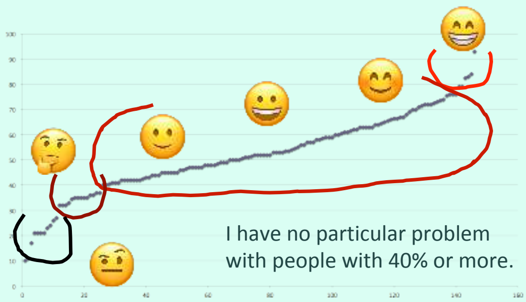
CS307

Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

MidTerm Exam



60%

is a GOOD grade

Final grade will be adjusted. There is no set threshold, if by and large you seem to have got a decent understanding of the subject you'll pass.

For naming conventions, some people advocate prefixing an index name woth something special.

IDX_MYTABLE_SOMECOL

If you have the choice, a suffix is probably a better option (with triggers too). It's possible to list all objects in a database, and, by sorting by name, suffixes allow to see all related objects grouped together.

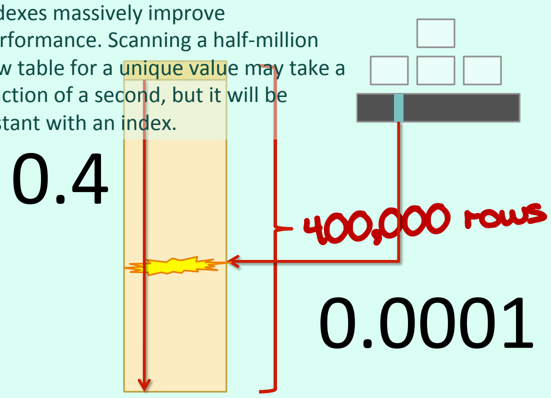
MYTABLE_SOMECOL_IDX

In any case, follow standards that are in place.

FOLLOW NAMING STANDARDS

What matters isn't so much the rule than the fact that everybody is following it and that a name tells you immediately what an object is.

Indexes massively improve performance. Scanning a half-million row table for a unique value may take a fraction of a second, but it will be instant with an index.

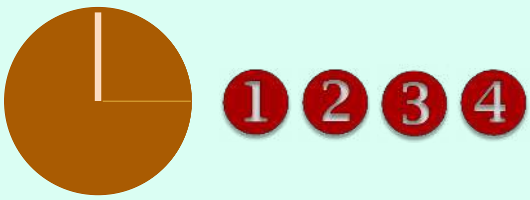


An end-user may not notice much of a difference, because a sub-second response time is quite acceptable. It's when the same action is repeated a large number of times that it makes a difference.

x 100

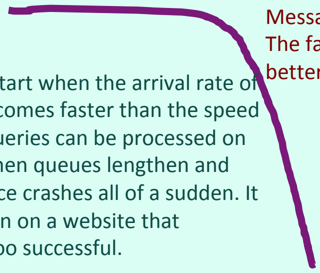
0.01 40

We are here right in the topic of scalability. Most computer systems are queueing systems. With few users, a mediocre response time is usually OK.



With a lot of queries, later queries will have to wait for earlier queries to be processed before they can be handled. For end-users, the wait time is part of the response time.

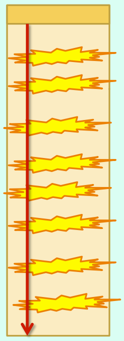
Performance



Message?
The faster the better.

Problems start when the arrival rate of queries becomes faster than the speed at which queries can be processed on average. Then queues lengthen and performance crashes all of a sudden. It may happen on a website that becomes too successful.

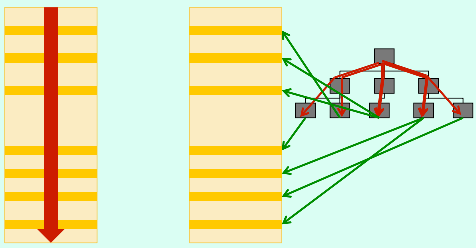
0.4



Does the super-performance of indexes mean that we should NEVER scan a table?
That would be too easy!
When we scan a table, the time to scan it is irrespective of how many rows we'll find to return: 1, 100, or several tens of thousands. An index search gives addresses of individual rows.

