Въведение в…

SQL

Съдържание

За какво служи SQL?

SQL е език за работа с релационни бази данни (БД). Релационните БД се състоят от таблици, които съдържат някаква информация. Езикът SQL ни предоставя средства за добавяне, преглеждане, променяне, изтриване на данни от БД.

Резюме

В тази статия ще ви запознаем накратко с релационните бази от данни и с основните операции, които можем да извършваме над тях чрез езика SQL. Статията не навлиза в големи подробности и много от важните възможности на езика остават встрани от нейния фокус. Така обръщаме внимание на основните неща, без доброто разбиране на които, по-нататъшната работа с бази от данни става доста по-трудна.

История

Съществуват различни модели на представянето на данните в БД – плосък (flat), йерархичен (hierarchical), пространствен (dimensional) и др. В началото на 70те години британският учен д-р Едгар Код публикува статия за релационния (relational) модел в базите данни. При него данните са структурирани в таблици, като допълнително са дефинирани връзки между таблиците. Заедно с релационния модел д-р Код предлага и език, наречен DSL/Alpha, за манипулиране на данните в базата. Скоро след публикацията IBM назначават екип за разработка на прототип, базиран на идеите в нея. Този екип създава езикът SQUARE – опростена версия на DSL/Alpha. След това SQUARE се развива до езика SEQUEL (Structured English Query Language), който в последствие е преименуван на по-краткото SQL, защото думата SEQUEL била запазена от британска авиокомпания.

През 1986г. SQL е стандартизиран от ANSI, а през 1987г. и от ISO. През 1989, 1992, 1999 и 2003 стандартът е доразвит до сегашната си версия, която можете да намерите на това място в on-line варианта на статията.

Основи на SQL

SQL определя резултата, не стъпките

Съкращението SEQUEL би могло да се преведе като "език за структурирани заявки на английски". И наистина кодът на SQL във своята същност представлява заявки т.е. заповедни изречения – въведи това, изтрий онова... SQL се различава от езици като C или Java по това, че кодът не определя какви действия трябва да се извършат, а само желания резултат. Ако, например, искаме да вземем всички записи в базата, за които е изпълнено условието X, не трябва да пишем цикъл, който ги обхожда и проверява условието за всеки поотделно, а направо правим заявка: "искам всички записи, които отговарят на условието X". Оттам нататък за конкретните стъпки по изпълнението на тази заявка се грижи програмата, която обслужва базата данни (database engine).

SQL е неизменна част от БД

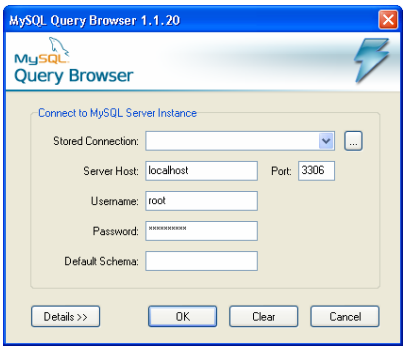
SQL е силно свързан с релационните бази от данни. Направен е за тях и отразява фундаменталните възможности и функционалност, които те предоставят. По-сложните езици за работа с бази от данни най-често просто надграждат SQL. Разбирането на SQL до голяма степен се припокрива с разбирането на релационните бази от данни – в базите всичко се държи в таблици, в SQL резултат е таблица; типовете на полетата в базите, са същите и в SQL; възможните операции в базите, са възможните операции и в SQL.

Какви знания ми трябват?

Езикът SQL е в основата на работата с релационни бази данни. Затова за разбирането на тази статия не са ви необходими предварителни знания.

