CS307

Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

Practical problem with disaster recovery

Disaster recovery, by definition, occurs in the middle of a disaster, which means complete panic. Database administrators and system managers can expect to have to switch systems in the middle of phone calls, and, if they don't answer, nervous managers around them asking when the system will be up. Try to imagine what it can be when computer systems crash at a busy airport. Recovery procedures must be as automated (and tested) as possible, because it's very hard to think straight in these cases.

Experience proves though that when the time comes of switching to a remote backup system, the most consuming may not be starting everything "for real" at the backup site and routing all applications there.
Switching a system is a big decision, and between the time when a problem is detected, its severity assessed and the "go" from management, quite a lot of time can fly away.

Switch

Diagnosis?

Decision taking?

One point often overseen is that the goal of a backup site isn't to replace the primary site forever. When the primary site is finally back on its feet, you must restore recent copies of databases, reset log shipping but the other way round, and finally, when ready, switch again (this time it can be scheduled at a low activity time), and reset the backup site to a backup role. All operations that aren't necessarily obvious.

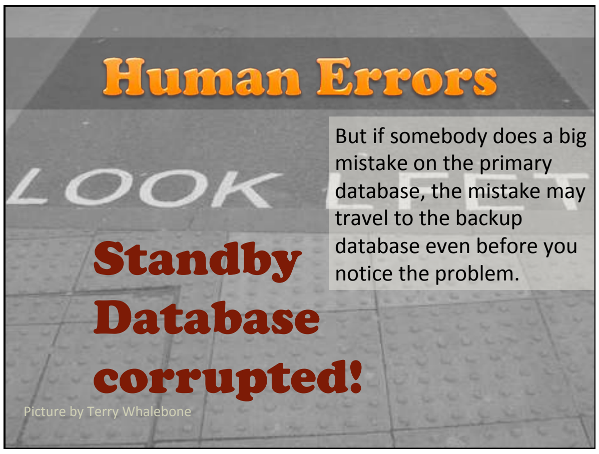
Think about return to normal ...

Hardware Failure

When you want protection against hardware failures, you want the backup site to run as close as possible behind the primary site.

Try to minimize lag

Picture by Philip Bitnar



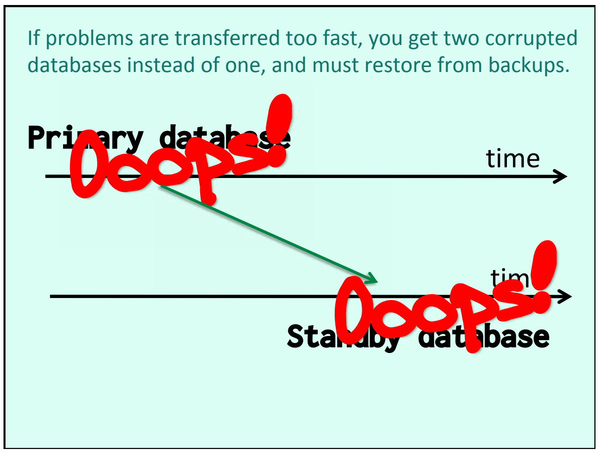
Human Errors

But if somebody does a big mistake on the primary database, the mistake may travel to the backup database even before you notice the problem.

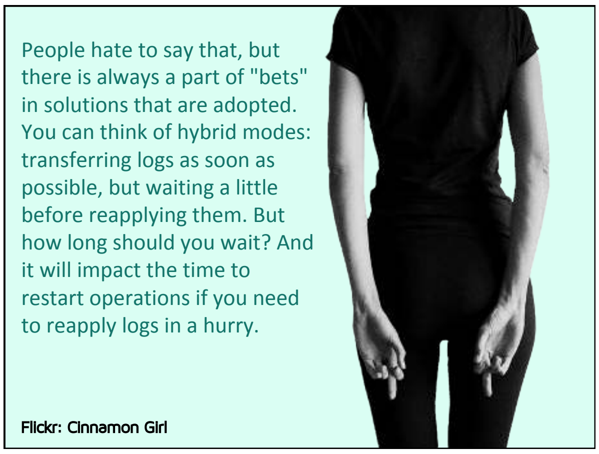
Standby Database corrupted!

Picture by Terry Whalebone

There are always conflicting issues with replication.

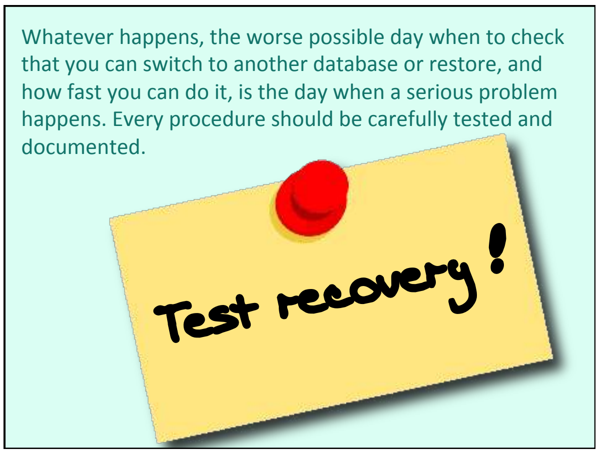


If problems are transferred too fast, you get two corrupted databases instead of one, and must restore from backups.



