БАЗЫ ДАННЫХ И ЯЗЫК SQL

Каждая область человеческой деятельности нуждается в хранении и обработке сопутствующей информации. Например, библиотека хранит сведения о книжных фондах и параллельно ведет списки читателей. Бухгалтерия оперирует счетами и производит различные операции над ними. Склад ведет учет наличия товаров и отслеживает их движение.

Под базой данных (БД) понимается некий организованный набор информации. В качестве примера простейшей БД можно привести список товаров, каждый из которых обладает набором стандартных характеристик (наименование, единица измерения, количество, цена и т.д.):

№	Наименование	Ед. изм.	Цена	Кол-во
1	Кирпич	штука	255	10000
2	Краска	литр	580	670
3	Шифер	лист	130	500
…	…	…	…	…
10001	Гвоздь	штука	20	8000
10002	Кабель	метр	100	200

Способов организации баз данных великое множество. В недавнем прошлом бумажные листки со списками товаров держали подшитыми в отдельную папку либо раскладывали по ящикам стола в соответствии с некоторыми критериями.

В наше время для подобных целей повсеместно используются компьютеры, что значительно облегчает процесс создания базы данных и дальнейшей работы с ней.

Существуют специальные компьютерные программы, позволяющие полностью автоматизировать процесс хранения, получения и модификации данных любого типа и назначения.

В общем случае они называются системами управления базами данных (СУБД) и состоят из языковых и программных средств, предназначенных для создания и эксплуатации баз данных.

Базовые свойства любой СУБД включают в себя:

- скорость (время доступа к данным);
- разграничение доступа (отдельные категории пользователей имеют доступ только к определенным категориям данных);
- гибкость (возможность формировать и обрабатывать сложные запросы к данным);
- целостность (средства поддержки согласованности взаимосвязанных данных при их изменении);
- отказоустойчивость (средства архивирования и восстановления данных на случай выхода из строя оборудования либо ошибок в программном обеспечении).

Базовые функции СУБД:

- интерпретация запросов пользователя, сформированных на специальном языке (обычно – SQL);
- определение данных (создание и поддержка специальных объектов, хранящих поступающие от пользователя данные, ведение внутреннего реестра объектов и их характеристик – так называемого словаря данных);
- исполнение запросов по выбору, изменению или удалению существующих данных или добавлению новых данных;
- безопасность (контроль запросов пользователя на предмет попытки нарушения правил безопасности и целостности, задаваемых при определении данных);
- производительность (поддержка специальных структур для обеспечения максимально быстрого поиска нужных данных);
- архивирование и восстановление данных.

(Понятие базы данных База данных)

Реляционные СУБД

Модель данных в реляционных СУБД

Прежде чем сохранять какие-либо данные в СУБД, необходимо описать модель этих данных.

По типу модели данных СУБД делятся на ***сетевые, иерархические* и *реляционные и др.***

СУБД реляционного типа являются наиболее распространенными и часто используемыми. В качестве примеров можно привести Oracle и Microsoft SQL Server.

Теория реляционных СУБД была разработана Коддом из IBM в 60-х годах XX века и базируется на математической теории отношений. Важнейшие понятия этой теории – таблица, строка, столбец, отношение, первичный и вторичный ключ.

Реляционная СУБД представляет собой совокупность именованных двумерных таблиц данных, логически связанных (находящихся в отношении) между собой.

Таблицы состоят из строк и именованных столбцов, строки представляют собой экземпляры информационного объекта, столбцы – атрибуты объекта.

В рассмотренном ранее примере таблица (назовем ее “Склад”) состоит из информационных объектов-строк, отдельная строка содержит сведения об отдельном товаре. Каждый товар характеризуется некими параметрами-атрибутами (“Наименование”, “Цена” и т.д.).

Строки иногда называют записями, а столбцы – полями записи.

Таким образом, в реляционной модели все данные представлены для пользователя в виде таблиц значений данных, и все операции над базой сводятся к манипулированию таблицами.

Связи между отдельными таблицами в реляционной модели в явном виде могут не описываться. Они устанавливаются пользователем при написании запроса на выборку данных и представляют собой условия равенства значений соответствующих полей.