Инсталиране

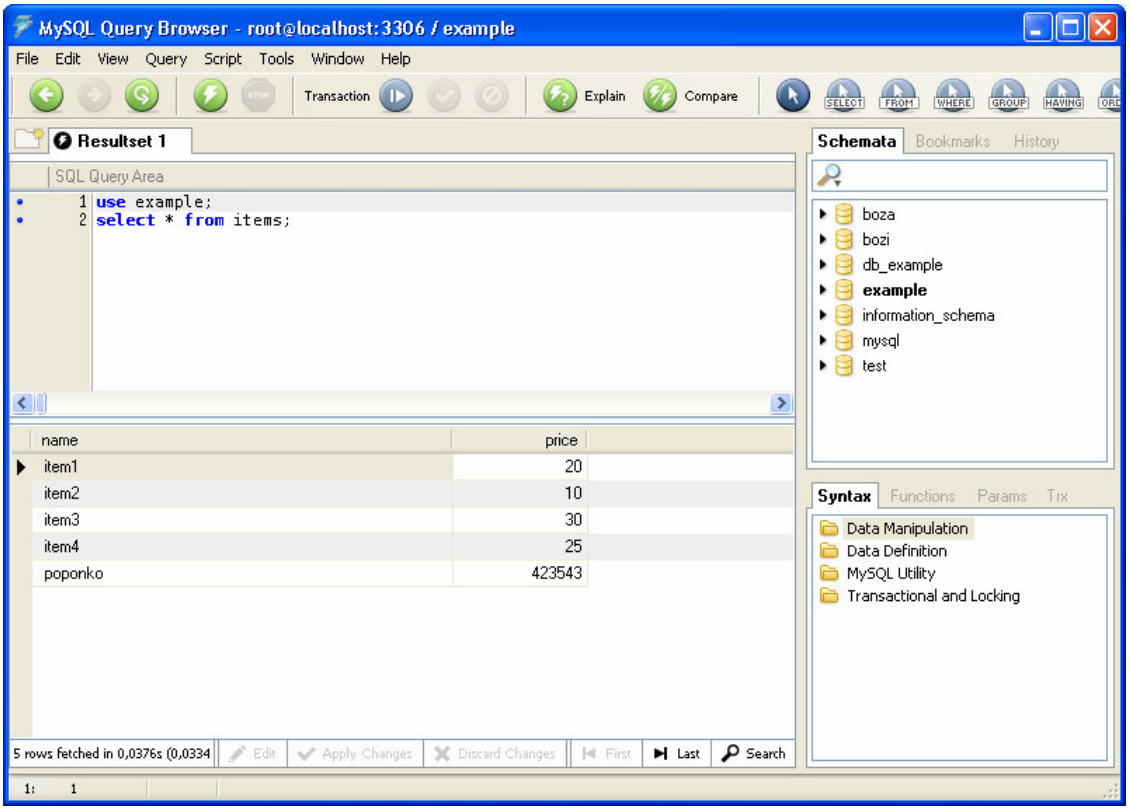
За да разполагате с база данни трябва да си инсталирате сървър за нея. Една безплатна възможност е MySQL. На адрес http://dev.mysql.com/downloads/ можете да намерите както последната версия на MySQL Server, така и инструменти за работа с него. Един такъв инструмент е MySQL Query Browser, който ще използваме, за да пишем заявки към базата данни.



Инсталирането на MySQL Server и MySQL Query Browser става лесно и бързо. Когато стартирате MySQL Query Browser трябва да въведете адреса на сървъра (в нашия случай локалния компютър), име и парола.

При инсталацията на MySQL Server трябва да сте въвели парола за root акаунт. Тогава срещу *Username* пишете root, а срещу *Password* – вашата парола. След натискане на бутона OK ще се отвори средата на MySQL Query Browser.

Работното поле е разделено на две големи части – едната за заявките, другата за резултата от тях. Вдясно има списък с всички бази на сървъра, към който сте се свързали:



Релационни бази данни

Преди да започнем да работим с релационни бази данни, трябва да се запознаем с принципите на организацията на данните в тях.

Таблици

Релационната база данни се състои от таблици. Всяка таблица си има име, по което може да бъде идентифицирана. Таблиците се състоят от редове, като всеки ред е един запис в базата, а клетките от таблицата са отделните полета в него. Таблиците имат по една или повече колони, като всяка колона има име и тип. Колоните определят съответните полета в записите от базата.

