

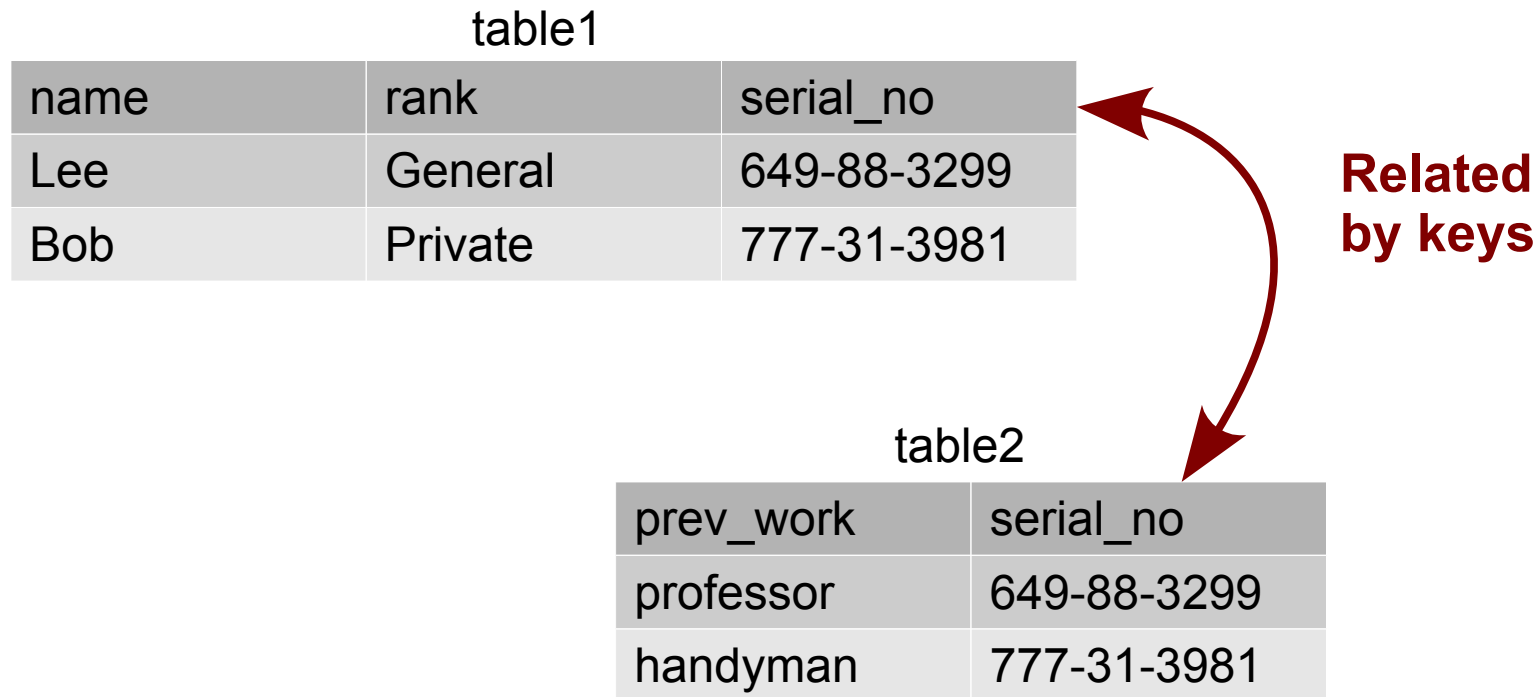
Relational databases and SQL

What is a relational database?

[The relational data model comprises] data items organized as a set of formally described tables from which data can be accessed or reassembled in many ways without having to reorganize the database tables. Each table (sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns.

A database organized according to the relational model is called a relational database.

