

Website Architecture

Lezione 13

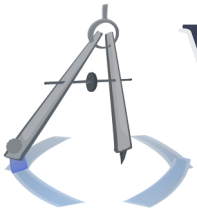
SQL

An overview of the basic and intermediate statements in standard SQL, including a more in-depth introduction to DBMS APIs.

Michael Serritella

Summer 2010





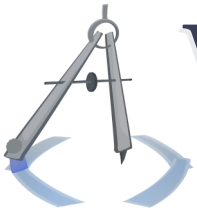
Website Architecture

Lezione 13 | SQL

Intro to SQL

SQL is a language which describes database commands, which are typically called queries. Hence, it is called Structured Query Language. It is a standardized language which is customized by every vendor whose DBMS implements it. SQL is a declarative language in that you simply (?) declare *what* you want rather than exactly how you want the DBMS to compute it.





Flavors

SQL has very many flavors. Every DBMS implements its own version of SQL, which is basically always a subset of the standard features and a set of new commands. The commands in this lesson are recognized by most all DBMSes unless otherwise noted.

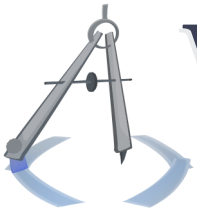
Basic grammar and statements

SQL grammar is fairly simple, especially for simple statements. Here we will see the core statements of the language.

Grammar characteristics

SQL is case-insensitive, and even your identifiers are (supposed to be) interpreted in a case-insensitive manner. Keywords are often written in uppercase for clarity. Whitespace is collapsed. Statements are quasi-English and are terminated with a semi-colon. Strings are single-quoted.





Website Architecture

Lezione 13 | SQL

CREATE TABLE

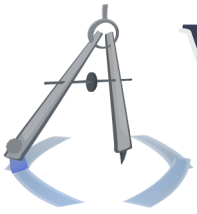
After you have created a database and then logged in (which is usually a little flavor-specific and will be covered later), you have to create a table. You give the set of possible columns and the data type of each column, along with an indication of the keys of the table and any constraints on the values that the DBMS should enforce.

First example

SQL

```
CREATE TABLE Workers
-- A table describing the workers of a company.
(Uno INTEGER PRIMARY KEY,
-- A 'character varying' - a string of variable length, up to 50.
FirstName VARCHAR[50] NOT NULL,
-- Also, NULL is a special value in SQL, and this column cannot
-- ever store it.
LastName VARCHAR[75] NOT NULL,
-- This could be NULL.
SupervisorUno INTEGER,
-- A Julian date.
HiringDate DATE,
-- A timestamp, of which varying levels of precision exist.
LastSignOut TIMESTAMP);
```





Website Architecture

Lezione 13 | SQL

Data types

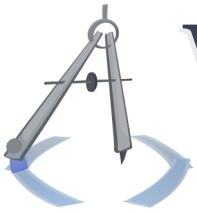
Each column must have a data type. There are standard SQL data types, like the ones seen above, but it's more important to know the vendor-specific data types. The standard types seem rather dated, and vendor-specific types are often useful abstractions, as well as new constructs entirely (e.g. arrays). Sometimes, the vendor doesn't even implement all of the core types but provides roughly equivalent alternatives. You can see the core types at [Wikipedia](#).

There is one particularly special type - the value **NULL**. This is treated differently when it is checked for a value or combined with other values, which we will see later.

Keys

You specify the key(s) of a table when you create it. Keys can be much more complicated than simply a sequential integer, although a sequential integer is basically always sufficient. So you can refer to the above example for practical use until you stumble upon the alternatives in your own experience. You may have to add the word "AUTOINCREMENT" in there





Website Architecture

Lezione 13 | SQL

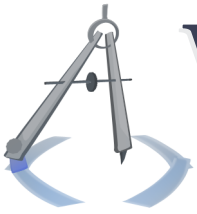
somewhere; check your documentation and/or search for that word.

Constraints

The database may enforce constraints on the data, like enforcing that an integer is always greater than 7 or enforcing that no two rows have the same value for a particular column. Constraints are typically specified when the table is created, although the syntax varies by DBMS. If a constraint is violated upon an insertion of a new row, the row will not be inserted. If a constraint is violated when a row is changed, the row will not be changed. In either case, an error message will be printed somewhere. If you are running the DBMS via the command prompt, it will print a message right there. Via an API is more tricky; we will see that case later.

You probably shouldn't make very detailed constraints if you're working with a website, which we'll see later, but if you want to add a constraint, here is one example:





Website Architecture

Lezione 13 | SQL

SQL

```
CREATE TABLE Dudes
-- A table describing dudes.
(Uno INTEGER PRIMARY KEY,
 Name VARCHAR[50] NOT NULL,
 ZipCode VARCHAR[12] NOT NULL,
 -- Hey, look: A default value.
 NumberOfChildren INTEGER DEFAULT 0,

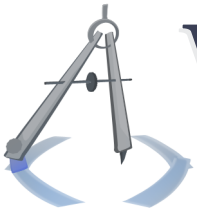
 -- Constraint! This applies to the pair, actually; no two rows
 -- may have the same pair of data for these columns.
 UNIQUE(Name, ZipCode),

 -- Constraint!
 CHECK(NumberOfChildren >= 0));
```

Constraints & Keys: Foreign Keys

Pretty often, you will want to store the primary key of one record as a piece of data in another table. This is a staple of the relational model. But how do you make sure that the key is correct when it is just some integer somewhere? You can make a constraint that a column value must be a valid key of another table. This column value is called a **foreign key**. Here is an example:





Website Architecture

Lezione 13 | SQL

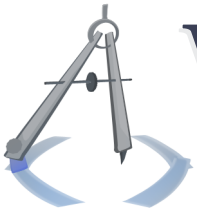
SQL

```
CREATE TABLE WorkersWithForeignKey
-- A table describing the workers of a company.
(UnO INTEGER PRIMARY KEY,
 Name VARCHAR[50] NOT NULL,

-- Foreign key, which has vendor-specific syntax:
 SupervisorUno INTEGER REFERENCES Supervisors(UnO)
);
```

Foreign keys seem nice - and they can be useful in some situations - but you may not want to use them so much in website databases, which we will see later when we talk about constraints. In any context, foreign keys impose a certain type of usage pattern once they are in place. If you delete a supervisor, in our above example, then the SupervisorUno entry corresponding to that supervisor would no longer be valid in the WorkersWithForeignKey table. So you have a choice: you can either prevent this kind of delete, or you can ensure that the deletion of a supervisor will also delete any workers who link to him/her; this is called a cascading delete. Check your DBMS documentation for syntax for cascading options.





INSERT INTO: Insert new data

Once a table is created, you populate it with the **INSERT** command, like so:

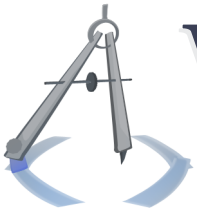
SQL

```
INSERT INTO Dudes (Name, ZipCode) VALUES ('Joe', '11223-4567');  
-- Giving your own key? Dumb.  
INSERT INTO Dudes VALUES (87, 'Bob', '77889', 3);
```

You can give a column list within the command, so that you may only provide a subset of column values, if you want. You may do this if the omitted columns have default values, which auto-incrementing primary keys will also have. You may also give the column values in a different order than they are given in the **CREATE TABLE**.

