

# Определение первичного ключа вставленной записи

*Сергей Супрунов*

**Первичные ключи имеют очень важное значение для работы с базой данных как в плане поиска нужных записей, так и с точки зрения обеспечения ссылочной целостности данных. Но при разработке приложений, взаимодействующих с БД, могут возникать специфические проблемы.**

**П**ри разработке структуры базы данных обычно используется один из двух подходов к выбору первичного ключа: назначение в качестве такового одного или нескольких столбцов таблицы, обеспечивающих возможность однозначно сослаться на данную запись (будем называть это естественным ключом, иногда используется термин «логический ключ»), либо ввод в структуру таблицы отдельного столбца, единственное назначение которого – уникально идентифицировать запись («искусственный ключ», называемый также суррогатным).

## Естественные и суррогатные ключи

Использование естественных ключей для работы с таблицами – наиболее прямолинейный и очевидный путь работы с базой данных. Например, в таблице пользователей Интернета в качестве первичного ключа вполне может использоваться имя учётной записи (login) абонента, в базе отдела кадров – табельный номер сотрудника, в телефонном справочнике – номер телефона и т. п. В ряде случаев в качес-

тве первичного можно задать составной ключ, если необходимая уникальность обеспечивается комбинацией нескольких полей.

Одним из недостатков этого подхода является то, что задача ввода, а порой и формирования ключа возлагается на пользователя. А следовательно, возможны ошибки – от банальных опечаток, когда вместо телефона 23456 случайно вводится 23356, до генерации неуникальных значений. Кроме того (и это является главной причиной, почему разработчики избегают естественных ключей), значение любого столбца, которое что-то означает, может измениться – абонент может поменять паспорт или потребовать смены логина, модернизация городской АТС может сопровождаться сменой плана нумерации телефонов, и т. п. Так что подобные ситуации приходится отслеживать, БД либо приложение должны обеспечивать сохранение целостности (а также и логичности) данных в случае модификации такого ключа, и так далее.

Также нужно указать, что использование слишком длинных естествен-

ных ключей (особенно составных) приводит к чрезмерному повышению избыточности БД – ведь значение такого ключа придётся указывать во всех связанных таблицах как внешний ключ (FOREIGN KEY). К тому же это снижает эффективность БД (увеличивается расход памяти, может возрасти сетевой трафик).

Поэтому в случаях, когда ни один столбец таблицы не может претендовать на роль первичного ключа, такой столбец можно придумать (то есть перейти к использованию суррогатного ключа). Например, что мешает в таблице, хранящей список городов, в качестве ключа использовать некую более или менее осмысленную аббревиатуру (типа «msk» (Москва), «spb» (Санкт-Петербург), «rnd» (Ростов-на-Дону)) или полный телефонный код районного центра? Поскольку речь здесь идёт о вводе суррогатного ключа, совершенно необязательно, чтобы он что-то означал, просто «осмысленность» может в некоторых случаях дать дополнительный положительный эффект – например, при необходимости обратиться к базе данных напрямую, а не через

приложение; также этот столбец вполне можно будет использовать и как альтернативный критерий поиска – при интенсивной работе с приложением всё-таки проще и быстрее запомнить эти аббревиатуры, чем вводить полные названия городов. Хотя в последнем случае ключ уже становится «естественным», поскольку его значение начинает использоваться как свойство объекта, данные о котором мы храним в таблице.

Однако придумывать значения для искусственного ключа порой становится весьма проблематичной задачей (тем более что полагаться в этом вопросе на фантазию пользователя не всегда разумно), поэтому разработчики предпочитают использовать методы автоматической генерации уникального числового значения, в том или ином виде предоставляемые практически всеми СУБД. В добавок, числовые столбцы наиболее эффективны с точки зрения ресурсозатрат системы. Однако здесь приходится сталкиваться с рядом трудностей.

Наиболее известная проблема, возникающая при разработке приложений, работающих с базой данных, активно использующей автоматически генерируемые значения, – это определение первичного ключа только что вставленной записи.

Например, рассмотрим такую ситуацию: некоторое приложение используется для ведения базы данных пользователей, и при заключении нового договора и добавлении связанной с ним новой учётной записи необходимо вставить строку в таблицу пользователей и строку в таблицу учётных записей, причём эти таблицы связаны ограничением ссылочной целостности (таблица учётных записей является «дочерней» по отношению к таблице пользователей, используя внешний ключ для ссылки на запись пользователя-владельца логина; поскольку один пользователь может иметь несколько учётных записей, здесь реализуется отношение «один ко многим»). То есть мы, добавляя запись в таблицу учётных записей, должны в поле внешнего ключа указать то же значение, которое задано в первой таблице в качестве первичного ключа. Проблема сводится к тому, что в общем случае оператор вставки (INSERT) не возвращает никакого значения, позволяющего сослаться на добавленную запись. Попытаемся рассмотреть проблему чуть глубже.