Пример логически взаимосвязанных таблиц:

Сотрудники

Табельный №	**Фамилия**	**Должность**	**№ отдела**
1	Иванов	Начальник	15
2	Петров	Инженер	15
3	Сидоров	Менеджер	10

Отделы

№ отдела	**Наименование**
15	Производственный отдел
10	Отдел продаж

В реляционной модели при логическом связывании таблиц применяется следующая терминология:

- Первичный ключ (или главный ключ, primary key, **PK**). Представляет собой столбец или совокупность столбцов, значения которых однозначно идентифицируют строки. В данном примере первичным ключом в таблице “Сотрудники” является столбец “Табельный

№”, ибо в одной организации не бывает сотрудников с одинаковыми табельными номерами. Очевидно, что в таблице “Отделы” первичным ключом является столбец, содержащий номер отдела;

- Вторичный (или внешний ключ, foreign key, **FK**). Столбец или совокупность столбцов, которые в данной таблице не являются первичными ключами, но являются первичными ключами в другой таблице. В рассматриваемом примере столбец “№ отдела” таблицы “Сотрудники” содержит вторичный ключ, с помощью которого может быть установлена логическая взаимосвязь строк таблицы с соответствующими строками таблицы “Отделы”.

Если какая-либо таблица содержит вторичный ключ, то она считается логически взаимосвязанной с таблицей, содержащей соответствующий первичный ключ.

В общем случае эта связь имеет характер “один ко многим” (одному значению первичного ключа может соответствовать несколько значений вторичного, пример – отдел № 15).

Возможны варианты, когда вторичный ключ входит в состав первичного ключа. Всё зависит от предметной области, которую описывает модель. В общем случае СУБД ничего “не знает” о логической взаимосвязи таблиц модели.

При обращении к СУБД **с запросом** пользователь должен в явном виде указать условия связывания двух таблиц.

В нашем примере условие будет выглядеть примерно так:

“Сотрудники”.”№ отдела” = “Отделы”.”№ отдела”.

Следовательно, в процессе написания запроса возможно связать две таблицы по любым произвольным полям (не только по первичным и вторичным ключам), которые в принципе могут быть сравнимы друг с другом. В этом случае связь носит характер “многие ко многим”. Иногда это бывает необходимо делать при написании сложных и специфических запросов, но в общем случае не рекомендуется и свидетельствует об ошибках при проектировании логической модели БД.

В некоторых реляционных СУБД возможно создавать так называемые ограничения целостности, которые в том числе контролируют взаимосвязь между **PK** и **FK**. Так, СУБД заблокирует попытки удалить запись из таблицы, на первичный ключ которой “ссылаются” вторич-

ные ключи в других таблицах. И наоборот – нельзя будет внести в поле вторичного ключа значение, отсутствующее в первичном ключе логически взаимосвязанной таблицы. Но это – только средство поддержания целостности данных и защиты от ошибок. Даже при наличии таких конструкций СУБД всё равно требует от пользователя логического связывания таблиц в явном виде при написании запросов к данным.

Нормализация модели данных

Основным критерием качества разработанной модели данных является ее соответствие так называемым нормальным формам (НФ).

Основная цель нормализации – устранение избыточности данных.

Коддом были определены три нормальные формы, которые включают одна другую. Другими словами, если модель данных соответствует 2НФ, то она одновременно соответствует и 1НФ. **Соответствие 3НФ подразумевает соответствие 1НФ и 2НФ.**

Первая нормальная форма гласит: информация в каждом поле таблицы является неделимой и не может быть разбита на подгруппы.

Пример информации, не соответствующей 1НФ:

…	Иванов, 15 отдел, начальник	…

Правильно:

Фамилия	**Должность**	**№ отдела**
Иванов	Начальник	15

Вторая нормальная форма гласит: таблица соответствует 1НФ и в таблице нет неключевых атрибутов, зависящих от части сложного (состоящего из нескольких столбцов) первичного ключа.

Пример информации, не соответствующей 2НФ:

№ отдела	**Должность**	**Отдел**	**Количество сотрудников**
15	Начальник	Производственный	1

		отдел	
15	Инженер	Производственный отдел	5
10	Начальник	Отдел продаж	1
10	Менеджер	Отдел продаж	10

Предположим, что данная модель описывает структуру отделов по должностям. Первичный ключ (выделен серым цветом) является сложным и состоит из двух столбцов (номер отдела и наименование должности).

В данном случае наименование отдела логически зависит только от номера отдела и не зависит от должности (одна и та же должность может существовать в разных отделах). Чтобы привести модель ко 2-й НФ, необходимо разбить эту таблицу на две логически взаимосвязанные:

Отделы

№ отдела	**Наименование**
15	Производственный отдел
10	Отдел продаж

Структура

№ отдела	**Должность**	**Количество сотрудников**
15	Начальник	1
15	Инженер	5
10	Начальник	1

10	Менеджер	10

Кстати, это пример случая, когда вторичный ключ (“№ отдела”) одновременно является частью первичного (“№ отдела”, “Должность”).

Следствие: если в таблице первичный ключ состоит из одного столбца, то эта таблица автоматически соответствует 2НФ (при условии соответствия и 1НФ).