If we need to retrieve MANY rows, there will be a time when plainly scanning the table will be faster.

where column_name in (select …)



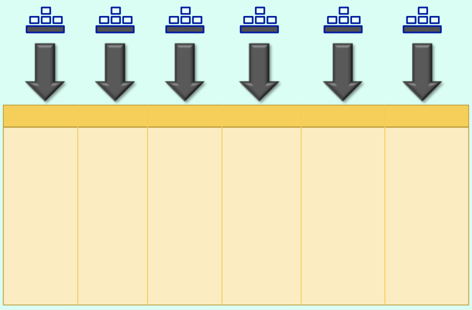
It arrives earlier than you might think.

What you are using and how you are doing it actually depends on how much data you are retrieving. What is known as "OLTP" (OnLine Transaction Processing) usually makes heavy use of indexes. By contrast, massive batch processes scan a lot, and it's how it should be.

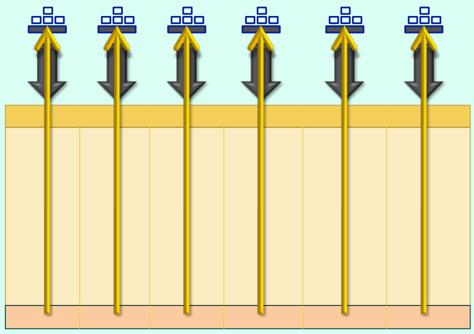
Right algorithm

⇕

Right volume



You sometimes find on forums people who are well-wishing but not very experienced who advise people who know hardly less than them to index all of their columns. Don't.

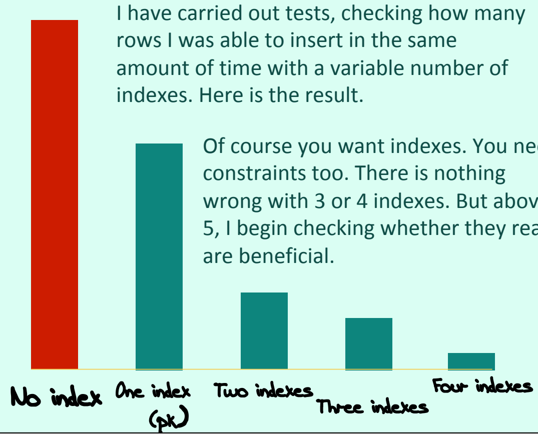


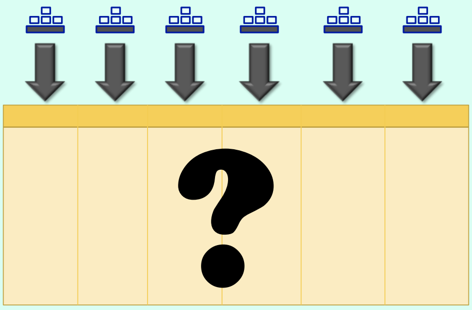
Indexing every column means, once again, that every INSERT will have to write not only into the table but into every index. Ouch.

You always have to compromise. Even if an index makes a search significantly faster,you have to put it in balance with the negative impact on inserts and deletes.

I have carried out tests, checking how many rows I was able to insert in the same amount of time with a variable number of indexes. Here is the result.

Of course you want indexes. You need constraints too. There is nothing wrong with 3 or 4 indexes. But above 5, I begin checking whether they really are beneficial.





But if we don't want to index ALL columns, the big question becomes: which ones?

search criteria often used

There is little need to index a column that you don't use as a search criterion: indexes are primarily here to help you find values faster. In the same way, it doesn't make much sense to index a column especially for the yearly report if it must penalize your inserts all year long.

Typically, in table containing exchange rates, you would index by currency code and date, because one as little meaning without the other. Some people might want to add the exchange rate to the index to find it there without needing to access the table (we may talk more about storage tricks later if we have time)

```
where column_name = ...
```

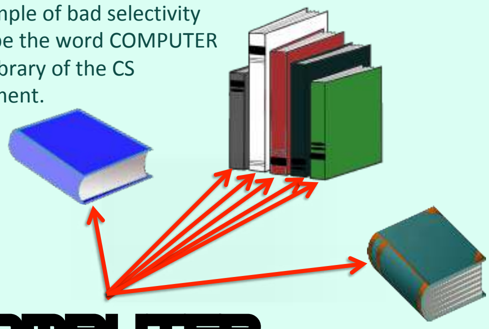
currency_code	as_of_date	exchange_rate

Another very important factor is that the column must be SELECTIVE, that means that the values it contains are rare and correspond to very few rows. Unique columns and PK are extremely selective by definition. Other columns are a mixed bag.