Таблицата има фиксиран брой колони (полета) и променлив брой редове (записи). Прост пример е таблица с две колони – едната за име на някаква стока, другата за цена:

name	price
item1	20
item2	10
item3	30
item4	25

На всеки ред от таблицата е описана по една стока – чрез името и цената си. Можем да добавяме и махаме стоки, но ако променим колоните, това вече се превръща в различна таблица.

Типове

Всяка колона определя типа на поле от запис. Типовете могат да са целочислени или числа с плаваща запетая, за символи и низове, за време и дата и т.н. Няма стандартни типове, които да могат да се ползват във всяка една база от данни. В повечето случаи има съответствия между типовете в различните бази, но най-малкото запазените думи, с които се декларират, може да са различни. Все пак принципът на работа е един и същи, затова ще се спрем на най-основните типове и в примерите ще работим с тях.

INTEGER

За цели числа се използва типът **INTEGER**. В MySQL типът **INT** е същия като **INTEGER**.Той е четири байтов и може да се използва за числа в интервала (-2147483648, 2147483647). Съществуват и други типове – **TINYINT** (1 байт), **SMALLINT** (2 байта), **BIGINT** (8 байта). Ако добавим думата **UNSIGNED** след някой от тези типове, тогава множеството от стойности ще има същият брой елементи, но започващи от 0 – например **INTEGER UNSIGNED** ще съдържа числата от 0 до 4294967295.

FLOAT

За дробни числа се използва типът **FLOAT**. Той обхваща интервалите [-3.402823466E+38; -1.175494351E-38], 0 и [1.175494351E-38; 3.402823466E+38]. С приблизително двойно по-голяма точност е типът **DOUBLE**. За тези типове също важи запазената дума **UNSIGNED**, както и при **INTEGER**.

VARCHAR

Символни последователности можем да съхраняваме в полета от тип **VARCHAR**. При деклариране на колона от такъв тип, в скоби се отбелязва максималния брой символи за съответната колона – например **VARCHAR**(**32**).

Стойността null

В базите от данни има една специална стойност, която може да бъде във всяко поле, независимо от какъв тип е. Това е стойността null. Ако по някаква причина не зададем стойност на поле от даден запис, това поле получава служебната стойност null. Тя означава, че това поле не е дефинирано и не е равно по стойност на никое друго поле. При дефинирането на колони можем да добавим израза **NOT NULL**, което ще забрани записи със стойност null в съответното поле.

Primary и foreign key

За да няма повтарящи се записи в дадена таблица, трябва да има нещо, по което да се идентифицират различните записи. Обикновено някоя колона се маркира като **primary key** (PK), което забранява два записа в таблицата да имат една и съща стойност в полето от тази колона. Възможно е няколко колони да бъдат PK, като по този начин се получава съставен първичен ключ. Това, обаче, е най-често е признак на лош дизайн на вашата база и най-вероятно дадената таблица може да бъде разделена на няколко по-малки. Затова ще приемем, че всяка таблица има точно една колона, която е PK и в която няма повтарящи се стойности.

В дадена таблица, някои от полетата може да са такива, които съответстват на полета от други таблици. Например в базата на МВР, в таблицата с действащите превозни средства може да има колона за собственик, в която да се записва ЕГН. В друга таблица са записани данните на гражданите, като PK е колона отново с ЕГН. Тогава полето от първата таблица е foreign key (FK), защото съответства на PK от друга таблица.

Ето една примерна таблица с хора, които са закупили дадена стока:

person	item
gosho	item1
gosho	item3
gosho	item5
pesho	item2
pesho	item5
ivan	item4
ivan	item1

name	price
item1	20
item2	10
item3	30
item4	25

Колоната item в първата таблица съответства на колоната name във втората. Полето name е PK, защото не може да има две стоки с едно и също име или пък една стока с няколко цени. Съответно полето item е FK, защото съответства на name. Забележете, че в първата таблица нито една от двете колони не може да е PK, защото един човек може да закупи много стоки и една стока може да бъде купена от много хора. Затова първата таблица или ще е без PK, или двете колони заедно трябва да са PK.