You should *always* give column names for a multitude of reasons. You don't want to have to remember the order in which the columns were written. And if you omit them, your code is much less readable. Furthermore, you may want to add columns to the table later (with **ALTER TABLE**), in which case you don't want to modify every **INSERT** if you don't have to.





Website Architecture

Lezione 13 | SQL

Bulk inserts

It is possible to insert more than one row in a single statement, provided the vendor's SQL allows for it. Example:

SQL

```
INSERT INTO Dudes
  (Name, ZipCode)
VALUES
  ('Joe', '11223-4567'),
  ('Mark', '22334', 2),
  ('Dave', '55443', 7);
```

SELECT: Retrieve data

Now comes **SELECT**. **SELECT** is actually the vast bulk of the language in terms of complexity, and you will use it more than any other type of command. Its simple case is easy enough:

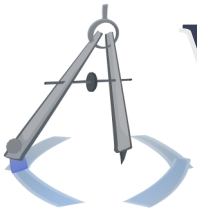
SQL

```
-- Get specific columns and only when a condition is matched.
SELECT Name, ZipCode FROM Dudes WHERE (NumberOfChildren > 1);

-- Parentheses are optional but nice.
SELECT Name, ZipCode FROM Dudes WHERE NumberOfChildren > 1;

-- Get all columns all the time; this could also have a WHERE clause.
SELECT * FROM Dudes;
```





Website Architecture

Lezione 13 | SQL

Using **SELECT**, you will get a set of results which is formatted like a table; in these examples, you would get a two-column table and then a four-column table from the last command. Generally speaking, we can say that both tables and results from **SELECT** are **relations**.

We will see more complicated examples later in the lesson.

UPDATE: Change data

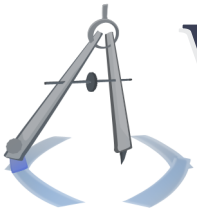
To change an existing row, use **UPDATE**. This has a sort of awkward syntax, including a humongous "gotcha":

SQL

```
UPDATE Workers
  SET LastSignOut = '2010-07-20 17:00:03',
      -- Just an example of updating multiple columns at once.
      LastName = 'Smith'
  WHERE Uno = 5;
-- You can actually set a timestamp equal to the function NOW()
-- and do other tricks.
-- Or perhaps it is not called NOW() if the DBMS is stubborn.
```

Did you see the **WHERE**? You need that, or else this change *will apply to all rows*.





Website Architecture

Lezione 13 | SQL

DELETE: Delete data

Use **DELETE** to remove rows of data. It has a syntax similar to **UPDATE**:

SQL

```
DELETE FROM Workers WHERE Uno > 3;
```

The same gotcha applies. You have been warned - twice!

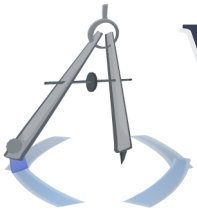
DROP: Discard an entire table

To delete an entire table, you don't use **DELETE**; you use **DROP**, like this:

SQL

```
DROP TABLE Workers;
```





SELECT options

When you use a database, you spend at least 90% of the time pulling data, and so the **SELECT** command is rather freakishly overdeveloped compared to the others. Here are some of the most common additions to **SELECT**.

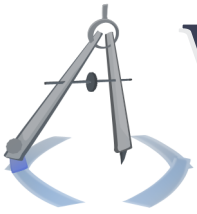
ORDER BY: Sorting results

You can add an **ORDER BY** clause to the end of a **SELECT** statement, which sorts the results, optionally with secondary, tertiary, or more columns to use in case the first column has some equal values. Here's a simple example:

```
SQL
SELECT *
  FROM Customers
  -- You can have an "ASC" or "DESC" modifier;
  -- without this, ascending is the default.
  ORDER BY Age DESC;
```

And here's a more complex example:





SQL

```
SELECT *  
  FROM Customers  
  -- First by last name, then by first name, then descending by age.  
 ORDER BY LastName, FirstName, Age DESC;
```

DISTINCT: Remove duplicates

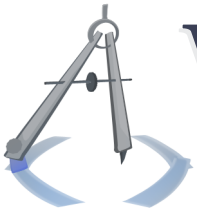
Let's say we have a very simple table that doesn't have a primary key. It just has names and ages, for instance. Suppose you want all unique combinations of name and age. For example, if there are three eighteen-year-old Davids, you only want to see "(18, David)" once. You can do this kind of thing with the **DISTINCT** modifier:

SQL

```
SELECT DISTINCT *  
  FROM JustNamesAndAges  
  -- Why not?  
 WHERE Age > 10;
```

More often, you want to select a subset of columns (called **projection**, BTW) and then ensure that your result set is unique. Perhaps your table contains a wealth of information, like name, age, height, SSN, etc. This is a pretty unique-ish combination





Website Architecture

Lezione 13 | SQL

of features, so if you were to do this:

SQL

```
SELECT DISTINCT * FROM ComprehensivePersonalInformation;
```

You would probably get the same result as you would without the **DISTINCT**. But let's say you want the distinct names and ages again; you could do this:

SQL

```
SELECT DISTINCT Name, Age FROM ComprehensivePersonalInformation;
```

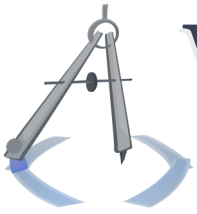
And your {Name, Age} result set would have no duplicates.

Using **DISTINCT** can come at a slight-to-moderate performance penalty, especially when there are no indices on the columns that you want to be distinct.

Aliases

You can give aliases to columns or tables as they appear within a **SELECT** statement. This will become useful in the immediate future, but let's see a few examples:





Website Architecture

Lezione 13 | SQL

SQL

```
-- Aliasing of column names:  
SELECT Name AS Nombre, City AS Ciudad FROM Personnel;  
  
-- First, note that you could do this 'dot' notation:  
SELECT Personnel.Name, Personnel.City FROM Personnel;  
  
-- Now, this; notice that you MAY omit "AS":  
SELECT Pers.Name, Pers.City FROM Personnel Pers;
```

When the result set is output, either to the console or a PHP program, for example, the output will only see the aliases. Keep this in mind for just a second.

Expressions

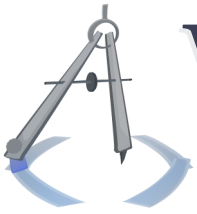
You can "select" mathematical expressions and other expressions. Let's start with this:

SQL

```
SELECT 5 + 8;
```

This will produce a 1x1 relation with the value 13, probably with the column name of "5 + 8", which is quite inconvenient. Let's make it more convenient:





Website Architecture

Lezione 13 | SQL

SQL

```
SELECT (5 + 8) AS TheSum;
```

That's a little better. Now let's make it useful:

SQL

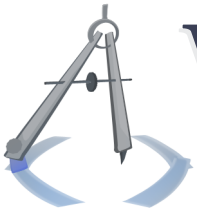
```
SELECT Name,  
       Weight,  
       Height,  
       -- Or something like that.  
       (Weight / Height) AS BodyMassIndex  
FROM People;
```

We will discuss related design principles when we see views.

