показано в примере 31.

**Пример 31. Подсчет количества разных должностей для каждого проекта**

Как можно видеть в результате выполнения запроса, представленного в примере 31, значения NULL функцией COUNT не принимались во внимание. (Сумма всех значений столбца должностей получилась равной 7, а не 11, как должно быть.)
Вторая форма функции COUNT, т. е. функция COUNT(*) подсчитывает количество строк в таблице. А если инструкция SELECT запроса с функцией COUNT(*) содержит предложение WHERE с условием, функция возвращает количество строк, удовлетворяющих указанному условию. В отличие от первого варианта функции COUNT вторая форма не игнорирует значения NULL, поскольку эта функция оперирует строками, а не столбцами. В примере 32 демонстрируется использование

функции COUNT(*) .

**Пример 32. Подсчет количества должностей во всех проектах**

Функция COUNT_BIG аналогична функции COUNT. Единственное различие между ними заключается в типе возвращаемого ими результата: функция COUNT_BIG всегда возвращает значения типа BIGINT, тогда как функция COUNT возвращает значения данных типа INTEGER.

**Статистические агрегатные функции**

Следующие функции составляют группу статистических агрегатных функций:

- VAR — вычисляет статистическую дисперсию всех значений, представленных в столбце или выражении;
- VARP — вычисляет статистическую дисперсию совокупности всех значений, представленных в столбце или выражении;
- STDEV — вычисляет среднеквадратическое отклонение (который рассчитывается как квадратный корень из соответствующей дисперсии) всех значений столбца или выражения;
- STDEVP — вычисляет среднеквадратическое отклонение совокупности всех значений столбца или выражения.

Примеры использования статистических агрегатных функций здесь не будем рассматривать.

**Агрегатные функции, определяемые пользователем**

Компонент Database Engine также поддерживает реализацию функций, определяемых пользователем. Эта возможность позволяет пользователям дополнить системные агрегатные функции функциями, которые они могут реализовывать и устанавливать самостоятельно. Эти функции представляют специальный класс определяемых пользователем функций. Примеры не будем рассматривать.

**Предложение *HAVING***

В предложении HAVING определяется условие, которое применяется к группе строк. Таким образом, это предложение имеет такой же смысл для групп строк, что и предложение WHERE для содержимого соответствующей таблицы. Синтаксис предложения HAVING следующий:
HAVING *condition*

Здесь параметр *condition* содержит агрегатные функции или константы.

Использование предложения HAVING совместно с агрегатной функцией COUNT(*) показать в примере 33.

**Пример 33. Выборка номеров проектов, в которых участвует меньше чем четыре сотрудника**

В примере 33 система посредством предложения GROUP BY группирует все строки по значениям столбца project_no. После этого подсчитывается количество строк в каждой группе и выбираются группы, содержащие менее четырех строк (три или меньше).
Предложение HAVING можно также использовать без агрегатных функций, как это показано в примере 34.

**Пример** 34. **Группирование строк таблицы** works_on **по должности и устранение тех должностей, которые не начинаются с буквы "М"**

Предложение HAVING можно также использовать без предложения GROUP BY, хотя это не является распространенной практикой. В таком случае все строки таблицы возвращаются в одной группе.

**Предложение *ORDER BY***

Предложение ORDER BY определяет порядок сортировки строк результирующего набора, возвращаемого запросом. Это предложение имеет следующий синтаксис:

ORDER BY {[ *col_name* | *col_number* [ASC | DESC]]} , ...

Порядок сортировки задается в параметре *coi_name*. Параметр *col_number* является альтернативным указателем порядка сортировки, который определяет столбцы по порядку их вхождения в список выборки инструкции select (1 — первый столбец, 2 — второй столбец и т. д.). Параметр ASC определяет сортировку в восходящем порядке, а параметр DESC — в нисходящем. По умолчанию применяется параметр ASC.

*ПРИМЕЧАНИЕ*

Имена столбцов в предложении order by не обязательно должны быть указаны в списке столбцов выборки. Но это не относится к запросам типа select distinct, т. к. в таких запросах имена столбцов, указанные в предложении order by, также должны быть указаны в списке столбцов выборки. Кроме этого, это предложение не может содержать имен столбцов из таблиц, не указанных в предложении from.

Как можно видеть по синтаксису предложения ORDER BY, сортировка результирующего набора может выполняться по нескольким столбцам. Такая сортировка показана в примере 35.

**Пример 35. Выборка номеров отделов и фамилий и имен сотрудников для сотрудников, чей табельный номер меньше чем 20000 а также с сортировкой по фамилии и имени.**

Столбцы в предложении ORDER BY можно указывать не по их именам, а по порядку в списке выборки. Соответственно, предложение в примере 35 можно переписать таким образом:

```
ORDER BY 2,1
```

Такой альтернативный способ указания столбцов по их позиции вместо имен применяется, если критерий упорядочивания содержит агрегатную функцию или вычисляемое выражение. (Другим способом является использование наименований столбцов, которые тогда отображаются в предложении ORDER BY.) Однако в предложении ORDER BY рекомендуется указывать столбцы по их именам, а не по номерам, чтобы упростить обновление запроса, если в списке выборки придется добавить или удалить столбцы. Указание столбцов в предложении ORDER BY по их номерам показать в примере 36.

**Пример 36. Для каждого номера проекта выбрать номер проекта и количество участвующих в нем сотрудников, упорядочив результат в убывающем порядке по числу сотрудников**

Язык Transact-SQL при сортировке в возрастающем порядке помещает значения NULL в начале списка, и в конце списка — при убывающем.

**Использование предложения *ORDER BY* для разбиения результатов на страницы**

Отображение результатов запроса на текущей странице можно или реализовать в пользовательском приложении, или же дать указание осуществить это серверу базы данных. В первом случае все строки базы данных отправляются приложению, чьей задачей является отобрать требуемые строки и отобразить их. Во втором случае, со стороны сервера выбираются и отображаются только строки, требуемые для текущей страницы. Как можно предположить, создание страниц на стороне сервера обычно обеспечивает лучшую производительность, т. к. клиенту отправляются только строки, необходимые для отображения.

Для поддержки создания страниц на стороне сервера в SQL Server 2012 вводится два новых предложения инструкции SELECT: OFFSET и FETCH. Применение этих двух предложений демонстрируется в примере 37. Здесь из базы данных (не нашей базы данных, поэтому делать не надо, только разобраться) извлекается идентификатор бизнеса, название должности и день рождения всех сотрудников женского пола с сортировкой результата по названию должности в возрастающем порядке. Результирующий набор строк разбивается на 10-строчные страницы и отображается третья страница.

**Пример 37. Применение предложений OFEEST И FETCH**

```
USE Adv12;
SELECT BusinessEntitylD, JobTitle, BirthDate
FROM HumanResources.Employee
```

```
WHERE Gender = 'F'
ORDER BY JobTitle
OFFSET 20 ROWS
FETCH NEXT 10 ROWS ONLY;
```

*ПРИМЕЧАНИЕ*

Есть и другие примеры использования предложения offset, которые рассмотрим позже.

В предложении OFFSET указывается количество строк результата, которые нужно пропустить в отображаемом результате. Это количество вычисляется после сортировки строк предложением ORDER BY. В предложении FETCH NEXT указывается количество удовлетворяющих условию WHERE и отсортированных строк, которое нужно возвратить. Параметром этого предложения может быть константа, выражение или результат другого запроса. Предложение FETCH NEXT аналогично предложению FETCH FIRST.

Основной целью при создании страниц на стороне сервера является возможность реализация общих страничных форм, используя переменные. Эту задачу можно выполнить посредством пакета SQL Server.