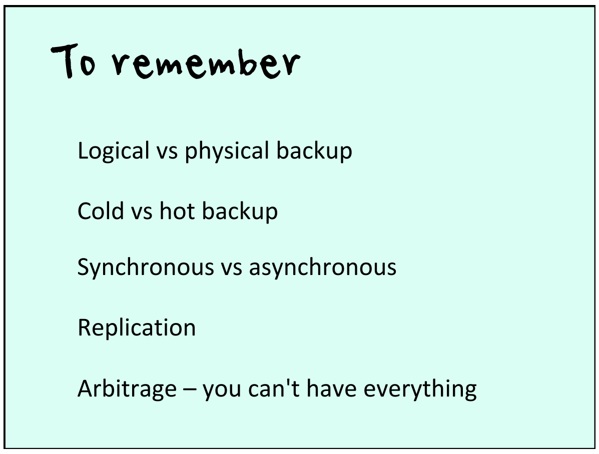
People hate to say that, but there is always a part of "bets" in solutions that are adopted. You can think of hybrid modes: transferring logs as soon as possible, but waiting a little before reapplying them. But how long should you wait? And it will impact the time to restart operations if you need to reapply logs in a hurry.

Flickr: Cinnamon Girl



Whatever happens, the worse possible day when to check that you can switch to another database or restore, and how fast you can do it, is the day when a serious problem happens. Every procedure should be carefully tested and documented.

Test recovery!



To remember

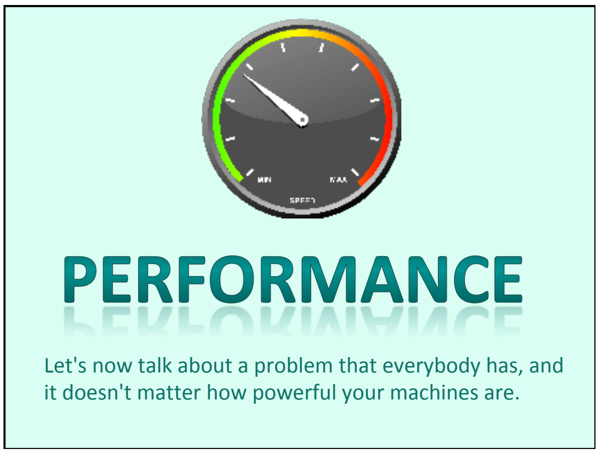
Logical vs physical backup

Cold vs hot backup

Synchronous vs asynchronous

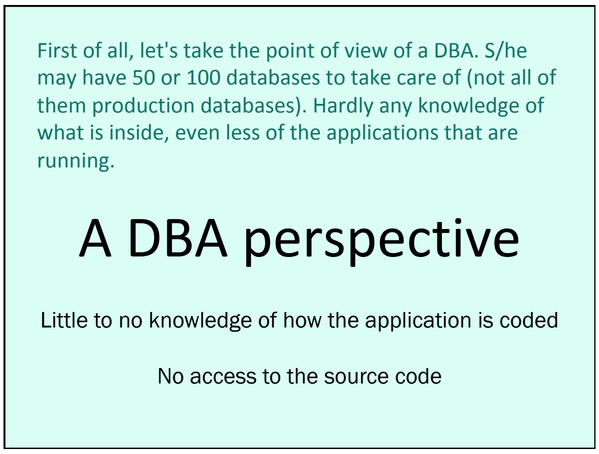
Replication

Arbitrage – you can't have everything


MIN
MAX
SPEED

PERFORMANCE

Let's now talk about a problem that everybody has, and it doesn't matter how powerful your machines are.

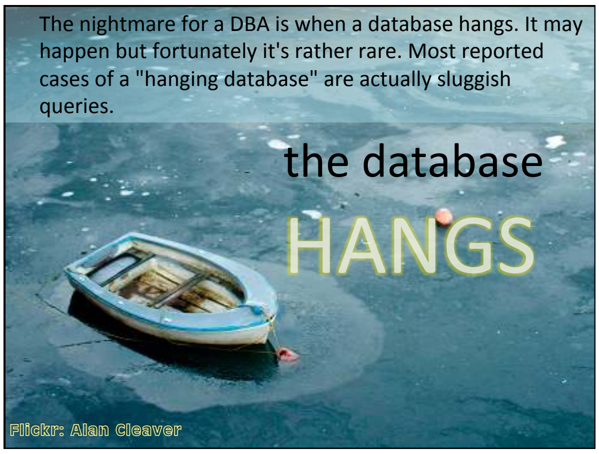


First of all, let's take the point of view of a DBA. S/he may have 50 or 100 databases to take care of (not all of them production databases). Hardly any knowledge of what is inside, even less of the applications that are running.