Joins

Joins are the primary tool of the relational model. They are **SELECT** statements that span across multiple tables, "joining" the rows of each table into one compound row and outputting them when some condition is true - usually when a value in one of the rows equals a value in another row. Let's see a first example that uses a sort of training-wheels syntax (with suboptimal performance):





Website Architecture

Lezione 13 | SQL

SQL

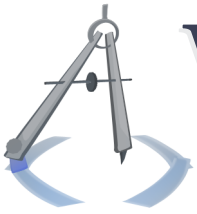
```
SELECT *  
  FROM Orders,  
       Customers  
 WHERE Orders.CustomerUno = Customers.Uno;
```

This will glue an Orders row next to a Customers row into one large row, so long as the customer unos match. The result set may have columns like this:

```
{Orders.Uno, Orders.TotalPrice,  
Orders.CustomerUno, Customer.Uno, Customer.Name}
```

This output format is far from ideal. The columns have these garish names, and by definition, some of the values will be redundant.. Every **Orders.CustomerUno** is going to be equal to the **Customer.Uno** next to it. So let's make a number of improvements to this query, using projection and aliases.





Website Architecture

Lezione 13 | SQL

SQL

```
SELECT O.Uno AS OrderUno,  
       O.TotalPrice AS TotalPrice,  
       C.Uno AS CustomerUno,  
       C.Name AS CustomerName  
FROM Orders AS O,  
     Customers AS C  
WHERE O.CustomerUno = C.Uno  
-- Big orders first. Note that we can't refer to column aliases  
-- in the ORDER BY clause; this is sort of just a quirk.  
ORDER BY O.TotalPrice DESC;
```

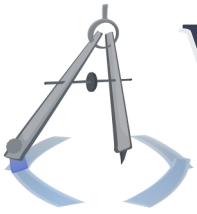
Aliases make the output format more easily readable and predictable, and they even make our SQL code more readable, since all the table names are abbreviated.

Proper syntax and different types of joins

We said before that we were using a training-wheels syntax. The more correct syntax requires that you know a little more about the different types of joins. There are a few different options in joins, primarily because you may want different behavior upon the condition that two rows do *not* match.

Let's stop for a second and imagine the code behind the code.





Website Architecture

Lezione 13 | SQL

There is probably some C code behind the DBMS which implements the SQL code. That C code is basically framed within a nested loop. That's the basic way to conceive of a join - it is a nested loop. Now, in the loop, what is the **if** statement that determines whether you glue two rows together and send them to output? Keep this in mind as we proceed.

If two rows don't match on the desired condition, should they still be output? What about all the rows on the "left", regardless of whether they match anything on the right, with blank (**NULL**) values for those phantom columns on the right? Or the opposite, prioritizing the right-hand row? These are the basic types of joins:

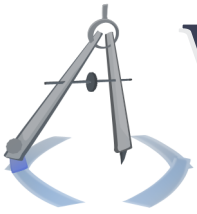
Inner Join What we have seen before. Only output composite rows if the component rows match.

Left Join A special type of "outer join." Output all rows from the left. If there is no match with a row on the right, output phantom values (like **NULL**) instead of the values from the right-hand row.

Right Join A special type of "outer join." Inverse of the left join.

Outer Join Output all row matches; additionally, output the unmatched rows of both sides, using phantom **NULL** values where





Website Architecture

Lezione 13 | SQL

appropriate. This is effectively equal to a left-join result set concatenated to a right-join one. It is also called a **Full Join** or **Full Outer Join**.

Remember the nested loop? Think inner loop/outer loop. It really isn't so foreign.

Now, let's apply this knowledge to some SQL:

SQL

```
SELECT O.Uno AS OrderUno,
       O.TotalPrice AS TotalPrice,
       C.Uno AS CustomerUno,
       C.Name AS CustomerName

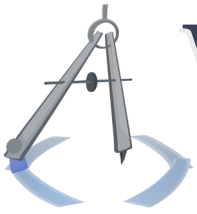
-- This is it; output all customers, even if they have no orders:
FROM Customers AS C
LEFT JOIN Orders AS O
  -- Essentially a special WHERE clause:
  ON C.Uno = O.CustomerUno

ORDER BY O.TotalPrice DESC;
```

You can do any of the joins instead of **LEFT JOIN**. However.. check your DBMS documentation. Some DBMSes don't support all types of joins, and some have even more than these.

Lastly, know that you *can* chain multiple joins in one query,





Website Architecture

Lezione 13 | SQL

should you be so ambitious:

SQL

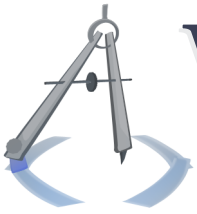
*-- But don't select *, since it becomes unreadable.*

```
SELECT *
  FROM Customers AS C
  LEFT JOIN Orders AS O
    ON C.Uno = O.CustomerUno
  INNER JOIN SalesReps AS SR
    ON O.SalesRepUno = SR.Uno
 ORDER BY O.TotalPrice DESC;
```

Unions and set operations

Recall your set-theory operations, like union, intersect, set subtraction, etc. You can apply these kinds of operations to the results of **SELECT** statements as long as they make sense. If the columns of two relations have the same types (i.e. they are structurally equivalent), they are said to be **union-compatible**. Then, you can do things like this:





Website Architecture

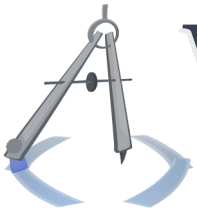
Lezione 13 | SQL

SQL

```
-- All names.  
SELECT Name, Age FROM Males  
UNION  
SELECT Name, Age FROM Females;  
  
-- Unisex names.  
SELECT Name, Age FROM Males  
INTERSECT  
SELECT Name, Age FROM Females;  
  
-- Manly names.  
SELECT Name, Age FROM Males  
EXCEPT  
SELECT Name, Age FROM Females;
```

These may have unintuitive behavior. They actually remove duplicates from the result, which is in keeping with their nature as set operations but a little anomalous within SQL. If you do not want to remove duplicates, add an " **ALL** " after the keyword; e.g. **UNION ALL**. You may want to do this for performance reasons.





Website Architecture

Lezione 13 | SQL

Miscellany: EXISTS; CASE; NULL

Now for a grab bag of still-common-enough **SELECT** features.

Sub-queries and EXISTS

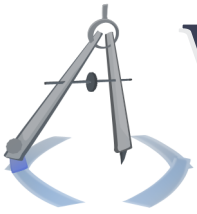
You can actually make **SELECT** queries within **SELECT** queries. Did you know that? Bet you didn't. This can get complicated, but there are some simpler cases. Here's a medium-complexity query that retrieves the names of customers who have ever made an order.

SQL

```
-- Using a (vendor-specific) concatenation operator.  
SELECT (LastName||', '||FistName) AS FormalName  
FROM Customers C  
WHERE EXISTS (SELECT AnyColumnNameHereMaybeStar  
                FROM Orders  
                WHERE CustomerUno = C.Uno)  
ORDER BY LastName;
```

An interesting point is that if you do this, it will not behave the way you want:





Website Architecture

Lezione 13 | SQL

SQL

```
-- Using a (vendor-specific) concatenation operator.
SELECT (LastName||', '||FistName) AS FormalName
FROM Customers C
WHERE EXISTS (SELECT COUNT(*)
              FROM Orders
              WHERE CustomerUno = C.Uno)
ORDER BY LastName;
```

The count always *exists*, even if its value is 0. Whoops.

There is also a **NOT EXISTS**.