Основни заявки в SQL

Създаване и изтриване на база

Първата стъпка при работата ни с нова база от данни, е да създадем празна база. В SQL създаваме със запазената дума **CREATE**:

```
CREATE DATABASE db_example;
```

Напишете този ред в прозореца за заявки и натиснете бутона Execute или *Ctrl+Enter* от клавиатурата. Ако всичко е наред, вече имате създадена нова база, която няма таблици в себе си. Името и е db_example. Ако от контекстното меню на списъка с базите изберете Refresh, db_example ще се появи наред с останалите в списъка. За да използвате тази база от данни трябва да укажете това със следната заявка:

```
USE db_example;
```

След тази заявка сървърът ще знае, че следващи заявки се отнасят за таблиците в тази база, а не за някоя друга.

Аналогично на създаването става и изтриването на база:

```
DROP DATABASE db_example;
```

При изпълнението на тази заявка, цялата база, с всички таблици в нея, ще бъде изтрита.

Създаване и изтриване на таблица

Вече имате създадена база, указали сте, че ще използвате нея и е време да създадете и таблици. Създаването и изтриването на таблица е аналогично на предните заявки. Тук трябва да зададем имена и типове на колоните:

```
CREATE TABLE items (
  name VARCHAR(20),
  price INTEGER UNSIGNED
);
```

След тази заявка се създава нова таблица с име items. Тя има две колони – name, от тип **VARCHAR**(**20**) и price от тип **INTEGER**. Дефинициите на колоните са в кръгли скоби след името на таблицата и са разделени със запетаи. Всяка дефиниция на поле започва с име, след това тип и други параметри, разделени с интервали. Добре е да дефинираме PK за тази таблица. По време на нейното създаване това става така:

```
CREATE TABLE items (
  name VARCHAR(20) PRIMARY KEY,
  price INTEGER UNSIGNED
);
```

Ако вече имаме таблицата и искаме да я променим като добавим PK, може да заявим това чрез следната заявка:

```
ALTER TABLE items ADD PRIMARY KEY (name);
```

В скобите указваме кои колони да бъдат PK. Тази заявка ще бъде изпълнена успешно само ако тези колони еднозначно определят записите в таблицата, т.е. когато няма повторения.

Така създадената таблица е празна – няма записи, т.е. редовете в нея са нула на брой. За добавяне на записи трябва да се използва запазената дума **INSERT**.

Изтриване на таблици става отново чрез запазената дума **DROP**:

```
DROP TABLE items;
```

INSERT заявки

Записи в дадена таблица се добавят чрез запазената дума **INSERT**:

```
INSERT INTO items (name, price) VALUES ('item1', 20);
INSERT INTO items (price, name) VALUES (20, 'item1');
INSERT INTO items VALUES ('item1', 20);
```

И трите реда от горния пример правят едно и също – в таблицата items добавят един запис, в който полето **name** има стойност *'item1'*, а полето ***price*** има стойност *'20'*. В скобите след името на таблицата изреждаме полетата, на които ще дадем стойност, а в скобите след **VALUES** – съответните стойности. На третия ред не са указани колони и в този случай се имат предвид всички колони на таблицата в реда, в който са били дефинирани при нейното създаване. Ако не дефинираме стойност за някоя от колоните, то тя получава стойност *null*.

Тъй като горните три заявки правят едно и също, те не могат да се изпълнят една след друга. Ако се опитаме да направим това, след първата от тях вече ще има запис за продукта item1. При опит втори път да добавим този запис, ще получим съобщение за грешка, защото колоната name е PK и не се позволява да има два продукта с еднакво име.

Запазената дума **INSERT** може да се използва и в друга форма:

```
INSERT INTO items SELECT …;
```

Тук **SELECT** заявката връща като резултат цяла таблица, всички записи на която се добавят в таблицата items.