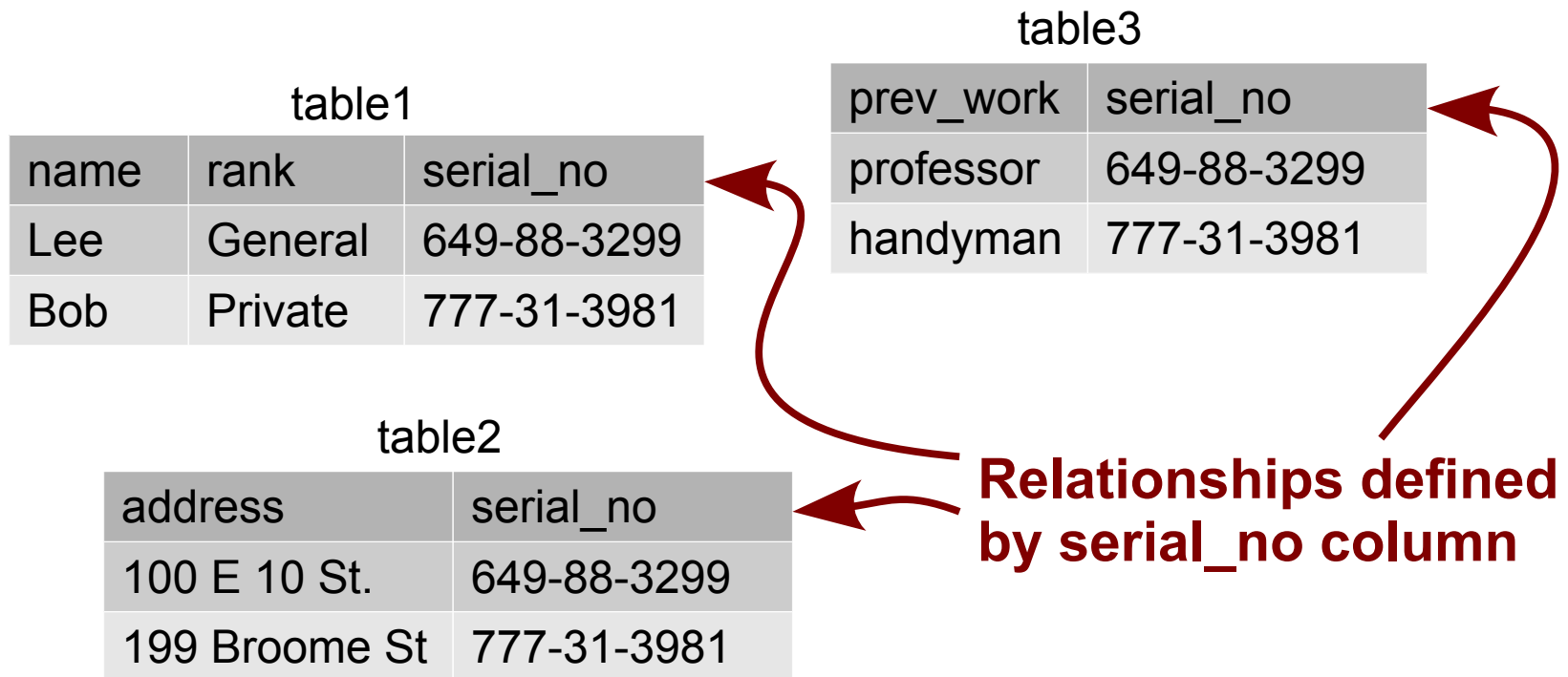
In the relational model, related records are linked together with a "key".