Третья нормальная форма гласит: таблица соответствует первым двум НФ и все неключевые атрибуты зависят только от первичного ключа и не зависят друг от друга.

Пример несоответствия 3НФ:

Сотрудники

Табельный №	Фамилия	Оклад	Наименование отдела	№ отдела
1	Иванов	500	Производственный отдел	15
2	Петров	400	Производственный отдел	15
3	Иванов	600	Отдел продаж	10

Очевидно, что неключевые атрибуты “Наименование отдела” и “№ отдела” логически взаимосвязаны друг с другом, в то время как комбинация фамилия-отдел-оклад имеет смысл только в сочетании с табельным номером сотрудника (предположим, что в организации работают два Иванова-однофамильца в разных отделах).

Решение может быть следующим:

Сотрудники

Табельный №	Фамилия	Оклад	№ отдела
1	Иванов	500	15
2	Петров	400	15

3	Иванов	600	10

Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

Язык SQL

Взаимодействие приложений и пользователей с реляционными СУБД осуществляется посредством специального языка структурированных запросов (Structured Query Language, сокращенно – SQL).

SQL был разработан еще в начале 70-х годов XX века и представляет собой непроцедурный язык, состоящий из набора стандартных команд на английском языке.

Термин “непроцедурный” означает, что изначально в языке отсутствуют алгоритмические конструкции (переменные, переходы по условию, циклы и т.д.) и возможность компоновать логически связанные команды в единые программные блоки (процедуры и функции).

Язык SQL в настоящий момент стандартизован, последний действующий стандарт носит название SQL2. Практически все известные СУБД поддерживают требования стандарта SQL2 плюс вводят собственные расширения языка SQL, учитывающие особенности конкретной СУБД (в том числе и процедурные расширения).

Общий принцип работы с СУБД посредством SQL можно кратко описать следующим образом: выдал команду – получил результат.

Отдельные команды изначально никак логически не связаны друг с другом. Например, для извлечения данных из таблицы пользователь должен сформировать специальное предложение

на языке SQL. СУБД обрабатывает запрос, извлекает нужные данные и возвращает их пользователю, после чего “забывает” об этом и переходит в состояние готовности выполнить любой очередной запрос SQL.

“Общение” пользователя с СУБД осуществляется с помощью специальных утилит, которые обычно входят в комплект поставки СУБД. В частности, у Oracle эта утилита называется SQL*Plus, а у MS SQL Server – Query Analyzer. Любая из них способна как минимум принять от пользователя SQL-команду, отправить ее на выполнение ядру СУБД и отобразить на экране результат операции. Вот как выглядит экран Oracle SQL*Plus:

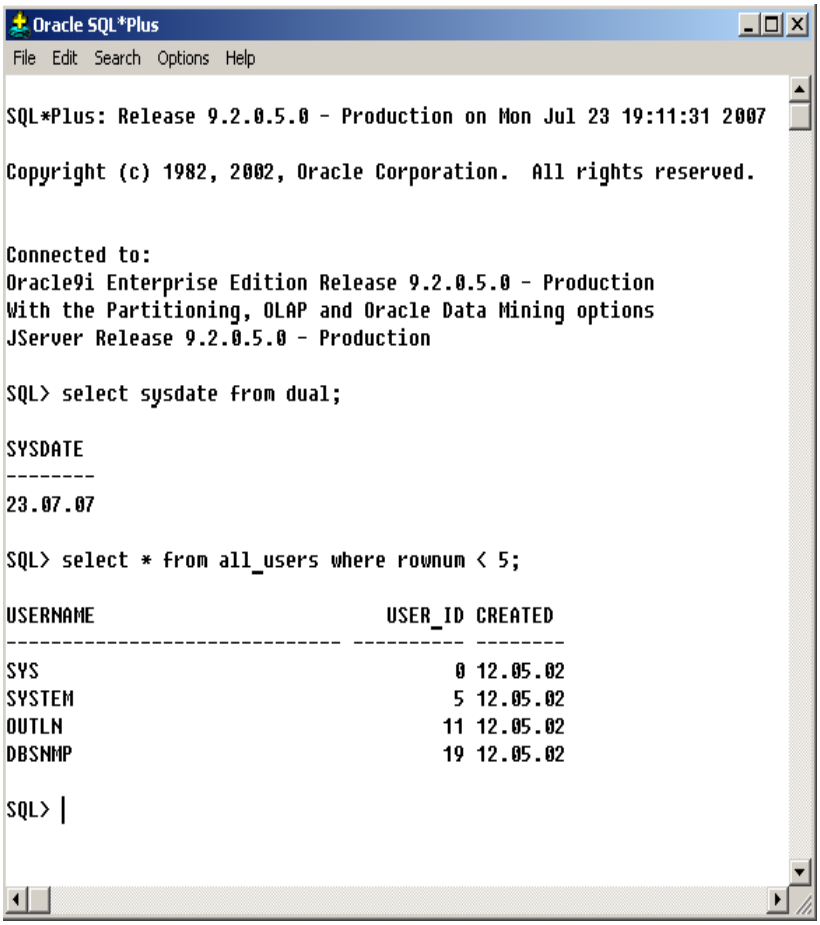

Рис. 1. Интерактивная утилита

В состав дистрибутива СУБД входят и API-библиотеки, позволяющие исполнять SQL-запросы из написанного пользователем программного кода.

Команды SQL

Как будет подробнее рассмотрено ниже, SQL позволяет не только извлекать данные, но и изменять их, добавлять новые данные, удалять данные, определять структуру данных, управлять пользователями, разграничивать доступ к данным и многое другое.

Базовый вариант SQL содержит порядка 40 команд (часто еще называемых запросами или операторами) для выполнения различных действий внутри СУБД.

Все SQL-команды начинаются с глагола (команды), определяющего, что именно нужно сделать.

Далее с помощью внутренних ключевых слов задаются дополнительные условия выполнения. Например, команда на выборку табельных номеров сотрудников с зарплатой больше 500 у.е. из таблицы, содержащей список сотрудников некоей организации, выглядит следующим образом:

```
SELECT TabNum FROM Employees WHERE Salary > 500
```

где:

- **SELECT** – глагол (“выбрать”);
- **FROM**, **WHERE** – ключевые слова (“откуда”, “где”);
- **Employees** – имя таблицы;
- **TabNum**, **Salary** – имена столбцов таблицы.

В общем случае структура каждой команды зависит от ее типа.

В зависимости от вида производимых действий все команды разбиты на несколько групп.

Команды определения структуры данных

(Data Definition Language – DDL)

В состав DDL-группы входят команды, позволяющие определять внутреннюю структуру базы данных.

Перед тем как сохранять данные в БД, необходимо создать в ней таблицы и, возможно, *некоторые другие сопутствующие объекты* (увеличивающие скорость поиска индексы, ограничения целостности и др.).

Пример некоторых DDL-команд:

Команда	Описание
CREATE TABLE	Создать новую таблицу
DROP TABLE	Удалить существующую таблицу
ALTER TABLE	Изменить структуру существующей таблицы

Команды манипулирования данными

(Data Manipulation Language – DML)

DML-группа содержит команды, позволяющие вносить, изменять, удалять и извлекать данные из таблиц.

Примеры DML-команд:

Команда	Описание
`SELECT`	Извлечь данные из таблицы
`INSERT`	Добавить новую строку данных в таблицу
`DELETE`	Удалить строки из таблицы
`UPDATE`	Изменить информацию в строках таблицы

Команды управления транзакциями

(Transaction Control Language – TCL)

TCL-команды используются для управления изменениями данных, производимыми DML-командами.

С их помощью несколько DML-команд могут быть объединены в единое логическое целое, называемое транзакцией.

При этом все команды на изменение данных в рамках одной транзакции либо завершаются успешно, либо все могут быть отменены в случае возникновения каких-либо проблем с выполнением любой из них.

Транзакции есть одно из средств поддержания целостности и непротиворечивости данных и являются одной из важнейших функций современных СУБД.

TCL-команды:

Команда	Описание
`COMMIT`	Завершить транзакцию и зафиксировать все изменения в БД

ROLLBACK	Отменить транзакцию и отменить все изменения в БД
SET TRANSACTION	Установить некоторые условия выполнения транзакции

Команды управления доступом

(Data Control Language – DCL)

DCL-команды управляют доступом пользователей к БД и отдельным объектам:

Команда	Описание
GRANT	Разрешить доступ
REVOKE	Отменить доступ

Работа с командами SQL

Извлечение данных, команда SELECT

Быстрое извлечение данных, хранящихся в таблицах, одна из основных задач СУБД. Для выборки данных используется команда **SELECT**.

В общем виде синтаксис этой команды выглядит следующим образом:

```
SELECT [DISTINCT] <список столбцов>
FROM <имя таблицы> [JOIN <имя таблицы> ON <условия связывания>]
[WHERE <условия выборки>]
[GROPUP BY <список столбцов для группировки>
                                    [HAVING <условия выборки групп>] ]
[ORDER BY <список столбцов для сортировки>]
```

В квадратных скобках указаны необязательные элементы команды. **Ключевые слова SELECT и FROM должны присутствовать всегда.**

Ниже рассмотрены возможные варианты написания этой команды подробнее.

