CS307

Database Principles

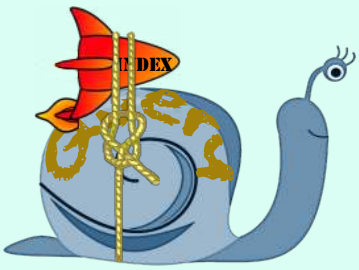
Stéphane Faroult
faroult@sustc.edu.cn

朱悦铭 Zhu Yueming zhuym@sustc.edu.cn

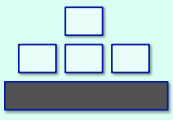
What we have seen last time:

Indexes

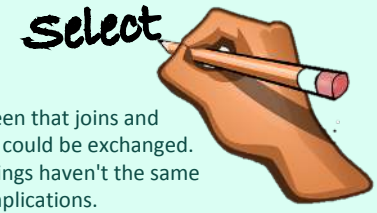
- B-Trees – **VERY** important for performance
- PK and unique constraints create indexes
- Has to be maintained / storage
- Cannot always be used
- EXPLAIN
- Indexing expressions



Don't fall into the trap that says "if I have a query that is slow, I just have to add an index and it will be fast". The index HAS TO BE USABLE, and besides remember that it depends on volumes returned.



In fact, proper index usage depends a lot on how you write your queries and this is what we are going to see now.



We have seen that joins and subqueries could be exchanged. But all writings haven't the same indexing implications.

```
select count(c.movieid)
from credits c
     inner join people p
             on p.peopleid = c.peopleid
where c.credited_as = 'A'
   and p.surname = 'Hanks'
   and p.first_name = 'Tom'
```

Let's take this join. Basically we can run it in two different ways, better explained with subqueries.

We can look for all peopleid values for actors, and check whether it's Tom Hanks'

```
select count(c.movieid)
from credits c
where c.credited_as = 'A'
   and exists (select null
               from people p
               where p.peopleid = c.peopleid
                 and p.surname = 'Hanks'
                 and p.first_name = 'Tom')
```

```
select count(c.movieid)
from credits c
where c.credited_as = 'A'
   and c.peopleid in (select peopleid
                      from people p
                      where p.surname = 'Hanks'
                        and p.first_name = 'Tom')
```

Or we can look for Tom Hanks' peopleid value, and look for films in which it's found with 'A'

```
select count(c.movieid)
from credits c
where c.credited_as = 'A'
   and exists (select null
               from people p
               where p.peopleid = c.peopleid
                 and p.surname = 'Hanks'
                 and p.first_name = 'Tom')
```

It means completely different things. In the first case I scan CREDITS and use the PK in PEOPLE each time

In the second case I use the PEOPLE index on (SURNAME, FIRST_NAME)

```
select count(c.movieid)
from credits c
where c.credited_as = 'A'
   and c.peopleid in (select peopleid
                      from people p
                      where p.surname = 'Hanks'
                        and p.first_name = 'Tom')
```

I may not be able to use the PK on CREDITS

```
(movieid, peopleid, credited_as)
```

Constraints
with an eye to
Performance

Which brings us to another topic: when I define constraints, can I do it in a clever way?

Primary key → **INDEX**

Unique constraint → **INDEX**

I have told you that PK and UNIQUE constraints were both creating an index, to quickly check whether an entry is already known.

UNIQUE

(first_name, surname)

(surname, first_name)

Which one?

From a logical point of view, when you say that a combination of columns is unique, the order doesn't matter. If it doesn't matter, perhaps we can use it at our benefit?

I can expect different types of queries that will refer to the columns in the unique constraint.

```
select *
from people
where surname = 'Spielberg'
  and first_name = 'Steven'

select *
from people
where surname = 'Spielberg'

select *
from people
where first_name = 'Steven'
```

In all likelihood the most common ones will be the first two ones. If I want the index to be usable in both cases SURNAME should come first.

```
select *
from people
where surname = 'Spielberg'
  and first_name = 'Steven'

select *
from people
where surname = 'Spielberg'

select *
from people
where first_name = 'Steven'
```

second

first

What about storing rows in some order?

Another idea to speed up queries would be to store rows in a given order. If order doesn't matter from a relational point of view, we can once again perhaps use it for our own benefit.

A regular table (unordered, rows are inserted where room is found) is called a

Heap-organized table

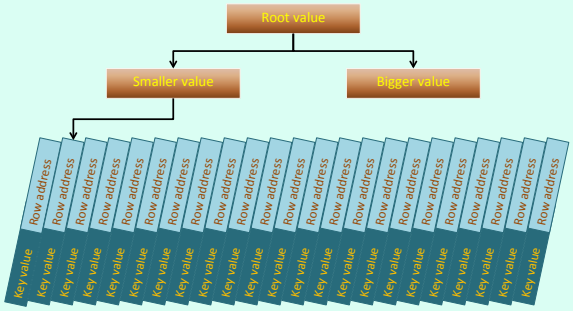
A table that is ordered is called a

Clustered index (Microsoft SQL Server)

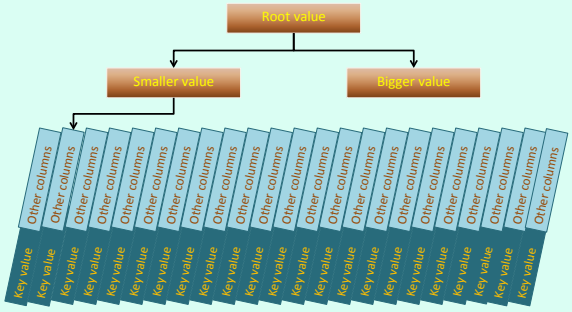
Index-organized table (ORACLE)

why "index"? Because indexes are strongly ordered structures, and an ordered table follows the same pattern.

In an index, you find key values and row addresses.

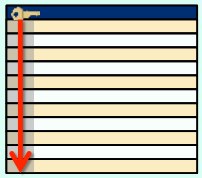


In an ordered table, row data replaces the row address.



For internal storage reasons, it works better with few columns.