**Инструкция *SELECT* и свойство *IDENTITY***

Свойство IDENTITY позволяет определить значения для конкретного столбца таблицы в виде автоматически возрастающего счетчика. Это свойство могут иметь столбцы численного типа данных, такого как TINYINT, SMALLINT, INT и BIGINT. Для такого столбца таблицы компонент Database Engine автоматически создает последовательные значения, начиная с указанного стартового значения. Таким образом, свойство IDENTITY можно использовать для создания однозначных числовых значений для выбранного столбца.

Таблица может содержать только один столбец со свойством IDENTITY. Владелец таблицы имеет возможность указать начальное значение и шаг приращения, как это показано в примере 38.

**Пример 38. Использование свойства IDENTITY**

```
USE sample;
CREATE TABLE product
    (product_no INTEGER IDENTITY(10000, 1) NOT NULL,
     product_name CHAR(30) NOT NULL,
    price MONEY);
SELECT $identity
FROM product
    WHERE product_name = 'Soap';
```

Результатом этого запроса может быть следующее:

```
product_no
10005
```

В примере 38 сначала создается таблица product, содержащая столбец product_no со свойством IDENTITY. Значения в столбце product_no создаются автоматически системой, начиная с 10 000 и увеличиваясь с единичным шагом для каждого последующего значения: 10 000, 10 001, 10 002 и т. д.

Со свойством IDENTITY связаны некоторые системные функции и переменные. Например, в коде примера 38 используется переменная $identity. Как можно видеть по результатам выполнения этого кода, эта переменная автоматически ссылается на свойство IDENTITY.

Начальное значение и шаг приращения столбца со свойством IDENTITY можно узнать с помощью функций IDENT_SEED и IDENT_INCR соответственно. Применяются эти функции следующим образом:

```
SELECT IDENT_SEED('product'), IDENT_INCR('product)'
```

Как уже упоминалось, значения IDENTITY устанавливаются автоматически системой. Но пользователь

может указать явно свои значения для определенных строк, присвоив параметру IDENTITY_INSERT значение ON перед вставкой явного значения:

SET IDENTITY INSERT *table name* ON

*ПРИМЕЧАНИЕ*

Поскольку с помощью параметра identity insert для столбца со свойством identity можно установить любое значение, в том числе и повторяющееся, свойство identity обычно не обеспечивает принудительную уникальность значений столбца. Поэтому для принудительного обеспечения уникальности значений столбца следует применять ограничения unique или primary key.

При вставке значений в таблицу после присвоения параметру IDENTITY_INSERT значения ON система создает следующее значение столбца IDENTITY, увеличивая наибольшее текущее значение этого столбца.

**Оператор *CREATE SEQUENCE***

Применение свойства IDENTITY имеет несколько значительных недостатков, наиболее существенными из которых являются следующие:

- применение свойства ограничивается указанной таблицей;
- новое значение столбца нельзя получить иным способом, кроме как применив его;
- свойство IDENTITY можно указать только при создании столбца.

По этим причинам в SQL Server 2012 вводятся последовательности, которые обладают той же семантикой, что и свойство identity, но при этом не имеют ранее перечисленных недостатков. В данном контексте *последовательностью* называется функциональность базы данных, позволяющая указывать значения счетчика для разных объектов базы данных, таких как столбцы и переменные.

Последовательности создаются с помощью инструкции CREATE SEQUENCE. Инструкция CREATE SEQUENCE определена в стандарте SQL и поддерживается другими реляционными системами баз данных, такими как IBM DB2 и Oracle.

В пример 39 показано создание последовательности в SQL Server.

**Пример 39. Использование оператора CREATE SEQUENCE**

```
USE sample;
CREATE SEQUENCE dbo.Sequence1 AS INT
   START WITH 1 INCREMENT BY 5 MINVALUE 1 MAXVALUE 256 CYCLE;
```

В примере 39 значения последовательности Sequence1 создаются автоматически системой, начиная со значения 1 с шагом 5 для каждого последующего значения. Таким образом, в предложении START указывается начальное значение, а в предложении INCREMENT — шаг. (Шаг может быть как положительным, так и отрицательным.)

В следующих двух, необязательных, предложениях MINVALUE и MAXVALUE указываются минимальное и максимальное значение объекта последовательности. (Обратите внимание, что значение MINVALUE должно быть меньшим или равным начальному значению, а значение MAXVALUE не может быть большим, чем верхний предел типа данных, указанных для последовательности.) В предложении CYCLE указывается, что последовательность повторяется с начала по превышению максимального (или минимального для последовательности с отрицательным шагом) значения. По умолчанию это предложение имеет значение NO CYCLE, что означает, что превышение максимального или минимального значения последовательности вызывает исключение.

Основной особенностью последовательностей является их независимость от таблиц, т. е. их можно использовать с любыми объектами базы данных, такими как столбцы таблицы или переменные. (Это свойство положительно влияет на хранение и, соответственно, на производительность.

Определенную последовательность хранить не требуется; сохраняется только ее последнее значение.)
Новые значения последовательности создаются с помощью выражения NEXT VALUE FOR, применение которого показано в примере 40.

**Пример 40. Создание новых значений последовательности**

```
USE sample;
SELECT NEXT VALUE FOR dbo.Sequence1;
SELECT NEXT VALUE FOR dbo.Sequence1;
```

Результат выполнения этого запроса:

1

6

С помощью выражения NEXT VALUE FOR можно присвоить результат последовательности переменной или ячейке столбца. В примере 41 показано использование этого выражения для присвоения результатов столбцу.

**Пример 41. Использование выражения NEXT VALUE FOR**

```
USE sample;
CREATE TABLE dbo.tablel
    (columnl INT NOT NULL PRIMARY KEY,
    column2 CHAR(10));
    INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'A');
    INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'B');
```

В примере 41 сначала создается таблица table1, состоящая из двух столбцов. Далее, две инструкции insert вставляют в эту таблицу две строки. Первые две ячейки первого столбца будут иметь значения 11 и 16. (Эти два значения соответствуют значениям, созданным в примере 40.)

В примере 42 показано использование представления каталога sys.sequences для просмотра текущего значения последовательности, не используя его.

**Пример 42. Просмотр текущего значения последовательности**

```
USE sample;
SELECT current_value
FROM sys.sequences
WHERE name = ’Sequence1’;
```

*ПРИМЕЧАНИЕ*

Обычно выражение next value for применяется в инструкции insert, чтобы система вставляла созданные значения. Это выражение также можно использовать, как часть многострочного запроса с помощью предложения over.

Для изменения свойства существующей последовательности применяется инструкция ALTER SEQUENCE. Одно из наиболее важных применений этой инструкции связано с параметром RESTART WITH, который переустанавливает указанную последовательность. В примере 43 показано использование инструкции ALTER SEQUENCE для переустановки почти всех свойств последовательности Sequence1.

**Пример 43. Переустановка свойств последовательности**

```
USE sample;
ALTER SEQUENCE dbo.sequence1 RESTART WITH 100 INCREMENT BY 50 MINVALUE 50
    MAXVALUE 200 NO CYCLE;
```

Удаляется последовательность с помощью инструкции DROP SEQUENCE.

**Операторы работы с наборами**

Кроме операторов, рассмотренных ранее, язык Transact-SQL поддерживает еще три оператора работы с наборами:

- UNION;
- INTERSECT;
- EXCEPT.

**Оператор *UNION***

Оператор объединяет результаты двух или более запросов в один результирующий набор, в который входят все строки, принадлежащие всем запросам в объединении. Соответственно, результатом объединения двух таблиц является новая таблица, содержащая все строки, входящие в одну из исходных таблиц или в обе эти таблицы.

