

Business Information Modeling

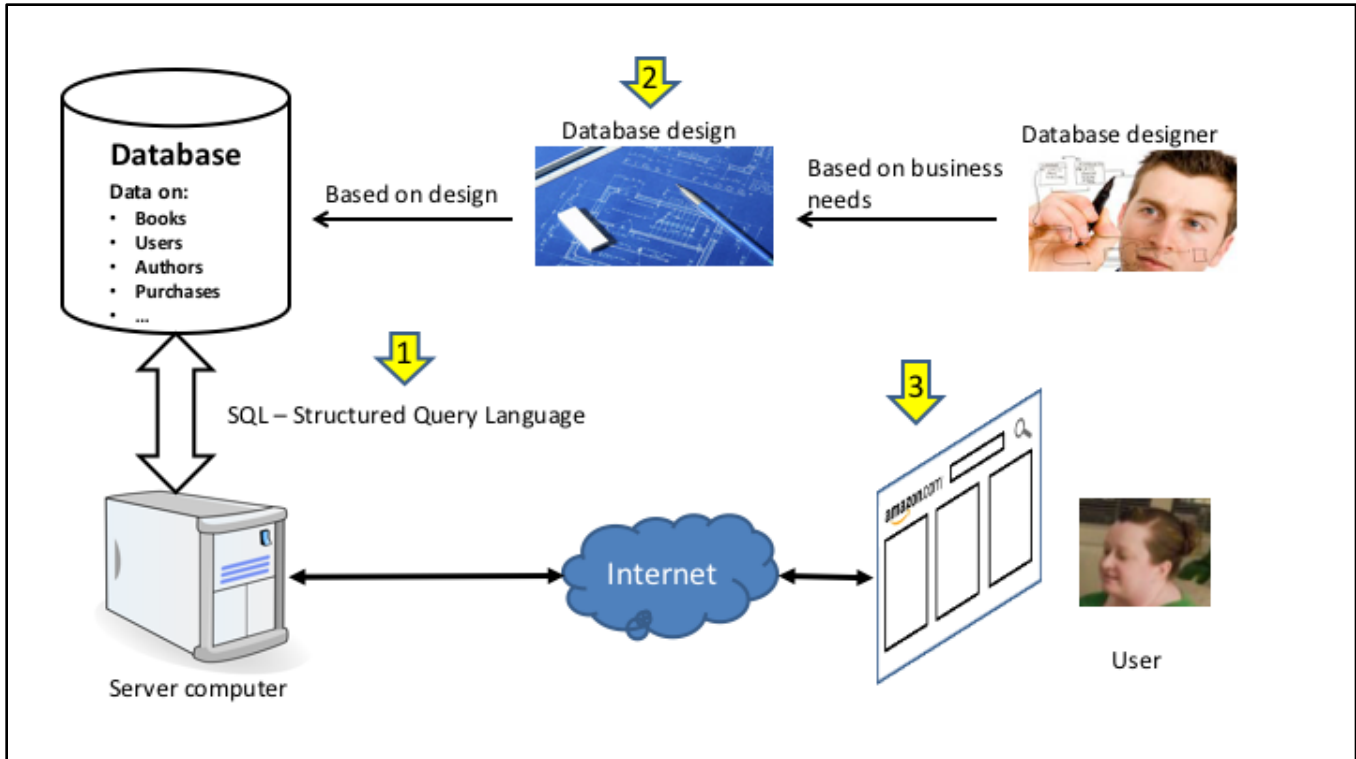
Enterprise-wide Accounting Information Systems -- I

The course focuses on the business application of information technology, with business coming first.

Information systems form the nervous systems of organizations today. Large organizations spend billions on technology infrastructure. People from all functional areas – Finance, Accounting, Marketing, Manufacturing, supply chain, etc, – participate in team efforts to build information systems and hence need at least a good grasp of the components of a business application and how they fit together.

