# CS307

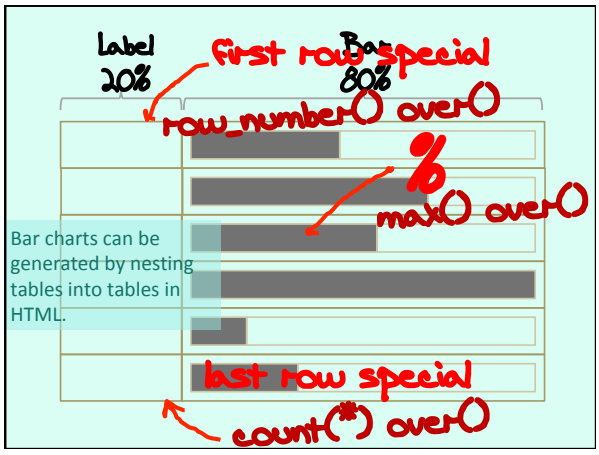
## Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

# HTML Charts

As pasting query output in Excel to generate charts (if you want an audit report to look credible, you need charts) is an exercise that is even more painful that reformatting plain text in Word, here is how you can generate HTML bar charts straight from SQL.




Bar charts can be generated by nesting tables into tables in HTML.

We need a number of numerical values for our HTML charts. We need percentages for bar sizes, and as this is HTML we must identify first and last rows for special processing.

```
select country_name,
       number_of_movies,
       round(100 * number_of_movies
             / max(number_of_movies) over (), 0) percentage,
       row_number() over (order by country_name) rn,
       count(*) over () cnt
from (select c.country_name,
             coalesce(x.number_of_movies, 0) as number_of_movies
      from countries c
           left outer join (select country as country_code,
                                   count(*) as number_of_movies
                            from movies
                            group by country) x
                on x.country_code = c.country_code) y
order by country_name
```

```
select case rn
         when 1 then
          '<html><body>' || '<h1>Movies per country</h1>'
          || '<table width="80%" style="border:none;margin:auto">'
         else ''
       end || '<tr><td width="20%">' || country_name
       || '</td><td width="80%">'
       || '<table width="100%"><tr>'
       || '<td style="background-color:grey" width="'
          || cast(percentage as varchar(5)) ||  '%" title="'
          || cast(number_of_movies as varchar(10))
          || ' movies"> </td>'
       || '<td width="' || cast(100 - percentage as varchar(5))
          || '%"> </td></tr></table></tr>'
       || case rn
            when cnt then '</table></body></html>'
            else ''
          end html
            from (
                                                    ) q
```

That's the previous query


And here is what you get, straight from SQL.
Good party trick.

# SQL Subtleties

There are a number of things that many people don't master too well and can be quite useful with SQL. Sadly, many people don't design their databases well enough, and this may require a bit of SQL back-bending.

```
select * from people
where born >= 1970
```

Relation

Relation

One thing which few people truly understand is that SQL isn't fully relational. If you start with a well-designed table with the required uniqueness constraints, it's a relation. When you apply a WHERE condition to it, you still have a relation.

```
select * from people
where born >= 1970
order by born
```

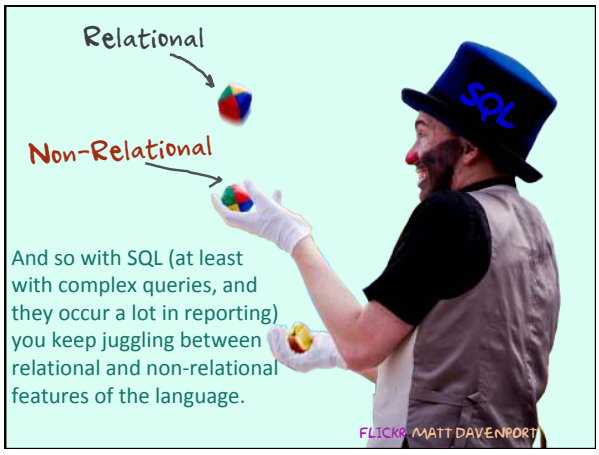
~~Relation~~ Array

Apply an ORDER BY to it, and you no longer have a relation. You have collected all the rows you wanted, you have sorted them, and now what you really have is more like an array.

```
select * from (
select * from people
where born >= 1970
order by born
)
where rownum <= 3
```

~~Relation~~ ~~Array~~ Relation

However, if you only want say the top three in your array, you can turn your array into a relation because then order no longer counts, just the fact that you have three of the oldest people born in 1970 or later.




And so with SQL (at least with complex queries, and they occur a lot in reporting) you keep juggling between relational and non-relational features of the language.

## Common Table Expressions

An interesting feature introduced mostly in the past 10 years in SQL is what is known in SQL Server circles as 'Common Table Expressions' or CTEs. They are available in every product, including SQLite (3.8 and above) except, at the time of writing, MySQL. MariaDB, compatible with MySQL, has them..

