Схемы таблиц. Представления.

5.1. Средства изменения описания таблиц и средства удаления таблиц

В стандарте *SQL2* имеются достаточно широкие возможности *по* модификации уже существующих схем таблиц. Для модификации таблиц используется оператор ALTER TABLE, который позволяет выполнить следующие *операции* изменения для схемы таблицы:

- добавить новый столбец в уже существующую и заполненную таблицу;
- изменить значение по умолчанию для какого-либо столбца;
- удалить столбец из существующей таблицы;
- добавить или удалить первичный ключ таблицы;
- добавить или удалить новый внешний ключ таблицы;
- добавить или удалить условие уникальности;
- добавить или удалить условие проверки для любого столбца или для таблицы в целом.

Синтаксис оператора ALTER TABLE:

```
<Изменить описание таблицы>::= ALTER TABLE <имя таблицы>
 { ADD определение столбца>
 ALTER <имя столбца> {SET DEFAULT <значение>
 DROP DEFAULT } |
 DROP <имя столбца> {CASCADE | RESTRICT} |
 ADD { <определение первичного ключа>|
 <определение внешнего ключа> |
 <условие уникальности данных> |
 <условие проверки> } |
 DROP CONSTRAINT имя условия { CASCADE |
 RESTRICT} }
```

Одним оператором ALTER TABLE можно провести только одно из перечисленных изменений, например, за один раз можно добавить один столбец. Если вам требуется добавить два столбца, то необходимо применить два оператора.

Давайте рассмотрим несколько примеров. Чаще всего применяется операция добавления столбца. Предложение определения нового столбца в операторе ALTER TABLE имеет точно такой же *синтаксис*, как и в операторе создания таблицы. Добавим столбец *EDUCATION* (образование), содержащий символьный *тип данных*, с заданным перечнем значений ("начальное", "среднее", "неоконченное высшее", "высшее").

```
ALTER TABLE READERS
```

```
ADD EDUCATION varchar (30) DEFAULT NULL
CHECK (EDUCATION IS NULL OR
 EDUCATION= "начальное" OR
 EDUCATION= "среднее" OR EDUCATION= "бакалавр" OR
 EDUCATION= "магистр")
```

В таблицу READERS будет добавлен столбец *EDUCATION*, в который *по* умолчанию будут добавлены все кортежи неопределенного значения. В дальнейшем эти значения могут быть заменены на одно из допустимых символьных значений.

Добавим ограничение на соответствие между датами взятия и возврата книги в таблице *EXEMPLAR*. Действительно, если даты введены, то требуется, чтобы дата возврата книги была бы больше на срок выдачи книги. Считаем, что стандартным сроком являются 2 недели. Теперь сформулируем оператор изменения таблицы *EXEMPLARE*:

```
ALTER TABLE EXEMPLAR
ADD CONSTRAINT CK_EXEMPLAR CHECK ((DATA_IN IS NULL AND DATA_OUT IS
NULL) OR
 /*Между датами 14 дней*/
DateDiff(day, DATA_IN,DATA_OUT)<=14
)
```

Операция удаления столбца связана с проверкой ссылочной целостности, и поэтому не разрешается удалять столбцы, связанные с *поддержкой ссылочной целостности* таблицы, то есть нельзя удалить столбцы *родительской таблицы*, входящие в *первичный ключ* таблицы, если на них есть ссылки в подчиненных таблицах.

При изменении первичного ключа таблицы следует быть внимательными. Во-первых, у исходной таблицы могут быть подчиненные, при этом *первичный ключ* исходной таблицы является внешним ключом для *подчиненных таблиц*, и просто его удалить невозможно, *СУБД* контролирует ссылочную *целостность* и не позволит выполнить операцию удаления первичного ключа таблицы, если на него имеются ссылки. Следовательно, в этом случае порядок изменения первичного ключа должен быть таким, как на рис. 5.1:

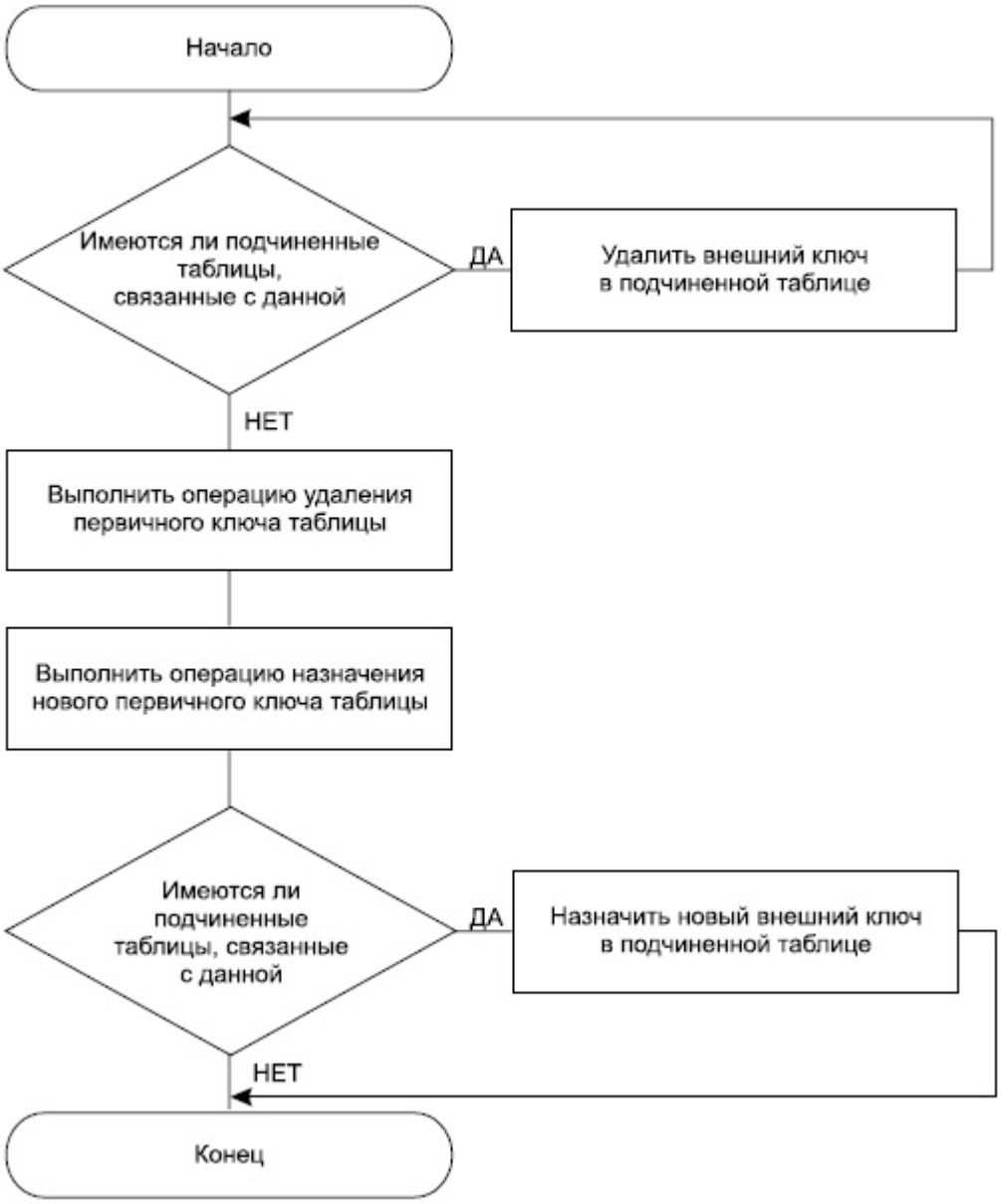

Рис. 5.1. Алгоритм изменения первичного ключа таблицы

Чаще всего операция ALTER TABLE применяется в CASE-системах при автоматической генерации скриптов создания таблиц в базе данных. В этих системах *универсальный алгоритм* предполагает сначала создание всех таблиц, которые заданы в даталогической модели, и только после этого добавляются соответствующие связи. И это понятно — в отличие от человеческого разума искусственный интеллект *CASE-системы* будет испытывать затруднения в определении иерархических взаимосвязей таблиц *базы данных*, поэтому он предпочитает использовать *универсальный алгоритм*, в котором сначала все объекты определяются, а затем добавляются соответствующие свойства для атрибутов, которые являются внешними ключами с указанием требуемых ссылок. В этом случае все *операции* назначения внешних ключей будут считаться корректными, потому что все объекты были описаны заранее, и для такого алгоритма порядок создания таблиц безразличен.