Week 2

Introduction to SQL



We noted several crucial points in the previous lecture:

- The database occupies a very central place in an enterprise system because it serves as the “memory” of the organization. It records the most important details of almost all important business transactions.
- Unless the database is designed properly, it can very adversely affect an organization.
- People from all functional areas and from Information Technology (IT) play a significant role in designing the database – after all the database must meet the operational requirements of the functional areas and IT people need to ensure that its technical aspects meet user needs as well. These two entities – functional managers and IT do not speak a common language and hence it is essential for IT to understand business as much as it is important for business to be able to spell out its needs clearly to IT.

While the course will cover all aspects diagrammed above, we will start with 1) Structured Query Language (SQL), the language used to interact with a relational database. We will then go on to look at 2) database design and then look at how we 3) develop web based applications that use a database.

The classic Supplier-Parts database of

Chris J. Date

Suppliers

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Parts

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

Projects

jno	jname	city
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Terminator	Oslo
J7	Tape	London

Shipments

sno	pno	jno	ship_date	qty
S1	P1	J1	3/3/1999	200
S1	P1	J4	4/4/1988	700
S2	P3	J1	6/6/1988	100
S2	P3	J2	8/8/1999	200
S2	P3	J3	7/7/1977	200
S2	P3	J4	8/9/1999	500
S2	P3	J6	8/7/1998	400
S2	P3	J7	8/8/1988	800
S2	P5	J2	9/9/1999	100
S4	P6	J3	9/9/1999	400
S4	P6	J7	9/8/1988	300

Copy of this provided as PDF document on Blackboard – use as reference as you watch the video

C. J. Date, an early researcher on Relational Database systems and wrote THE BOOK on the topic – at least it remained as the go-to book for a long time.

He used the above database structure to introduce SQL. We use the same structure for its simultaneous simplicity and illustrative power. It enables us to see some of the key features without being overwhelmed by extraneous details that would come to play in any realistic example.

Once we learn the basics with this database, we will then turn our attention to more complex, real-life examples in the time worn spirit of “learn to walk before you learn to run.”

The database shows the following tables:

- Suppliers
- Parts
- Projects
- Shipments

Each of the first three tables has a “primary key” field. In relational databases we use a primary key as a unique identifier of a row of a table. Within a table the primary key is not allowed to repeat. Why do we need a primary key? Because we can never guarantee that the values in the other columns will not repeat. For example, we could very well have two suppliers named “Smith” with a rating of 20 in “London.” If this happens we will have no way to distinguish between them. The primary key enables us to distinguish between them because they both cannot have the same value for it.

The rest of the first three tables should be self-explanatory, except possibly for a clarification regarding the field “city” in each of them. For suppliers, the city represents the city they are located in; for parts let us take this to mean the city where our company stores the part and for projects, let us take it to mean the city where the project is being executed.

We could have given each of the city columns different names, like “supplier_city,” “part_city” and so on, but the current choice illustrates that different tables can have columns with the same name and can use the columns differently, as the example illustrates.

We will now look at how we can use the SQL language to retrieve information from the above “database.”

Query 1: All details of all suppliers

What we expect

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Used for all information retrieval operations in SQL -- Reserved word

Short cut to get all columns from the table mentioned in the FROM clause.

SELECT *
FROM suppliers

Reserved word to specify the table(s) from which we want information

Name(s) of the table(s)

We expect to get back everything from the suppliers table – all rows, all columns.

In SQL, we use the SELECT statement to retrieve information.

SQL is not case sensitive, but I have used uppercase for all SQL reserved words – words that we cannot use in naming tables or columns and lowercase for all of our names – table and column names. Remember **SQL is not case sensitive**.

In a SELECT statement we mention the column(s) we want and the tables from which we want them. Very soon we will also see that we can get a subset of the rows by specifying conditions.

SQL

Non Procedural

Say what you want without giving the system detailed step by step instructions.

Let the system figure out how best to fetch your results. Might not seem like a big deal, but it is in fact huge.

In complex database systems, we often need to change around various aspects of how information is stored internally. If we had to know how things are stored and write our queries accordingly, then when some of the mechanics change, our queries will stop working. Computer applications will become very brittle – or alternately people will be very scared to change anything lest the system breaks. Which also inhibits new developments and enhancements to a running system.