Общая форма оператора UNION выглядит таким образом:

*select_1* UNION [ALL] *select_2* {[UNION [ALL] select_3]}...

Параметры *seiect_l, seiect_2, ...* представляют собой инструкции select, которые создают объединение. Если используется параметр all, отображаются все строки, включая дубликаты. В операторе union параметр all имеет то же самое значение, что и в списке выбора select, но с одним отличием: для списка выбора select этот параметр применяется по умолчанию, а для оператора union его нужно указывать явно.

В своей исходной форме база данных sample не подходит для демонстрации применения оператора UNION. Поэтому в этом разделе создается новая таблица employee_enh, которая идентична существующей таблице employee, но имеет дополнительный столбец domicile. В этом столбце указывается место жительства сотрудников.

Новая таблица employee_enh выглядит следующим образом:

| emp no | emp fname | emp lname | dept no | domicile |
|---|---|---|---|---|
| 25348 | Matthew | Smith | d3 | San Antonio |
| 10102 | Ann | Jones | d3 | Houston |
| 18316 | John | Barrimore | d1 | San Antonio |
| 29346 | James | James | d2 | Seattle |
| 9031 | Elke | Hansel | d2 | Portland |
| 2581 | Elsa | Bertoni | d2 | Tacoma |
| 28559 | Sybil] | Moser | dl | Houston |

Создание таблицы employee_enh предоставляет нам удобный случай продемонстрировать использование предложения INTO в инструкции SELECT. Инструкция SELECT INTO выполняет две операции. Сначала создается новая таблица со столбцами, перечисленными в списке выбора SELECT. Потом строки исходной таблицы вставляются в новую таблицу. Имя новой таблицы указывается в предложении INTO, а имя таблицы-источника указывается в предложении FROM.

В примере 44 показано создание таблицы employee_enh.

**Пример 44. Применение инструкции SELECT INTO для создания новой таблицы**

```
USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no INTO employee_enh FROM employee;
ALTER TABLE employee_enh
    ADD domicile CHAR(25) NULL;
```

В примере 44 инструкция SELECT INTO создает таблицу employee_enh, вставляет в нее все строки из таблицы-источника employee, после чего инструкция ALTER TABLE добавляет в новую таблицу столбец domicile.

Но добавленный столбец domicile не содержит никаких значений. Значения в этот столбец можно вставить посредством среды Management Studio (см. *работу 1)* или же с помощью следующего кода:

```
USE sample;
UPDATE employee_enh SET domicile = 'San Antonio'
    WHERE emp_no = 25348;
UPDATE employee_enh SET domicile = 'Houston'
    WHERE emp_no = 10102;
UPDATE employee_enh SET domicile = 'San Antonio'
    WHERE emp_no = 18316;
UPDATE employee_enh SET domicile = 'Seattle'
    WHERE emp_no = 2 934 6;
UPDATE employee_enh SET domicile = 'Portland'
    WHERE emp_no = 9031;
UPDATE employee_enh SET domicile = 'Tacoma'
    WHERE emp_no = 2581;
UPDATE employee_enh SET domicile = 'Houston'
    WHERE emp_no = 28559;
```

Теперь мы готовы продемонстрировать использование инструкции UNION. В примере 45 показан запрос для создания соединения таблиц employee_enh и department, используя эту инструкцию.

**Пример 45. Объединение таблиц с помощью инструкции UNION**

```
USE sample;
SELECT domicile
    FROM employee_enh
UNION
SELECT location
    FROM department;
```

Объединять с помощью инструкции UNION можно только совместимые таблицы. Под совместимыми таблицами имеется в виду, что оба списка столбцов выборки должны содержать одинаковое число столбцов, а соответствующие столбцы должны иметь совместимые типы данных. (В отношении совместимости типы данных INT и SMALLINT не являются совместимыми.)

Результат объединения можно упорядочить, только используя предложение ORDER BY в последней инструкции SELECT, как это показано в примере 46. Предложения GROUP BY и HAVING можно применять с отдельными инструкциями SELECT, но не в самом объединении.

**Пример 46. Упорядочивание результатов объединения двух таблиц**

```
USE sample;
SELECT emp_no FROM employee WHERE dept_no = 'd1'
UNION
SELECT emp_no FROM works_on
    WHERE enter_date < '01.01.2007'
ORDER BY 1;
```

Запрос в примере 46 осуществляет выборку сотрудников, которые или работают в отделе d1, или начали работать над проектом до 1 января 2007 г.

*ПРИМЕЧАНИЕ*

Оператор union поддерживает параметр all. При использовании этого параметра дубликаты не удаляются из результирующего набора.

Вместо оператора UNION можно применить оператор OR, если все инструкции SELECT, соединенные одним или несколькими операторами UNION, ссылаются на одну и ту же таблицу. В таком случае набор инструкций SELECT заменяется одной инструкцией SELECT с набором операторов OR.

**Операторы *INTERSECT и EXCEPT***

Два других оператора для работы с наборами, INTERSECT и EXCEPT, определяют пересечение и разность соответственно. Под пересечением в данном контексте имеется набор строк, которые принадлежат к обеим таблицам. А разность двух таблиц определяется как все значения, которые принадлежат к первой таблице и не присутствуют во второй. В примере 47 показано использование оператора INTERSECT.

Для примеров 47 и 48 сформулируйте, что получаем в результате выполнения.

**Пример 47. Применение оператора INTERSECT**

```
USE sample;
SELECT emp_no
FROM employee
WHERE dept_no = 'dl'
INTERSECT
SELECT emp_no
FROM works_on
WHERE enter_date < '0l.0l.2008';
```

*ПРИМЕЧАНИЕ*

Язык Transact-SQL не поддерживает использование параметра ALL ни с оператором INTERSECT, ни с оператором EXCEPT.

**Пример 48. Применение оператора EXCEPT**

```
USE sample;
SELECT emp_no FROM employee
WHERE dept_no = 'd3'
EXCEPT
SELECT emp_no
FROM works_on
WHERE enter_date > '01.01.2008';
```

*ПРИМЕЧАНИЕ*

Следует помнить, что эти три оператора над множествами имеют разный приоритет выполнения: оператор intersect имеет наивысший приоритет, за ним следует оператор except, а оператор union имеет самый низкий приоритет. Невнимательность к приоритету выполнения при использовании нескольких разных операторов для работы с наборами может повлечь неожиданные результаты.

**Выражения *CASE***

В области прикладного программирования баз данных иногда требуется модифицировать представление данных. Например, людей можно подразделить, закодировав их по их социальной принадлежности, используя значения 1, 2 и 3, обозначив так мужчин, женщин и детей соответственно. Такой прием программирования может уменьшить время, необходимое для реализации программы. Выражение CASE языка Transact-SQL позволяет с легкостью реализовать такой тип кодировки.

*ПРИМЕЧАНИЕ*

В отличие от большинства языков программирования, CASE не является инструкцией, а выражением. Поэтому выражение case можно использовать почти везде, где язык Transact-SQL позволяет применять выражения.

Выражение CASE имеет две формы:

- простое выражение CASE;
- поисковое выражение CASE.

Синтаксис простого выражения CASE следующий:

CASE *expression_1*

{WHEN *expression_2* THEN *result_1*}…
[ELSE *result_n]*
END

Инструкция с простым выражением CASE сначала ищет в списке всех выражений в предложении WHEN первое выражение, совпадающее с выражением *expression_l,* после чего выполняет соответствующее предложение THEN. В случае отсутствия в списке WHEN совпадающего выражения, выполняется предложение ELSE.
Синтаксис поискового выражения CASE следующий:

```
CASE
   {WHEN condition_1 THEN result_1} ...
      [ELSE result_n]
END
```

В данном случае выполняется поиск первого отвечающего требованиям условия, после чего выполняется соответствующее предложение THEN. Если ни одно из условий не отвечает требованиям, выполняется предложение ELSE. Применение поискового выражения CASE показано в примере 49.