CTEs are simply a kind of factorization of subqueries. They use WITH, like recursive queries, but implement no recursion: it's just a convenient way of writing a query.

```
select c.continent,
       count(m.movieid) nb_films
from countries c
     inner join movies m
     on m.country = c.country_code
group by c.continent
```

Let's ask the question "what is the continent with the most films" (something which can be elegantly answered with a Window function, by the way)

```
select c.continent,
       count(m.movieid) nb_films
from countries c
     inner join movies m
     on m.country = c.country_code
group by c.continent
```

In this query, you need without a window function to compute twice the number of films per continent, once to find the maximum, and once to find the continent matching this maximum.

```
      select max(nb_films)
      from (select c.continent,
                   count(m.movieid) nb_films
            from countries c
                 inner join movies m
                 on m.country = c.country_code
            group by c.continent) continent_films
```

```
select c.continent,
       count(m.movieid) nb_films
from countries c
     inner join movies m
     on m.country = c.country_code
group by c.continent
having count(*) =
     (select max(nb_films)
      from (select c.continent,
                   count(m.movieid) nb_films
            from countries c
                 inner join movies m
                 on m.country = c.country_code
            group by c.continent) continent_films)
```

It can be done with a join, or as shown here, but in all cases you get a rather unwieldly and slightly scary query.

```
select c.continent,
       count(m.movieid) nb_films
from countries c
     inner join movies m
     on m.country = c.country_code
group by c.continent
having count(*) =
     (select max(nb_films)
      from (select c.continent,
                   count(m.movieid) nb_films
            from countries c
                 inner join movies m
                 on m.country = c.country_code
            group by c.continent) continent_films)
```

So, the idea is, as with numerical factorisation, to extract the common part and WRITE it only once. Whether it will be EXECUTED only once is the choice of the query optimizer.

```
with continent_films as
(select c.continent,
        count(m.movieid) nb_films
 from countries c
      inner join movies m
      on m.country = c.country_code
 group by c.continent)
```

Common Table Expression (CTE)

We give it a name using WITH, after which we can use it. Note that you can define a comma-separated CTEs (with q1 as (), q2 as (), ...) and that any CTE can refer to a CTE that has been defined before it.
Another important thing to notice is that, contrary to what happens with recursive queries (also defined using WITH) there is no parenthesed list of columns returned after the name given to the query.

```
with continent_films as
(select c.continent,
        count(m.movieid) nb_films
 from countries c
      inner join movies m
      on m.country = c.country_code
 group by c.continent)
 select *
 from continent_films
 where nb_films = (select max(nb_films)
                   from continent_films)
```

Makes for a query that is easier to read. This is frequently used in UNION statements, because it is relatively common to encounter identical subqueries in different parts of a UNION statement.

Interesting use of left outer join

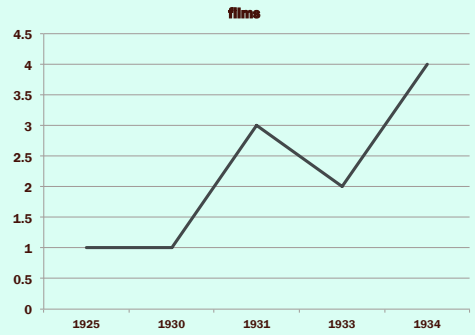
Left outer joins are extremely useful for "filling the gaps", especially in time series. This is a usage that may seem a bit remote from joins as we have seen them, but which is quite practical.

```
select year_released, count(*) as films
from movies
where year_released between 1925 and 1934
group by year_released
```

| year_released | films |
|---|---|
| 1925 | 1 |
| 1930 | 1 |
| 1931 | 3 |
| 1933 | 2 |
| 1934 | 4 |

Problem

If we look at very old films, we may not have films released every year, and gaps in the series of years. This may be a problem if we want to generate a chart from this data, or compute values (rate of increase?) that assume regular intervals.




```
select year_released, count(*) as films
from movies
where year_released between 1925 and 1934
group by year_released
```

One way to solve this problem is to "outer join" the result of our query to a full, gap-less, list of years.

```
select x.year_released, count(m.movieid) as films
from (select 1925 as year_released
      union all select 1926
      union all select 1927
      union all select 1928
      union all select 1929
      union all select 1930
      union all select 1931
      union all select 1932
      union all select 1933
      union all select 1934) x
    left outer join movies m
    on m.year_released = x.year_released
group by x.year_released
```

There are various tricks for generating lists of values

You can sometimes be cleverer

23

| year_released | films |
|---|---|
| 1925 | 1 |
| 1926 | 0 |
| 1927 | 0 |
| 1928 | 0 |
| 1929 | 0 |
| 1930 | 1 |
| 1931 | 3 |
| 1932 | 0 |
| 1933 | 2 |
| 1934 | 4 |

With the outer join, count(movied), which only counts not null values (remember that aggregate functions ignore nulls) will return 0 when we have no films for that year (count() never returns NULL). Note that count(*) wouldn't work as it counts rows.