Список столбцов содержит перечень имен столбцов таблицы, которые должны быть включены в результат. Имена, если их несколько, отделяются друг от друга запятой:

```
SELECT TabNum FROM Employees
```

```
SELECT TabNum, Name FROM Employees
```

Звездочка (*****) на месте списка столбцов обозначает все столбцы таблицы:

```
SELECT * FROM Employees
```

При выборке столбцов с одинаковыми именами из нескольких таблиц перед именем каждого столбца надо указать через точку имя таблицы:

```
SELECT Employees.Name, Departments.Name FROM …
```

Ключевое слово DISTINCT

Если в результирующем наборе данных встречаются одинаковые строки (значения всех полей совпадают), можно от них избавиться, указав ключевое слово **DISTINCT** перед списком столбцов. Приведенный ниже запрос вернет уникальный список должностей, существующих в организации:

```
SELECT DISTINCT Position FROM Employees
```

Секция FROM, логическое связывание таблиц

Перечень таблиц, из которых производится выборка данных, указывается в секции **FROM**. Выборка возможна как из одной таблицы, так и из нескольких логически взаимосвязанных. Логическая взаимосвязь осуществляется с помощью подсекции **JOIN**.

На каждую логическую связь пишется отдельная подсекция. Внутри подсекции указывается условие связи двух таблиц (обычно по условию равенства первичных и вторичных ключей). Примеры для модели данных `Сотрудники-Отделы-Города`:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инж.	15	500
3	Сидоров	Менеджер	10	700

Departments

DeptNum	City	Name
15	1	Производственный отдел
10	2	Отдел продаж

Cities

City	Name
1	Минск
2	Москва

```
SELECT Employees.TabNum, Employees.Name, Departments.Name
    FROM Employees
        JOIN Departments ON Employees.DeptNum =
            Departments.DeptNum
```

Результат запроса будет выглядеть следующим образом:

1	Иванов	Производственный отдел
2	Петров	Производственный отдел
3	Сидоров	Отдел продаж

```
SELECT Employees.TabNum, Employees.Name, Departments.Name,
                                                Cities.Name
                    FROM Employees
JOIN Departments ON Employees.DeptNum = Departments.DeptNum
JOIN Cities ON Departments.City = Cities.City
```

Результат запроса будет выглядеть следующим образом:

1	Иванов	Производственный отдел	Минск
2	Петров	Производственный отдел	Минск
3	Сидоров	Отдел продаж	Москва

Пример связывания таблиц по нескольким полям:

```
SELECT Table1.Field1, Table2.Field2
    FROM Table1
    JOIN Table2
ON Table2.ID1 =Table1.ID1
AND Table2.ID2 =Table1.ID2
        AND ….
```

Существует несколько типов связывания:

Тип	Результат
JOIN	Внутреннее соединение. В результирующем наборе присутствуют только записи, значения связанных полей в которых совпадают
LEFT JOIN	Левое внешнее соединение. В результирующем наборе присутствуют все записи из Table1 и соответствующие им записи из Table2. *Если соответствия нет, поля из Table2 будут пустыми*
RIGHT JOIN	Правое внешнее соединение. В результирующем наборе присутствуют все записи из Table2 и соответствующие им записи из Table1. *Если соответствия нет, поля из Table1 будут пустыми*
FULL JOIN	Полное внешнее соединение. Комбинация двух предыдущих. В результирующем наборе присутствуют все записи из Table1 и соответствующие им записи из Table2. Если соответствия нет – поля из Table2 будут пустыми.

	Записи из Table2, которым не нашлось пары в Table1, тоже будут присутствовать в результирующем наборе. В этом случае поля из Table1 будут пустыми.
CROSS JOIN	Cartesian product. Результирующий набор содержит все варианты комбинации строк из Table1 и Table2. Условие соединения при этом не указывается.

Проиллюстрируем каждый тип примерами. Модель данных:

Table1

Key1	Field1
1	A
2	B
3	C

Table2

Key2	Field2
1	AAA
2	BBB
2	CCC
4	DDD

```
SELECT Table1.Field1, Table2.Field2
    FROM Table1
    JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC

```
SELECT Table1.Field1, Table2.Field2
    FROM Table1
    LEFT JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
C	

```
SELECT Table1.Field1, Table2.Field2
    FROM Table1
    RIGHT JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
	DDD

```
SELECT Table1.Field1, Table2.Field2
    FROM Table1
    FULL JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
	DDD
C	

```
SELECT Table1.Field1, Table2.Field2
```

```
FROM Table1
CROSS JOIN Table2
```

Результат:

A	AAA
A	BBB
A	CCC
A	DDD
B	AAA
B	BBB
B	CCC
B	DDD
C	AAA
C	BBB
C	CCC
C	DDD

Секция WHERE

Для фильтрации результатов выполнения запроса можно использовать условия выборки в секции **WHERE**.