In fact, prior to relational databases and SQL, this was a huge issue and people found it extremely difficult to make any changes. In large systems, making a very small change took a huge amount of effort and people started referring to this phenomenon as the “maintenance tarpit” – imagine walking in a tarpit where each step forward requires great effort.

Truly, relational databases with SQL came as a knight in shining armor to save the software development community.

An example might make this clearer.

What If



You are returning to your city after a business trip. You get into a cab at the airport and give the cabbie step by step directions. The cabbie does not like it one bit and mutters something to himself and you hear something like “clo...,” but couldn’t care less – after all you are paying the big bucks, right?.

After several steps, you find that the only road leading to the great shortcut you were guiding the cabbie to is closed – which is what the cabbie was muttering to himself. You do not know any alternate route and have to now eat humble pie and let the cabbie get you out of trouble -- \$15 extra because of the detours.

Great story, but what does this have to do with SQL?

Everything.

Since you “knew” the details of all the streets, you gave the cabbie detailed step-by-step guidance. But when you found that the details had changed you were in a soup. Instead if you had got into the cab and told the cabbie “50 South orange avenue” then the problem of finding the route would be the cabbie’s and she probably knows more about the routes and is more aware of changes than you were and would find a good route taking all changes into account.

In an organization, if the people retrieving information from a database had to give step-by-step instructions in a procedural language then they would need to know the internal details of how the database is organized. When that changes – as it always does as organizations evolve – then all the programs that had detailed instructions for retrieving data based on how the data was organized earlier will fail to work because something had changed. Like you were stymied when the road was closed.

Using SQL is like specifying the final destination (non-procedural, no step by step instructions) and letting the database manager (cabbie) find the best way (route) to the destination, based on detailed knowledge.

Contrary to popular belief, knowing too much can sometimes be a problem. “Ignorance is bliss,” at least in SQL!!

Basic structure of SQL SELECT statement

```
SELECT <comma separated column names>  
FROM   <comma separated table names>  
WHERE  <condition>
```

The slide shows the prototypical SELECT statement.

The elements in uppercase represent reserved words that must be used as is (upper, lower or mixed case).

You need to replace the elements in angle brackets with whatever the specific query requires.

All details of all suppliers. Order the results in ascending order of names

What we expect

sno	sname	status	city
S5	Adams	30	Athens
S3	Blake	30	Paris
S4	Clark	20	London
S2	Jones	10	Paris
S1	Smith	20	London

```
SELECT *  
FROM suppliers  
ORDER BY sname
```

Use ORDER BY to control the order of rows in the output.

Default ordering is ascending. We can specify ASC or DESC.

Query 2: All details of all parts

Your turn -- pause the video, answer the question and then proceed.

What we expect

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

```
SELECT *  
FROM parts
```

If you did not get this right (very unlikely), be sure to review the previous slides before proceeding – otherwise you will be wasting your time.

Query 2: All details of all parts , but in descending order of weight.

Your turn -- pause the video, answer the question and then proceed.

What we expect

pno	pname	color	weight	city
P6	Cog	Red	19	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P1	Nut	Red	12	London
P5	Cam	Blue	12	Paris

```
SELECT *  
FROM parts  
ORDER BY weight DESC
```

Query 3: Supplier name and status for all suppliers

What we expect

sname	status
Smith	20
Jones	10
Blake	30
Clark	20
Adams	30

Since we want only some of the columns, not all, we specify the columns we want.

```
SELECT sname, status  
FROM suppliers
```

Here we want a few specific columns and not all. So we mention these after the SELECT clause. If we have more than one column, we separate column names with commas.

Column names must be spelled correctly – otherwise the system will not know what to do. They can be entered in any case – upper, lower or mixed.

Query 4: Part name, city and weight for all parts

Your turn -- pause the video, answer the question and then proceed.

What we expect

pname	city	weight
Nut	London	12
Bolt	Paris	17
Screw	Rome	17
Screw	London	14
Cam	Paris	12
Cog	London	19

```
SELECT pname, city, weight  
FROM parts
```

Query 5: All details for suppliers with status 20 or less

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S4	Clark	20	London

```
SELECT *  
FROM suppliers  
WHERE status <= 20
```

Now we add a condition

Query 6: Part number and color for parts that weigh 15 or more

Your turn -- pause the video, answer the question and then proceed.