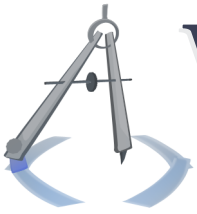
CASE: For control flow

In making expression-based columns, you may be looking for an if-statement or a ternary operator. The **CASE** command is basically this. Here's a quick example:

SQL

```
SELECT LastName,
       FirstName,
       (CASE
        WHEN (IsDoctor) THEN 'Dr.'
        WHEN (Gender = 'M') THEN 'Mr.'
        ELSE THEN 'Ms.'
       END) AS SimpleTitle -- Note that the name comes down here.
FROM Customers C;
```





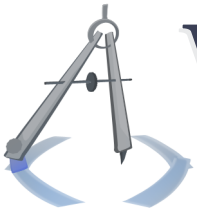
Website Architecture

Lezione 13 | SQL

Wrangling of NULL

The value **NULL** behaves strangely when used in an expression or tested for **NULL**ness. The SQL standard is apparently vague on how this should work, so check your DBMS documentation. But when used in an expression, generally, the result of the expression becomes **NULL**, so you should avoid this. How might you avoid it? You check for **NULL**. How do you check? You use the expression "value **IS NULL**" or **IS NOT NULL**.





Advanced constructs

That's the basics. Now we can use some advanced constructs on top of tables, which give us a more software-engineered way of designing databases.

Views: Virtualized tables

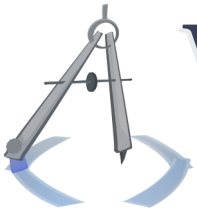
Views are basically macros - wrappers around **SELECT** statements. But they offer you a great layer of abstraction between the data and the application. First, look at this nearly-trivial example:

SQL

```
CREATE View AView AS  
SELECT * FROM ATable WHERE (AValue > 30);
```

You can select from AView just like a table (or ATable, rather), and this will only show rows with AValue greater than 30. The user of the view does not know that it's a view. Even aside from security, views provide an opportunity to separate interface from implementation. Thus, views will afford you some flexibility as a software engineer. You should consider only us-





Website Architecture

Lezione 13 | SQL

ing views in your application's queries. If your DBMS supports user accounts and permissions, make it so that only the view are accessible to the application's database user account.

Execution semantics

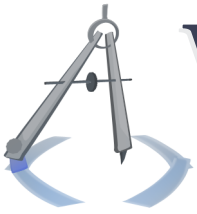
Firstly, how do views really behave? Are they copies of the data? If the table is updated, is the view updated? A view does not store a separate copy of the data. It is just a macro. So it is always updated whenever the underlying data is updated.

Can you insert into a view? No - not unless you hack up something to allow it, which we will see later. So that is another type of protection offered by a view, although it has its downsides.

Projection, renaming, and reordering

Using a view, you can reshape the structure of a table, including projection (excluding columns), renaming columns via aliasing, and reordering them. For example:





Website Architecture

Lezione 13 | SQL

SQL

```
CREATE View AnotherView AS
SELECT Name AS Nome, Age AS Eta, Sex AS Sesso
FROM People
WHERE Age > 10;
```

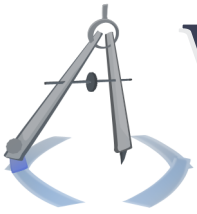
You may want to do this to hide primary keys, for instance, or just to create a simpler interface. It is much easier to select everything from a view than to specify columns from a table; the code would also be more readable and verifiable.

Composition and multiplicity of interfaces

Let's really begin to separate interface from implementation. Tables.. are for data. That sounds silly until you know the alternatives.

Tables are for raw data. Perhaps you have a ton of raw data - thirty columns of personal information on your employees - but you don't always want to select all of it. More accurately, you probably have some well-defined roles in your business, like a payroll dude, an office-manager dude, a secretary dude, etc. Each of them may only care about a subset of these columns,





Website Architecture

Lezione 13 | SQL

and more importantly, you may want to block their access to the rest of them. If you didn't have views, perhaps you would (ab)use the relational model by creating a set of tables like PeopleForPayroll, PeopleForSecretaries, etc., and perhaps they would even be linked by some keys. Who knows. You don't have to worry about that now. Just dump all the data in the table in whatever way is most efficient, and create views that are most convenient.

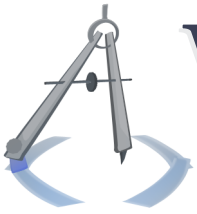
Aside from role separation, you may want to just cast the data in a new light. You can use a view to wrap a join or a nested query (similar to the **EXISTS** examples).

SQL

```
CREATE View TopSalesReps AS
SELECT *
  FROM SalesReps AS SR
  -- Select them as long as their entry in the Earnings relation
  -- says that they've sold over $50,000.
 WHERE (SELECT TotalEarnings
        FROM Earnings
        WHERE SalesRepUno = SR.Uno) > 50000;
```

Perhaps you want to take this idea further and select a very specific slice of your personnel database. Top sales reps who





Website Architecture

Lezione 13 | SQL

have sold mostly in the eastern United States. Sure, you could wrap this all into one **SELECT**. But more than likely, you'd want to make views for each significant component of this query and then unify them via one final view - perhaps a view that joins them together or unions them. Views can help you compose a multi-layered structure of interfaces in order to produce more-and-more-specific subsets of data.

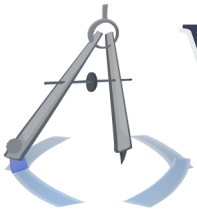
Derived computation

Suppose you have some raw data and some derived data. Some data that can be easily computed from the raw data, like our earlier example of BodyMassIndex. Again: Tables are for data. The rawest of the raw. If you have derived data, compute it on the fly by rolling it into a view.

Efficiency

Views can offer some performance advantages, but in practice, they are probably a meager detriment to performance overall. Depending on the DBMS, a view's query execution strategy may be optimized better than an equivalent **SELECT** statement that gets compiled on the fly. And when building the actual





Website Architecture

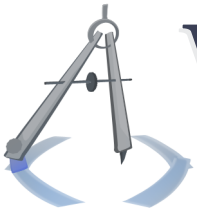
Lezione 13 | SQL

SQL strings and passing them through the API, selecting everything from a view is a very small and simple SQL statement when compared to the complicated **SELECT** that is behind the view.

On the downside, though, a view is just a macro. And especially if you have done renaming or reordering of columns, it is an extra layer of translation that needs to be computed. At a deeper level, you should consider that the DBMS may be less likely to optimize its query execution plans if you have views that wrap views that wrap views, *ad nauseam*. The DBMS may stop at a certain depth, so the execution plans may be more naïve than you would expect or perhaps more naïve than what you would get if you were to literally execute the **SELECT** statements.

In order to optimize view access, some DBMSes will "materialize" a view, in which they store copies of the underlying data and automatically keep them concurrent. This can be a great optimization for DBMSes who do it and a sharp disadvantage to DBMSes who don't.





Website Architecture

Lezione 13 | SQL

Functions

SQL allows for function syntax, and, like regular programming languages, you can use functions to compute & return a value, and you can also use them to execute some statements and produce side effects (i.e. procedures).

SQL functions can be handy for computation, but they can also be an ultimate tool in software engineering.