Для примеров 49 и 50 сформулируйте, что получаем в результате выполнения.

**Пример 49. Применение поискового выражения CASE**

```
USE sample;
SELECT project_name,
   CASE
      WHEN budget > 0 AND budget < 100000 THEN 1
      WHEN budget >= 100000 AND budget < 200000 THEN 2
      WHEN budget >= 200000 AND budget < 300000 THEN 3
      ELSE 4
    END
    budget_weight FROM project;
```

В примере 50 показан другой способ применения выражения CASE, где предложение WHEN содержит вложенные запросы, составляющие часть выражения.

**Пример 50. Применение выражения CASE С вложенными запросами**

```
USE sample;
SELECT project_name,
   CASE
      WHEN p1.budget < (SELECT AVG(p2.budget) FROM project p2) THEN 'below average'
      WHEN p1.budget = (SELECT AVG(p2.budget) FROM project p2) THEN 'on average'
      WHEN p1.budget > (SELECT AVG(p2.budget) FROM project p2) THEN 'above average'
    END budget category FROM project p1;
```

**Подзапросы**

Во всех рассмотренных ранее примерах значения столбцов сравниваются с выражением, константой или набором констант. Кроме таких возможностей сравнения язык Transact-SQL позволяет сравнивать значения столбца с результатом другой инструкции select. Такая конструкция, где предложение where инструкции select содержит одну или больше вложенных инструкций select, называется *подзапросом* (subquery). Первая инструкция select подзапроса называется *внешним запросом* (outer query), а внутренняя инструкция (или инструкции) select, используемая в сравнении, называется *вложенным запросом* (inner query). Первым выполняется вложенный запрос, а его результат передается внешнему запросу.

*ПРИМЕЧАНИЕ*

Вложенные запросы также могут содержать инструкции insert, update и delete, которые рассматриваются в следующей работе.

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. В этом разделе приводится несколько примеров независимых подзапросов. Связанные подзапросы рассматриваются далее в этой работе совместно с оператором соединения.

Независимый подзапрос может применяться со следующими операторами:
- операторами сравнения;
- оператором IN;
- операторами ANY и ALL;

**Подзапросы и операторы сравнения**

Использование оператора равенства (=) в независимом подзапросе показано в примере 51.

**Пример 51. Выборка имен и фамилий сотрудников отдела** Research

```
USE sample;
SELECT emp_fname, emp_lname FROM employee WHERE dept_no =
        SELECT dept_no
            FROM department
            WHERE dept_name = 'Research');
```

Сформулируйте, что делает этот запрос.

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. В последующем разделе рассматривается, 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 52.

**Пример 52. Получение всей информации о сотрудниках, чей отдел находится в Далласе**

```
USE sample;
SELECT *
    FROM employee
    WHERE dept_no IN (SELECT dept_no FROM department WHERE location = 'Dallas');
```

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.

Запрос с несколькими уровнями вложенности показан в примере 53.

**Пример 53. Выборка фамилий всех сотрудников, работающих над проектом Apollo**

```
USE sample;
SELECT emp_lname
FROM employee
WHERE emp_no IN (SELECT emp_no
                 FROM works_on
                 WHERE project_no IN (SELECT project_no
                                      FROM project
                                      WHERE project_name = 'Apollo'));
```

Сформулируйте, что делает этот запрос.

**Подзапросы и операторы ANY и *ALL***

Операторы ANY и ALL всегда используются в комбинации с одним из операторов сравнения. Оба оператора имеют одинаковый синтаксис:
*column_name operator* [ANY|ALL] *query*

Параметр *operator* обозначает оператор сравнения, а параметр *query* — вложенный запрос.
Оператор ANY возвращает значение TRUE (истина), если результат соответствующего вложенного запроса содержит хотя бы одну строку, удовлетворяющую условию сравнения. Ключевое слово SOME является синонимом ANY. Использование оператора ANY показано в примере 54.

**Пример 54. Выборка табельного номера, номера проекта и названия должности для сотрудников, которые не затратили большую часть своего времени при работе над одним из проектов**

```
USE sample;
SELECT DISTINCT emp_no, project_no, job
FROM works_on
WHERE enter_date > ANY (SELECT enter_date FROM works_on);
```

В примере 54 каждое значение столбца enter_date сравнивается со всеми другими значениями этого же столбца. Для всех дат этого столбца, за исключением самой ранней, сравнение возвращает значение true (истина), по крайней мере, один раз. Строка с самой ранней датой не попадает в результирующий набор, поскольку сравнение ее даты со всеми другими датами никогда не возвращает значение true (истина). Иными словами, выражение enter_date > ANY (SELECT enter_date FROM works_on) возвращает значение true, если в таблице works_on имеется *любое* количество строк (одна или больше), для которых значение столбца enter_date меньше, чем значение enter_date текущей строки. Этому условию удовлетворяют все значения столбца enter_date, за исключением наиболее раннего.

Оператор ALL возвращает значение TRUE, если вложенный запрос возвращает все значения, обрабатываемого им столбца.

*ПРИМЕЧАНИЕ*

> Настоятельно рекомендуется избегать использования операторов any и all. Любой запрос с применением этих операторов можно сформулировать лучшим образом посредством функции exists, которая рассматривается далее в этой работе (см. далее *разд. "Подзапросы и функция EXISTS"*). Кроме этого, семантическое значение оператора ANY можно легко принять за семантическое значение оператора all и наоборот.

**Временные таблицы**

*Временная таблица* — это объект базы данных, который хранится и управляется системой базы данных на временной основе. Временные таблицы могут быть локальными или глобальными. Локальные временные таблицы представлены физически, т. е. они хранятся в системной базе данных tempdb. Имена временных таблиц начинаются с префикса #, например #table_name.
Временная таблица принадлежит создавшему ее сеансу, и видима только этому сеансу. Временная

таблица удаляется по завершению создавшего ее сеанса. (Также локальная временная таблица, определенная в хранимой процедуре, удаляется по завершению выполнения этой процедуры.)

Глобальные временные таблицы видимы любому пользователю и любому соединению и удаляются после отключения от сервера базы данных всех обращающихся к ним пользователей. В отличие от локальных временных таблиц имена глобальных временных таблиц начинаются с префикса ##.

В примерах 55 и 56 показано создание временной таблицы, называющейся project_temp, используя две разные инструкции языка Transact-SQL.

**Пример 55. Создание временной таблицы посредством инструкции CREATE TABLE**

```
USE sample;
CREATE TABLE #project_temp
    (project_no CHAR(4) NOT NULL,
    project_name CHAR(25) NOT NULL);
```

**Пример 56. Создание временной таблицы посредством инструкции SELECT**

```
USE sample;
SELECT project_no, project_name INTO #project_temp1 FROM project;
```

Примеры 55 и 56 похожи в том, что в обоих создается локальная временная таблица #project_temp and #project_temp1. При этом таблица, созданная в примере 55 инструкцией CREATE TABLE, остается пустой, а созданная в примере инструкцией SELECT заполняется данными из таблицы project.

**Оператор соединения *JOIN***

В предшествующих разделах мы рассмотрели применение инструкции SELECT для выборки данных из одной таблицы базы данных. Если бы возможности языка Transact-SQL ограничивались поддержкой только таких простых инструкций SELECT, то присоединение в запросе двух или больше таблиц для выборки из них данных было бы невозможно. Следственно, все данные базы данных требовалось бы хранить в одной таблице. Хотя такой подход является вполне возможным, ему присущ один значительный недостаток — хранимые таким образом данные характеризуются высокой избыточностью.