selective

rare

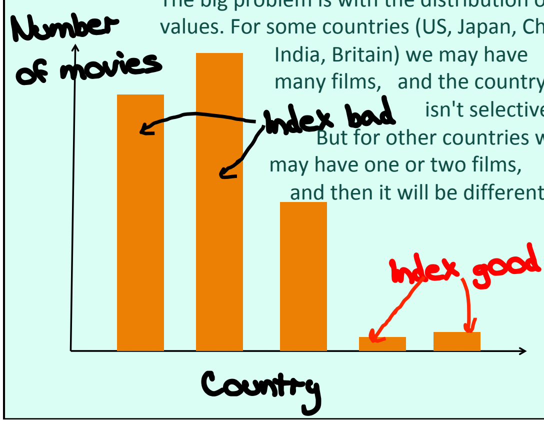
An example of bad selectivity would be the word COMPUTER in the library of the CS department.



COMPUTER

It's probably in every book.

The big problem is with the distribution of values. For some countries (US, Japan, China, India, Britain) we may have many films, and the country isn't selective. But for other countries we may have one or two films, and then it will be different.



If there is an index on a column such as COUNTRY, will the DBMS use it? Not necessarily. The optimizer may decide not to use an index. Is there a way to know if it will? Good news: yes.

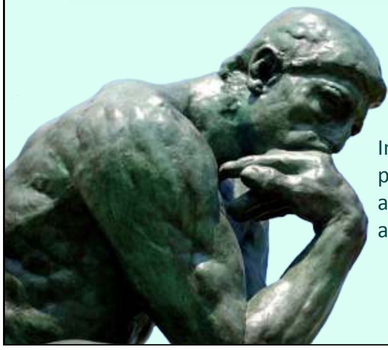
What is the DBMS actually DOING?

All DBMS products implement (with slight syntax differences, it won't surprise you) a way to display what is known as the "execution plan", what the optimizer would choose to do to run a query.

```
explain <select statement>
```

SQL Server `set showplan_all on`

execution plan

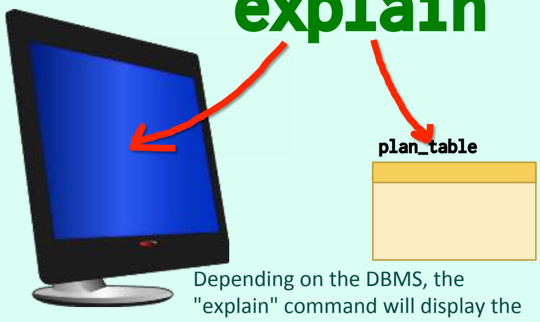


Tables
Indexes

In this execution plan, you see tables and indexes that are accessed.

Flickr: Brian Hillegas

explain



Depending on the DBMS, the "explain" command will display the plan directly or populate a table that must be queried.

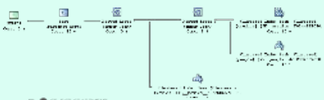
Many tools can also display graphically an execution plan on demand.

Graphical Environments

- SQL Server Management Studio
- SQL Developer
- IBM Data Studio

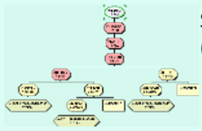
etc.

Microsoft SQL Server



ORACLE

IBM DB2



Some examples (same query)

It's worth noting that the same query on the same tables with the same indexes and the same data in the tables may very well result in different execution plans with different products. Internal algorithms are different, internal data storage is different, some products may be better than others at processing data in a particular way, and of course optimizers are different and may choose different courses.

Beginners often assume that there are "good" execution plans (that only use indexes) and "bad" ones (which scan tables). In fact, it's impossible in most cases to say, by reading two different plans, which one is fastest.

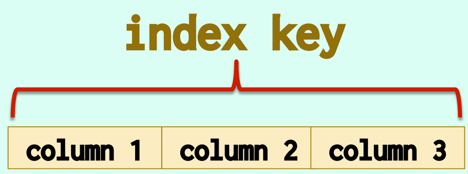
execution plans

The main benefit of execution plans it to check whether the optimizer is more or less doing what you thought it would do.

explain

In particular whether an index is used. The optimizer may choose not to use it. It may also be unable to use it.

Index used?



The impossibility of using an index (or of using it as intended) may come from several reasons. One is with composite indexes, the key of which is concatenated values from several columns. The index can only be used if the lead column(s) appear in the query.

The problem is the same as looking up for someone's number in a phone book when you don't know the surname. If you had the surname and address but not the first name, you could do it.