В общем виде синтаксис **WHERE** выглядит следующим образом:

```
WHERE [NOT] <условие1>  [ AND | OR <условие2>]
```

Условие представляет собой конструкцию вида:

```
<столбец таблицы, константа или  выражение>
```

<оператор сравнения> <столбец таблицы, константа или выражение>

или

IS [NOT] NULL

или

[NOT] LIKE <шаблон>

или

[NOT] IN (<список значений>)

или

[NOT] BETWEEN <нижняя граница> **AND** <верхняя граница>

Операторы сравнения:

<	Меньше
<=	Меньше либо равно
<>	Не равно
>	Больше
>=	Больше либо равно
=	Равно

Примеры запросов с операторами сравнения:

```
SELECT * FROM Table WHERE Field > 100
SELECT * FROM Table WHERE Field1 <= (Field2 + 25)
```

Выражение **IS [NOT] NULL** проверяет данные на [не]пустые значения:

```
SELECT * FROM Table WHERE Field IS NOT NULL
SELECT * FROM Table WHERE Field IS NULL
```

Необходимо отметить, что язык SQL, в отличие от языков программирования, имеет встроенные средства поддержки факта отсутствия каких-либо данных.

Осуществляется это с помощью **NULL**-концепции.

NULL не является каким-то фиксированным значением, хранящимся в поле записи вместо реальных данных. Значение **NULL** не имеет определенного типа. **NULL** – это индикатор, говорящий пользователю (и SQL) о том, что данные в поле записи отсутствуют.

Поэтому его нельзя использовать в операциях сравнения.

Для проверки факта наличия-отсутствия данных в SQL введены специальные выражения.

Выражение [**NOT**] **LIKE** используется при проверке текстовых данных на [не]соответствие заданному шаблону. Символ ‘%’ (процент) в шаблоне заменяет собой любую последовательность символов, а символ ‘_’ (подчеркивание) – один любой символ.

```
SELECT * FROM Employees WHERE Name LIKE ‘Иван%’
```

Попадающие под заданное условие фамилии: **Иван**ов

```
SELECT * FROM Employees WHERE Name LIKE ‘__д%’
```

Попадающие под заданное условие фамилии: Си**д**оров

Выражение [NOT] IN проверяет значения на [не]вхождение в определенный список:

```
SELECT * FROM Employees WHERE Position IN (‘Начальник’, ‘Менеджер’)
```

Выражение [**NOT**] **BETWEEN** проверяет значения на [не]попадание в некоторый диапазон:

```
SELECT * FROM Employees WHERE Salary BETWEEN 200 AND 500
```

Этот запрос вернет список работников, зарплата которых больше либо равна 200 у.е. и меньше либо равна 500 у.е.

Несколько условий поиска могут комбинироваться посредством логических операторов ***AND***, ***OR*** *или* ***NOT****:*

```
SELECT *
    FROM Employees
    WHERE Position IN (‘Начальник’, ‘Менеджер’)
        AND Salary BETWEEN 200 AND 500
SELECT *
    FROM Employees
    WHERE (Position = ‘Начальник’ OR  Position =
                                    ‘Менеджер’)
        AND Salary BETWEEN 200 AND 500
SELECT *
    FROM Employees
    WHERE NOT (Position = ‘Начальник’ OR  Position =
                                    ‘Менеджер’)
```

Секция ORDER BY

Необязательная секция **ORDER BY** в команде **SELECT** предназначена для сортировки строк результирующего набора данных.

Формат этой секции в общем виде выглядит так:

```
ORDER BY Field1 [ASC | DESC] [, Field2 [ASC | DESC] ] [, …]
```

Ключевое слово **ASC** предписывает производить сортировку по возрастанию, а **DESC** – по убыванию. Если **ASC** и **DESC** отсутствуют, по умолчанию подразумевается **ASC**. Например, выберем записи о начальниках и отсортируем результат в порядке убывания размера зарплат:

```
SELECT *
    FROM Employees
    WHERE Position = 'Начальник'
    ORDER BY Salary DESC
```

Следующий запрос отсортирует сотрудников по отделам (в порядке возрастания номера отдела) и по размеру зарплат внутри каждого отдела (в порядке убывания зарплаты):

```
SELECT *
    FROM Employees
    ORDER BY DeptNum ASC, Salary DESC
```