24

## Interesting use of min/max

Aggregate functions are often an elegant way to solve problems in which there is no explicit reference to an aggregate, but in which comparing several rows is required for answering the question.
Take for instance this case, which probably occurs pretty often in web stores when every ordered item in a command isn't in stock.

Orders composed of a variable number of articles

Articles may be in status available (A) or pending (P)

Shipment occurs when ALL articles are in status A

Most people (and books) will show you a query like this.

```
select o.order_id, ...
from orders o
where not exists
      (select null
       from order_detail d
       where d.order_id = o.order_id
         and d.status <> 'A')
```

Correlated: inefficient with big volumes

It hurts because you need to fully scan the table of orders, and for each row check that there is nothing pending.

In this query you execute some massive operations, but not tons of small queries, and on big volumes it will be far more efficient. It takes advantage of the fact that 'A' comes before 'P' alphabetically. If the max is 'A', then there is no 'P'

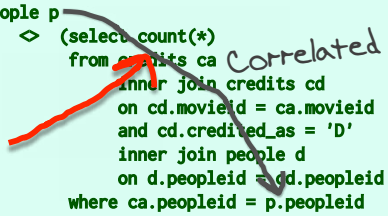
```
select o.order_id, ...
from orders o
     inner join order_detail d
     on d.order_id = o.order_id
group by o.order_id, ...
having max(d.status) = 'A'
```

All values are equal :
having min(...) = max(...)

## exists compared to count

When it comes to correlated subqueries, if you execute them a number of times, then you don't want to spend too much time in each one.
A surprising high number of people use count(*) when they are only interested in existence and only want to check whether the result of the count is zero or not. Suppose that you are scanning a 1,000,000 row table. If you use EXISTS and if the third row you inspect verifies the condition, you can stop here. If you use COUNT, you must check all rows to find how many other rows also verify the condition.

```
select p.first_name, p.surname
from people p
where 0  <>  (select count(*)
              from credits ca
                   inner join credits cd
                   on cd.movieid = ca.movieid
                   and cd.credited_as = 'D'
                   inner join people d
                   on d.peopleid = cd.peopleid
              where ca.peopleid = p.peopleid
                and ca.credited_as = 'A'
                and d.first_name = 'Howard'
                and d.surname = 'Hawks')
```




Some people would look for people who have played in a Howard Hawks film that way, counting each time in how many Howard Hawks films they appear. One is enough to qualify.

```
select p.first_name, p.surname
from people p
where exists (select null
              from credits ca
                   inner join credits cd
                   on cd.movieid = ca.movieid
                   and cd.credited_as = 'D'
                   inner join people d
                   on d.peopleid = cd.peopleid
              where ca.peopleid = p.peopleid
                and ca.credited_as = 'A'
                and d.first_name = 'Howard'
                and d.surname = 'Hawks')
```

I wouldn't use a correlated query in that case, but this would hurt less.

GROUP BY: Multiple joins with the same table

```
select d.city, a.city,
...
from flights f
     inner join airports d
        on d.code = f.departure
     inner join airports a
        on a.code = f.arrival
```

GROUP BY can sometimes replace self-joins, especially when a table appears many, many times. You can display the names of cities between which you have flights with this (legitimate) query.

Multiple joins with the same table


```
select a.city,
...
from flights f
     inner join airports a
        on a.code in (f.departure,
                      f.arrival)
```

You can also join once, but in that case your query will return two rows in the same column, which is probably not how you want your result displayed.

Multiple joins with the same table

Paris

Beijing

```
select case a.code
         when f.departure then a.city
         else null
       end, ...
from flights f
     inner join airports a
        on a.code in (f.departure,
                      f.arrival)
```

With CASE you can spread your result over two columns, with a name and a NULL, in opposite positions.

Multiple joins with the same table

Paris | Beijing

```
select max(case a.code
         when f.departure then a.city
         else null
       end), ...
from flights f
     inner join airports a
        on a.code in (f.departure,
                      f.arrival)
group by ...
```

Apply MAX(), which ignores NULL, and you squash your two rows into one.

## Limiting damage

Another interesting use of window functions is limiting damage with runaway queries, especially in interactive environments where queries are dynamically built from user input that may not be very selective, thus leading to queries that return h*uuuu*ge numbers of rows.

```
select count(*)
from (original query)
```

```
if count <= maxcnt:
    original query
```

I have seen people trying to solve this problem in this way, wrapping the query first into a count(*) and checking how many rows it retrieves. If the first query saves on data transfers and data rendition, it still executes the painful part. And if the query is OK, in practice you run it twice.

```
select ... ,
       count(*) over () cnt
from ...
where ...
  and rownum <= maxcnt + 1
order by ...
```

A far better way to do it (here with Oracle) is to limit the output, then sort, and compute with a window function the number of rows returned.