Язык Transact-SQL устраняет этот недостаток, предоставляя для этого оператор соединения JOIN, который позволяет извлекать данные более чем из одной таблицы. Этот оператор, наверное, является наиболее важным оператором для реляционных систем баз данных, поскольку благодаря ему имеется возможность распределять данные по нескольким таблицам, обеспечивая, таким образом, важное свойство систем баз данных — отсутствие избыточности данных.

*ПРИМЕЧАНИЕ*

> Оператор UNION также позволяет выполнять запрос по нескольким таблицам. Но этот оператор позволяет присоединить несколько инструкций SELECT, тогда как оператор соединения join соединяет несколько таблиц с использованием всего лишь одной инструкции SELECT. Кроме этого, оператор union объединяет строки таблиц, в то время как оператор join соединяет столбцы (как мы увидим далее).

Оператор соединения также можно применять с базовыми таблицами и представлениями. В этой работе рассматривается соединение базовых таблиц, а соединения представлений будем рассматривать в следующих работах.

Оператор соединения JOIN имеет несколько разных форм. В этом разделе рассматриваются следующие основные формы этого оператора:

- естественное соединение;
- декартово произведение или перекрестное соединение;
- внешнее соединение;
- тета-соединение, самосоединение и полусоединение.

Прежде чем приступить к рассмотрению разных форм соединений, в этом разделе мы рассмотрим

разные варианты оператора соединения JOIN.

**Две синтаксические формы реализации соединений**

Для соединения таблиц можно использовать две разные синтаксические формы оператора соединения:

- явный синтаксис соединения (синтаксис соединения ANSI SQL:1992);
- неявный синтаксис соединения (синтаксис соединения "старого стиля").

Синтаксис соединения ANSI SQL:1992 был введен стандартом SQL92 и определяет операции соединения явно, т. е. используя соответствующее имя для каждого типа операции соединения. При явном объявлении соединения используются следующие ключевые слова:

- CROSS JOIN;
- [INNER] JOIN;
- LEFT [OUTER] JOIN;
- RIGHT [OUTER] JOIN;
- FULL [OUTER] JOIN.

Ключевое слово CROSS JOIN определяет декартово произведение двух таблиц. Ключевое слово INNER JOIN определяет естественное соединение двух таблиц, а LEFT OUTER JOIN и RIGHT OUTER JOIN определяют одноименные операции соединения. Наконец, ключевое слово FULL OUTER JOIN определяет соединение правого и левого внешнего соединений. Все эти операции соединения рассматриваются в последующих разделах.

Неявный синтаксис оператора соединения является синтаксисом "старого стиля", где каждая операция соединения определяется неявно посредством предложения where, используя так называемые *столбцы соединения* (см. вторую инструкцию в примере 57).

*ПРИМЕЧАНИЕ*

Для операций соединения рекомендуется использовать явный синтаксис, т. к. это повышает надежность запросов. По этой причине во всех примерах, связанных с операциями соединения, используются формы явного синтаксиса. Но в нескольких первых примерах также будет продемонстрирован и синтаксис "старого стиля".

**Естественное соединение**

Наилучшим способом объяснить *естественное соединение* можно посредством примера 57.

*ПРИМЕЧАНИЕ*

Термины "естественное соединение" (natural join) и "соединение по эквивалентности" (equi-join) часто используют синонимично, но между ними есть небольшое различие. Операция соединения по эквивалентности всегда имеет одну или несколько пар столбцов с идентичными значениями в каждой строке. Операция, которая устраняет такие столбцы из результатов операции соединения по эквивалентности, называется *естественным соединением.*

Запрос в примере 57 возвращает всю информацию обо всех сотрудниках: имя и фамилию, табельный номер, а также имя, номер и местонахождение отдела, при этом для номера отдела отображаются дубликаты столбцов из разных таблиц.

**Пример 57. Явный синтаксис оператора соединения**

```
USE sample;
SELECT employee.*, department.*
    FROM employee INNER JOIN department ON employee.dept_no = department.dept_no;
```

В примере 57 в инструкции select для выборки указаны все столбцы таблиц для сотрудника employee и отдела department. Предложение from инструкции SELECT определяет соединяемые таблицы, а

также явно указывает тип операции соединения — INNER JOIN. Предложение on является частью предложения from и указывает соединяемые столбцы в обеих таблицах. Выражение employee. dept_no = department .dept_no определяет условие соединения, а оба столбца условия называются *столбцами соединения.*

Эквивалентный запрос с применением неявного синтаксиса ("старого стиля") будет выглядеть следующим образом:

```
USE sample;
SELECT employee.*, department.*
    FROM employee, department
    WHERE employee.dept_no = department.dept_no;
```

Эта форма синтаксиса имеет два значительных различия с явной формой: список соединяемых таблиц указывается в предложении FROM, а соответствующее условие соединения указывается в предложении WHERE посредством соединяемых столбцов.

*ПРИМЕЧАНИЕ*

Настоятельно рекомендуется использовать подстановочный знак * в списке выбора select только при работе с SQL в интерактивном режиме, и избегать его применения в прикладных программах.

*ПРИМЕЧАНИЕ*

Система базы данных не может определить логическое значение столбца. Например, она не может определить, что между столбцами номера проекта и табельного номера сотрудника нет ничего общего, хотя оба они имеют целочисленный тип данных. Поэтому система базы данных может только проверить тип данных и длину строк. Компонент Database Engine требует, что соединяемые столбцы имели совместимые типы данных, например int и smallint.

База данных sample содержит три пары столбцов, где каждый столбец в паре имеет одинаковое логическое значение (а также одинаковые имена). Таблицы employee и department можно соединить по столбцам employee.dept_no и department.dept_no. Столбцами соединения таблиц employee и works_on являются столбцы employee.emp_no и works_on.emp_no. Наконец, таблицы project и works_on можно соединить по столбцам project.project_no и works_on.project_no.
Имена столбцов в инструкции select можно уточнить. В данном контексте под уточнением имеется в виду, что во избежание неопределенности относительно того, какой таблице принадлежит столбец, в имя столбца включается имя его таблицы (или псевдоним таблицы), отделенное точкой: *table_name.column_name (имя_ таблицы.имя_столбца).*

В большинстве инструкций select столбцы не требуют уточнения, хотя обычно рекомендуется применять уточнение столбцов с целью улучшения понимания кода. Если же имена столбцов в инструкции select неоднозначны (как, например, столбцы dept_no в таблицах employee и department в примере 57) использование уточненных имен столбцов является *обязательным.*
В инструкции SELECT с операцией соединения, кроме условия соединения предложение WHERE может содержать и другие условия, как это показано в примере 58.

**Пример 58. Выборка полной информации сотрудников, работающих над проектом Gemini**
Явный синтаксис соединения:

```
USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on JOIN project ON project.project_no = works_on.project_no
WHERE project_name = 'Gemini';
```

Неявный синтаксис соединения ("старый стиль"):

```
USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on, project
    WHERE project.project_no = works_on.project_no AND project_name = 'Gemini';
```

В дальнейшем во всех примерах будет использоваться только явный синтаксис соединения.

**Пример 59. Выборка номера отдела сотрудников, приступивших к работе над проектами 15 октября 2007 г.**

**Соединение более чем двух таблиц**

Теоретически количество таблиц, которые можно соединить в инструкции SELECT, неограниченно. (Но одно условие соединения совмещает только две таблицы!) Однако для компонента Database Engine количество соединяемых таблиц в инструкции SELECT ограничено 64 таблицами.