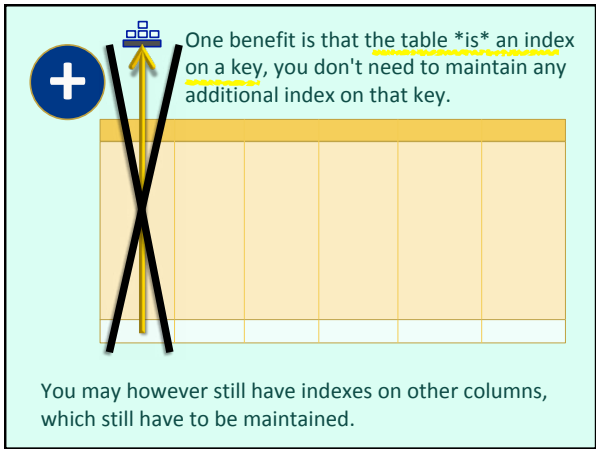
That way you have all rows ordered according to the key value.

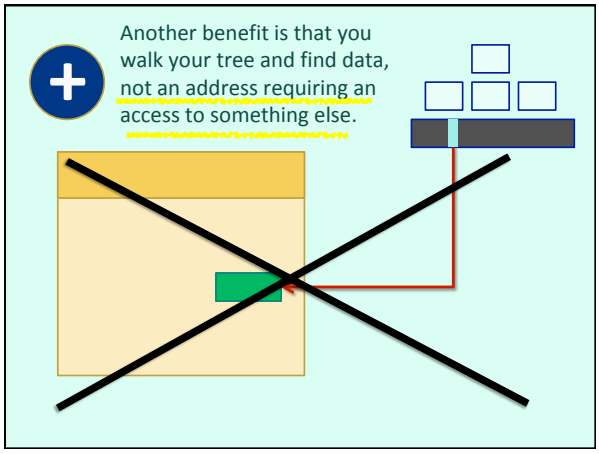


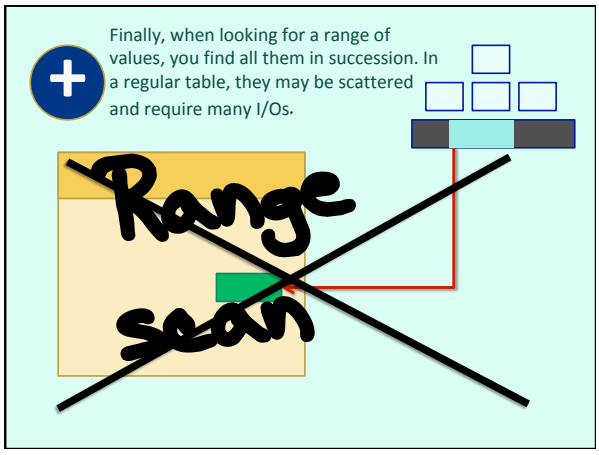
Benefits

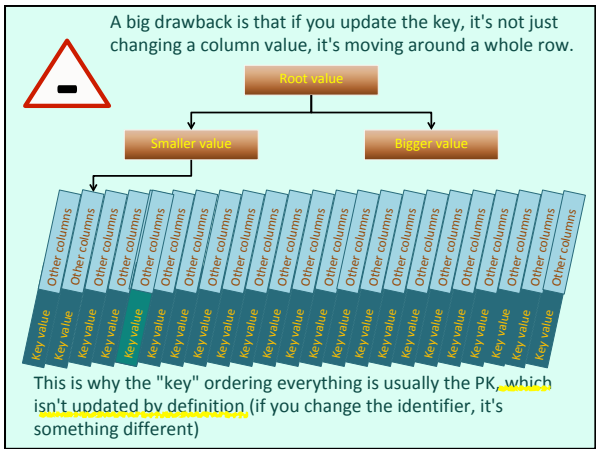
Drawbacks

Like any clever idea, it has its benefits and drawbacks. Interestingly, this structure is much used with SQL Server (almost standard) and with MySQL/InnoDB, and rarely with Oracle.


One benefit is that the table *is* an index on a key, you don't need to maintain any additional index on that key.
You may however still have indexes on other columns, which still have to be maintained.


Another benefit is that you walk your tree and find data, not an address requiring an access to something else.


Finally, when looking for a range of values, you find all them in succession. In a regular table, they may be scattered and require many I/Os.
Range scan


A big drawback is that if you update the key, it's not just changing a column value, it's moving around a whole row.
Root value
Smaller value
Bigger value
Other columns
Key value
This is why the "key" ordering everything is usually the PK, which isn't updated by definition (if you change the identifier, it's something different)

insert

Inserting a row also means inserting it at the right place, which may involve shifting around a lot of bytes to make room. Not an issue if rows are inserted in the same order as the key (for instance if the key is an auto-numbered column or the timestamp at insertion). Big issue if the order of insertion is more or less random in reference to the key.

Right place

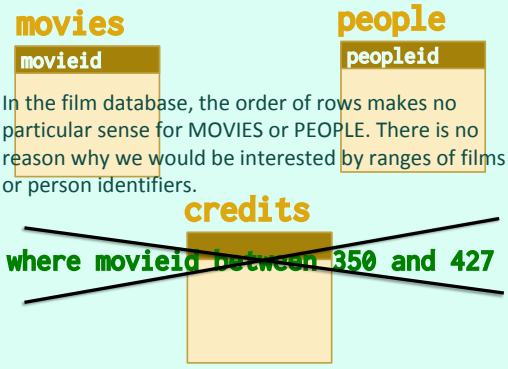
Natural candidates for this type of organization is tables in the "tall and thin" (many rows, few columns) category, not in the "short and plump" one, and for which the key order makes some real-life sense for range scans.

Many rows

Few columns

Tables that implement a many-to-many relationship may fall into this category.

Natural order



In the film database, the order of rows makes no particular sense for MOVIES or PEOPLE. There is no reason why we would be interested by ranges of films or person identifiers.