If cnt > maxcnt:
"Refine your query"

What are we doing here?

Suppose that we consider that the query should normally not return more than 2,000 rows. We limit what we get to 2,001 at most, only sort these rows (not a big number, should be fast) and count in the process how many rows we have.
If we find 2,000 rows or less, we can be certain that we have retrieved all rows of interest. If we have 2,001 rows, the ordered set may be wrong, but we won't display it so it doesn't matter. We're putting a cap over what may go wrong.

# Fuzzy searches

Life would be easy with databases if we were looking all the time for strict equality of data. Unfortunately, this is rarely the case, especially with text data. Even if we try to normalize and standardize data as much as we can, constraints will do little against typos and some misspellings (surnames are a nightmare for emergency services at hospitals). We'll explore a few problems and possible solutions.

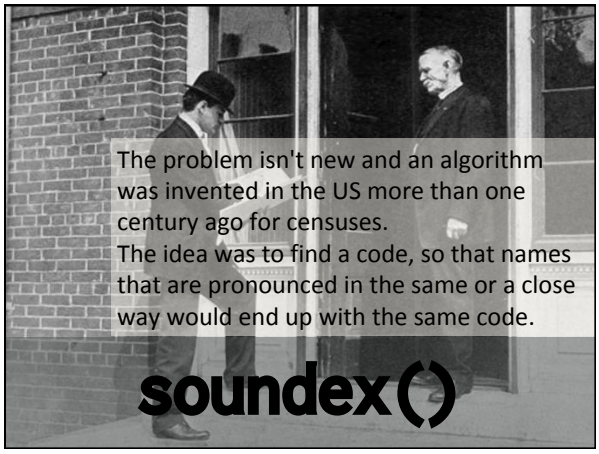
Guilgood
Gillgood
Gielgud
Gilgud

Picture by Allan Warren

This famous British Shakespearean actor probably had one of the most misspelt names in his country (Gielgud is correct). His family was of German origin and they never felt a need to make their name look more English (contrary to the von Battenbergs who became Mountbattens, or Sachs-Coburg Gothas who became Windsors during WWI, much to the amusement of Wilhelm II of Prussia, himself a grandson of Queen Victoria)

Zhou
Chiew
Chiau
周
Chu
Chou
Jhou

Spelling issues are even worse with name originally written in another script than the Latin alphabet, and for which transcription is merely based on sound (and pronunciation may vary). This common Chinese surname may be turned into many variants.




The problem isn't new and an algorithm was invented in the US more than one century ago for censuses.
The idea was to find a code, so that names that are pronounced in the same or a close way would end up with the same code.

**G 4 2 3**

Basically you retain the first letter, drop vowels, letters that sound like vowels (w = oo), h (often silent), then replace similarly sounding consonents by the same digit, before eliminating one of two successive identical digits. Gielgud and Guilgood both become G423.

| | |
|---|---|
| B, F, P, V | 1 |
| C, G, J, K, Q, V, X, Z | 2 |
| D, T | 3 |
| L | 4 |
| M, N | 5 |
| R | 6 |

wells

W420

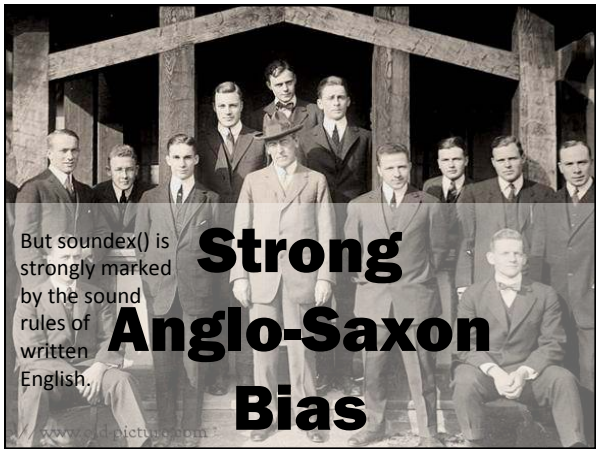
Welles
Willis
Wallach

Now, if people look for "Wells", as in H.G. Wells, a name absent from the film database, nothing will be found. However, if we look for names the soundex of which is the same as the soundex of "Wells", we find many, among which those three.