В примере 60 показано соединение трех таблиц базы данных sample.

**Пример 60. Выборка имен и фамилий всех аналитиков** (job = 'analyst[1]), **чей отдел находится в Сиэтле** (location = 'Seattle')

Результат запроса, приведенного в примере 60, можно получить только в том случае, если соединить, по крайней мере, три таблицы: works_on, employee и department. Эти таблицы можно соединить, используя две пары столбцов соединения:

(works_on.emp_no, employee.emp_no)
(employee.dept_no, department.dept_no)

В примере 61 показано соединение четырех таблиц базы данных sample.

**Пример 61. Выборка наименований проектов (с удалением избыточных дубликатов), в которых участвуют сотрудники бухгалтерии (отдел** Accounting)

Обратите внимание, что для осуществления естественного соединения трех таблиц используется два условия соединения, каждое из которых соединяет по две таблицы. А при соединении четырех таблиц таких условий соединения требуется три. В общем, чтобы избежать получения декартового продукта при соединении *n* таблиц, требуется применять *n* - 1 условий соединения. Конечно же, допустимо использование более чем *n* - 1 условий соединения, а также других условий, для того чтобы еще больше уменьшить количество элементов в результирующем наборе данных.

**Декартово произведение**

В предшествующем разделе мы рассмотрели возможный способ создания естественного соединения. На первом шаге этого процесса каждая строка таблицы employee соединяется с каждой строкой таблицы department. Эта операция называется *декартовым произведением* (cartesian product). Запрос для создания соединения таблиц employee и department, используя декартово произведение, показан в примере 62.

**Пример 62. Соединение таблиц декартовым произведением**

```
USE sample;
SELECT employee.*, department.*
    FROM employee CROSS JOIN department;
```

Декартово произведение соединяет каждую строку первой таблицы с каждой строкой второй. В общем, результатом декартового произведения первой таблицы с *n* строками и второй таблицы с *m* строками будет таблица с *n x m* строками. Таким образом, результирующий набор запроса в примере 62 имеет 7 x 3 = 21 строку.

**Внешнее соединение**

В предшествующих примерах естественного соединения, результирующий набор содержал только те строки с одной таблицы, для которых имелись соответствующие строки в другой таблице. Но иногда кроме совпадающих строк бывает необходимым извлечь из одной или обеих таблиц строки без совпадений. Такая операция называется *внешним соединением* (outer join).

В примерах 63 и 64 показано различие между естественным соединением и соответствующим ему

внешним соединением. (Во всех примерах в этом разделе используется таблица employee_enh.)

**Пример 63. Выборка всей информации для сотрудников, которые проживают и работают в одном и том же городе**

```
USE sample;
SELECT employee_enh.*, department.location FROM employee_enh JOIN department ON domicile =
location;
```

Результат выполнения этого запроса:

| emp_no | emp_fname | emp_lname | dept_no | domicile | location |
|---|---|---|---|---|---|
| 29346 | James | James | d2 | Seattle | Seattle |

В примере 63 получение требуемых строк осуществляется посредством естественного соединения. Если бы в этот результат потребовалось включить сотрудников, проживающих в других местах, то нужно было применить левое внешнее соединение. Данное внешнее соединение называется *левым* потому, что оно возвращает все строки из таблицы с *левой* стороны оператора сравнения, независимо от того, имеются ли совпадающие строки в таблице с правой стороны. Иными словами, данное внешнее соединение возвратит строку с левой таблицы, даже если для нее нет совпадения в правой таблице, со значением null соответствующего столбца для всех строк с несовпадающим значением столбца другой, правой, таблицы (см. пример 64). Для выполнения операции левого внешнего соединения компонент Database Engine использует оператор LEFT OUTER JOIN.

Операция *правого* внешнего соединения аналогична левому, но возвращаются все строки таблицы с *правой* части выражения. Для выполнения операции правого внешнего соединения компонент Database Engine использует оператор RIGHT OUTER JOIN.

**Пример 64. Выборка сотрудников (с включением полной информации) для таких городов, в которых сотрудники или только проживают, или проживают и работают**

```
USE sample;
SELECT employee_enh.*, department.location
    FROM employee_enh LEFT OUTER JOIN department ON domicile = location;
```

Проанализируйте результаты выполнения этих запросов, в чем разность этих видов соединения

**Пример 65. Выборка отделов (с включением полной информации о них) для таких городов, в которых сотрудники или только проживают, или проживают и работают**

```
USE sample;
SELECT employee_enh.domicile, department.*
    FROM employee_enh RIGHT OUTER JOIN department ON domicile =location;
```

Кроме левого и правого внешнего соединения, также существует полное внешнее соединение, которое является объединением левого и правого внешних соединений. Иными словами, результирующий набор такого соединения состоит из всех строк обеих таблиц. Если для строки одной из таблиц нет соответствующей строки в другой таблице, всем ячейкам строки второй таблицы присваивается значение NULL. Для выполнения операции правого внешнего соединения используется оператор FULL OUTER JOIN.

**Пример 66. Эмуляция левого внешнего соединения**

```
USE sample;
SELECT employee_enh.*, department.location FROM employee_enh JOIN department ON domicile =
location UNION
SELECT employee_enh.*, 'NULL'
    FROM employee_enh WHERE NOT EXISTS (SELECT *
```

```
FROM department
WHERE location = domicile);
```

Первая инструкция SELECT объединения определяет естественное соединение таблиц employee_enh и department по столбцам соединения domicile и location. Эта инструкция возвращает все города для всех сотрудников, в которых сотрудники и проживают и работают. Дополнительно, вторая инструкция SELECT объединения возвращает все строки таблицы employee_enh, которые не отвечают условию в естественном соединении.

**Другие формы операций соединения**

В предшествующих разделах мы рассмотрели наиболее важные формы соединения. Но существуют и другие формы этой операции, которые мы рассмотрим в этом разделе. А именно:

- тета-соединение;
- самосоединение;
- полусоединение.

Эти формы рассматриваются в следующих подразделах.

**Тета-соединение**

Условие сравнения столбцов соединения не обязательно должно быть равенством, но может быть любым другим сравнением. Соединение, в котором используется общее условие сравнения столбцов соединения, называется *тета-соединением.* В примере 67 показана операция тета-соединения, в которой используется условие "меньше чем". Данный запрос возвращает все комбинации информации о сотрудниках и отделах для тех случаев, когда место проживания сотрудника по алфавиту идет перед месторасположением любого отдела, в котором работает этот служащий.

**Пример 67. Операция тета-соединения с условием "меньше чем"**

```
USE sample;
SELECT emp_fname, emp_lname, domicile, location FROM employee_enh JOIN department ON
    domicile < location;
```

В примере 67 сравниваются соответствующие значения столбцов domicile и location. В каждой строке результата значение столбца domicile сравнивается в алфавитном порядке с соответствующим значением столбца location.

**Самосоединение, или соединение таблицы самой с собой**

Кроме соединения двух или больше разных таблиц, операцию естественного соединения можно применить к одной таблице. В данной операции таблица соединяется сама с собой, при этом один столбец таблицы сравнивается сам с собой. Сравнивание столбца с самим собой означает, что в предложении FROM инструкции SELECT имя таблицы употребляется дважды. Поэтому необходимо иметь возможность ссылаться на имя одной и той же таблицы дважды. Это можно осуществить, используя, по крайней мере, один псевдоним. То же самое относится и к именам столбцов в условии соединения в инструкции SELECT. Для того чтобы различить столбцы с одинаковыми именами, необходимо использовать уточненные имена. Соединение таблицы с самой собой демонстрируется в примере 68.