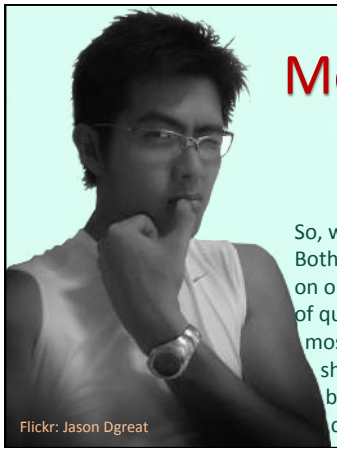
CREDITS is a different case. Its key is the concatenation of MOVIEID, PEOPLEID and CREDITED_AS. If we look for a MOVIEID value, it will be a range scan (of all keys starting with this value). Besides, it's useless to have an index that stores exactly the same information as the table, plus locators.

And here, it makes some real-life sense of finding people involved in a film clustered together.

movieid	peopleid	credited_as
123	178	D
123	312	A
123	86	A
123	105	A
123	237	A

However, it would also make some real-life sense of finding all the films of one person clustered together.

peopleid	movieid	credited_as
97	178	D
97	178	A
97	212	D
97	212	A
97	126	A



Flickr: Jason Dgreat

Movies ?

People ?

So, what should we put first? Both make sense. It depends on our audience, and the type of query that we expect to be most frequent. But in fact we shouldn't worry too much, because we can have our cake and eat it.

Index both!
(constraint + additional index)
Order by the most important for you

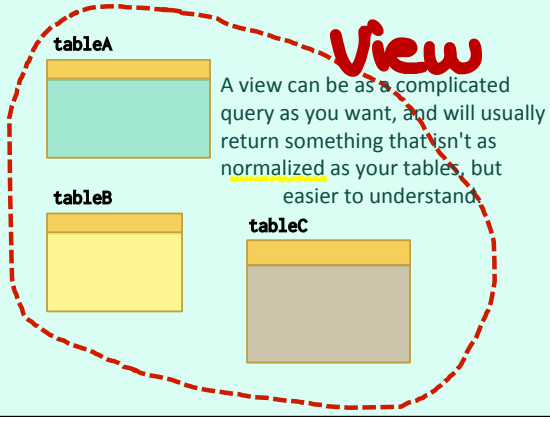
```
main() {
  my_func()
  my_func()
  my_func()
}
```

```
my_func() {
}
```

We have seen stored functions, but these functions are just returning numbers, strings or dates. Can we have relational functions, allowing to reuse relational operations as we would reuse code in a program?


```
create view viewname (col1, ..., coln)
as
select ...
```

In practice (theory is a bit more complicated) there isn't much to a view: it's basically a named query. If the query is correct, it should return a valid relation, so why not consider it as if it were a table?
You can optionally rename columns after the view name (if you don't, the view uses column names from the query result)



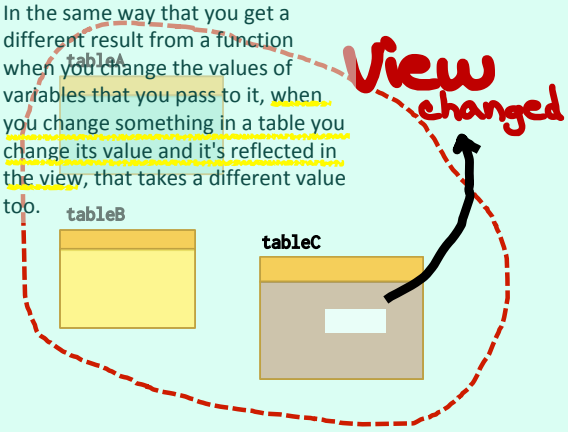
Tables = variables

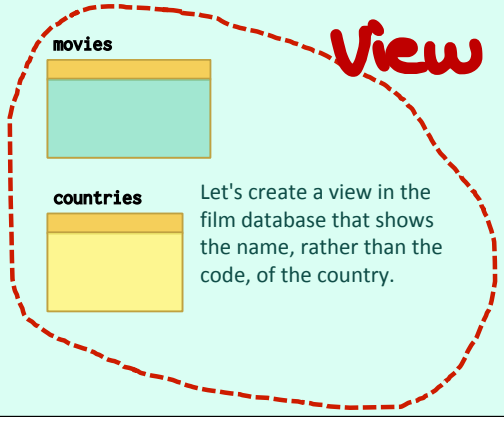
rows

Remember that tables are relational variables, and that the rows they contain are nothing more than their "value"

values

In the same way that you get a different result from a function when you change the values of variables that you pass to it, when you change something in a table you change its value and it's reflected in the view, that takes a different value too.





Let's create a view in the film database that shows the name, rather than the code, of the country.

```
create view vmovies
as select m.movieid,
          m.title,
          m.year_released,
          c.country_name
   from movies m
        inner join countries c
           on c.country_code = m.country
```

```
select *
from vmovies
where country_name = 'Italy'
```

Once the view is created, I can query the view exactly as if it were a table; nothing says that it's a view, except the name that *I* have chosen. I like to give a special name to views to make it clear that it's a view (discussion about practical differences between views and tables comes soon) but I could have masked a change in table design to allow old programs to run by having a changed table T renamed T_V2 and creating a view T rendering the old version.

```
select *
from (select m.movieid,
             m.title,
             m.year_released,
             c.country_name
      from movies m
           inner join countries c
             on c.country_code = m.country)
           vmovies
where country_name = 'Italy'
```

Result-wise, the previous query is strictly equivalent to this one.

Some optimizers are able to push the condition up into the view.

However, there is far more than this to views. As I have said earlier, views are just the relational equivalent of functions: the ability to store (and reuse) a relational expression, in other words something that returns a relation and not simply a value like what you usually do with a stored function.
If we step back to design issues, you remember that modelling a database is basically distributing data between normalized tables, and there are often ways of organizing data that are more suitable for a given application. In some respect, views provide a way of creating an "alternate model".