SELECT заявки

За да извличаме данни от базата се използва ключовата дума **SELECT**. Най-простият неин вариант е следният:

```
SELECT * FROM items;
```

Тази заявка в буквален превод означава "избери всичко от items" и като резултат връща цялата таблица items. За да изберем само няколко колони, ги изреждаме на мястото на звездичката (символа '*'), разделени със запетаи. Например, за да видим само имената на всички продукти ще използваме следната заявка:

```
SELECT name FROM items;
```

Ако пък искаме да получим таблица, в която на първо място са цените, а след това имената пишем следното:

```
SELECT price, name FROM items;
```

По този начин можем да избираме и подреждаме колони. За да вземем част от редовете, трябва да поставим някакво по-абстрактно условие, защото редовете нямат номера или имена:

```
SELECT name FROM items WHERE price < 50 AND price > 10;
```

Тази **SELECT** заявка ще върне като резултат таблица с една колона, която съдържа имената на стоките, чиято цена е между 10 и 50. За поставяне на условия се използва се използва ключовата дума **WHERE**. След нея е записано някакво условие, логически израз, който се прилага на всеки един от записите. Ако резултатът е "истина", записът участва в резултата, а ако е "лъжа" се пренебрегва.

За да получим записите на таблицата в определен ред, можем да използваме **ORDER BY**:

```
SELECT price, name FROM items WHERE price < 50 AND price > 10
ORDER BY price DESC, name;
```

На края на **SELECT** заявката добавяме **ORDER BY** и след това изреждаме по кои колони да се сортира. В случая първо сортираме по price, а ако има няколко записа с еднаква цена, сортираме по name. По подразбиране сортирането става в нарастващ ред. Ако искаме някоя колона да се сортира в намаляващ, след името и добавяме думата **DESC**.

Друга полезна запазена дума е **LIMIT**. Тя служи за ограничаване на броя на записите в резултата от **SELECT** заявка. Например, ако искате само първите пет реда от горната заявка (петте най-скъпи стоки), тя би изглеждала така:

```
SELECT price, name FROM items WHERE price < 50 AND price > 10
ORDER BY price DESC, name LIMIT 5;
```

DELETE заявки

Изтриването на записи е аналогично на избирането на записи. Ако вземем една **SELECT** заявка и частта преди **FROM** заменим с **DELETE**, тогава всички записи, които щяха да участват в резултата на **SELECT**, ще бъдат изтрити:

```
DELETE FROM items WHERE price < 50 AND price > 10;
```

Тази заявка изтрива всички записи, в които цената е между 10 и 50.

Ако в **DELETE** заявка няма условие **WHERE**, всички записи от дадената таблица ще бъдат изтрити и тя ще остане празна. Забележете, че това е различно от заявката **DROP**, при която цялата таблица се изтрива.

Разширения на заявките

SELECT заявка за две таблици

До този момент разгледахме заявки, които се отнасяха за отделни таблици и ги разглеждаха сами за себе си. Обикновено таблиците в една база от данни имат много връзки помежду си и тези връзки неминуемо се отразяват и на заявките, които пишем. Нека сега създадем следната таблица:

```
CREATE TABLE stores (
  id INTEGER UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(20),
  price_limit INTEGER UNSIGNED
);
```

Това е таблица с магазини, всеки от които си има уникален номер, име и ограничение за най-скъпа стока. Номерът е уникален, затова е PK. Запазената дума **AUTO_INCREMENT** означава, че ако не се задава стойност на това поле, то автоматично ще получи номер, с едно по-голям от това на предния създаден запис. Тази таблица ще ни е нужна за следващите примери.

Казахме, че таблиците в една база от данни обикновено имат множество връзки помежду си. Те могат да са както явно зададени, така и просто логически. Пример за логическа връзка са двете таблици items и stores, за които няма конкретно дефинирани връзки, но можем да напишем следната заявка:

```
SELECT * FROM stores, items WHERE price_limit >= price;
```

id	name	price_limit	name	price
2	City MOLL	200	item1	100
2	City MOLL	200	item2	70
2	City MOLL	200	item3	50
4	big shop	60	item3	50
2	City MOLL	200	item4	10
3	shop	40	item4	10
4	big shop	60	item4	10
5	market	30	item4	10
7	local store	20	item4	10
8	local store	30	item4	10
2	City MOLL	200	item5	50
4	big shop	60	item5	50

Чрез тази заявка ще получим като отговор всички двойки от магазини и стоки такива, че стоката не превишава границата за най-скъпа стока в магазина. За да се изпълни тази заявка, се създава временна таблица, в която са комбинирани всички редове от едната с всички от другата и съдържа колоните и от двете таблици.