**Пример 68. Выборка всех отделов (с полной информацией), расположенных в том же самом месте, как и, по крайней мере, один другой отдел**

```
USE sample;
SELECT tl.dept_no, tl.dept_name, tl.location
FROM department tl JOIN department t2 ON tl.location = t2.location
WHERE tl.dept_no <> t2.dept_no;
```

В примере 68 предложение from содержит два псевдонима для таблицы department: tl и t2. Первое условие в предложении where определят столбцы соединения, а второе — удаляет ненужные дубликаты, обеспечивая сравнение каждого отдела с *другими* отделами.

**Полусоединение**

Полусоединение похоже на естественное соединение, но возвращает только набор всех строк из одной таблицы, для которой в другой таблице есть одно или несколько совпадений. Использование полусоединения показано в примере 69.

**Пример 69. Использование полусоединения**

```
USE sample;
SELECT emp_no, emp_lname, e.dept_no FROM employee e JOIN department d
      ON e.dept_no = d.dept_no WHERE location = 'Dallas';
```

Как можно видеть в примере 69, список выбора select в полусоединении содержит только столбцы из таблицы employee. Это и есть характерной особенностью операции полусоединения. Эта операция обычно применяется в распределенной обработке запросов, чтобы свести к минимуму объем передаваемых данных. Компонент Database Engine использует операцию полусоединения для реализации функциональности, называющейся *соединением типа "звезда"*.

**Связанные подзапросы**

Подзапрос называется *связанным* (correlated), если любые значения вложенного запроса зависят от внешнего запроса. В примере 70 показано использование связанного подзапроса.

**Пример 70. Выборка фамилий всех сотрудников, работающих над проектом p3**

```
USE sample;
SELECT emp_lname
FROM employee
WHERE 'p3' IN (SELECT project_no
                    FROM works_on
                    WHERE works_on.emp_no = employee.emp_no);
```

В примере 70 вложенный запрос должен логически выполниться несколько раз, поскольку он содержит столбец emp_no, который принадлежит таблице employee во внешнем запросе, и значение столбца emp_no изменяется каждый раз, когда проверяется другая строка таблицы employee во внешнем запросе.

Давайте проследим, как система может выполнять запрос в примере 70. Сначала система выбирает первую строку таблицы employee (для внешнего запроса) и сравнивает табельный номер сотрудника в этом столбце (25348) со значениями столбца works_on.emp_no вложенного запроса. Поскольку для этого сотрудника имеется только одно значение project_no равное p2, вложенный запрос возвращает значение p2. Это единственное значение результирующего набора вложенного запроса не равно значению p3 внешнего запроса, условие внешнего запроса (WHERE 'p3' IN... ) не удовлетворяется и, следовательно, внешний запрос не возвращает никаких строк для этого сотрудника. Далее система берет следующую строку таблицы employee и снова сравнивает номера сотрудников в обеих таблицах. Для этой строки в таблице works_on есть две строки, для которых значение project_no равно p1 и p3 соответственно. Следовательно, вложенный запрос возвращает результат p1 и p3. Значение одного из элементов этого результирующего набора равно константе $p^{3,}$ поэтому условие удовлетворяется, и отображается соответствующее значение второй строки столбца emp_lname (Jones). Такой же обработке подвергаются все остальные строки таблицы employee, и в конечном результате возвращается набор из трех строк.

В следующем разделе приводятся дополнительные примеры по связанным подзапросам.

**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. Рассмотрим работу этой функции на нескольких примерах, начиная с примера 71.

**Пример 71. Выборка фамилий всех сотрудников, работающих над проектом p1**

```
USE sample;
SELECT emp_lname
FROM employee
WHERE EXISTS (SELECT *
                    FROM works_on
                    WHERE employee.emp_no = works_on.emp_no AND project_no = ’pi’);
```

Вложенный запрос функции EXISTS почти всегда зависит от переменной с внешнего запроса. Поэтому функция EXISTS обычно определяет связанный подзапрос.
Давайте проследим, как Database Engine может обрабатывать запрос в примере 71. Сначала внешний запрос рассматривает первую строку таблицы employee (сотрудник Smith). Далее функция EXISTS определяет, есть ли в таблице works_on строки, чьи номера сотрудников совпадают с номером сотрудника в текущей строке во внешнем запросе и чей project_no равен pi. Поскольку сотрудник Smith не работает над проектом pi, вложенный запрос возвращает пустой набор, вследствие чего функция EXISTS возвращает значение FALSE. Таким образом, сотрудник Smith не включается в конечный результирующий набор. Этому процессу подвергаются все строки таблицы employee, после чего выводится конечный результирующий набор.
В примере 72 показано использование функции NOT EXISTS.

**Пример 72. Выборка фамилий сотрудников, чей отдел не расположен в Сиэтле**

```
USE sample; SELECT emp_lname
FROM employee
WHERE NOT EXISTS (SELECT *
                        FROM department
                        WHERE employee.dept_no = department.dept_no AND location = 'Seattle');
```

Список выбора инструкции SELECT во внешнем запросе с функцией EXISTS не обязательно должен быть в форме SELECT *, как в предыдущем примере. Можно использовать альтернативную форму SELECT colum_list, где column_list представляет список из одного или нескольких столбцов таблицы. Обе формы равнозначны, потому что функция EXIST только проверяет на наличие (или отсутствие) строк в результирующем наборе. По этой причине в данном случае правильнее использовать форму SELECT *.

**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 73.

**Пример 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 соединений**

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 74.

**Пример 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 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.

**Табличные выражения**

*Табличными выражениями* называются подзапросы, которые используются там, где ожидается наличие таблицы. Существует два типа табличных выражений:

- производные таблицы;
- обобщенные табличные выражения.

Эти две формы табличных выражений рассматриваются в следующих подразделах.

**Производные таблицы**

*Производная таблица* (derived table) — это табличное выражение, входящее в предложение from запроса. Производные таблицы можно применять в тех случаях, когда использование псевдонимов столбцов не представляется возможным, поскольку транслятор SQL обрабатывает другое предложение до того, как псевдоним станет известным. В примере 75 показана попытка использовать псевдоним столбца в ситуации, когда другое предложение обрабатывается до того, как станет известным псевдоним.

**Пример 75. Неправильная попытка выбрать все группы месяцев из значений столбца** enter date **таблицы** works on

```
USE sample;
SELECT MONTH(enter_date) as enter_month FROM works_on GROUP BY enter_month;
```

Попытка выполнить этот запрос выдаст сообщение об ошибке. Почему?

Эту проблему можно решить, используя производную таблицу, содержащую предшествующий запрос (без предложения GROUP BY), поскольку предложение FROM исполняется перед предложением GROUP BY (пример 76).

**Пример 76. Правильная конструкция запроса из примера 75**

```
USE sample;
SELECTenter_month
    FROM (SELECT MONTH(enter_date) as enter_month FROM works_on) AS m GROUP BY
    enter_month;
```

Обычно табличное выражение можно разместить в любом месте инструкции SELECT, где может

появиться имя таблицы. (Результатом табличного выражения всегда является таблица или, в особых случаях, выражение.) В примере 77 показывается использование табличного выражения в списке выбора инструкции
SELECT.

**Пример 77. Использование табличного выражения**

```
USE sample;
SELECT w.job, (SELECT e.emp_lname
        FROM employee e WHERE e.emp_no = w.emp_no) AS name FROM works_on w
    WHERE w.job IN('Manager', 'Analyst');
```

Что делает этот запрос?

**Обобщенные табличные выражения**

*Обобщенным табличным выражением* (ОТВ) (Common Table Expression — сокращенно CTE) называется именованное табличное выражение, поддерживаемое языком Transact-SQL. Обобщенные табличные выражения используются в следующих двух типах запросов:

- нерекурсивных;
- рекурсивных.