What is important is that views are permanent objects in the database - needless to say, their content will change with the data in the underlying tables, but the structure will remain constant and can be described in the same way as the structure of a table can be described: columns are typed. Beware that columns are the one in tables when the view was created. Columns added later to tables in the view won't be added even if the view was created with SELECT * (bad practice)

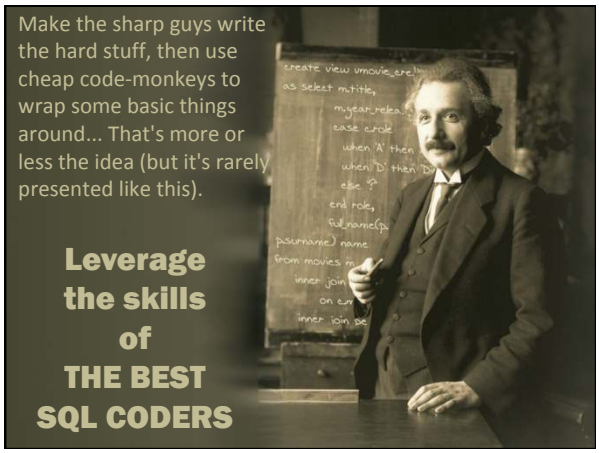
Permanent object

Permanent structure

In real life, views are much used for simplifying queries. Many business reports are based on the same set of hairy joins, with just variations on the columns that you aggregate or order by. Somehow, views allow to come back to that old fancy of the early days of SQL, having something that anybody can query with simple commands, without having to master the intricacies of the querying language.

Simplify queries

Make the sharp guys write the hard stuff, then use cheap code-monkeys to wrap some basic things around... That's more or less the idea (but it's rarely presented like this).




```
select *
from vmovies
where country_name = 'Italy'
```

This is something that a cheap beginner completely ignorant of databases should be able to write after having been briefed for about three minutes.

Looks like a table

Tastes like a table

But ...

It's again an area where theory is bruised by practice.

Let's say that we have this view, which nicely displays film credits, including people names like 'Erich von Stroheim' as they should appear.

```
create view vmovie_credits
as select m.title,
          m.year_released release,
          case c.credited_as
            when 'A' then 'Actor'
            when 'D' then 'Director'
            else '?'
          end duty,
          full_name(p.first_name, p.surname) name
from movies m
    inner join credits c
         on c.movieid = m.movieid
     inner join people p
         on p.peopleid = c.peopleid
```

vmovie_credits

title	release	duty	name
Casablanca	1942	Director	Michael Curtiz
Casablanca	1942	Actor	Humphrey Bogart
Casablanca	1942	Actor	Ingrid Bergman
Casablanca	1942	Actor	Conrad Veidt
Casablanca	1942	Actor	Claude Rains
...	...	...	...

When we query the view, it looks really good and user-friendly. Well, it actually depend on HOW we query it, and on which column. Querying by title will be fine.

ORACLE

And sometimes the optimizer can do very clever things.

```
select country, count(*) as num_films
from movies
group by country
having country = 'ar'
```

I have told you that Oracle (at least the last time I checked) would, with this type of query, process the aggregate, then discard everything that isn't Argentine. It should be a WHERE condition.

ORACLE

If you create this view

```
create view movie_count
select country, count(*) as num_films
from movies
group by country
```

and run this query you might expect the same phenomenon to occur

```
select *
from movie_count
where country = 'ar'
```

ORACLE

In fact, no. The optimizer will see the problem.

```
select country, count(*) as num_films
from movies
where country = 'ar'
group by country
```

It will "push" the condition up into the query and only count Argentine films, running in effect the query above.

```
create view vmovie_credits
as select m.title,
          m.year_released release,
          case c.credited_as
            when 'A' then 'Actor'
            when 'D' then 'Director'
            else '?'
          end duty,
          full_name(p.first_name, p.surname) name
from movies m
    inner join credits c
         on c.movieid = m.movieid
     inner join people p
         on p.peopleid = c.peopleid
```

There are times, though, when all the benevolence of the optimizer cannot do anything for you. You may remember how awful function full_name() is

```
select *
from vmovie_credits
where name = 'Humphrey Bogart'
```

Dreadful expression

If you are writing something like this, what looks like a column (NAME) is in fact the result of a function. There is no way the index on (SURNAME, FIRST_NAME) can be used. We'll have to scan the full table, compute the function, and compare its result to the constant. Unless you do some tricky stuff to index in a way or the other the result of the function (not always possible).

VIEWS

Hide complexity

The problem with views is that as long as you haven't seen how they have been defined, you have no idea how complex they may be. They may be fairly innocuous, or they may be queries of death (they often are)

```
select distinct title
from vmovie_credits
```

Difficulties usually increase sharply as a young developer gets with time more confident, not to say bold, with SQL. Being so accustomed to working with this convenient "table", VMOVIES_CREDITS (it may not bear a name that makes it obvious it's a view), the developer may think of this as a way to return all the different titles in the database. Technically speaking, it will return the desired result, and it may even do it reasonably fast.

```
select distinct title
from
  (select m.title,
          m.year_released release,
          case c.credited_as
            when 'A' then 'Actor'
            when 'D' then 'Director'
            else '?'
          end duty,
          full_name(p.first_name, p.surname) name
   from movies m
     inner join credits c
       on c.movieid = m.movieid
     inner join people p
       on p.peopleid = c.peopleid) vmovie_credits
```

Lot of useless work for what we want

Do we really want the join?

Scalability

And here we are coming to one of the great issues with databases and information systems generally speaking, namely the ability to deliver response times that remain acceptable when the number of users, data volumes, or both, sharply increase.
The computer system of any retailer must survive Black Friday in the US or 11/11 in China.

Computing power is always

LIMITED

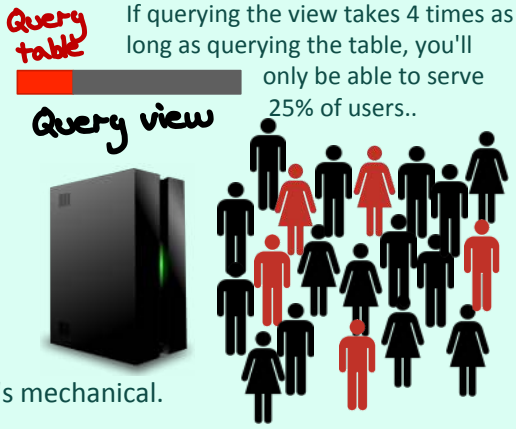
And the problem is that it doesn't matter how big and powerful your computers are, computing power will always be a limited resource.