[redacted]	Tom	Main St 123

Flickr: Larry Page

PEOPLE

SURNAME	FIRST_NAME

```
explain select * from people
 where surname = '...'
   and first_name = '...'
```

index can be used

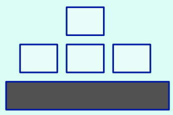
```
 where first_name = '...'
```

index cannot be used

```
like '%something%'
```

Exactly the same problem happens when you are using a LIKE expression that STARTS with a %, or the SUBSTR() function. Without the leading part, no way you can walk the tree that takes you to row addresses.

function()



Actually, the problem is more general than that. Suppose that you have a tree built upon the values that you can find in column C. If you say that f(C) is equal to something, there is no way to find the corresponding value for C in the tree.

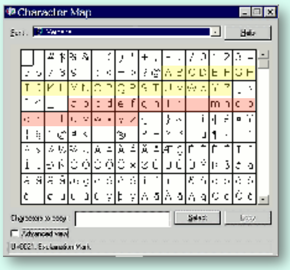
In a way that is very similar to a search in a phonebok when you don't know the surname, it's like searching a dictionary for words that present some special property, such as containing some special letters or being also readable backwards (palindromes). Mission impossible.

• Flickr: TEDxNJLibraries

Case-sensitive DBMS:

```
where upper(surname) = upper('some input')
```

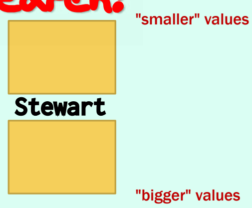
A typical, and common, problem is to perform, with a case-sensitive DBMS a case insentive search in a column when data is in mixed case.
A tree is based on the ordering of values. It's how values compare to the values in the node that you are visiting which tells you which subtree you should visit next.



The problem is that the internal codes of letters consider that "a" is greater than "Z" (and don't talk about accented letters)

```
where upper(surname) = 'MARVIN'
```

Where to search?



If you imagine that you have a binary tree, find 'Stewart' at the root, and look for something equal to MARVIN when set to uppercase without knowing in which case that something is written, you are toast.

```
where upper(surname) = 'MARVIN'
```

Where to search?

"smaller" values

MILES
O'brien
Stewart
marvin
wayne

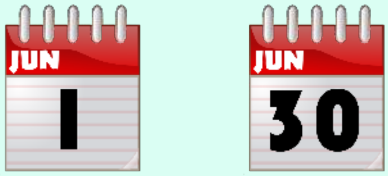
"bigger" values

It could actually be anywhere in the tree.

```
where upper(surname) = 'MARVIN'
```

Same story with functions that extract date parts. Use them in a WHERE clause, your index is dead. You should always express conditions on dates as range conditions.

```
where extract(month from date_column) = 6
  and extract(year from date_column) =
           extract(year from current_date)
```



Same story again with implicit conversions. If they are performed in the wrong direction, wave farewell to the index.

```
and varchar_code = 12345678
```

```
select cast('12345678' as varbinary(max)) as string_12345678,
       cast(12345678 as varbinary(max)) as number_12345678;
```

string_12345678	number_12345678
0x3132333435363738	0x00BC614E

Ooops ...

Numeric	String	Date
4	'14'	'25-JUL-1603'
14	'25'	'4-JUL-1776'
25	'4'	'14-JUL-1789'

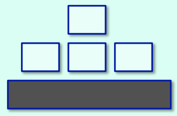
The reason is simple. Index search is based on ordering and ordering is different with different datatypes. Convert datatypes, you break the ordering.

Some of these issues can be taken care of by only storing appropriate data in your tables, for instance not storing mixed case.

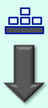
```
insert into table_name(column_name)
values(upper(<input>))
```

In some other cases, though, the search MAY require applying a function to the indexed column.

```
select *
from people
where soundex(surname) = soundex('Stuart')
```



Is there a way to apply a function to a column and yet benefit of the quick access provided by an index?
Yes, by cheating.



people

peopleid	first_name	surname	born	died	surname_soundex
1	James	Stewart	1908	1997	S363
2	Humphrey	Bogart	1899	1957	B263

What you can do for instance is add a column that stores the soundex. This happily violates the rules of good normalization (... it depends on another column that is a part of a key), but then you can index it and apply the search to this new redundant column.

people

peopleid	first_name	surname	born	died	surname_soundex
1	James	Stewart	1908	1997	S363
2	Humphrey	Bogart	1899	1957	B263

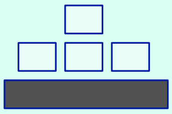
If you have full control of the program that inserts data into the table you can modify it so as to insert soundex(SURNAME) at the same time as you insert SURNAME. If not, there is (at the cost of far slower uploads) the option of a before insert/for each row trigger that does it on the fly.