**wells**

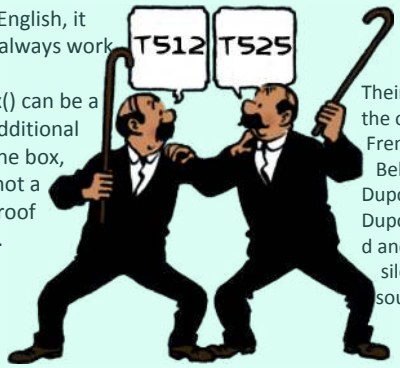
The question becomes: which is the best match? one way to find it is to remove from the candidate names every letter from the user input, and see what remains.

| Welles | Willis | Wallach |
|---|---|---|
| i | i | a ach |





Even in English, it doesn't always work so well. soundex() can be a useful additional tool in the box, but it's not a bullet-proof method.




Their names in the original French (from Belgium) are Dupond and Dupont. Final d and t are silent. Fools soundex too.

**Thompson** **Thomson**

**Title to search:** 2001, a space odyssey

## 2001: A Space Odyssey

Even if finding a simple surname when you hesitate about spelling isn't easy (well, there is LIKE), difficulty pales in comparison of searching by title. The longer the title, the more opportunities for having something wrong, and sometimes it may be as little as punctuation.

What people use is known as "full-text search" and here is a simplified version of how it works. You split a text in (pure) words, eliminate words that are too common, then associate each word with the film identifier.

## FULL-TEXT SEARCH

**2001: a space odyssey**

**2001**
~~**a**~~
**SPACE**
**ODYSSEY**


These words are stored in a special table. Note that a true full-text search engine "stems" words - it will recognize singular and plural as well as infinitive and preterit or past participle as the same word.

```
create table movie_title_ft_index
      (title_word    varchar(30) not null,
       movieid       int not null,
        primary key(title_word, movieid),
        foreign key (movieid)
            references movies(movieid))
```

Title to search: 2001, a space odyssey

**2001**

**SPACE**

**ODYSSEY**

How are we going to use this? When people enter a title, we perform the same operation as before of isolating and standardizing words that look important.
Then we are going to look for the MOVIEID values associated to these words in the previous table.

Trouble is, the same word will probably occur in several titles


**2001**

**SPACE**

**ODYSSEY**

We need a way to qualify matches.

```
select movieid
from (select movieid,
             rank()
             over (order by hits desc) as rnk
      from (select movieid,
                   count(*) as hits
            from movie_title_ft_index
            where title_word in
               ('SPACE', 'ODYSSEY', '2001')
            group by movieid) q1) q2
where rnk = 1
```

Such a query might serve the purpose. We count how many of the words we find by film for which at least one is found.

Then we rank. Notice that here we are interested by ties.

# Ties ?

Talking about ties, we should be able to solve them.

Title to search: Casablanca


The first one is probably the one that was meant.

# Ties ?

**Title to search:** Casablanca

Different strategies may be adopted. And there is always the option of displaying multiple films in case of doubt (remakes, for instance).

Remove input words from title found

See what remains!

Compare lengths

# Typos ?

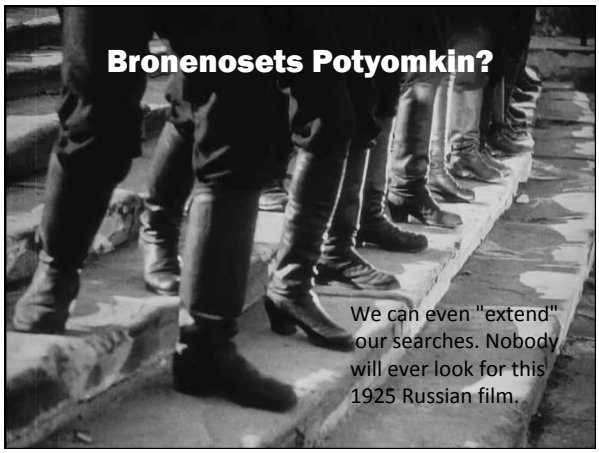
**Title to search:** Casabalnca

We may even extend our "index" with common misspellings and typos (what do they do in search engines?)

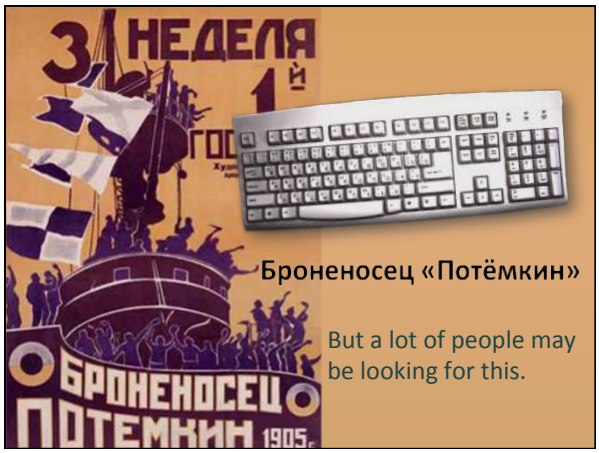
CAZABLANCA

CASBLANCA

CASABALNCA




We can even "extend" our searches. Nobody will ever look for this 1925 Russian film.