Slower query to retrieve the same data

=

Fewer simultaneous users served

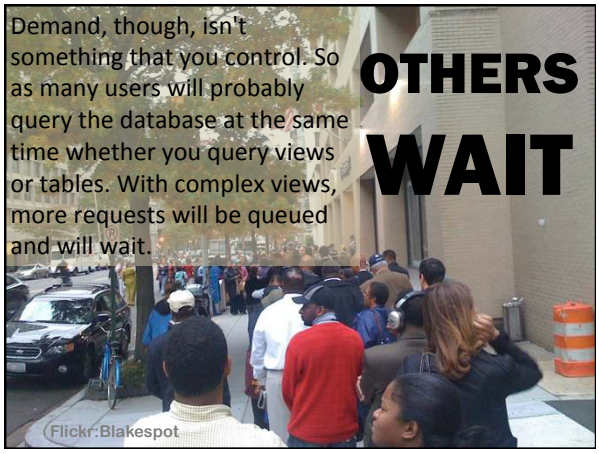
You will only be able to serve *that* many users simultaneously.
You don't want to see everything crawl during peak time.



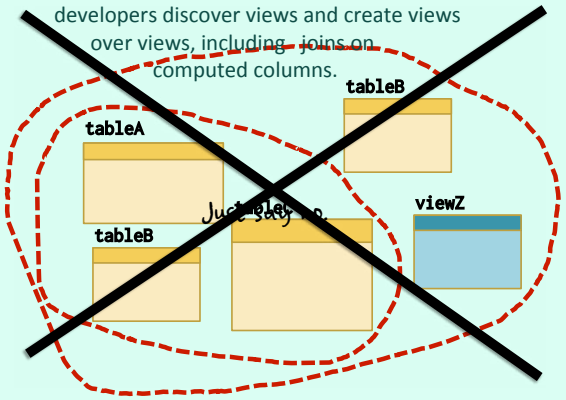
If querying the view takes 4 times as long as querying the table, you'll only be able to serve 25% of users..

It's mechanical.

Demand, though, isn't something that you control. So as many users will probably query the database at the same time whether you query views or tables. With complex views, more requests will be queued and will wait.



Performance issues are usually compounded when developers discover views and create views over views, including joins on computed columns.



Nevertheless, there are three areas where views are very useful. I have mentioned reporting, user interfaces are a bit in the same spirit (more later) the third area is security.

Reports

User Interface

SECURITY

How is security managed?

Before we see how views can help, we need to review how security is managed in a database.

To access a database, you must be authenticated, which often means entering a username and a password.

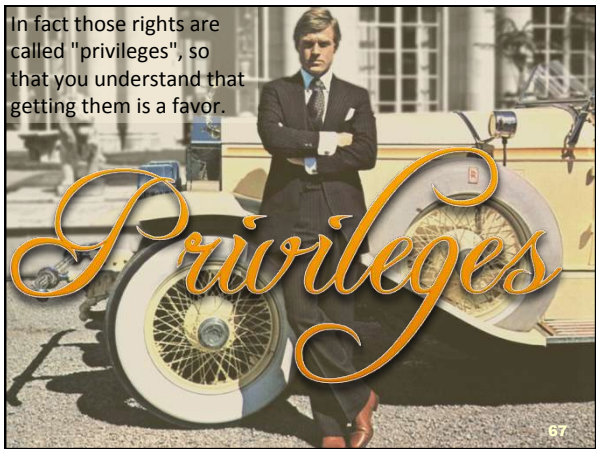
There are other means of authentication, and for some products database authentication is tied to operating system authentication, but in any case the database knows who you are.

So you end up being connected to a database account, and this account as a set of rights.

Database Account

RIGHTS

In fact those rights are called "privileges", so that you understand that getting them is a favor.



A privilege is given to a user account using this command:

```
grant <right> to <account>
```

and can be taken back using this one:

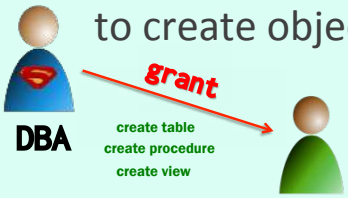
```
revoke <right> from <account>
```

GRANT and REVOKE are the two pillars of what is sometimes called DCL, Data Control Language.

System rights

Rights

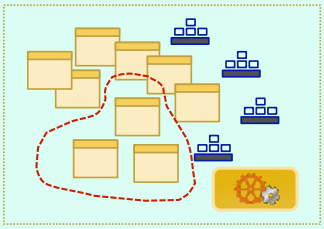
to create objects



Privileges fall into two categories: system rights give users the right to issue DDL commands and change the structure of the database. Not many people get them.

A user who has these (or some of these) privileges will be able to create objects in a SCHEMA, which is usually a set of database objects required by one application.

Schema



The other category of privileges is composed of privileges to access and change the data. Everybody who accesses the database must have some privileges of that category, otherwise there would be no point in accessing the database ...

Table rights

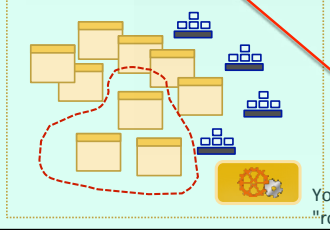
Rights

to access the data

Some people can only access some of the data, some can modify "current" data but not reference tables, some data administrators may have the right to modify any table ... but not necessarily to create even a view!

When accounts have been created by a DBA, the schema owner can grant them specific privileges.

Schema



You can also grant privileges to "roles", and "roles" to users

GRANT commands to give privileges on a table look like this. You can give one or several privileges at once. Sometimes you can give privileges over all the tables in a schema, existing tables and tables still to be created. The UPDATE privilege can also be restricted to some columns only. Some products may require special additional rights (with PostgreSQL "usage" on a schema)

```
grant select, insert on tablename
to accountname
```

And for users who have been naughty:

```
revoke privilege on tablename
from accountname
```

So ...