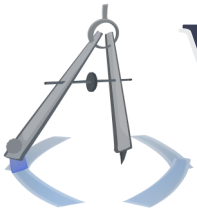
Basic operators

The basic operators, like $+$ $-$ $/$ $*$, are pretty much universal for numeric types. You will see some variation in more complicated ones, though, like for string concatenation. Some DBMSes do this, which is a demonstration of the basic (and familiar) function syntax:

SQL

```
CREATE View PeopleWithFancyNames AS
SELECT LastName,
       FirstName,
       CONCAT(LastName, ', ', FirstName) AS FormalName,
       CONCAT(Title, ' ', FirstName, ' ', LastName) AS TitleName
FROM People;
```





Website Architecture

Lezione 13 | SQL

Check your documentation for a complete list of functions and expressions, which will vary significantly.

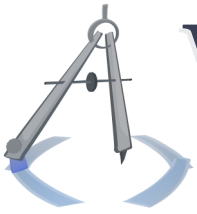
Date/time functions

There are some built-in date and time functions. For instance, to get the current time, you can almost always call **NOW()** (which, like anything, is case-insensitive); sometimes it is **DATETIME()**. Or you can call **DATE()**, for instance. And there are bound to be some date and time manipulation functions, as people often rely on the database for generating and manipulating dates and times, rather than inserting them from the application (e.g. getting the date from PHP and inserting it).

Aggregate functions

Some functions apply to an *entire column* of data, like to get the count, the sum, or the average. Example:





Website Architecture

Lezione 13 | SQL

SQL

```
-- Returns a 1x1 relation with the number of entries.  
SELECT COUNT(*) FROM People;  
  
-- Returns a 1x1 relation with the sum of people's ages (?).  
SELECT SUM(Age) FROM People;  
  
-- Returns a 1x1 relation with the average of people's ages.  
SELECT AVG(Age) FROM People;  
  
-- Returns a 1x1 relation with the lowest age; MAX() also exists.  
SELECT MIN(Age) FROM People;
```

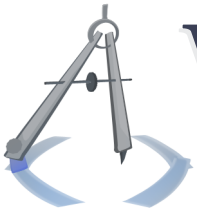
The majority of aggregate functions will probably be DBMS-specific, but these are more or less guaranteed to exist in all DBMSes.

So, what if you want to incorporate an aggregate value into a result set, not just have it by itself? You can do that, but be aware of the semantics.

SQL

```
SELECT Name,  
        Score,  
        AVG(Score) AS AverageScore,  
        (Score - AverageScore) AS DistanceFromAverage  
FROM Students;
```





Website Architecture

Lezione 13 | SQL

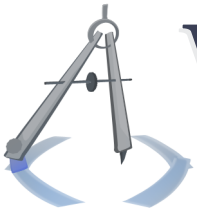
In the above example, *AverageScore* will be the same for all rows. This is the same behavior as with any aggregate function. From here, you have a few options: 1) Use aggregate functions for one-row queries only; 2) Use aggregate functions that naturally apply to every row in some way (as above); 3) You want some more advanced grouping and separation. If you want more advanced behavior, you will need to investigate **GROUP BY** and **HAVING**, which is like a **WHERE** clause for aggregate values.

User-defined functions

The best part about functions is actually that you can define your own functions. User-defined functions are just blocks of SQL statements, and they can take parameters. They can even return a result set instead of just a 1x1 value; in this case, you can consider the function a wrapper for a **SELECT** statement or a sequence of non-**SELECT** statements ending with a **SELECT**.

Here is an example to give you an idea, although syntax will vary by DBMS:





Website Architecture

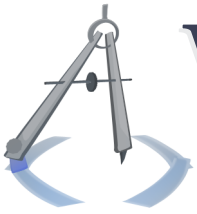
Lezione 13 | SQL

SQL

```
CREATE FUNCTION HighestGrade(TestNumber INTEGER) RETURNS REAL
AS $_$
SELECT MAX(Grade) FROM TestGrades WHERE (TestNumber = $1);
$_$
LANGUAGE sql STABLE;
```

Some DBMSes allow a mixture of a procedural language with SQL. This is generally termed PL/SQL. The language is DBMS-specific; for example, PostgreSQL has PL/pgSQL. These procedural languages are reminiscent of Pascal; they simply allow you to declare variables and incorporate them into queries. You can almost certainly do the same thing without procedural languages and variables, though it would be more arduous, and you may want to give yourself some intermediate functions or views to assist in the process. In any case, here is a PL/pgSQL function:





Website Architecture

Lezione 13 | SQL

SQL

```
CREATE FUNCTION LowestGradeOfStudentWithHighestGrade(TestNumber INTEGER)
RETURNS REAL
AS $$
DECLARE
    highestScoreStudentUno INTEGER;
    lowestScoreOfStudent REAL;
BEGIN
    -- Check inputs.
    if $1 < 0 THEN
        RETURN -1;
    END IF;

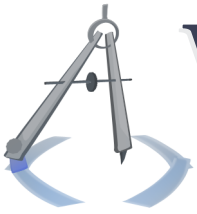
    -- Find the highest-scoring student
    -- (or, really, a student with the high score).
    SELECT StudentUno INTO highestScoreStudentUno
        FROM TestGrades
        WHERE (TestNumber = $1) AND
            (Grade = MAX(Grade));

    -- Get the lowest score of this student across all exams.
    SELECT MIN(Grade) INTO lowestScoreOfStudent
        FROM TestGrades
        WHERE (StudentUno = highestScoreStudentUno);

    RETURN lowestScoreOfStudent;
END
$$
LANGUAGE plpgsql STABLE;
```

The one quirk about functions is that you must **SELECT** from them, even if they don't return a value (they are said to "return void", and they return a 1x1 **NULL**). For example:





Website Architecture

Lezione 13 | SQL

SQL

```
SELECT * FROM HighestGrade(3);
```

```
SELECT CleanUpDatabaseOrDoSomeMaintenanceOrSomething();
```

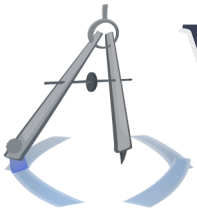
User-defined function binding

If you are connected to the database via a C program or a PHP program, for instance, you can usually register a SQL function name that calls one of your C or PHP functions. Thus, you can run any code (e.g. a shell script) when a function is called. There is certainly some function-call overhead, so make sure that you need to do this. You almost never need to do this. Explore the limits of what the database can do for your particular problem before you resort to binding your own functions.

For software engineering in database design

Now comes the best part. We mentioned with views that your DBMS probably allows some kind of user-account permissions and that you can set them specifically so that the user can only access views, not tables. Well, why stop there? If you're running a website, you almost always have a well-defined set





Website Architecture

Lezione 13 | SQL

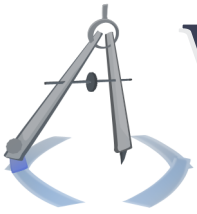
of database tasks, and if you execute a different query, it's because you're getting hacked. If you only expose functions to the database user of your Web application, then you completely button down the functional privileges of that user. You can even validate your inputs first!

Unsurprisingly, coming from a programming perspective, functions allow the ultimate in separation of interface from implementation. And if you are making a site of any appreciable size or popularity, you should consider investing your time into this method. It offers maximum efficiency and maximum security, with only a startup cost of your time. It is not a widely advocated method, probably because of its slight complexity or its initial tedium, but so what? You should do it.