## Автогенерация ключей

Но для начала более подробно поговорим о том, как выполняется автогенерация ключей в различных СУБД (остановимся только на наиболее популярных открытых SQL-решениях – MySQL, PostgreSQL и Firebird).

В PostgreSQL автоинкремент столбца реализуется с помощью специального типа – serial. Назначение этого типа данных столбцу приводит к автоматическому созданию последовательности (SEQUENCE), из которой при каждом добавлении записи в таблицу выбирается очередное значение типа integer. Механизм последовательностей гарантирует уникальность генерируемых значений в пределах

## Вопросы терминологии

В теории реляционных баз данных принята терминология, несколько отличающаяся от используемой «практикующими» сисадминами. Данные, хранимые в реляционной БД, принято представлять в виде двумерных таблиц, столбцы которых описывают те или иные свойства объектов, информацию о которых мы храним, а строки хранят данные по конкретным объектам. В теории объекты называются сущностями, таблицы – от-

ношениями, столбцы – атрибутами, а строки – кортежами. Однако в практической работе с конкретными СУБД всё же более привычной и устоявшейся является «бытовая» терминология – таблицы (tables), строки/записи (rows/records) и столбцы/поля (columns/fields). Также имеет смысл упомянуть ещё одну группу терминов – соответственно классы (classes), экземпляры (instances) и атрибуты (attributes), характерную для объектного подхода к базам данных.

всей базы данных (то есть он работает вне транзакций). На практике это выглядит так:

```
CREATE TABLE users (id serial, fio varchar);
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

Поскольку столбцу id мы задали тип serial, то при создании таблицы users автоматически будет создана и последовательность users\_id\_seq. Вставка в таблицу новой записи (обратите внимание, что столбец id мы в команде INSERT опускаем) приведёт к автоматическому присвоению столбцу id следующего значения, сгенерированного последовательностью.

При желании аналогичного результата можно добиться, создав вручную последовательность и указав при создании таблицы для соответствующего столбца значение DEFAULT nextval(sequence):

```
CREATE SEQUENCE seq;
CREATE TABLE users (id integer default nextval('seq'),
fio varchar);
```

Firebird предлагает несколько менее удобный механизм – эта СУБД поддерживает так называемые генераторы (GENERATOR, во 2-й версии для них реализован стандартный SQL-синтаксис – SEQUENCE), функция которых аналогична функции последовательностей PostgreSQL. Однако специальный тип данных для автоматического использования генераторов не поддерживается, поэтому разработчик БД должен самостоятельно заботиться о вставке значения, возвращаемого генератором – либо указывать в запросах INSERT значение соответствующего поля (см. пример ниже), либо писать триггеры, эмулирующие работу автоинкрементных типов данных других СУБД.

```
CREATE SEQUENCE users_id_seq;
INSERT INTO users (id, fio) VALUES
(NEXT VALUE FOR users_id_seq, 'Иванов И.И.');
```

Здесь, как видите, столбец id, и его значение нужно указывать (сам id должен иметь один из целочисленных типов, обычно INTEGER или BIGINT). Можно использовать синтаксис версии 1.5 (CREATE GENERATOR users\_id\_seq и GEN\_ID(users\_id\_seq, 1) соответственно), но он является устаревшим, и его рекомендуется избегать (если вам, конечно, не нужен инкремент на значение, отличное от единицы, который в новом синтаксисе пока не поддерживается).

А вот установить GEN\_ID() в качестве DEFAULT-значения в Firebird (аналогично описанному выше методу

для PostgreSQL), к сожалению, не получится – в текущих версиях синтаксис допускает использовать в качестве значения по умолчанию только константу либо контекстную переменную.

СУБД MySQL реализует автоматическую генерацию первичного ключа через механизм автоинкремента. Вам достаточно указать ключевое слово `AUTO_INCREMENT` в описании соответствующего столбца (при этом столбец обязательно должен быть описан и как ключ), и MySQL самостоятельно позаботится о выборе уникального значения:

```
CREATE TABLE users (id int not null auto_increment ↵
    primary key, fio varchar(50));
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

Так же, как и в PostgreSQL, чтобы автоинкремент сработал, в запросе `INSERT` столбец `id` не должен фигурировать (что равнозначно значению `NULL`), либо ему в качестве значения должен передаваться 0 – если указать другое значение, то оно и будет сохранено (если не возникнет противоречия с другими ограничениями и типом столбца).