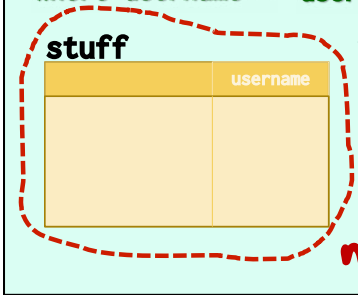
How can views help with security?

The trick is to use a view that only shows what people are supposed to see, and grant SELECT on the view and not on the table..

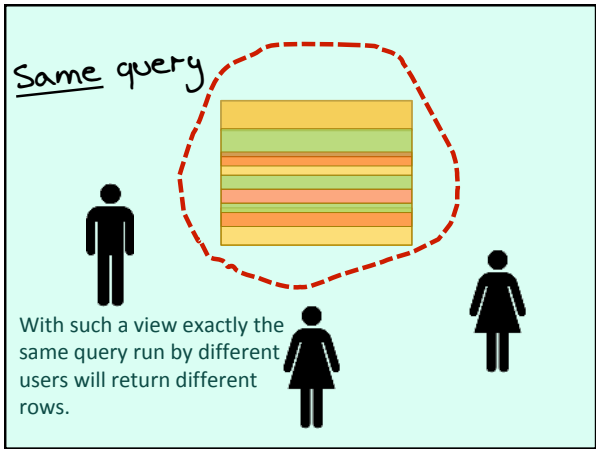
You can hide sensitive columns

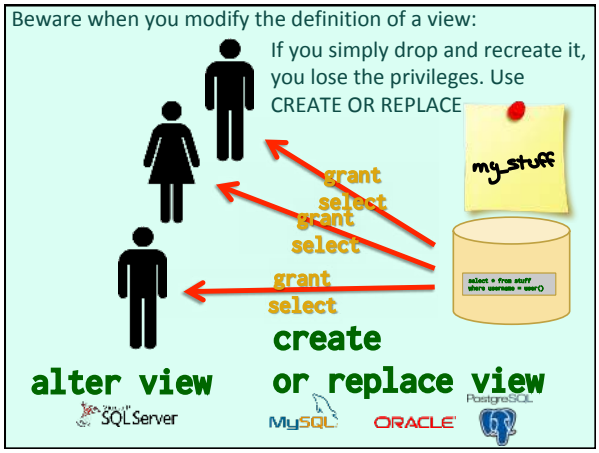
```
create view my_stuff
as
select * from stuff
where username = user()
```

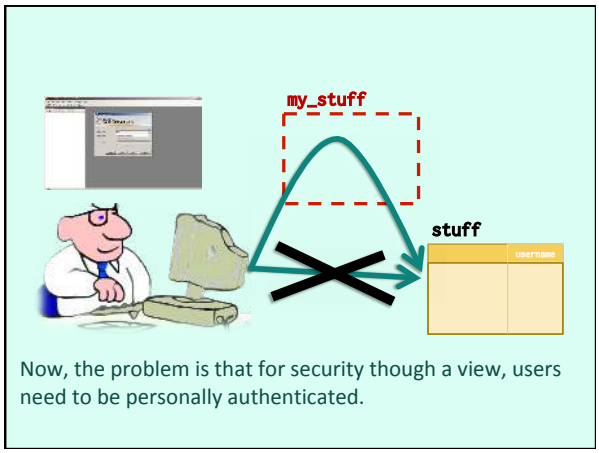
Syntax for identifying the current user varies.

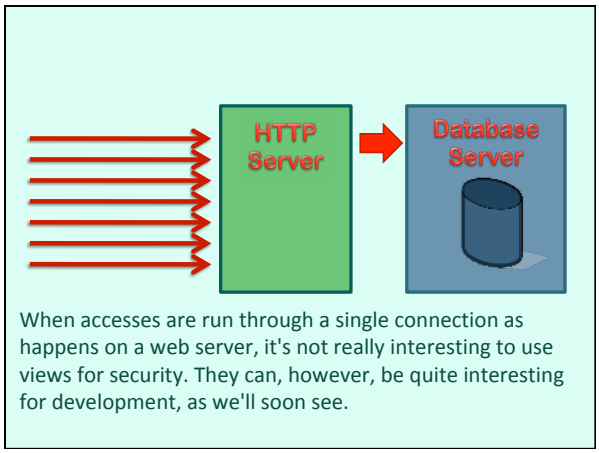


You can even hide rows by only returning rows "owned" by the user currently connected.


Same query
With such a view exactly the same query run by different users will return different rows.


Beware when you modify the definition of a view:
If you simply drop and recreate it, you lose the privileges. Use CREATE OR REPLACE
my_stuff
grant select
grant select
grant select
alter view
create or replace view
SQLServer
MySQL
ORACLE
PostgreSQL


my_stuff
stuff
username
Now, the problem is that for security though a view, users need to be personally authenticated.


HTTP Server
Database Server
When accesses are run through a single connection as happens on a web server, it's not really interesting to use views for security. They can, however, be quite interesting for development, as we'll soon see.

What about CHANGING DATA through views?

If views are in theory like tables, why not using them for controlling not only what you SEE, but what you CHANGE?

Lots of things can go wrong

It all depends on the view ... The problem is that most view are designed to provide a more user-friendly view of data: joins transforming codes into more legible values, functions making data prettier (date formatting, for instance). And by doing so you often lose information.

For instance if your view concatenates first_name and surname, splitting a single string in two parts is tough if you want to insert through the view.

```
case
  when p.first_name is null then p.surname
  else p.first_name || ' ' || p.surname
end name
```

Tommy Lee Jones

Benicio Del Toro

Everybody isn't called 'Gary Cooper'.

And for updates ... Let's have a view that displays the country name rather than code.

```
create view vmovies
as select m.movieid,
          m.title,
          m.year_released,
          c.country_name
   from movies m
        inner join countries c
          on c.country_code = m.country
```

from movies

from countries

movieid	title	year_released	country_name
2	Blade Runner	1982	United States
6	Das Boot	1985	Germany
9	Goodfelas	1990	United States
15	Le cinquième élément	1997	France
22	The Lord of the Rings	2001	New Zealand
27	We Feed the World	2005	Australia
25	Ying hung boon sik	1986	Hong Kong

Never understood why people were confusing both.