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

pno	color
P2	Green
P3	Blue
P6	Red

```
SELECT  pno, color
FROM    parts
WHERE   weight >= 15
```

Query 7: Names of all suppliers from Paris

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

sname
Jones
Blake

```
SELECT  sname
FROM    suppliers
WHERE   city = 'Paris'
```

Condition looks similar, but
don't miss the single quotes
around 'Paris'


Always surround text values in
queries with single quotes

Don't do this for numbers.

Query 8: All details of Red parts

Your turn -- pause the video, answer the question and then proceed.

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



pno	pname	color	weight	city
P1	Nut	Red	12	London
P4	Screw	Red	14	London
P6	Cog	Red	19	London

```
SELECT  *  
FROM    parts  
WHERE   color = 'Red'
```

Query 9: All details of suppliers from Paris with a status of 20 or more

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

sno	sname	status	city
S3	Blake	30	Paris

```
SELECT *  
FROM suppliers  
WHERE city = 'Paris'  
AND status >= 20
```

Use AND to connect simple conditions to make more complex ones

Again, note quotes around 'Paris'

No quotes around numbers

Query 10: Part numbers of Red parts weighing less than 15

Your turn -- pause the video, answer the question and then proceed.

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



pno
P1
P4

```
SELECT  pno
FROM    parts
WHERE   color = 'Red'
AND     weight < 15
```

We used "<"
because our
condition required
that.

Query 11: Names of parts with weight between 10 and 15 (inclusive)

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

pname
Nut
Screw
Cam

```
SELECT pname
FROM parts
WHERE weight >= 10
AND weight <= 15
```

Correct, but verbose

```
SELECT    pname
FROM      parts
WHERE     weight BETWEEN 10 AND 15
```

Correct, and easier to follow

Query 12: Supplier number of Paris suppliers with status between 10 and 20 (inclusive). Your turn.

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



```
SELECT    sno
FROM      suppliers
WHERE     status BETWEEN 10 AND 20
AND       city = 'Paris'
```

Query 13: How many suppliers are there overall?

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



count(*)
5

```
SELECT COUNT (*)  
FROM suppliers
```

Aggregate function.
Calculates a value based on
elements from several rows
(possibly)

count(*) simply tells us how
many rows were returned.

Till now each row of output from each of our queries was created from just a single input row – you should go back and verify that this was indeed the case.

The queries only selected a subset of columns and rows as needed, but each row of output was based on just one row of input.

With aggregate functions this changes. Each row of output can be computed based on values from many input rows – aggregation.

In this slide we see the simple case of aggregation when all the rows in a table are counted.

Incidentally, we can put any column name instead of “*” in count(*), but since we are only counting the number of rows, “*” does just as well.

Getting a different column name to display in the output by using "AS"

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



Supplier count
5

```
SELECT      COUNT(*) AS "Supplier count"  
FROM        suppliers
```

By adding "AS" after the column name and before the comma, we can control the column name that appears in the output. You can do this for any column, not just for aggregate functions.

Query 14: How many parts are there overall? (Your turn)

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



count(*)
6

```
SELECT    COUNT (*)
FROM      parts
```

Query 15: City names and number of suppliers in each city

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



city	count(*)
London	2
Paris	2
Athens	1

```
SELECT    city, COUNT(*)
FROM      suppliers
GROUP BY  city
```

With a GROUP BY clause, the aggregate function applies to the groups and not the whole table.

Without it, the aggregate function applies to the whole table as in the previous two examples.

GROUP BY, if it appears, can only occur after the WHERE clause, unless there is no WHERE clause as in the above example.

Conceptually, you can think of GROUP BY as gathering together all the rows that have the same value for the GROUP BY fields (city, in this example).

So we can think conceptually that first the SQL processor creates the following:

sno	sname	status	city
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

London group

Paris group

Athens "group"

Note that rows having the same value for city have been gathered together.