S363

```
select *
from people
where surname_soundex = soundex('Stuart')
```

Having the column and having indexed it, you no longer need to apply any function to the searched column, and everything is fine.

Most products actually allow you to do something cleaner by indexing an expression (sometimes called "generated" or "virtual" column), which can be the result of a function.



It only works if the expression or function is

Deterministic

$f()$

transformation

Same input

function

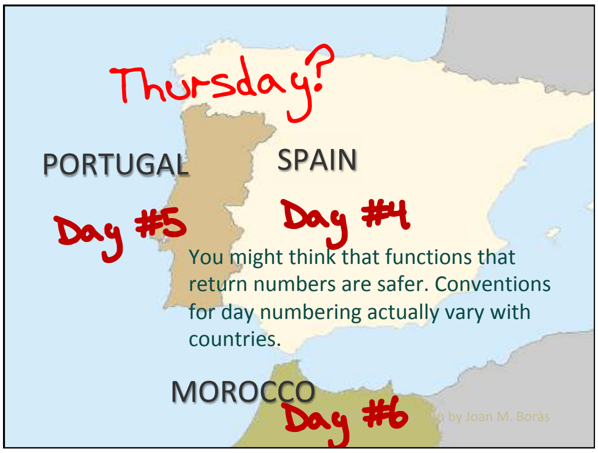
Same output

ALWAYS !

"Deterministic" means that the same input will always generate the same output. Many commonly used functions aren't deterministic because their result is affected by database settings (localization settings most often).

```
datename(month, '1970-01-01')
```

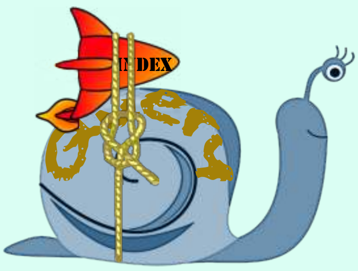
This SQL Server function isn't deterministic, because if the DBA changes the language by default of the database, what will be returned will be different. You can imagine the scenario: language is English, you create an index, plenty of "January" stored in the index, the DBA switches the language to Spanish and you search the index for "Enero" ... SQL Server prevents you from indexing this function.



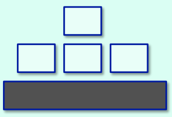
You might think that functions that return numbers are safer. Conventions for day numbering actually vary with countries.

upper(column_name)

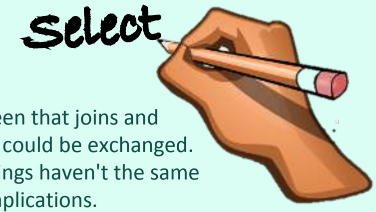
Fortunately, there are still many functions that truly are deterministic!
If your DBMS allows it, you can index UPPER(...) without any problem. Same story with SOUNDEX(...). When your statement will be analyzed the SQL engine will be able to notice your using the function and will use the index.



In any case, 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

```
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')
```