Пълни имена

В примера можем да забележим, че в новополучената таблица има две колони с име name. Това води до известни проблеми, ако искаме например да напишем нещо такова:

```
SELECT * FROM stores, items WHERE name = name;
```

За да можем да разграничаваме двете колони с еднакви имена, трябва да укажем от коя таблица е колоната:

```
SELECT * FROM stores, items WHERE stores.name = items.name;
```

Псевдоними

В SQL заявките често се налага да пишем дълги имена, както в горния пример. За улеснение и по-добра четимост, на всяка таблица можем да съпоставим псевдоним:

```
SELECT * FROM stores s, items i WHERE s.name = i.name;
```

След името на таблицата, от която ще извличаме информация, пишем нейния псевдоним. В случая псевдонимът на stores е s, а на items – i.

Имена на колоните

Дори и да използваме псевдоними, в таблицата, която получаваме като резултат, ще има еднакви колони. Това можем да избегнем като им дадем нови имена:

```
SELECT
   s.name AS store_name, i.name AS item_name, price
FROM stores s, items i WHERE price_limit >= price;
```

Задаването на нови имена става чрез запазената дума **AS**, като след нея указваме името, под което искаме да се появи в новата таблица. Резултатът от горния пример ще съдържа три колони: *store_name*, *item_name* и *price*.

SELECT от SELECT

Както вече знаем, **SELECT** заявките връщат като резултат таблица. Поради тази причина нищо не пречи да ги влагаме една в друга по следния начин:

```
SELECT name
FROM (SELECT name, price AS pr FROM items WHERE price < 50) a
WHERE a.pr > 10;
```

Забележете, че таблицата резултат от вложения SELECT трябва да си има псевдоним (в случая **a**). След като сме дали друго име на колоната *price*, вече трябва да се обръщаме към новото – *pr*. Влагането понякога може да бъде много полезно за логическото разграничение на нещата, но в случая същия резултат можехме да получим и по-лесно:

```
SELECT name FROM items WHERE price < 50 AND price > 10;
```

JOIN на таблици

Думата join означава свързвам, съединявам. Същото означава и запазената дума **JOIN** в SQL, където се използва за съединяване на таблици. За следващите примери ще имаме нужда от следната таблица:

```
CREATE TABLE items_in_stores (
  item VARCHAR(20),
  store_id INTEGER UNSIGNED,
  PRIMARY KEY (store_id, item),
  CONSTRAINT FK_to_stores FOREIGN KEY FK_to_stores (store_id)
    REFERENCES stores (id),
  CONSTRAINT FK_to_items FOREIGN KEY FK_to_items (item)
    REFERENCES items (name)
);
```

Таблицата съдържа наредени двойки от стоки и магазини, които определят коя стока в кой магазин е налична. Дефинираме съставен PK, който включва и двете колони, защото една стока може да е в няколко магазина и един магазин може да съдържа няколко стоки. Запазената дума **CONSTRAINT** се използва за дефиниране на ограничения, които в случая са две дефиниции на FK – store_id е FK към полето id в stores, а item е FK към полето name в items. В случая тези дефиниции на FK са само информативни, без да оказват някакво влияние на таблицата и операциите с нея.