Triggers

A trigger is a procedure that runs when a certain event is fired. The procedure is a block of SQL, like a simple user-defined function. The event is more interesting. You can make triggers that execute before or after - *or instead of* - an **INSERT**, **UPDATE**, or **DELETE**. You can even add a **WHERE** clause so that only some





Website Architecture

Lezione 13 | SQL

particular inserts, for instance, will fire off your trigger.

Here is a sort of contrived example, although syntax will vary by DBMS:

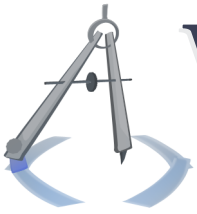
```
SQL
-- Keep track of when male customers are deleted
-- (with a contrived table structure).
CREATE TRIGGER ATriggerName
  AFTER DELETE ON Customers
  WHEN old.Gender = 'M'
  BEGIN
    -- Maybe this only has one row, BTW.
    UPDATE CustomerStatistics SET MaleCustomers = (MaleCustomers - 1);
  END;
```

Triggers can play an important role in database design; they can more or less break the models we have seen so far, allowing for new types of database designs altogether.

Gaming the relational model

In the last lesson, we saw the classic tradeoff of the relational model. In the relational model, you can separate data into encapsulated tables and then link the tables via keys, and so whenever you want a set of related data from all tables, you





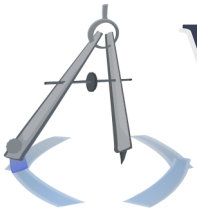
Website Architecture

Lezione 13 | SQL

have to do a join query and link them up via their keys. In the non-relational model, you store all information where it is needed, even if it is redundant. The upside to the relational model is that it more easily preserves correctness of data. The downside is that joins are relatively expensive. Well, now we have triggers.

So what, though? Well, what if you could keep data everywhere it is needed but you can *ensure that it is correct*? Triggers can do this. So, let's build an example. We have a Customers table, an Orders table, and an OrdersConsolidated table, or whatever you'd like to call it. The Customers and Orders tables are self-contained, like they are in the relational model. The OrdersConsolidated table has order information and customer names, like in the non-relational model. If the user wants this kind of information, he/she can select from OrdersConsolidated and not incur the cost of a join. Now, whenever a user updates Orders or Customers, triggers will manage OrdersConsolidated; they will propagate any changes to that table. The user just has to make sure not to modify the OrdersConsolidated table directly. And, after all, you should probably hide this entire system behind views and functions,





Website Architecture

Lezione 13 | SQL

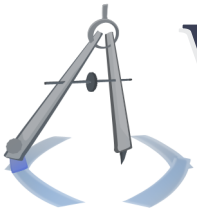
anyway, so that shouldn't be a problem.

A comprehensive example of SQL would be quite tedious. In fact, writing this trigger management system in real life is tedious, since you have to account for every type of operation on Customers and Orders. But once you set it up, it can make the difference between a completely unusably slow database and a quick one.

This system does have its disadvantages. The amount of data stored is on the order of double, and inserts and changes are slower. However, in website databases, the data is never *that* big, and, as we will see, **SELECT** performance is almost always prioritized.

As with any optimization, make sure there is a real need for this before you spend the time and change your structure so dramatically. And if you exclusively use functions (or maybe views) for your interface, you leave the door open for yourself to make this optimization later and integrate it seamlessly with your application, should the need for this optimization arise over time. So, *if* your database is too slow with the purely





Website Architecture

Lezione 13 | SQL

relational model, this structure is a win; you should do it.

For inserting into views

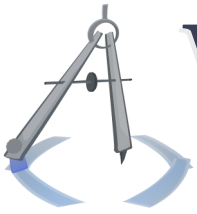
You can actually use triggers to change the rules on inserting into views. You can allow insertion into views if you create a trigger that fires *instead of* the insert on the view. Then you can do whatever behavior you want in the trigger, like maybe inserting the data into three different tables.

This is neat, but you probably won't have too much of a need for this. You should probably make a function-only interface in any case. But even if you don't do that and you have primarily views, you probably have read-only users and read-write users. You probably trust the read-write users pretty well, so you don't need this layer of abstraction; you can just let them insert into a table.

Transactions

Transactions are cohesive groupings of SQL statements. You would group statements into a transaction if they should either





Website Architecture

Lezione 13 | SQL

all execute together or not execute at all, should something fail. Transactions will modify the database all at once, when they are completed, or not at all.

This can also simplify problems of concurrency. If multiple users are connected to the database and they are all making changes, perhaps it would be a problem if the changes of one user affected another user prematurely. Perhaps one user is making an intermediate change to the database while another user is reading. Transactions can encapsulate all these intermediate changes so that they don't affect concurrent queries. Since most all websites have concurrent database access, you should know about transactions.

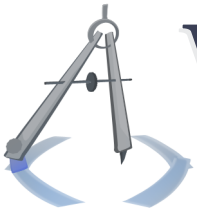
How do you do it? Fairly easily, actually. The syntax is DBMS-specific, but it's never that complicated. Here's an example:

```
SQL
BEGIN TRANSACTION;

-- More statements..
UPDATE Customers SET Name = 'David' WHERE Uno = 8;
-- More statements..

COMMIT TRANSACTION;
```





Website Architecture

Lezione 13 | SQL

If there is a problem, you can "roll back" a transaction, which cancels it.

SQL

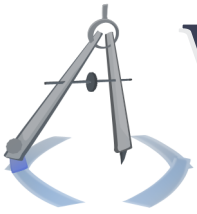
```
BEGIN TRANSACTION;
```

```
-- User just gave some bogus data.. eject!
```

```
ROLLBACK TRANSACTION;
```

There are some more subtle options with transactions, like the ability to save checkpoints, but you probably don't care about these subtleties for most websites. If you're curious, check out your DBMS documentation.





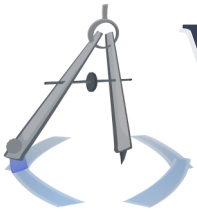
DBMS APIs

In designing your website - depending on the DBMS - you may never need to write these schema-related SQL statements purely by hand; you may have nice control panels or APIs to help you. When you render the website in PHP, you will have a fairly nice API.

Typical use of API

Database APIs in all programming languages seem to share a common design. They encourage similar usage patterns (e.g. a function for queries, a function for retrieving a row of results, etc.), and they usually offer you a similar set of functions that conveniently wrap SQL statements. For instance, most APIs have functions for beginning or ending transactions, so you don't have to pass the appropriate SQL. Most have functions for telling you the number of rows in the result set or the number of rows that was just recently inserted (which would be DBMS-specific SQL commands). Here we will see a typical example of syntax and a list of typical facilities offered by database APIs.





Website Architecture

Lezione 13 | SQL

Syntax in PHP

The PHP API for databases has evolved over these past few years. The old school looks like this, which is still a good demonstration of APIs in general:

PHP

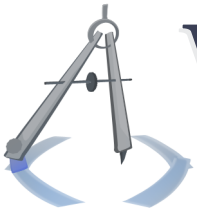
```
<?php
$result = mysql_query("SELECT * FROM People;");

// Keep passing around this $result as an opaque token to functions
// who understand it.
print("There are ".mysql_num_rows($result)." rows.");

// Iterate through the result set, with $row representing each row
// using an associative array. Column names are the array keys.
while ($row = mysql_fetch_assoc($result))
{
    // Print the keys and values of this row.
    print_r($row);
}
?>
```

The new school in PHP is object-oriented and presents a unified interface that works with any DBMS for which there is support. There may be additional functions for the unique features of each DBMS, but there is a large common ground. This library is called **PDO: PHP Data Objects (docs)**.