## Получение значения, установленного автоматически

Теперь перейдём непосредственно к вопросу получения значения ключа только что добавленной в таблицу записи и рассмотрим наиболее популярные методы.

```
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

```
SELECT MAX(id) FROM users;
```

Одно из самых опасных, но, как ни странно, распространённых решений: делаем вставку и следом «быстренько» считываем максимальное значение ключа. В однопользовательских реализациях такой подход допустим (хотя и может приводить к дополнительной нагрузке на СУБД). А вот в системах, где можно установить сразу несколько соединений, нельзя исключать вероятность того, что в промежутке между `INSERT` и `SELECT` одного сеанса вторым пользователем будет выполнена другая вставка, в результате чего максимальное значение идентификатора изменится.

Безусловно, если используются сравнительно «глубокие» уровни изоляции транзакций, этот метод использовать можно. Например, в Firebird на уровне изоляции «`READ COMMITTED`» и выше записи, вставляемые в других сеансах, не будут видны до подтверждения (`COMMIT`) текущей транзакции, так что `MAX(id)` будет возвращать либо максимальное значение, установленное в рамках этой транзакции, либо (если записи ещё не добавлялись) максимальное значение на момент начала транзакции. А вот при уровне «`READ UNCOMMITTED`» уже следует быть осторожным – новые записи, подтверждённые конкурирующей транзакцией, станут доступны сразу.

```
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

```
SELECT id FROM users WHERE fio = 'Иванов И.И.';
```

Данный метод работает только в том случае, если комбинация других полей гарантированно позволяет сослаться на одну и только одну запись. То есть этот метод можно с успехом применять к таблицам, записи которых име-

ют уникальный «естественный» столбец (или уникальную комбинацию столбцов), но который мало пригоден для использования в качестве первичного ключа, и потому на эту роль «назначается» суррогатный ключ.

```
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

```
SELECT LAST_INSERT_ID();
```

Работает только в MySQL (но в некоторых других СУБД есть аналогичные функции, например, `SCOPE_IDENTITY()` в MS SQL для получения значения, присвоенного полю `IDENTITY`, выполняющему функции автоинкремента). Функция возвращает значение, «выданное» полю `AUTO_INCREMENT` в рамках данного сеанса, так что можно не опасаться пересечений с параллельными сеансами, где также выполняется вставка в эту же таблицу. Обратите внимание на одну особенность MySQL – если вы используете синтаксис «множественной» записи, когда одним оператором вставляется сразу несколько строк, то `LAST_INSERT_ID()` возвращает значение автоинкрементного столбца первой (а не последней, как можно было бы ожидать) из вставленных записей.

```
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

```
SELECT currval('users_id_seq');
```

Решение, аналогичное приведённому выше, но для PostgreSQL – после вставки записи в таблицу мы можем запросить текущее значение последовательности («текущее» в рамках данной транзакции, с параллельно выполняемыми запросами конфликтов не будет). Правда, чтобы `currval()` вернула значение, вызову этой функции в рамках данной транзакции должен предшествовать явный или неявный (при автоматической генерации значения для `serial`-столбца) вызов функции `nextval()`.

PostgreSQL помимо работы с последовательностями предоставляет и более универсальное решение – позволяет получить внутренний уникальный идентификатор (OID) последней вставленной в рамках данной транзакции строки независимо от наличия в таблице `serial`-полей. Правда, для этого таблица должна быть создана с поддержкой OID (начиная с 8-й версии все таблицы по умолчанию создаются без OID, так что нужно либо использовать фразу `WITH OIDS` в операторе `CREATE TABLE`, либо раскомментировать строку «`#default_with_oids = true`» в `postgresql.conf`: тогда все таблицы будут создаваться с полем OID). Например, так это выглядит в Python-скрипте, использующем модуль `pgdb`:

```
import pgdb
db = pgdb.connect(user='user', database='database')
cr = db.cursor()
cr.execute('INSERT INTO users (fio) VALUES ↵
    ('Иванов И.И.')')
# Здесь получаем OID вставленной записи
lastoid = cr.lastrowid
cr.execute('SELECT id FROM users ↵
    WHERE oid = %d' % lastoid)
id = cr.fetchone()[0] # А это – искомое значение ключа
cr.execute('INSERT INTO logins (id, login) ↵
    VALUE (%d, 'ivanov')' % id)
cr.close()
db.commit()
db.close()
```

Другие интерфейсные модули могут предоставлять аналогичные функции под другим именем (скажем, метод



pg\_oid\_status в драйвере DBD::Pg языка Perl, pg\_last\_oid() в PHP, и т. п.).

```
INSERT INTO users (fio) VALUES ('Иванов И.И.');
```

```
SELECT GEN_ID(users_id_seq, 0);
```

Казалось бы, аналогичное решение для Firebird – делать вставку (предполагая, что поле id заполняется триггером, использующим генератор users\_id\_seq) и запрашивая текущее значение генератора. Но нет! Функция GEN\_ID() возвращает глобальное текущее значение – если другой пользователь в другом сеансе (вне зависимости от уровня изоляции транзакции) воспользуется этим же генератором, то текущее значение изменится и в вашем сеансе! То есть вы получите самое последнее сгенерированное значение в рамках всей БД, а не в рамках вашей транзакции, как это имеет место быть для currval() в PostgreSQL.

```
INSERT INTO users (fio) VALUES ('Иванов И.И.') RETURNING id;
```