В языке *SQL* присутствует и операция *удаления таблиц*. *Синтаксис* этой *операции* предельно прост:

```
<Удалить таблицу> ::= DROP TABLE <имя таблицы> [CASCADE | RESTRICT]
```

Параметр CASCADE означает, что при *удалении таблицы* одновременно удаляются и все объекты, связанные с ней. С таблицей, кроме рассмотренных ранее ограничений, могут быть связаны также объекты *типа триггеров* и представления. Однако операция удаления объектов определяется еще правами пользователей, что связано с концепцией безопасности в базах данных. Это значит, что если вы не являетесь владельцем объекта, то вы можете не иметь прав на его удаление. И в этом случае синтаксически правильный оператор DROP TABLE не может быть выполнен системой в силу отсутствия прав на удаление связанных с удаляемой таблицей объектов. Кроме того, операция удаления таблицы не должна нарушать *целостность базы данных*, поэтому удалять таблицу, на которую имеются ссылки других таблиц, невозможно.

Например, в нашей схеме, связанной с библиотекой, мы не можем удалить ни таблицу BOOKS, ни таблицу READERS, ни таблицу CATALOG. У этих таблиц есть *связь* с *подчиненными таблицами EXEMPLAR* и RELATION_67. Поэтому если вы хотите удалить некоторый набор таблиц, то сначала необходимо грамотно построить последовательность их удаления, которая не нарушит базовых принципов поддержки целостности вашей схемы *БД*. В нашем примере *последовательность операторов удаления таблиц* может быть следующей:

```
DROP TABLE EXEMPLAR
DROP TABLE RELATION_67
DROP TABLE CATALOG
DROP TABLE READERS
DROP TABLE BOOKS
```

5.2. Понятие представления. Операции создания представлений

Для описания внешних моделей в реляционной модели могут использоваться представления. *Представление (View)* — это *SQL-запрос* на выборку, который *пользователь* воспринимает как некоторое *виртуальное отношение*. Задание представлений входит в описание схемы *БД* в реляционных *СУБД*. Представления позволяют скрыть ненужные несущественные детали для разных пользователей, модифицировать реальные структуры данных в удобном для приложений виде и, наконец, разграничить *права* доступа к данным и тем самым повысить защиту данных от несанкционированного доступа.

В отличие от реальной таблицы *представление* в том виде, как оно сконструировано, не существует в базе данных, это действительно только *виртуальное отношение*, хотя все данные, которые представлены в нем,

действительно существуют в базе данных, но в разных отношениях. Они скомпонованы для пользователя в удобном виде из *реальных таблиц* с помощью некоторого запроса. Однако *пользователь* может этого не знать, он может обращаться с этим представлением как со стандартной таблицей. *Представление* при создании получает некоторое уникальное имя, его описание хранится в описании схемы *базы данных*, и *СУБД* в любой момент времени при обращении к этому представлению выполняет *запрос*, соответствующий его описанию, поэтому *пользователь*, работая с представлением, в каждый момент времени видит действительно реальные, актуальные на настоящий момент данные. Оно формируется как бы на лету, в момент обращения.

Оператор определения представления имеет следующий вид:

```
<создание представления>::= CREATE VIEW <имя представления>
[ (<список столбцов>)] AS <SQL-запрос>
```

При необходимости в представлении можно задать новое имя для каждого столбца виртуальной таблицы. При этом надо помнить, что если указывается *список* столбцов, то он должен содержать ровно столько столбцов, сколько содержит их *SQL-запрос*.

Если *список* имен столбцов в представлении не задан, то каждый столбец представления получает имя соответствующего столбца запроса.

Рассмотрим типичные виды представлений и их назначение.