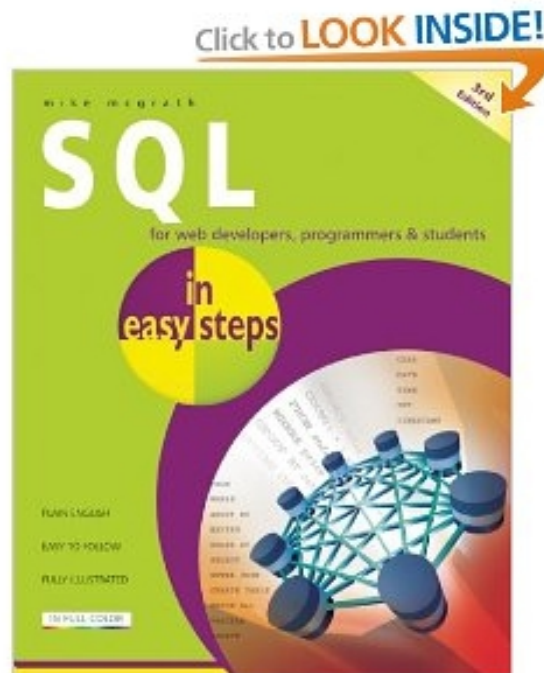
Structured query language (SQL) is the most common means of querying relational databases

Normalization

Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency. Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them. **The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.**



An easy book on SQL



[Share your own customer images](#)


[Search inside this book](#)

SQL in Easy Steps [Paperback]

[Mike McGrath](#) (Author)

[Be the first to review this item](#) |  Like (0)

List Price: ~~\$14.99~~

Price: **\$11.69** 

You Save: **\$3.30 (22%)**

Only 6 left in stock (more on the way).

Ships from and sold by **Amazon.com**. Gift-wrap available.

Want it delivered Monday, December 10? Express delivery is available to 10003. Order it in the next **13 hours and 26 minutes**, and choose **Local Express Delivery** at checkout.

23 new from **\$8.17** **6 used** from **\$11.61**

**12 Days,
12 Deals**



Each Day, a Best Seller for 75% Off or More

From 9 a.m. to 1 p.m. Pacific time, December 3-14, we'll be offering a select best seller in Books at 75% or more off its cover price--while supplies last. Don't miss your chance at unbelievable savings during **12 Days, 12 Deals**.

> [See more product promotions](#)

Reasons for using a traditional relational database and SQL

- Easy access across languages and platforms.
- Rich query language with powerful capabilities.
- Data integrity across multiple, potentially distributed processes.
 - What if we had to write our own access to databases that were simultaneously being used by several processes? It would be hard to keep a consistent state.
- Easy ad hoc query capability.
- Easy maintenance.

Reasons for NOT using a traditional relational database and SQL

- Fast reads / writes (as in simulation)
- Fast access of pre-populated data structures (NoSQL movement)
- High availability persistence
 - Example: You are a market maker who takes no overnight positions. Your machine crashes at 345pm.
 - Can you bring it back in time to dump your positions? Every second of delay increases your rate of trading and, hence, your temporary impact!
 - Hence, waiting means more than just added risk. It means an expectation of higher costs!

Installing MySQL

- Downloading
 - Mac / Linux
 - <http://dev.mysql.com/downloads/mysql>
 - Windows
 - <http://dev.mysql.com/downloads/installer>
- Just for the sake of making this example simple, I installed mySQL as the root user
 - For real applications, you should think hard about permissions and access privileges

Starting SQL server and client

- Starting server (from terminal)

```
sudo /usr/local/mysql/bin/mysqld_safe
```

<Enter root password>

Pause the task with: Ctrl-Z

Background task with: **bg**

- Starting client

```
/usr/local/mysql/bin/mysql -u root
```

- The windows client is a separate program so command line statements are not required to start it
- After client and server are started, we can issue commands to the client that are processed by the server
- To shut down server

```
sudo /usr/local/mysql/bin/mysqladmin shutdown
```

Creating databases

```
CREATE DATABASE findata;  
USE findata;  
DROP DATABASE findata;
```

Creating tables and deleting tables

```
CREATE TABLE trades (  
    id INT,  
    secsFromEpochToMidN INT,  
    msecFromMidN INT,  
    price FLOAT,  
    size INT  
);
```

```
DROP TABLE trades;
```

Inserting rows into tables

```
INSERT INTO trades (  
    id,  
    secsFromEpochToMidN,  
    msecFromMidN,  
    price,  
    size  
) VALUES (  
    323,  
    15000,  
    34200000,  
    88.10,  
    300  
);
```



We want to retrieve this
record using Java

Selecting rows from tables

```
SELECT * FROM trades;
```

```
SELECT * FROM trades  
WHERE price = 88.10
```

Potential round-off
error



```
SELECT * FROM trades  
WHERE price > 88.0999 AND price < 88.1001
```

```
SELECT * FROM trades, quotes  
WHERE trades.id = 323 and quotes.id = 323
```

Join data from two
tables. This join
is called an *inner join*.