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

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

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'
```

second

```
select *
from people
where surname = 'Spielberg'
```

first

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

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

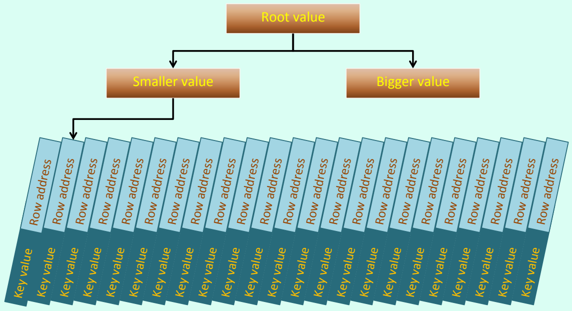
A table that is ordered is called a

Microsoft SQL Server **Clustered index**

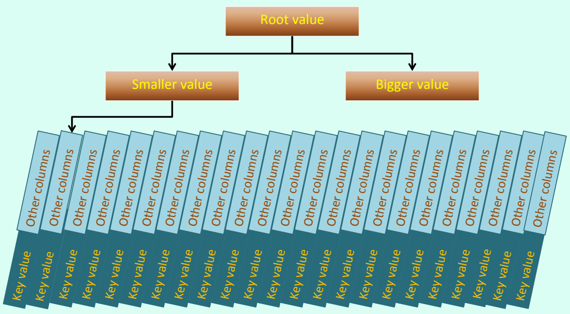
ORACLE **Index-organized table**

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.

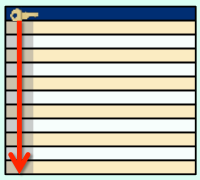


In an ordered table, row data replaces the row address.



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

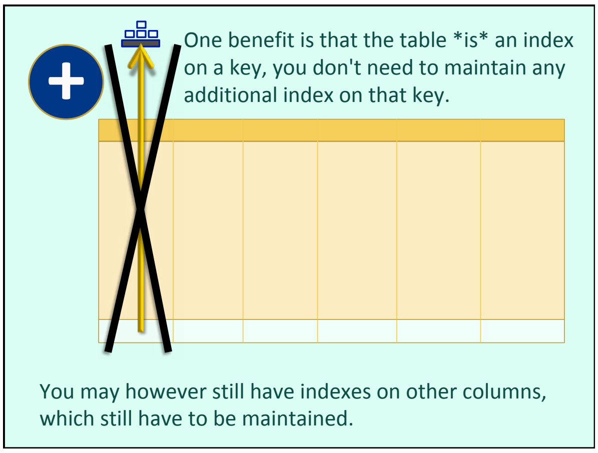
That way you have all rows ordered according to the key value.

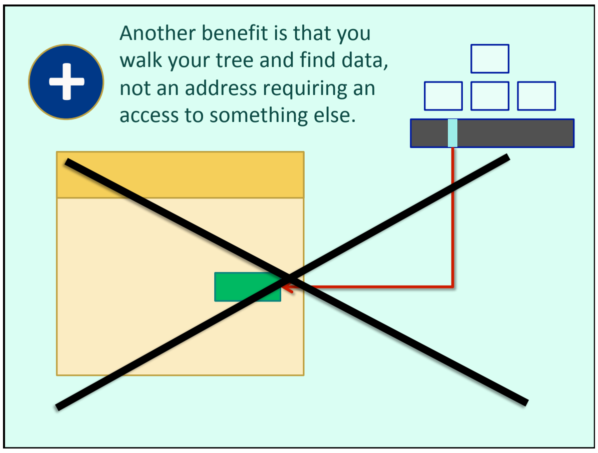


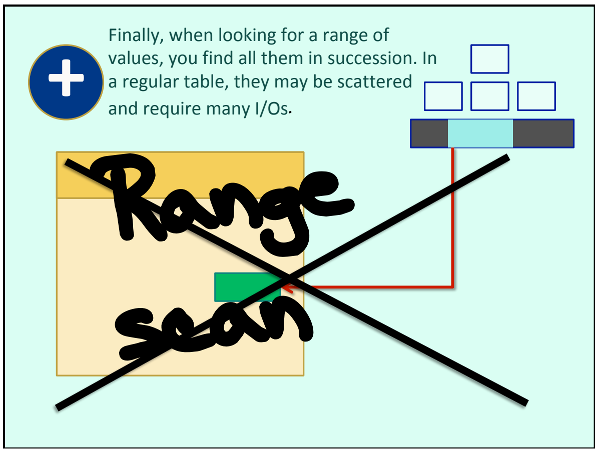
Benefits

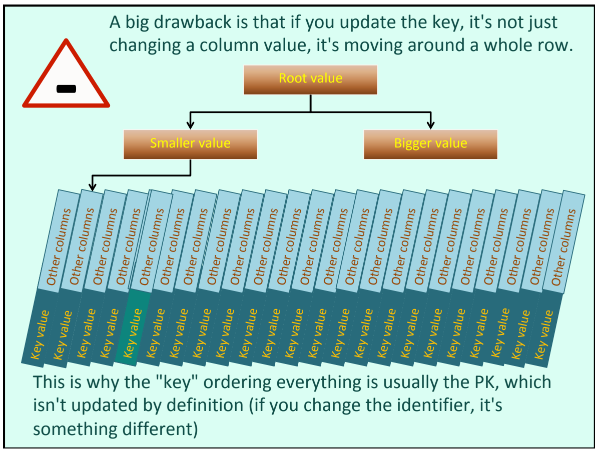
Drawbacks

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 ofinsertion is more or less random.

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

---

movies

movieid

people

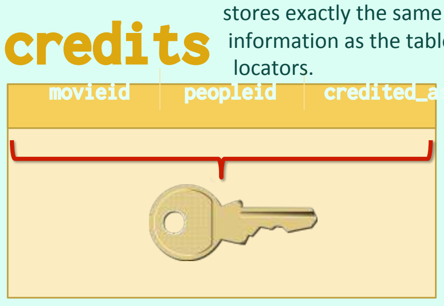
peopleid

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

~~where movieid between 350 and 427~~

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.

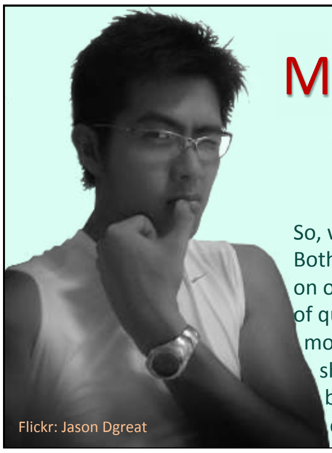


And here, it makes some real-life sense of finding people involved in a film clustered together.

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()
}
```