**ОТВ и нерекурсивные запросы**

Нерекурсивную форму ОТВ можно использовать в качестве альтернативы производным таблицам и представлениям. Обычно ОТВ определяется посредством предложения WITH и дополнительного запроса, который ссылается на имя, используемое в предложении WITH (см. пример 79).

*ПРИМЕЧАНИЕ*

В языке Transact-SQL значение ключевого слова with неоднозначно. Чтобы избежать неопределенности, инструкцию, предшествующую оператору with, следует завершать точкой с запятой (;).

В примерах 78 и 79 показано использование ОТВ в нерекурсивных запросах. В примере 78 используется стандартное решение, в примере 79 та же задача решается посредством нерекурсивного запроса.

**Пример 78. Стандартное решение**

```
USE AdventureWorks;
SELECT SalesOrderlD
    FROM Sales.SalesOrderHeader
    WHERE TotalDue > (SELECT AVG(TotalDue)
                        FROM Sales.SalesOrderHeader
                        WHERE YEAR(OrderDate) = ’2002’)
    AND Freight > (SELECT AVG(TotalDue)
                    FROM Sales.SalesOrderHeader
                    WHERE YEAR(OrderDate) = ’2002’)/2.5;
```

Что делает этот запрос

Но это решение несколько сложно, поскольку требует создания представления, а потом его удаления после окончания выполнения запроса. Лучшим подходом будет создать ОТВ. В примере 79 показывается использование нерекурсивного ОТВ, которое сокращает определение запроса, приведенного в примере 78.

**Пример 79. Использование ОТВ для сокращения объема запроса**

```
USE AdventureWorks;
```

```
WITH price_calc(year_2002) AS (SELECT AVG(TotalDue)
      FROM Sales.SalesOrderHeader WHERE YEAR(OrderDate) = '2002')
SELECT SalesOrderlD
   FROM Sales.SalesOrderHeader
   WHERE TotalDue > (SELECT year_2002 FROM price_calc)
AND Freight > (SELECT year_2002 FROM price_calc)/2.5;
```

Синтаксис предложения WITH в нерекурсивных запросах имеет следующий вид:
WITH cte_name (column_list) AS (inner_query) outer_query

Параметр cte_name представляет имя ОТВ, которое определяет результирующую таблицу, а параметр column_list — список столбцов табличного выражения.

(В примере 79 ОТВ называется price_calc и имеет один столбец — year_2002.) Параметр inner_query представляет инструкцию select, которая определяет результирующий набор соответствующего табличного выражения. После этого определенное табличное выражение можно использовать во внешнем запросе outer_query. (Внешний запрос в примере 79 использует ОТВ price_calc и ее столбец year_2002, чтобы упростить употребляющийся дважды вложенный запрос.)

**ОТВ и рекурсивные запросы**

Посредством ОТВ можно реализовывать рекурсии, поскольку ОТВ могут содержать ссылки на самих себя. Этот материал пока слишком сложен, поэтому рассматривать его не будем.

**Резюме**

В этой работе мы рассмотрели все возможности инструкции SELECT, касающиеся выборки данных из одной или нескольких таблиц. Каждая инструкция SELECT, которая извлекает данные из таблицы, должна содержать как минимум список столбцов и предложение FROM. Предложение FROM указывает таблицу или таблицы, из которых выбираются данные. Наиболее важным необязательным предложением является предложение WHERE, содержащее одно или несколько условий, объединяемых логическими операторами AND, OR или NOT. Таким образом, условия в предложении WHERE накладывают ограничения на выбираемые строки.

**Упражнения**

**Упражнение 1**

Выполните выборку всех строк из таблицы works_on.

**Упражнение 2**

Выполните выборку табельных номеров всех сотрудников с должностью клерк (clerk).

**Упражнение 3**

Выполните выборку табельных номеров всех сотрудников, которые работают над проектом p2 и чей табельный номер меньше, чем 10 000. Решите эту задачу, используя два различных, но эквивалентных запроса с инструкцией select.

**Упражнение 4**

Выполните выборку табельных номеров всех сотрудников, которые не приступили к работе над проектом в 2007 г.

**Упражнение 5**

Выполните выборку табельных номеров всех сотрудников проекта p1 с ведущими должностями (т. е. аналитик analyst и менеджер manager).

**Упражнение 6**

Выполните выборку всех сотрудников проекта p2, чья должность еще не определена.

**Упражнение 7**

Выполните выборку табельных номеров и фамилий сотрудников, чьи имена содержат две буквы "t".

**Упражнение 8**

Выполните выборку табельных номеров и имен всех сотрудников, у которых вторая буква фамилии "o" или "a" (буквы английские) и последние буквы фамилии "es".

**Упражнение 9**

Выполните выборку табельных номеров сотрудников, чьи отделы расположены в Сиэтле (Seattle).

**Упражнение 10**

Выполните выборку фамилий и имен сотрудников, которые приступили к работе над проектами 4 января 2007 г.

**Упражнение 11**

Сгруппируйте все отделы по их местонахождению.

**Упражнение 12**

Объясните разницу между предложениями DISTINCT и GROUP BY.

**Упражнение 13**

Как предложение GROUP BY обрабатывает значения NULL? Подобна ли эта обработка обычной обработке этих значений?

**Упражнение 14**

Объясните разницу между агрегатными функциями COUNT(*) и COUNT(column) .

**Упражнение 15**

Выполните выборку наибольшего табельного номера сотрудника.

**Упражнение 16**

Выполните выборку должностей, занимаемых больше, чем двумя сотрудниками. **Упражнение 17**

Выполните выборку табельных номеров сотрудников, которые или имеют должность клерк clerk, или работают в отделе d3.

**Упражнение 18**

Объясните, почему следующий запрос неправильный.

```
SELECT project name FROM project WHERE project_no =
    (SELECT project_no FROM works_on WHERE Job = 'Clerk')
```

Исправьте синтаксис запроса.

**Упражнение 19**

Каково практическое применение временных таблиц?

**Упражнение 20**

Объясните разницу между глобальными и локальными временными таблицами.

*ПРИМЕЧАНИЕ*

В решениях всех последующих упражнений по операции соединения используйте явный синтаксис.

**Упражнение 21**

Создайте следующие соединения таблиц project и works_on, выполнив:

- естественное соединение;
- декартово произведение.

**Упражнение 22**

Сколько условий соединения необходимо для соединения в запросе ***n*** таблиц?

**Упражнение 23**

Выполните выборку табельного номера сотрудника и должности для всех сотрудников, работающих над проектом Gemini.

**Упражнение 24**

Выполните выборку имен и фамилий всех сотрудников, работающих в отделе

Research или Accounting.

**Упражнение 25**

Выполните выборку всех дат начала работы для всех клерков (clerk), работающих в отделе d1.

**Упражнение 26**

Выполните выборку всех проектов, над которыми работают двое или больше сотрудников с должностью клерк (clerk).

**Упражнение 27**

Выполните выборку имен и фамилий сотрудников, которые имеют должность менеджер (manager) и работают над проектом Mercury.

**Упражнение 28**

Выполните выборку имен и фамилий всех сотрудников, которые начали работать над проектом одновременно, по крайней мере, еще с одним другим сотрудником.

**Упражнение 29**

Выполните выборку табельных номеров сотрудников, которые живут в том же городе, где находится их отдел. (Используйте расширенную таблицу enployee_ enh базы данных sample.)

**Упражнение 30**

Выполните выборку табельных номеров всех сотрудников, работающих в отделе маркетинга marketing. Создайте два равнозначных запроса, используя:

- оператор соединения;
- связанный подзапрос.