Join types

- Inner
- Equi-join
 - Natural join
- Cross join
- Outer join
 - Left outer join
 - Right outer join
 - Full outer join

[http://en.wikipedia.org/wiki/Join_\(SQL\)](http://en.wikipedia.org/wiki/Join_(SQL))

SQL from Java: *MySQL_ex1.java*

```
// This will load the MySQL driver
Class.forName("com.mysql.jdbc.Driver");


// Setup the connection with the DB
connect = DriverManager.getConnection(
    "jdbc:mysql://localhost/findata?user=root"
);

// Statements allow us to issue SQL queries to the database
statement = connect.createStatement();

// Get the results of the SQL query
resultSet = statement.executeQuery(
    "SELECT * FROM trades WHERE trades.id = 323"
);

while (resultSet.next()) {
    Integer id = resultSet.getInt("id");
    Float price = resultSet.getFloat("price");

    System.out.println(
        "Stock " + id + " has price of " + price
    );
}
```



Anything we can
do from the SQL
client terminal, we
can do from Java

Installing JDBC driver in Eclipse

- Find the appropriate version of JDBC driver for MySQL. Example:

<http://dev.mysql.com/downloads/connector/j/>

- After downloading and unzipping the package, I get the following file somewhere on my machine:

mysql-connector-java-5.1.18-bin.jar

Installing JDBC driver in Eclipse

- I click on my project
- Click on Properties
- Select Java Build Path
- Click on Add External Jar
- Locate and select the previously mentioned jar file
- And that's it – I have the same access to my SQL database as I do from the console

Problem solving with SQL

- We're going to use a common problem – bucketing prices – to demonstrate the power of SQL
- Suppose we have a simple price table comprised of the following columns
 - *id* – A stock id
 - *ts* – A time stamp telling us how many seconds elapsed from the start of the trading day to this record
 - *price* – The price of the stock at the time this record was written
- We want to make 1 minute (60 seconds) buckets of prices
 - In other words, the last price to appear for stock n before the 60 sec mark is the one we will use at the 60 second mark. The last price to appear before the 120 second mark is the price we will use at the 120 second mark. For example, we would do this if we want prices at 9:30, 9:31, 9:32, etc.

Suppose these are our prices

id	ts	price
1	30	89.90
1	40	90.00
1	110	90.10
2	50	45.00
2	70	44.90
2	80	44.85
3	90	75.00

We want the last price in 1 min buckets

id	ts	price
1	30	89.90
1	40 ★	90.00
1	110 ★	90.10
2	50 ★	45.00
2	70	44.90
2	80 ★	44.85
3	90 ★	75.00


★ Records to be used at 60 sec mark

★ Records to be used at 120 sec mark

The output we want

id	ts	price
1	40	90.00
2	50	45.00
3	60	NULL
1	110	90.10
2	80	44.85
3	90	75.00

Note that there was no price for stock id 3 in this 60 second bucket, so it was filled in with a value of NULL



Constructing this table with SQL

- In SQL, we can write procedures that are similar to standard programming languages such as Java and C++, procedures with loops and if statements
- However, this is not the most efficient use of SQL
- We will try to minimize the use of loops and create this table using mostly queries and joins
 - In MySQL, some lightweight loops are required, but other versions of SQL can do it without loops