Wrong! AUSTRIA, not Australia!

CORRECTION

SQL Server would let you update ... and try to change the name in table COUNTRIES.

```
create view vmovies
as select m.movieid,
          m.title,
          m.year_released,
          c.country_name
   from movies m
        inner join countries c
          on c.country_code = m.country
```

NOT this

Most products will express concern and prevent you from doing it.

THIS

Abandon all hope, ye who enter here

In many cases, view update is simply impossible.

- **Most joins**
- **Aggregates**
- **Expressions**
- **Omitted**
  - **mandatory columns**
  - **(insert)**

Sometimes it works very well

In some cases, view update is quite possible.

This will work fine with Oracle, which would have complained with a join

One table

```
create or replace view vmy_movies
as select m.movieid,
          m.title,
          m.year_released,
          m.country
   from movies m
   where m.country in
     (select c.country_code
      from countries c
           inner join user_scope u
              on u.continent = c.continent
      where u.username = user)
```

Everything else in a subquery

USER_SCOPE

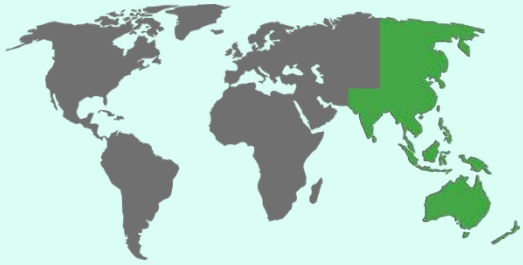
Username	Continent
HUIZHONG	ASIA
PAVEL	EUROPE
IBRAHIM	AFRICA
AMINATA	AFRICA
MICHAEL	EUROPE
JUAN_CARLOS	AMERICA
SANDEEP	ASIA
PATRICIA	AMERICA
PATRICIA	EUROPE

Which proves that in some cases join and subquery aren't exactly equivalent ...

There is no problem because the view update maps to a simple table update.

Plain insert/update/delete of movies

Now, there may STILL be a problem.



Suppose that you are in charge of Asia/Pacific, and only see films from this region.

Consistency Issue

```
select * from vmy_movies;
```

movieid	title	year_released	country
19	Pather Panchali	1955	in
20	Shichinin no Samurai	1954	jp
21	Sholay	1975	in
22	The Lord of the Rings	2001	nz
25	Ying hung boon sik	1986	hk
26	We Feed the World	2005	au

Ooops

Only from Asia/Oceania

If you change the country from Australia to Austria (in Europe), poof! you no longer see it.

```
update vmy_movies
set country = 'at'
where movieid = 26
```

Nothing prevents from

```
insert into vmy_movies(title, year_released, country)
values ('Snow White and the Seven Dwarfs', 1937, 'us')
```

UNLESS

There is one special constraint, though, that exists for views: **WITH CHECK OPTION**.

```
create or replace view vmy_movies
as select m.movieid,
          m.title,
          m.year_released,
          m.country
   from movies m
   where m.country in
     (select c.country_code
      from countries c
           inner join user_scope u
              on u.continent = c.continent
      where u.username = user)
with check option
```

It prevents you from making a change that will make a row disappear from the view (other than a DELETE)

CHECK OPTION would let you update from Australia to any Asian country

au 

cn 

in 

jp 

id 

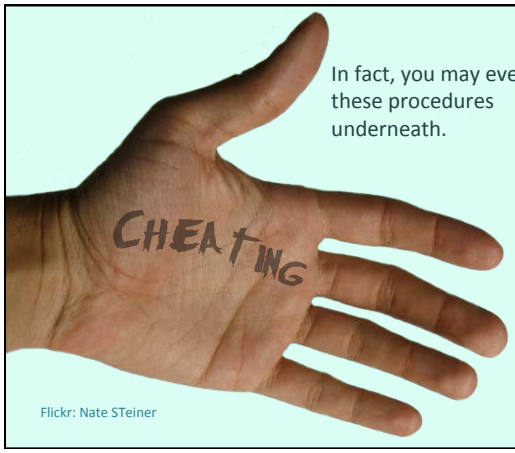
But not to a country from another region

at 

Solution in some cases:

- insert procedure
- update procedure
- delete procedure

If updating the view directly is impossible, in many cases (remember when we were displaying the country name) what should be applied to base tables is fairly obvious and can be performed by dedicated stored procedures.



In fact, you may even call these procedures underneath.

Flickr: Nate STeiner

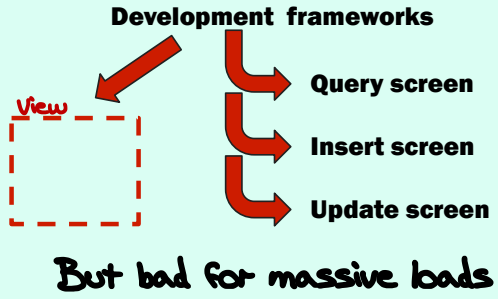
There is a special type of trigger called an

instead of trigger

It can be created on a view and lets you call a procedure "instead of" performing the triggering event

- insert procedure
- update procedure
- delete procedure

This can be quite useful with web development frameworks. Map your framework objects to views with "instead of" triggers, and let the framework generate the maintenance screens for you.



But bad for massive loads

Sometimes another option than views

Function returning a table

They aren't supported by all DBMS products.

Views are cleaner ...

The advantage of a function returning a table is that it allows you to inject a parameter deep inside a query: with a view, itss just like with a table, you can only apply conditions to the data it returns. No such restriction with a function returning a table, but they are more in the "useful dirty trick" category, and once again every DBMS isn't supporting them, or supporting them in the same way (some products may return "streamed" data,returning rows as they are retrieved, some products may buffer them, retrieving data in memory before returning it). Dangerous if you ever migrate to another DBMS.