5.2.1. Горизонтальное представление

Этот вид представления широко применяется для уменьшения объема *реальных таблиц* в обработке и ограничения доступа пользователей к закрытой для них информации. Так, например, правилом хорошего тона считается, что руководитель подразделения в некоторой фирме может видеть оклады и результаты работы только своих сотрудников, в этом случае для него создается горизонтальное представление, в которое загружены строки общей таблицы сотрудников, работающих в его подразделении.

Например, у нас есть таблица "Сотрудник" (EMPLOYEE) с полями "Табельный номер" (T_NUM), "ФИО" (NAME), "должность"(POSITION), "оклад"(SALARY), "надбавка"(PREMIUM), "отдел" (DEPARTMENT).

Для приложения, с которым работает начальник отдела продаж, будет создано представление

```
CREATE VIEW SAL_DEPT
 AS
 SELECT *
 FROM EMPLOYEE
 WHERE DEPARTMENT= "Отдел продаж"
```

5.2.2. Вертикальное представление

Этот вид представления практически соответствует выполнению операции проектирования некоторого отношения на ряд столбцов. Он используется в основном для скрытия информации, которая не должна быть доступна в конкретной внешней модели.

Например, для работника табельной службы, который учитывает присутствие сотрудников на работе, информация об окладе и надбавке должна быть закрыта. Для него можно создать следующее вертикальное представление:

```
CREATE VIEW TABEL
 AS
 SELECT T_NUM, NAME, POSITION, DEPARTMENT
 FROM EMPLOYEE
```

5.2.3. Сгруппированные представления

Эти представления содержат запросы, которые имеют группировку. Сгруппированные представления всегда должны содержать список столбцов. Они могут использовать агрегированные функции в качестве результирующих столбцов, а в дальнейшем это представление может использоваться как виртуальная таблица, например, в других запросах.

Создадим представление, которое определяет суммарный фонд заработной платы и надбавок по каждому подразделению с указанием количества сотрудников, минимальной, максимальной и средней зарплаты и надбавки по подразделению. Такой запрос позволяет сравнить заработную плату и надбавки прямо по всем подразделениям, и он может быть очень эффективно использован администрацией при проведении сравнительного анализа подразделений фирмы.

```
CREATE VIEW RATE
(DEPARTMENT, COUNT_EMP, SUM_SALARY, SUM_PREMIUM, MAX_SALARY,
MIN_SALARY,
 AVERAGE_SALARY, MAX_PREMIUM, MIN_PREMIUM, AVERAGE_PREMIUM)
AS
SELECT DEPARTMENT, COUNT(*), SUM(SALARY), SUM(PREMIUM),
MAX(SALARY), MIN(SALARY), AVERAGE (SALARY), MAX(PREMIUM),
MIN(PREMIUM),AVERAGE (PREMIUM)
FROM EMPLOYEE
GROUP BY DEPARTMENT
```

5.2.4. Объединенные представления

Часто представления базируются на многотабличных запросах. Такое использование позволяет упростить разработку пользовательского интерфейса, сохранив при этом корректность схемы базы данных. Для примера снова обратимся к базе данных "Библиотека" и создадим

представление, которое содержит список читателей-должников с указанием книг, которые у них на руках, и указанных в базе сроков сдачи этих книг. Такое представление может понадобиться для административного приложения, которое разрабатывается для директора библиотеки или его заместителя, они должны принимать административные меры для наказания нарушителей и возврата книг в библиотеку.

```
ALTER VIEW [dbo].[DEBTORS] (
 ISBN, [Название], [Читательский билет], [Фаммилия], [Имя], [Адрес],
[Домашний тел.], [Моб. тел.], [Дата возврата], [Просрочено, дней]
)
AS
SELECT books.ISBN, TITLE, Readers.[READER_ID], readers.LAST_NAME,
READERS.FIRST_NAME, ADRES, HOME_PHONE, CELL_PHONE, DATA_OUT,
DateDiff(DAY, EXEMPLAR.DATA_OUT, GetDate())
FROM BOOKS, EXEMPLAR, READERS
WHERE BOOKS.ISBN = EXEMPLAR.ISBN AND
EXEMPLAR.READER_ID = READERS.READER_ID AND
EXEMPLAR.DATA_OUT < GetDate()
```