Website Architecture

Lezione 13 | SQL

Connections to the database are managed by objects of the **PDO** class, and individual query result sets are managed by **PDOStatement** objects. Here's an example:

PHP

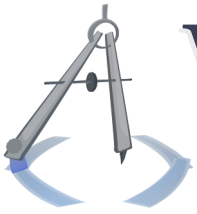
```
<?php
// The connection parameters obviously must specify the DBMS product.
$dbConnection = new PDO('mysql:dbname=MyDatabaseName;host=localhost',
                        'myDBMSUsername',
                        'myPassword');

// But at this point, we don't care that we're using MySQL.
$dbResult = $dbConnection->query("SELECT * FROM People;");

// Iterate through the results.
// Pass in a constant to fetch() which tells it how to populate the
// return-value array; in this case, we want an associative array.
while ($row = $dbResult->fetch(PDO::FETCH_ASSOC))
{
    // Print the keys and values of this row.
    print_r($row);
}
?>
```

See the PDO documentation for more functions and lots of examples. However, you may want to use the database module of the MiniArc framework instead, which wraps PDO. It does return **PDOStatement** objects from queries, so you should still know that part of PDO.





Website Architecture

Lezione 13 | SQL

General powers and abilities

In general, APIs typically offer you these features:

Connection Easy/parameterized connection to a database.

Query Functions for queries, where "query" may be preferred to mean **SELECT** statements but is not necessarily so.

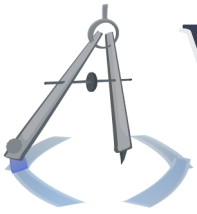
Results Functions for returning results as arrays, possibly associative with the column names as keys. Functions for telling you the number of columns and rows (if possible).

Execute Functions for executing queries that may change the database, such as **INSERTS** and **UPDATES**. You could pass these to the query functions, but the execute functions typically give you more relevant information afterwards, like the number of rows affected, number of rows inserted, or the primary key of the row last inserted. Using this type of function also avoids confusion from these queries not returning a result set; query functions may expect or prefer a result set.

Errors Returns the most recent error code (usually numeric) and/or error string, probably the same as you would see if you were using the console.

Transactions Begin/end/rollback a transaction.





Website Architecture

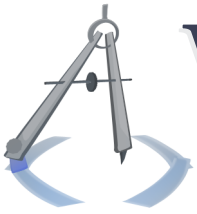
Lezione 13 | SQL

Administration panels

Pro-level DBMSes will have an administration panel with a very slick graphical interface. This could be a standalone program or a website. The panel is most helpful for commands that are not basic queries, such as the commands for creating users and passwords, setting permissions, creating tables and functions, and much more. You can browse the data in the database, search for rows, and execute arbitrary SQL. View statistics, change configurations related to performance.. the list goes on and on. You would want to use this in the design phase of your site, so that you run basically everything except your most basic queries through the administration panel.

We will see PostgreSQL and SQLite in this course. Unfortunately, SQLite does not have such a panel at the time of this writing. It's likely inevitable, but it hasn't happened yet. Fortunately, PostgreSQL has some nice panels. There is the standalone program **pgAdmin III** (binary **pgadmin3**), which works especially well if you are running the PostgreSQL database on the same computer. If you're using a database provided by a Web hosting company, you probably can't use pgAdmin.





Website Architecture

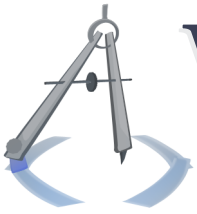
Lezione 13 | SQL

However, there is a website-based control panel called **phpPgAdmin**, which is basically a clone of pgAdmin¹. Your hosting company will almost certainly provide this to you from your hosting control panel. And if you're running PostgreSQL on your own computer, you can still use it, though you should probably just use pgAdmin.

In general, there is one special power that administration panels will give you. From the panel, you can easily dump the state of the database to a text file, which consists of all the SQL commands needed to rebuild the schema and re-insert all the data. You could do this without the panel, but why? Once you have the SQL text file, you can store it or transfer it, of course; this is often a convenient way to backup or checkpoint your database, especially if it is not monstrous in size; otherwise the DBMS will have its own checkpoint features that you should probably use. For another practical use: You may want to set up your database using pgAdmin and then dump it to a file and rebuild it via the phpPgAdmin of your hosting account. That way, you don't have to connect to the Internet to do all of this initial administration, which is often time-consuming.

¹Its interface and functionality are very much like pgAdmin, although the project was inspired by phpMyAdmin, which was developed for MySQL.





Website Architecture

Lezione 13 | SQL

Overlap of responsibilities

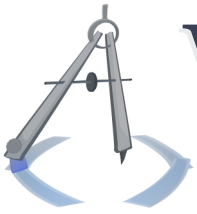
So, you have a DBMS and you have a PHP program. They actually have quite an overlap of capabilities. You can add constraints in the database, or you can check user input in the PHP program before it gets to the database. You can make complex join queries in the database, or you can make simple queries from PHP and link them together with PHP code in a serial process. Which should you do?

Performance

For issues of performance, you should basically always push the code into the database. Make even highly complex queries in the database, so your PHP code is simple and the SQL strings that you build in PHP are simple. There are a few reasons for this.

When you send SQL code to the DBMS, PHP has to build the string, and then the API needs to act as a middleman to deliver it to the DBMS, and then the DBMS has to compile your SQL and then run it. When the results are ready, the DBMS tells the API, the API tells you, and a bunch of little variables





Website Architecture

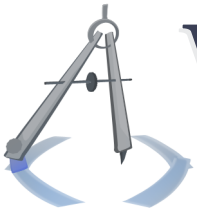
Lezione 13 | SQL

are built, using all kinds of nicey-nice arrays and types. This construction process takes time and should be avoided when possible. Make as few queries as possible to incur the smallest overhead in this sense.

More importantly, if you piecemeal the queries to the DBMS, it cannot take advantage of any optimizations it would make from long-term planning. If you pass a three-deep nested **SELECT** to the DBMS, the DBMS may be able to see that the outermost and innermost queries are related.. they use the same column which has an index.. etc. If you make a three-nested loop in PHP and make three simple queries from them, you rob the DBMS of the chance to make these more complicated optimizations, and it will cost you bigtime.

Why should you "basically always" do this and not always? Well, sometimes you may need to break up a complex query into multiple queries from PHP. You may have to do this when you are very tight on RAM and the large query could use a sizeable amount of RAM. You can't really control how the DBMS manages its memory within a query execution plan; after all, SQL is a declarative language, and you only specify





Website Architecture

Lezione 13 | SQL

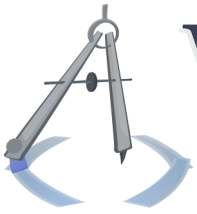
what you want, not how the DBMS computes it. The DBMS will do a good job for the given task - better than you - but you still may need to throttle the RAM usage. In this case, see if you can break up the query into smaller segments. Obviously, you should only do this optimization when you have a clear need for it.