One very good example of view application is the set of tables that contain information about the objects in the database, collectively known as the

Data Dictionary

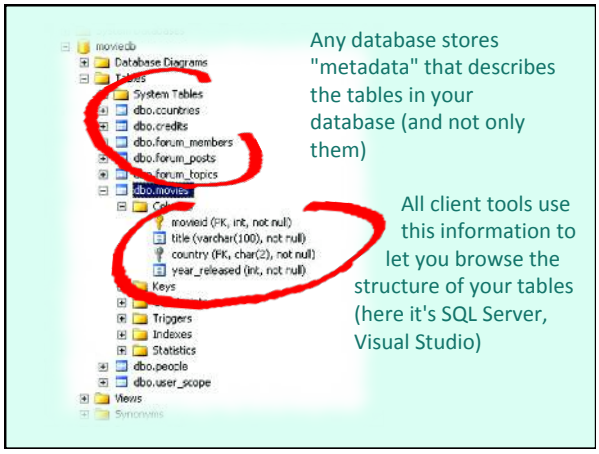
or sometime called the

Catalog

They are using all the features we have seen (you only have privileges to read views and only see what is relevant to your account)

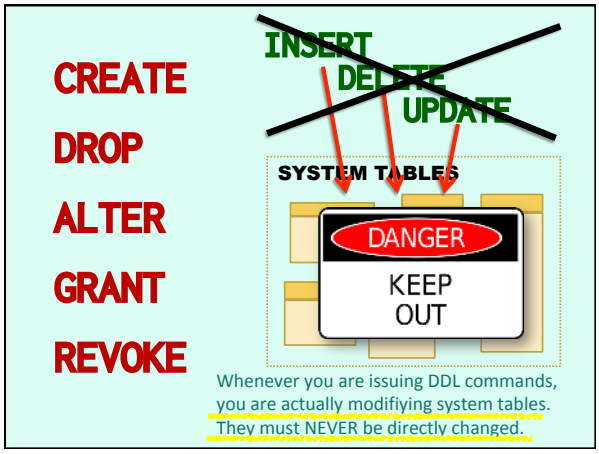
One catalog per database

You always have ONE catalog per database. A database is an independent unit and you can have foreign keys only within (inside) one database; however, you can have several schemas in a database, and you can reference tables in another schema. There may also be metadata such as user accounts that is shared among databases. Most DBMS products can manage several databases at once; other than SQLite, the exception is MySQL that only has ONE catalog. What MySQL calls a *database* is actually a schema.

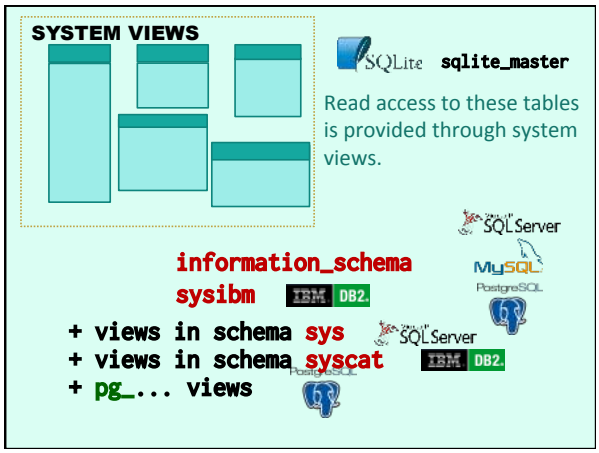


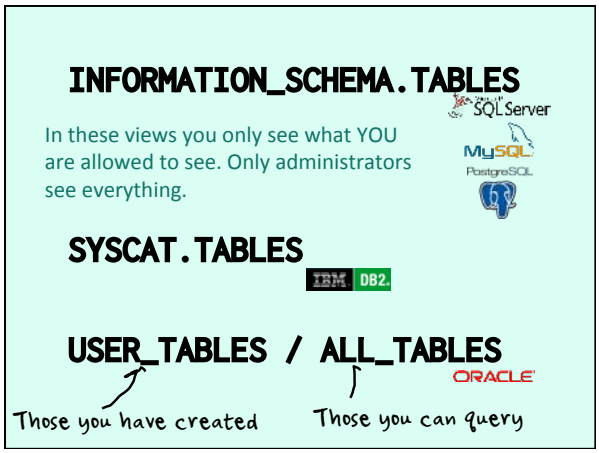
Any database stores "metadata" that describes the tables in your database (and not only them)

All client tools use this information to let you browse the structure of your tables (here it's SQL Server, Visual Studio)



Whenever you are issuing DDL commands, you are actually modifying system tables. They must NEVER be directly changed.





standard

INFORMATION_SCHEMA

= minimum ...

The "SQL standard" defines a schema for the catalog that several DBMS vendors try to implement (Oracle, so far, doesn't follow it, perhaps because Oracle has no schemas independent from user accounts, and DB2 doesn't call it INFORMATION_SCHEMA). However, you only find minimum information in INFORMATION_SCHEMA. Some products have views to describe triggers, others haven't, for instance. Other than a small common set, many columns may alos be different simply because implementations are different.

```
moviedb=> select table_name,
moviedb->        column_name,
moviedb->        ordinal_position,
moviedb->        data_type
moviedb-> from information_schema.columns
moviedb-> where table_name = 'movies';

 table_name |  column_name  | ordinal_position |     data_type
------------+---------------+------------------+-------------------
 movies     | year_released |                4 | integer
 movies     | country       |                3 | character
 movies     | title         |                2 | character varying
 movies     | movieid       |                1 | integer
(4 rows)
```

There are usually simpler commands to display the structure of a table, but these commands execute nothing more than this type of query. Everything is pulled out of the data dictionary. This MySQL query gives the same result on PostgreSQL (very rare!).

INFORMATION

As a developer, you can get from the data dictionary some information that is hard to get elsewhere (constraints, for instance).
Database administrators use them a lot for scripting, because the data dictionary always reflects the current state of a database.

SCRIPTING (DBA )

```
select 'drop table ' ||
       table_name || ';'
from information_schema.tables
where table_name like 'TMP%'
  and ...
```

DBAs often use queries on the catalog to generate other SQL queries; it can be done in a script, or in a procedure with a cursor (a case when cursors are mandatory). They sometimes generate other commands, such as shell script, for instance for backing up database files (the name of which can be found in some remote corners of the catalog).