The SQL processor then applies the aggregate function to each group (London, Paris and Athens) in the example, to create the output containing the number of rows in each group.

Crucial point about GROUP BY clause

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



city	count(*)
London	2
Paris	2
Athens	1

City names and number of suppliers in each city

```
SELECT    city, COUNT(*)  
FROM      suppliers  
GROUP BY  city
```

With a GROUP BY clause, the fields mentioned in GROUP BY must also appear in the SELECT clause. Otherwise the query would be nonsensical. Note that city is included in the SELECT clause.

When you have a GROUP BY clause, the SELECT clause can only have aggregate functions and column names that appear in the GROUP BY clause.

For example the following SQL would be nonsensical:

```
SELECT sno, city, count(*)  
FROM suppliers  
GROUP BY city
```

Just try to visualize the output.

The city field in the SELECT clause has only 3 values (the three cities) because we have grouped by city. The count(*) field also has three values – the count for each of the groups. But the sno field still has 5 values, because there are 5 suppliers. So there is a mismatch in the number of values for the fields mentioned in the SELECT clause and hence the query does not specify any meaningful table. To make it even clearer, the row for London in the output actually corresponds to two sno values. If we were to go ahead and produce any output, which of the two supplier numbers do we include?

Crucial point about aggregate functions

What output would you expect from the following SQL query?

```
SELECT sno, COUNT(*)  
FROM suppliers
```

Without GROUP BY we cannot mix aggregates and non-aggregates in the SELECT clause

With GROUP BY, all fields mentioned in the GROUP BY clause must appear in the SELECT clause.

We have 5 suppliers, but only one value for count(*), the number of rows in the table.

So the SQL processor can produce no meaningful output.

The query does not make sense.

Query 16: City names and number of projects in each city. (Your turn.)

```
SELECT    city, COUNT(*)  
FROM      projects  
GROUP BY  city
```


Query 17: City names and average status of suppliers from each city.

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



city	Avg status
London	20
Paris	20
Athens	30

```
SELECT      city, AVG(status) AS "Avg status"
FROM        suppliers
GROUP BY    city
```

City names and average status of suppliers from each city – order results by city.

```
SELECT    city, AVG(status) AS "Avg status"
FROM      suppliers
GROUP BY  city
ORDER BY  city
```

Query 18: City names and the number of projects in each city – your turn

jno	jname	city
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Termina	Oslo
J7	Tape	London



city	No. of projects
Paris	1
Rome	1
Athens	2
London	2
Oslo	1

```
SELECT      city, COUNT(*) AS "No. of projects"
FROM        projects
GROUP BY    city
```

Query 19: Part color and the average weight of parts of that color – your turn
Hint: use the AVG aggregate function

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



color	avg(weight)
Red	15
Green	17
Blue	14.5

```
SELECT    color, AVG(weight)
FROM      parts
GROUP BY  color
```

Query 20: Maximum and minimum weight in the parts table

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



Max wt	Min wt
19	12

```
SELECT      MAX(weight) AS "Max wt",  
            MIN(weight) AS "Min wt"  
  
FROM        parts
```

Query 21: City names and average status of suppliers from each city. List only cities with average status greater than 20.

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens



city	Avg status
London	20
Paris	20
Athens	30



city	Avg status
Athens	30

```
SELECT      city, AVG(status) AS "Avg status"
FROM        suppliers
GROUP BY    city
HAVING      AVG(status) > 20
```

Use HAVING to filter among groups
Be sure to understand difference between WHERE and HAVING

We used WHERE clauses to filter from the rows in a table.

We use HAVING not to filter rows, but to filter groups created by GROUP BY.

Pay careful attention to this distinction.

Query 22: Part color and average weight of parts having that color. List only parts with average weight below 17 – your turn.

pno	pname	color	weight	city
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London



color	avg(weight)
Red	15
Green	17
Blue	14.5



color	avg(weight)
Red	15
Blue	14.5

```
SELECT      color, AVG(weight)
FROM        parts
GROUP BY    color
HAVING      AVG(weight) < 17
```