Одно из самых простых и логичных решений – ключевое слово RETURNING позволяет вернуть приложению любую комбинацию полей только что вставленной записи, в том числе и заполняемых автоматически. Остаётся лишь обработать возвращаемое значение обычным образом (так же, как при отправке SELECT-запроса). К сожалению, данный синтаксис не является стандартным и реализуется как расширение лишь некоторыми СУБД (например, PostgreSQL начиная с версии 8.2 и Firebird начиная с версии 2.0). Если когда-нибудь этот синтаксис войдёт в стандарт SQL, методы, описанные в данной статье, видимо, потеряют свою актуальность.

## Предварительная генерация ключа

Как видите, получить идентификатор только что вставленной записи в общем случае проблематично. В конкретных условиях можно воспользоваться встроенными возможностями СУБД, при условии что используемые интерфейсные модули их поддерживают.

Однако, имея в виду, что в большинстве случаев необходимо просто знать значение ключа соответствующей записи, можно пойти другим путём – сначала генерировать это значение в приложении, а потом лишь вставлять его во все нужные таблицы, не полагаясь на возможности СУБД. Решений здесь тоже несколько.

```
SELECT MAX(id) FROM users
```

```
INSERT INTO users ...
```

Ничуть не лучше нашего первого примера, с той лишь разницей, что теперь существует весьма высокая вероятность, что несколько клиентов одновременно попытаются вставить записи с одним и тем же значением ключа (глубокая изоляция транзакции лишь усугубляет ситуацию). Хорошо, если столбец id имеет ограничение уникальности (явный UNIQUE либо в составе ограничения PRIMARY KEY) – тогда ничего страшного не произойдёт (при условии, конечно, что приложение корректно обрабатывает исключения). Но в целом этот метод не рекомендуется к использованию.

```
SELECT nextval() FROM users_id_seq
```

```
INSERT INTO users...
```

Это пример для PostgreSQL, но данный подход применим к любой СУБД, поддерживающей работу с последовательностями или генераторами – сначала получаем следующее значение последовательности (сама последовательность при этом «сдвигается» дальше, так что можно быть уверенным, что никакой другой пользователь это же значение уже не получит), а затем вручную передаём его при вставке записей во все связанные таблицы.

В Firebird следующее значение генератора, как уже упоминалось, можно получить функцией GEN\_ID(users\_id\_seq, 1), где 1 – приращение генератора, либо «стандартным» синтаксисом – NEXT VALUE FOR users\_id\_seq (появился в версии 2.0). Поскольку генераторы работают вне транзакций, можно быть уверенным, что ни в каком другом сеансе данное значение выдано уже не будет.

Адекватного решения для MySQL, позволяющего получить будущее значение автоинкрементного столбца, мне найти не удалось. Но в качестве обходного пути можно предложить нечто не слишком красивое, но работающее:

```
CREATE TABLE users_id_seq (id int not null);
```

```
INSERT INTO users_id_seq VALUES (0);
```

Теперь получить следующее значение «последовательности» можно так:

```
UPDATE users_id_seq SET id = last_insert_id(id + 1);
```

```
SELECT last_insert_id();
```

Казалось бы, можно обновлять столбец как id = id + 1 и потом получать значение id. Но в многопользовательском приложении это небезопасно, а last\_insert\_id() как раз гарантирует «атомарность» всей операции в рамках текущего соединения. Теперь осталось лишь воспользоваться сгенерированным значением для вставки данных во все связанные таблицы. Наконец, можно генерировать значение ключа вообще только силами приложения (например, как функцию текущего времени и номера сеанса, чтобы обеспечить необходимую уникальность). В этом случае мы вообще никак не будем зависеть от возможностей той или иной СУБД, а также сможем при необходимости генерировать значения не только числового типа.

Итак, в случае «предгенерации» ключа проблема тоже вполне решается, но по-прежнему требуются учёт специфики используемой СУБД либо дополнительные «извращения».

## Заключение

Как видите, проблема «поиска» автоматически сгенерированного значения первичного ключа довольно многогранна и имеет не одно решение, и в целом её решение сильно зависит от конкретной реализации СУБД. Окончательный вердикт о том, как именно следует поступать, выносить не буду – да это и невозможно, поскольку в каждой конкретной ситуации могут быть свои «за» и «против» при обсуждении того или иного подхода. Пожалуй, главное, о чём не следует забывать, так это о многопользовательской природе почти всех активно используемых в наши дни систем управления базами данных. 