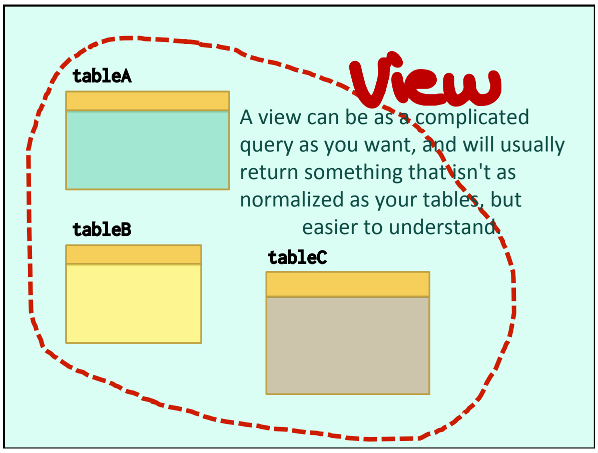
```
my_func() {

}
```

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 ...

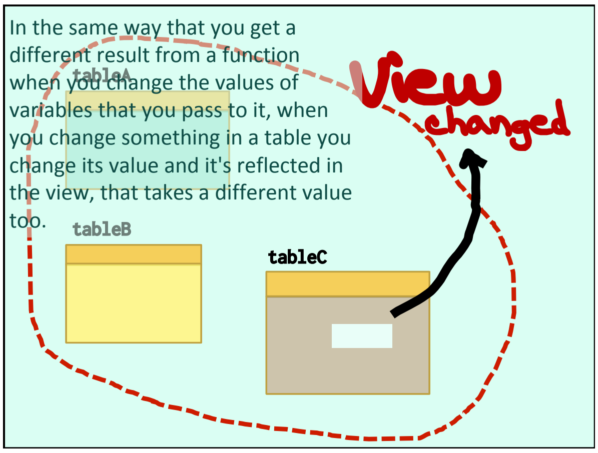
In practice (theory is a bit more complicated) there isn't much to a view: it's basically a named query. If the query is correct, it should return a valid relation, so why not consider it as if it were a table?
You can optionally rename columns after the view name (if you don't, the view uses column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

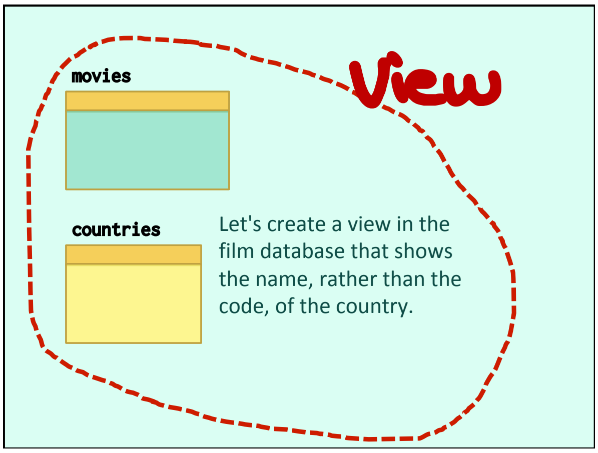


Tables = variables

rows → values

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






```
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.

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