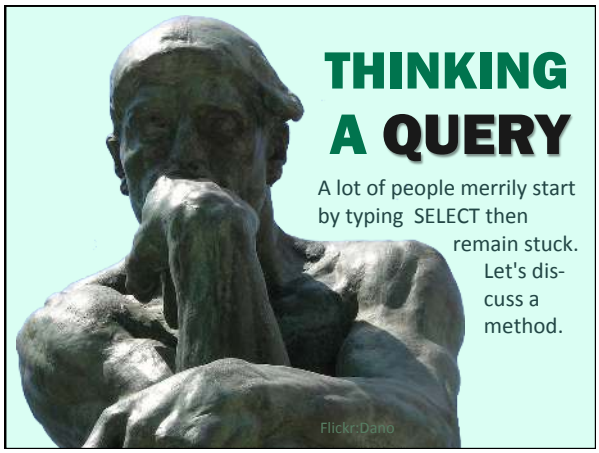



Броненосец «Потёмкин»

But a lot of people may be looking for this.


3 НЕДЕЛЯ 1 Й ГОСКИНО
БРОНЕНОСЕЦ ПОТЕМКИН 1905
Battleship Potemkin
Panzerkreuzer Potemkin
Le cuirassé Potemkine
El acorazado Potemkin
المدمرة بوتمكين
Or for this.


БРОНЕНОСЕЦ
ПОТЕМКИН
POTEMKINE
CUIRASSE
ACORAZADO
PANZERKREUZER
BATTLESHIP
POTEMKIN
BRONENOSETS
POTYOMKIN
This might be a solution.
How do you think IMDB are doing?


THINKING A QUERY
A lot of people merrily start by typing SELECT then remain stuck. Let's discuss a method.
Flickr:Dano


IMPORTANT: queries must be thought as successive layers, from the inside out.
Photo © 2011 J. Ronald Lee, CC Attribution 3.0

Let's take an example and see how we can turn the expression of a query problem into SQL, and if possible efficient SQL.

If we want to do it right, we must proceed in about 5 steps.

Find the titles and the directors of movies for which we have three actors or more.

movies
credits
+ people
credits

## STEP 1: The Scope

The first step consists in identifying the tables we need. Note that often some entities aren't required, just relationships pointing to them. We don't need here PEOPLE for actors, we just count them and CREDITS is good enough.

movies for which we have three actors or more.

## STEP 2: Aggregates

The second step consists in computing aggregates. A huge number of queries involve aggregates, in a way or another. We must perform them before joining when possible.

```
select movieid
from credits
where credited_as = 'A'
group by movieid
having count(*) >= 3
```

## STEP 2: Aggregates

In that case we'll have to look for all actors in CREDITS, but we are operating against small amounts of data (just identifiers). If a condition other than being an actor had allowed to reduce the number of rows before aggregating, we should have taken advantage of it.

```
select movieid
from credits
where credited_as = 'A'
group by movieid
having count(*) >= 3
```

## STEP 3: Main Filter

There is usually in a query a main filter, THE condition that defines the most precisely the subset of rows we want to retrieve. In that case it's the aggregate computed earlier.

```
select m.movieid, m.title, m.year_released
from (select movieid
      from credits
      where credited_as = 'A'
      group by movieid
      having count(*) >= 3) m0
     inner join movies m
       on m.movieid = m0.movieid
```

## STEP 4: Core Joins

At this stage you should add "core joins". Core joins are either the ones that contribute to filtering (not in that case) or that returns information that you should return and that shall be here.

```
select core.title, core.year_released,
       p.first_name, p.surname
from (select m.movieid, m.title, m.year_released
      from (select movieid
            from credits
            where credited_as = 'A'
            group by movieid
            having count(*) >= 3) m0
           inner join movies m
             on m.movieid = m0.movieid) core
left outer join credits c
  on c.movieid = core.movieid
 and c.credited_as = 'D'
left outer join people p
  on p.peopleid  = c.peopleid
order by core.title,
         core.year_released,
         p.surname
```

## STEP 5: Polish

And finally you deal with everything else: outer joins that may return `core` additional information (or not), ordering. All this is just to make the result nicer.

A brief summary of points that have little to do with SQL syntax or relational theory but are often of high practical importance.