Можем да я попълним като заредим всички възможни магазини с всички възможни стоки ето така:

```
INSERT INTO items_in_stores
SELECT
   i.name AS item,
   s.id AS store_id
FROM stores s, items i WHERE s.price_limit >= i.price;
```

INNER JOIN

Първия вид **JOIN**, който ще разгледаме е **INNER JOIN**. Този вид **JOIN** съединява две таблици по зададени по една колона от всяка от тях. Всеки два реда, за които стойностите в тези колони съвпадат, се 'залепят' един за друг. Като резултат получаваме таблицата от така получените редове. Следващите две заявки правят едно и също:

```
SELECT *
FROM items a, items_in_stores b
WHERE a.name = b.item;
```

```
SELECT *
FROM items a INNER JOIN items_in_stores b ON a.name = b.item;
```

Синтаксисът на JOIN заявките е:

[таблица1] **[някакъв] JOIN** [таблица2] **ON** [колона1] = [колона2];

Така указваме, че съединяваме таблиците 1 и 2 по колони 1 и 2. Виждаме, че това може да се постигне и с обикновена **SELECT** заявка. На практика разлика между едното и другото няма, но логически има значение кое от двете ще изберем. При първия пример таблицата, над която работим, е декартовото произведение на двете таблици и допълнително налагаме условие над нейните записи. При втория пример таблицата, над която работим е вече получената с това условие. Важно е да разграничаваме логически над коя таблица искаме да работим, за да са по-ясни и лесни за разбиране нашите заявки. Нека например поискаме цените на всички стоки, налични в магазина с номер 2:

```
SELECT price
FROM items a, items_in_stores b
WHERE a.name = b.item AND b.store_id = 2;
```

```
SELECT price
FROM items a INNER JOIN items_in_stores b ON a.name = b.item
WHERE store_id = 2;
```

Тук отново двата примера връщат един и същи резултат. В първия пример, обаче, нещата започват да стават неясни, защото двете условия след **WHERE** нямат логическа връзка. При втората заявка всичко е ясно обособено и по-лесно за разбиране.

OUTER JOIN

При **INNER JOIN** заявките резултатът съдържа само записи, които имат съответствие и в двете таблици. Понякога обаче, искаме да вземем всички редове от дадена таблица и да им съпоставим редове от другата, само ако е възможно. Например, искаме за всички продукти справка в кои магазини са налични, а ако няма такива магазини, полето да е със стойност null. Този ефект ще постигнем със следната заявка:

```
SELECT *
FROM items a LEFT OUTER JOIN items_in_stores b ON a.name = b.item;
```

Запазената дума **LEFT** се използва, за да укаже, че всички записи от лявата таблица (в случая items) трябва да присъстват в резултата. Ако вместо това използваме **RIGHT**, тогава всички редове от дясната таблица ще участват в резултата. Ако на всеки запис от първата таблица (която сме определили с **LEFT** или **RIGHT**) съответства запис от втората, тогава **OUTER JOIN** ще дава същите резултати като **INNER JOIN**.

Влагане

Също както и при **SELECT**-ите, резултатът от **JOIN**-овете е таблица. Следователно, навсякъде, където трябва да сложим име на таблица, можем да напишем както **SELECT**, така и **JOIN**. Това нерядко може да се случи. Например, ако искаме да боравим с имената на магазините, а не техните id номера, трябва един допълнителен **JOIN**:

```
SELECT items.name AS iname, items.price, stores.name AS sname
FROM
  (items JOIN items_in_stores ON name = item)
  JOIN
  stores
  ON store_id = id;
```

Резултатът от тази заявка съдържа точно толкова реда, колкото и таблицата items_in_stores, като във всеки ред е заместен номера на магазина с неговото име и е добавена цената съответната стока. Ако ни е по-удобно да работим с имената на магазините, можем да използваме резултата от този **SELECT** в други наши заявки.

Бъдещето на SQL

SQL е от малкото компютърни езици, които остават живи и актуални в продължение на десетилетия. Това се дължи както на неговата простота и интуитивен синтаксис, така и на силната му връзка с релационните бази от данни. Тези негови качества го правят мощен инструмент за работа с бази от данни. SQL върви ръка за ръка с релационните бази от данни и, поне засега, няма причина да бъде заменен с нещо друго докато те се използват.

Полезни връзки

http://www.w3schools.com/sql/ - това е сайт, на който достъпно и с много примери са обяснени основните конструкции в SQL, както и такива, които не са включени в тази статия

MySQL Query Browser – в долния десен край, под областта с наличните бази, има дефиниции и пояснения за заявки, които са валидни в MySQL