```
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'
```

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

However, there is far more than this to views. As said in the preceding lectures, 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".

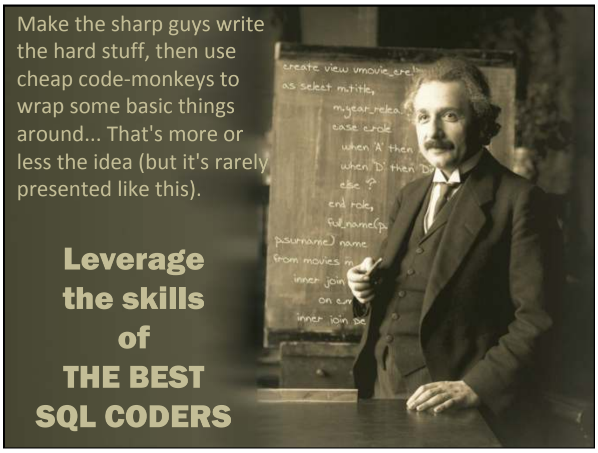
What is important is that views are permanent objects in the database - needless to say, their content will change with the data in the underlying tables, but the structure will remain constant and can be described in the same way as the structure of a table can be described: columns are typed. Beware 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

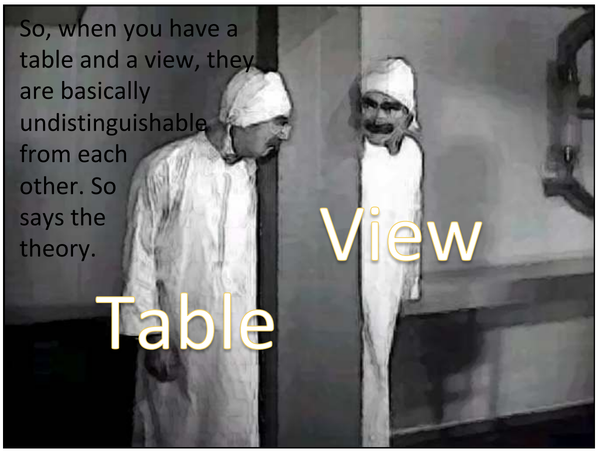
In real life, views are much used for simplifying queries. Many business reports are based on the same set of hairy joins, with just variations on the columns that you aggregate or order by. Somehow, views allow to come back to that old fancy of the early days of SQL, having something that anybody can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

Simplify queries




```
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

=

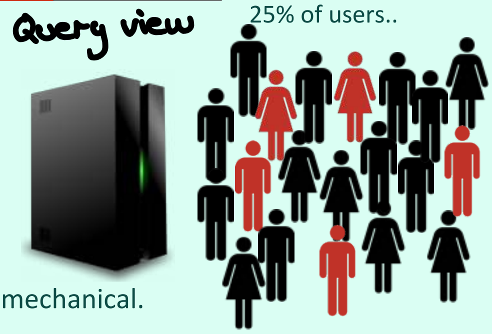
Fewer simultaneous users served

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

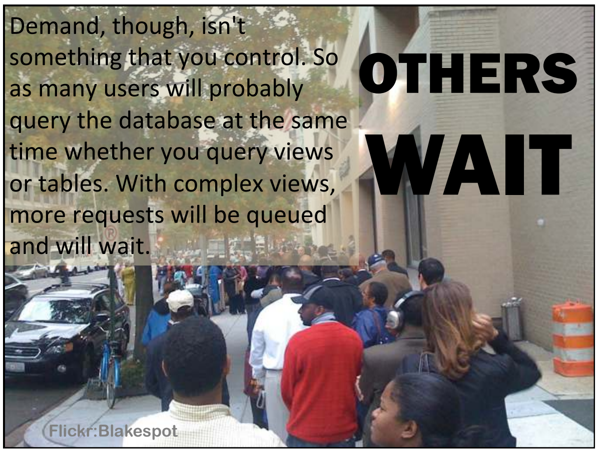
Query table

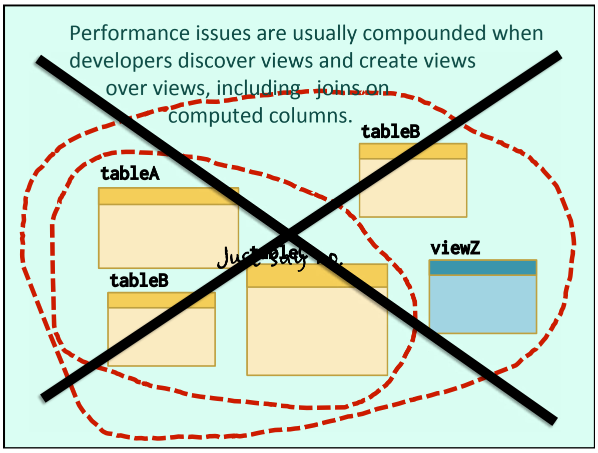
If querying the view takes 4 times as long as querying the table, you'll only be able to serve 25% of users..

Query view



It's mechanical.





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., which we'll discuss next time.

Reports

User Interface

SECURITY