You may also need to do this so you can come up for air every once in a while and write a progress message from PHP (e.g. "54% done.."). However, you can avoid splitting the large query by hacking up some triggers and user-defined SQL functions bound to PHP functions. For instance, when inserting into a table, upon every 10,000th insert, call some SQL (PHP) function. The **WHERE** clause of a trigger could assist you in this.

Correctness

For issues of correctness and constraints, you should probably not have constraints in databases that drive websites. The application (i.e. PHP) is much better suited to handle these problems.





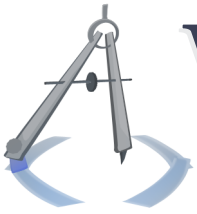
Website Architecture

Lezione 13 | SQL

Remember that if you try to insert a row and it fails to satisfy a constraint, the row will simply not be inserted. An error will be raised in the DBMS. You have to check for this error via the API, and maybe you will get a numeric error code and/or an error string. Then what - you parse the string? Chances are that this string wouldn't even be very helpful to you (e.g. "Column 'age' fails constraint"), since you might normally want to give the user a very detailed and helpful description of his/her mistake. On top of all this, there is the performance issue of checking with the DBMS *after* every single operation just to make sure there was no error.

You generally shouldn't have constraints in website databases, although there are a few exceptions to this rule. First, we're assuming that the database and the PHP program are pretty well coupled. The database will only ever be used by the PHP program - they are a pair - so we might as well put the error checking in either place. However, this may not be the case. You may have some people working on the back end of your site or your company's database, and they could use the command line or whatever other program. In this case, you might want to use constraints, and unfortunately, you may have to check





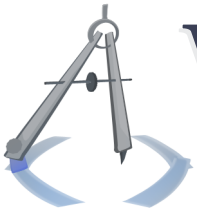
Website Architecture

Lezione 13 | SQL

for errors in your PHP program. However.. if your company will let you do it, you may want to create a separate interface for the other users of the database which imposes constraints on them but not on you. If they insert to an intermediary table, like `PeopleWithConstraints`, then that can have all the constraints that you conceptually want for the table `People`. Then, upon a successful insert, a trigger copies the new row to the `People` table. You have to make sure that those users don't insert into `People`, and they should get their data from `People` and not `PeopleWithConstraints`. You can enforce all of this with a nimble interface and permissions system.

For a second exception: You may want to utilize the **UNIQUE** constraint. This one is often quite important, quite frankly, and may trump PHP efficiency. However, you don't always have to check for errors afterwards. Your PHP program could take advantage of a **UNIQUE** constraint by assuming that duplicate insertion is idempotent. If you want the idempotence semantic, you can write the same PHP code as before and still ensure correctness from the DBMS.





Website Architecture

Lezione 13 | SQL

Security issues?

PHP handles user data, and it also writes SQL code. Fundamentally, this poses a problem. You can probably imagine that you want to integrate the user data into SQL queries at some point ("What is your name?" \Rightarrow write `$Name` into a **WHERE** clause). Is this a security hole? It is, actually. Let's see the problem of **SQL injection**.

SQL injection works like any code injection technique. The **SELECT** may look like this:

SQL

```
SELECT * FROM Users WHERE Username = 'Bob';
```

And the PHP may look like this:

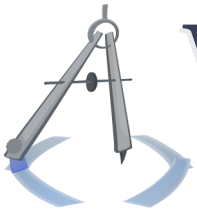
PHP

```
<?php
$sql = "SELECT * FROM Users WHERE Username = '". $GivenUsername. "'";
?>
```

So, what if `$GivenUsername` looks like this?

```
asdf' OR 1 = 1; DROP TABLE Payroll;
```





Website Architecture

Lezione 13 | SQL

The resulting `$sql` will have two SQL statements (or more, if the attacker wants). Using this, it is rather easy to destroy your database, and it may even be possible to elicit more information. How do you fix this?

Over the years, there have been a few decent solutions. The most successful one is to escape the string for SQL syntax, so it ends up looking like this:

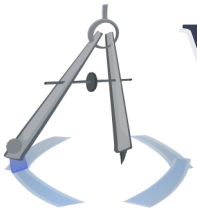
```
asdf\' OR 1 = 1; DROP TABLE Payroll;
```

In this case, the quote is never closed, and the user's name looks like this long string of gibberish. This leads to a single **SELECT** query which is likely harmless.

Aside from this, some DBMS APIs have changed their behavior to only execute the first SQL statement they see in an SQL string. You may be able to circumvent this behavior explicitly.

In any case, there's a much better solution now, which also offers a speedup in other situations.





Prepared statements

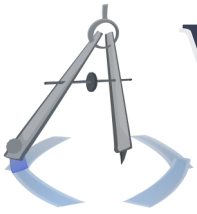
Enter the prepared statement. This is the most significant change to DBMS APIs in recent years. In a prepared statement, you pre-compile some SQL, optionally with placeholder variables. The SQL is compiled once, and then you can execute it again and again without incurring the compilation fee, and you can pass new data every time that will populate the placeholder variables. For instance, the SQL of a prepared statement may look like this:

SQL

```
SELECT * FROM Users WHERE Username = ?;
```

In PHP, you can use **PDO** objects to create prepared statements, which are **PDOStatement** objects. Then you hold on to the **PDOStatement** and use its methods to (re-)execute the query with new data. For example:





Website Architecture

Lezione 13 | SQL

PHP

```
<?php
// Put the SQL in its own string for convenience.
$sql = "SELECT * FROM Users WHERE (Age > ?) AND (Gender = ?)";

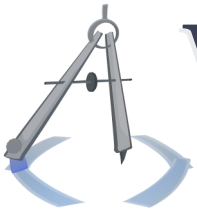
// Prepare the statement.
$dbStatement = $dbConnection->prepare($sql);

// Get results as usual, though we use execute() to fetch them.
$dbResult = $dbStatement->execute(array(33, 'F'));
?>
```

This can have a great performance benefit. But most importantly, it solves our problem of SQL injection!

Previously, the entire SQL string was built and then it was expected to contain valid SQL code. The user data was integrated into a string which, at some point, was expected to be SQL code. Now, the user data is passed *separately* from the SQL string, and so *the DBMS makes no such assumptions*. The data is passed via a different channel of communication, so to speak. It is similar to `printf()`, where the expectations of forthcoming data are incorporated into a string and then the data itself is sent later.





Website Architecture

Lezione 13 | SQL

Using prepared statements, you do not have to escape the given strings for SQL quotes or anything like that. The data is passed very literally into the database. This also saves on the performance penalty of having to parse the user input or even pass over it in any way.

In MiniArc

The MiniArc framework has a more convenient interface for preparing and executing these statements. It wraps the process of creating a new statement and building an array of arguments. Here are a few examples:

PHP

```
<?php
$sql = "SELECT * FROM Users WHERE (Age > ?) AND (Gender = ?)";
$dbResult = Database::Query($sql, 33, 'F');

// Say you still want to prepare a statement for efficiency.
$dbStatement = Database::PrepareStatement($sql);
$dbResult1 = Database::QueryPrepared($dbStatement, 33, 'F');
$dbResult2 = Database::QueryPrepared($dbStatement, 25, 'M');
?>
```

The **Query()** function is variadic, so it can take any number of arguments. See the MiniArc documentation for more.