First, we create the data we'll be using

```
DROP DATABASE priceData;  
CREATE DATABASE priceData;  
USE priceData;
```

```
CREATE TABLE prices ( id INT, ts INT, price FLOAT );
```

```
INSERT INTO prices ( id, ts, price ) VALUES ( 1, 30, 89.90 );  
INSERT INTO prices ( id, ts, price ) VALUES ( 1, 40, 90.00 );  
INSERT INTO prices ( id, ts, price ) VALUES ( 1, 110, 90.10 );
```

```
INSERT INTO prices ( id, ts, price ) VALUES ( 2, 50, 45.00 );  
INSERT INTO prices ( id, ts, price ) VALUES ( 2, 70, 44.90 );  
INSERT INTO prices ( id, ts, price ) VALUES ( 2, 80, 44.85 );
```

```
INSERT INTO prices ( id, ts, price ) VALUES ( 3, 90, 75.00 );
```

```
select * from prices order by id, ts;
```

Price for our example

id	ts	price
1	30	89.9
1	40	90
1	110	90.1
2	50	45
2	70	44.9
2	80	44.85
3	90	75

The *group by* statement

```
mysql> select max( ts ) from prices;
```

```
+-----+  
| max( ts ) |  
+-----+  
|          10 |  
+-----+
```

```
1 row in set (0.00 sec)
```

Without *group by*, one
max time stamp for
entire
table

```
mysql> select max( ts ) from prices group by id;
```

```
+-----+  
| max( ts ) |  
+-----+  
|          110 |  
|           80 |  
|           90 |  
+-----+
```

```
3 rows in set (0.01 sec)
```


With *group by*, one
max
time stamp for each id
in table

Using *group by* we can get almost the answer we want with just two queries


```
select id, ( max( ts ) ) ts
from prices where ( ts < 60 )
group by id order by ts, id;
```

For each id, we are now looking only at the last price in each 60 second bucket

```
select id, ( max( ts ) ) ts
from prices where ( ts < 120 )
group by id order by ts, id;
```



id	ts	price
1	40	89.9
2	50	45



id	ts	price
2	80	45
3	90	75
1	110	89.9

The time stamps and ids have to be joined with the full price record to get the correct prices

Thinking about the solution

- The above result was obtained with two separate queries. If we had a database of enough 1 minute buckets, this would be extremely slow
 - 37620 separate queries for three months of data
- As well, we want to fill in a null record for stocks that do not have a record in a given bucket
 - For example, stock id 3 has no record at all in the first bucket
- So what we want is a table that has both, real records and fake records with NULL values
- For a given stock, we'll use real records when we have those and fake records when we don't

How do we build the table we need?

- (1) For all records, compute the bucket number, ***bkt***
- (2) For each stock id and bucket, keep only the last record
- (3) Construct a second table of filler records that fall precisely on end-of-bucket time stamps and have NULL ***price*** values
 - These will be our *type 2* records, for which the price will always be NULL and the *bkt* column will be the *ts* column divided by 60. We will use these when, for a given combination of stock id and bucket number, there is no real price record.
 - To construct this table, we will join a list of all unique stock ids with a list of boundary time stamps (60,120,180,240, etc). In MySQL, creating this list of time stamps requires a loop, which we will implement as an SQL procedure
- (4) Insert our *type 2* records into the table that contains our *type 1* records – as in the table, all buckets now have at least one record per stock id, even if the price is NULL

This is the table we need to perform our calculations

$\text{floor}(\text{ts}/60)+1$

type 1 = original records

id	ts	price	bkt	type
1	30	89.90	1	1
1	40	90.00	1	1
1	60	NULL	1	2
1	110	90.10	2	1
1	120	NULL	2	2
2	50	45.00	1	1
2	60	NULL	1	2
2	70	44.90	2	1
2	80	44.85	2	1
2	120	NULL	2	2
3	90	75.00	2	1
3	120	NULL	2	2

remove older recs in each bucket

insert type 2 records

Final calculation

- For each combination of stock id and bucket number, select the record with the min(type)
 - Type 1 instead of type 2, unless type 1 is not available
- In other words, if, for a give stock id, a bucket has a real price and one of our NULL price filler records, we want to use the real price. Otherwise, we want to use the filler record.