Ключевое слово **ASC** можно опустить, ибо оно действует по умолчанию:

```
SELECT *
    FROM Employees
    ORDER BY DeptNum, Salary DESC
```

Групповые функции

Если нас не интересуют строки таблицы как таковые, а интересуют некоторые итоги, мы можем использовать в процессе выборки колонок таблиц групповые функции. Основные групповые функции представлены ниже:

Функция	Описание
SUM(Field)	Вычисляет сумму по указанной колонке
MIN(Field)	Вычисляет минимальное значение по указанной колонке
MAX(Field)	Вычисляет максимальное значение по указанной колонке
AVG(Field)	Вычисляет среднее значение по указанной колонке
COUNT(*)	Вычисляет количество строк в результирующей выборке
COUNT(Field)	Вычисляет количество не пустых значений в колонке

Например, чтобы узнать максимальную зарплату, получаемую сотрудниками в организации, можно выполнить запрос вида:

```
SELECT MAX(SALARY)
    FROM Employees
```

Общее количество записей в таблице вернет запрос вида:

```
SELECT COUNT(*)
    FROM Employees
```

Секция GROUP BY

По умолчанию группой, на которой вычисляется групповая функция, является вся результирующая выборка. Если мы нуждаемся в вычислении промежуточных итогов, мы можем разбить итоговую выборку на подгруппы с помощью необязательной секции **GROUP BY**:

```
GROUP BY Field1 [, Field2] [, …]
```

Например, подсчитаем максимальную зарплату по отделам организации:

```
SELECT DeptNum, MAX(SALARY)
    FROM Employees
    GROUP BY DeptNum
```

В этом случае функция **MAX** будет считаться отдельно для всех записей с одинаковым значением поля **DeptNum**.

Секция HAVING

На промежуточные итоги может быть наложен *дополнительный фильтр* посредством секции **HAVING**. В нижеприведенном примере в результат попадут только отделы, максимальная зарплата в которых превышает 1000 у.е.:

```
SELECT DeptNum, MAX(SALARY)
    FROM Employees
    GROUP BY DeptNum
    HAVING MAX(SALARY) > 1000
```

Важно понимать, что секции **HAVING** и **WHERE** взаимно дополняют друг друга. Сначала с помощью ограничений **WHERE** формируется итоговая выборка, затем выполняется разбивка на группы по значениям полей, заданных в **GROUP** BY. Далее по каждой группе вычисляется групповая функция и в заключение накладывается условие **HAVING**.

Изменение данных

Под изменением данных понимаются следующие операции:

- вставка новых строк в таблицу;
- изменение существующих строк;
- удаление существующих строк.

Команда INSERT

Добавление новых записей в таблицу осуществляется посредством команды **INSERT**. Она имеет следующий синтаксис:

```
INSERT INTO <имя таблицы> [(<список имен колонок>)]
   VALUES(<список констант>)
```

Например, для внесения сведений о новом работнике необходимо выполнить следующую команду:

```
INSERT INTO Employees(TabNum, Name, Position, DeptNum,
                                          Salary)
   VALUES(45, 'Сергеев', 'Старший менеджер', 15, 850)
```

После выполнения команды таблица Employees будет выглядеть следующим образом:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инженер	15	500
3	Сидоров	Менеджер	10	700
45	Сергеев	Старший менеджер	15	850

Если какая-либо колонка в списке будет опущена при вставке, в соответствующее поле записи автоматически будет занесено пустое значение (**NULL**):

```
INSERT INTO Employees(TabNum, Name, DeptNum, Salary)
    VALUES(45, 'Сергеев', 15, 850)
```

После выполнения команды таблица Employees будет выглядеть следующим образом:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инженер	15	500
3	Сидоров	Менеджер	10	700
45	Сергеев		15	850

Количество констант в секции **VALUES** всегда должно соответствовать количеству колонок.

Список колонок в команде **INSERT** может быть опущен целиком. В этом случае список констант в секции **VALUES** должен точно соответствовать описанию колонок таблицы в словаре данных СУБД, иначе команда будет отвергнута ядром БД.

Пример правильной команды:

```
INSERT INTO Employees VALUES(45, 'Сергеев',
                        'Старший менеджер', 15, 850)
```

Команда вида:

```
INSERT INTO Employees VALUES(45, 'Сергеев', 15, 850)
```

завершится ошибкой, так как количество констант не соответствует реальному количеству колонок в таблице.

В колонку можно в явном виде внести пустое значение посредством ключевого слова **NULL**.