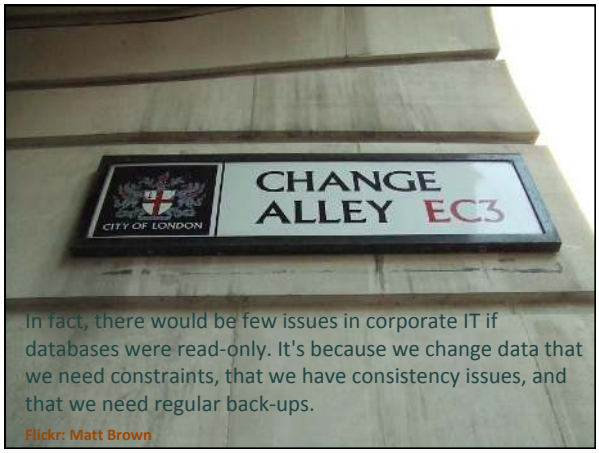
Searching for names and in text is tricky.
**soundex()** is no silver bullet. Full-text search is helpful for articles.

Fuzzy searches are basically counting exact matches on "atomic" elements, and ranking.

Build queries bit by bit. Scope, aggregates, main filter, core joins, then qualifying joins, ordering etc.

SELECT is the very big chunk in SQL (for one good reason, a large part of what you have seen with SELECT is also used in DELETE or UPDATE that have to retrieve rows before deleting or changing them)


But there are interesting things to say about other commands too.

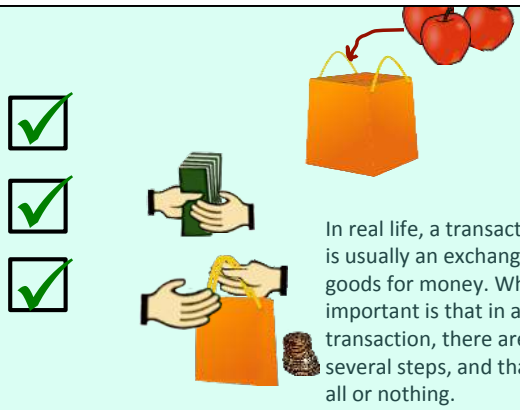



In fact, there would be few issues in corporate IT if databases were read-only. It's because we change data that we need constraints, that we have consistency issues, and that we need regular back-ups.

Flickr: Matt Brown

# Transaction

First of all, let's talk about something very, very important which is directly linked to the idea of consistency: transactions.



In real life, a transaction is usually an exchange of goods for money. What is important is that in a transaction, there are several steps, and that i's all or nothing.

Flickr: Randen Pederson


The classical database example is transfering money from your current account to your saving account. You need to change the balance in two rows, what happens if the system crashes in the middle?

| Account type | Account number | Balance |
|---|---|---|
| CURRENT ACNT | 1234567 | 300.00 |
| SAVINGS ACNT | 8765432 | 1600.00 |

At worst, balances should remain what they were before you initiated the transfer.

# Single Unit

This idea that one business operation may translate into several database operations that must all succeed or fail is of prime importance to a DBMS.
Some products require a special command to start a transaction (BEGIN is sometimes START)

```
begin transaction
  insert
  update
  delete
```

Other products such as Oracle or DB2 automatically start a transaction if you aren't already in one when you start modifying data.

A transaction ends when you issue either COMMIT (which is like an OK button) or ROLLBACK, which cancels everything you have done since the beginning of a transaction (you can sometimes cancel a subpart of a transaction, but it's not much used)

```
commit
```

OK

ROLLBACK automatically undoes everything. You can no longer do it after COMMIT.

```
rollback
```

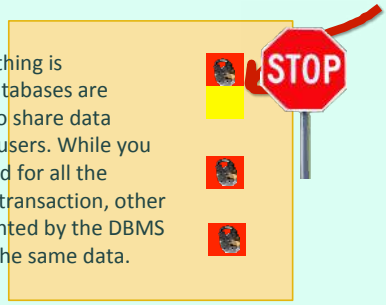
Cancel

## rollback

| Account type | Account number | Balance |
|---|---|---|
| CURRENT ACNT | 1234567 | 300.00 |
| SAVINGS ACNT | 8765432 | 1600.00 |

If during a transfer the debit from one account went well but you couldn't credit the other one (in some countries, some saving accounts cannot hold more than a given amount, it would be a reason for failure) ROLLBACK will restore the original balance.

## begin transaction




One important thing is concurrency: databases are usually meant to share data between many users. While you change data, and for all the duration of the transaction, other users are prevented by the DBMS from changing the same data.

commit

Flickr: Andrew Magill


Many products (MySQL is one of them) start in "autocommit" mode, which means that every change is automatically commited and cannot be undone otherwise than by running the reverse operation (not always easy). Some interfaces such as JDBC (Java DataBase Connectivity) always start in autocommit mode, even when accessing products such as Oracle that never natively work in such a mode.

Flickr:Mr.McClurken

For products such as Oracle and MySQL, any change to the structure of the database (DDL operations) automatically commits all pending changes to data; DDL operations cannot usually be rolled back.
This isn't the case with SQL Server, for wich a transaction can contain DDL statements.