Последний запрос можно переписать следующим образом:

```
INSERT INTO Employees VALUES(45, 'Сергеев', NULL, 15, 850)
```

В этом случае команда вставки отработает корректно, и в поле Position будет внесено пустое значение. Очевидно, что к аналогичному результату приведет и команда:

```
INSERT INTO Employees(TabNum, Name, Position, DeptNum,
                                        Salary)
   VALUES(45, 'Сергеев', NULL, 15, 850)
```

Кроме простого добавления новых записей, команда **INSERT** позволяет осуществлять пакетную перекачку данных из таблицы в таблицу.

Синтаксис подобной команды следующий:

```
INSERT INTO <имя таблицы> [(<список имен колонок>)]
    <команда SELECT>
```

Например:

```
INSERT INTO Table1(Field1, Field2)
    SELECT Field3, (Field4 + 5) FROM Table2
```

Команда DELETE

Чтобы удалить ненужные записи из таблицы, следует использовать команду **DELETE**:

```
DELETE FROM <имя таблицы> [WHERE <условия поиска>]
```

Если опустить секцию условий поиска **WHERE**, из таблицы будут удалены все записи. Иначе – только записи, удовлетворяющие критериям поиска.

Форматы секций **WHERE** команд **SELECT** и **DELETE** аналогичны.

Примеры команды **DELETE**:

```
DELETE FROM Employees
DELETE FROM Employees WHERE TabNum = 45
```

Команда UPDATE

Изменить ранее внесенные командой **INSERT** данные можно с помощью команды **UPDATE**:

```
UPDATE < имя таблицы>
```

```
SET <имя колонки> = <новое значение> , <имя колонки> =
                                <новое значение>, …
WHERE <условия поиска>]
```

Как и в случае команды **DELETE**, при отсутствии секции **WHERE** обновлены будут все строки таблицы. Иначе – только подходящие под заданные условия.

Примеры:

```
UPDATE Employees SET Salary = Salary + 100
UPDATE Employees
SET Position = ‘Старший менеджер’, Salary = 1000
WHERE TabNum = 45 AND Position IS NULL
```

Определение структуры данных

Команда CREATE TABLE

Для создания новых таблиц используется команда **CREATE TABLE**. В общем виде ее синтаксис следующий:

```
CREATE TABLE <имя таблицы>
(
 <имя колонки> <тип колонки>[(<размер колонки>)]
                        [<ограничение целостности уровня колонки>]
[, <имя колонки> <тип колонки>[(<размер колонки>)]
                        [<ограничение целостности уровня колонки>]]
[, …]
[<ограничение целостности уровня таблицы>]
[,<ограничение целостности уровня таблицы>]
[, …]
)
```

Примеры:

```
CREATE TABLE Departments
(
DeptNum int NOT NULL PRIMARY KEY,
Name varchar(80) NOT NULL
)
CREATE TABLE Employees
(
 TabNum int NOT NULL PRIMARY KEY,
 Name varchar(100) NOT NULL,
 Position varchar(200),
 DeptNum int,
Salary decimal(10, 2) DEFAULT 0,
CONSTRAINT FK_DEPARTMENT FOREIGN KEY (DeptNum)
    REFERENCES Departments(DeptNum)
)
```

Помимо команды **CREATE TABLE**, можно создать новую таблицу с помощью специальной формы команды **SELECT**:

```
SELECT [DISTINCT] <список колонок>
INTO <имя новой таблицы>
FROM <имя таблицы> [JOIN <имя таблицы> ON <условия связывания>]
[WHERE <условия выборки>]
[GROUP BY <список колонок для группировки> [HAVING <условия выборки групп>] ]
[ORDER BY <список колонок для сортировки>]
```

При наличии ключевого слова **INTO** в команде **SELECT** ядро СУБД не вернет результирующую выборку пользователю, а автоматически создаст новую таблицу с указанным именем и заполнит ее данными из результирующей выборки.

Имена колонок таблицы и типы будут определены автоматически при анализе команды **SELECT** и словаря базы данных.

Команда ALTER TABLE

Созданную таблицу можно впоследствии изменить с помощью команды **ALTER TABLE**. Команда **ALTER TABLE** позволяет добавлять новые колонки и ограничения целостности, удалять их, менять типы колонок, переименовывать колонки.

Примеры различных вариантов команды **ALTER TABLE**:

```
ALTER TABLE Departments ADD COLUMN City int

ALTER TABLE Departments DROP COLUMN City

ALTER TABLE Departments ADD
    CONSTRAINT FK_City
        FOREIGN KEY (City)
        REFERENCES Cities(City)

ALTER TABLE Departments DROP CONSTRAINT FK_City
```

Команда DROP TABLE

Удаление ранее созданной таблицы производится командой **DROP TABLE**:

```
DROP TABLE Departments
```