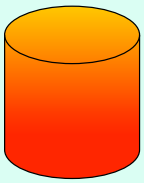
ORACLE MySQL

create
drop, alter

commit

# Data Change

One thing which is important to understand (we'll come back to it) is that transactions are a kind of "blind spot" in which the database switches from a consistent state to another consistent state, but during which internally things may not be consistent. This has a huge impact on a number of highly important practical questions: what will other users see? Can we backup the database while it's active?



**Begin Transaction**

**Commit**

## ACCOUNTS

| Account type | Number | Balance |
|---|---|---|
| CURRENT ACNT | 1234567 | 300.00 |
| SAVINGS ACNT | 8765432 | 1600.00 |

After having used the transfer-between-accounts example that you see everywhere, let's haste to say that a bank will never run two updates in one transaction for this kind of operation. Why? If you have ever looked at one of your bank statements, you have seen the list of operations since the last statement. What is stored is operations, and balances are recomputed once in a while.

## ACCOUNTS

| Account type | Number | Balance | Date |
|---|---|---|---|
| CURRENT ACNT | 1234567 | 300.00 | 1-Sep |
| SAVINGS ACNT | 8765432 | 1600.00 | 1-Sep |

## OPERATIONS

| Account number | Amount | Operation | Date |
|---|---|---|---|
| 1234567 | 100.00 | DEBIT | 3-Sep |
| 8765432 | 100.00 | CREDIT | 3-Sep |

This is what will happen in the real world. A batch program may run daily, weekly or monthly to recompute the new balance. In between, balances can be recomputed from the old balance and by aggregating the latest operations. There are few updates, and a lot of inserts.

In the same way, do you think that when you click on a "Like" button, you simply update a counter? Certainly not, because you could artificially inflate popularity. Usually, on a web site you can only vote once for something from either the same IP address (which is transmitted to the web server), or the same member id if you must be signed-in to be allowed to vote (safer)

Primary Key

| Videoid | Memberid |
|---|---|
| AGOivHmYbh8TGWzQ | 27097019 |
| BZVGhkMurOZDwqTH | 27097019 |
| FRpohgTYVbDEhkMr | 22109719 |
| AGOivHmYbh8TGWzQ | 78943285 |

A primary key will ensure that everybody can only vote once for a video. Once again, what appears as an update is in fact an insert.

On the other hand, deletes often are virtual deletes; in fact they are updates of a flag. One good reason is foreign keys, which prevent physical deletion. A web store cannot physically delete an item no longer on sale but which appears in many past orders, you would have to delete much of your sales history.

| Id | Label | Active |
|---|---|---|
| 123 | TEST | N |

We have already seen the syntax for inserting one row.

```
insert into table_name
  (column1, column2, ..., columnn)
values (value1, value2, ..., valuen),
       ...
       (valuep, valueq, ..., valuez)
```

Most products (exceptions are Oracle and SQLite) allow inserting several rows in one statement, with a comma-separated list of row data between parentheses.


```
countries
    country_code char(2) primary key
    country_name varchar(50) not null
    continent    varchar(20) not null
    population   int
```

This works with any SQL DBMS

```
insert into countries(country_code,
country_name, continent, population)
values('us', 'United States', 'AMERICA',
       314000000)
```

If you don't specify the columns, it's understood as "all the columns, in the same order as they are displayed when running select *"

```
insert into table_name
values (value1, value2, ..., valuen)
```


This is very dangerous in a program, because you can always add a column to a table, and often remove a column or change the default order.

What happens when you omit a column? The value inserted is the default one if defined, otherwise it will be NULL.

```
insert into table_name
  (col1, col2, col4)
values (value1, value2, value4)
```


```
insert into countries(country_code,
country_name, continent)
values('kz', 'Kazakhstan', 'ASIA')
```

For instance, using the same table as before, this will insert NULL for the population of Kazakhstan. That's allowed, because I haven't made population a mandatory attribute.

```
create table <table_name>
            (...
             <column_name>    <data type>
                   default <default_value> not null,
             ...)
```

If I want to specify a default value, I do it when I create the table. If I have a default value for a mandatory column, it will be OK to omit it in an insert statement. Note that if you have a default value for a nullable column, nothing prevents you from explicitly inserting NULL, and the default value won't be used.

An interesting question is how we populate these numerical identifiers that we are using as primary keys when what identifies a row in real life is a complication combination of columns.

**movieid ?**

Querying the next value to use is a sure recipe for conflicts. Several users may get the same one.

```
select max(movieid) + 1
from movies
```


User1 User2

## SEQUENCE

```
create sequence movie_seq
```

There are two approaches. One is to use special database objects called sequences, which are simply number generators.