A DBA perspective

Little to no knowledge of how the application is coded

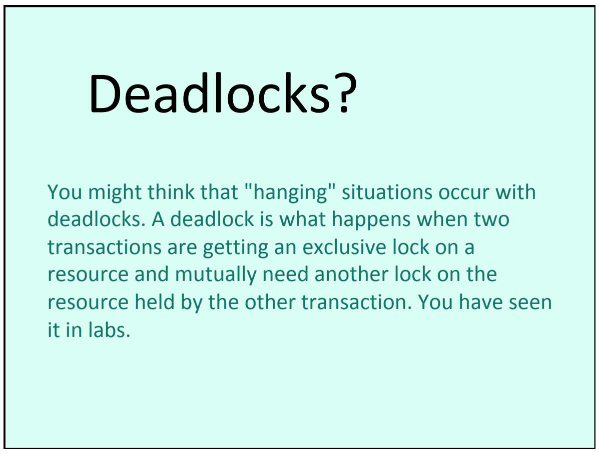
No access to the source code



The nightmare for a DBA is when a database hangs. It may happen but fortunately it's rather rare. Most reported cases of a "hanging database" are actually sluggish queries.

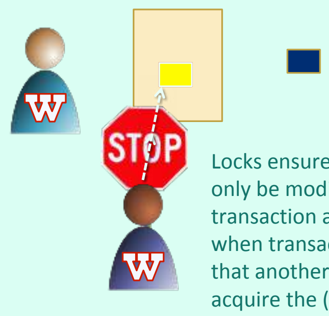
the database HANGS

Flickr: Alan Cleaver



Deadlocks?

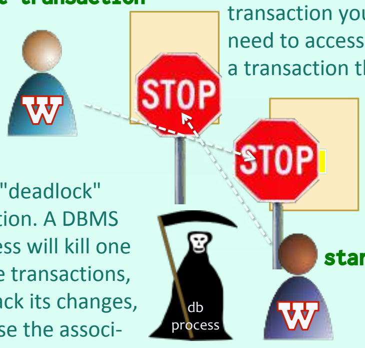
You might think that "hanging" situations occur with deadlocks. A deadlock is what happens when two transactions are getting an exclusive lock on a resource and mutually need another lock on the resource held by the other transaction. You have seen it in labs.



Locks ensure that one row can only be modified by one transaction at a time. It's only when transaction commits that another transaction can acquire the (exclusive) row lock.

start transaction

You have a problem when in a transaction you lock rows and need to access rows locked by a transaction that wants to modify the rows that you lock.



start transaction

It's a "deadlock" situation. A DBMS process will kill one of the transactions, rollback its changes, release the associated locks and allow the other transaction to complete.

Deadlocks:

Annoying, but a problem for developers, not DBAs

If you except the case of bugs in the DBMS software (yes, it happens) deadlocks are usually a development problem solved by always locking resources in the same order.

Why a DBMS may freeze:

File-system full

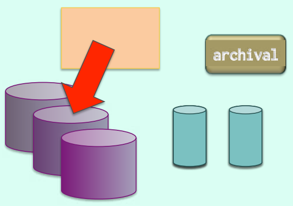
You can see real problems of a hanging database when a file-system where the DBMS software needs to write becomes full, which blocks everything.

With every complex system such as a DBMS you have lots of log files written. When a problem occurs, they can grow very big. Disk space should be monitored.

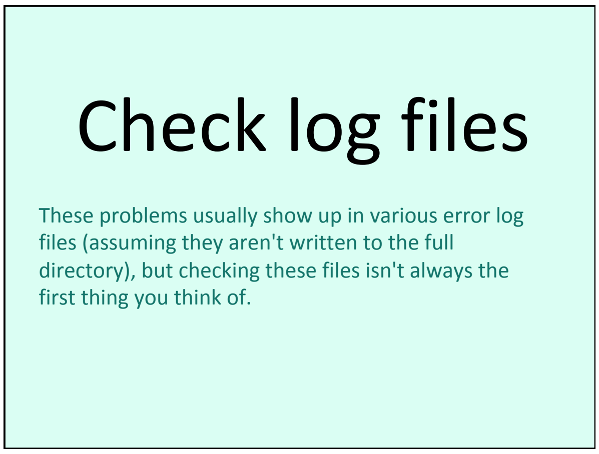
Why a DBMS may freeze:

ORACLE

Log Archival inactive/blocked

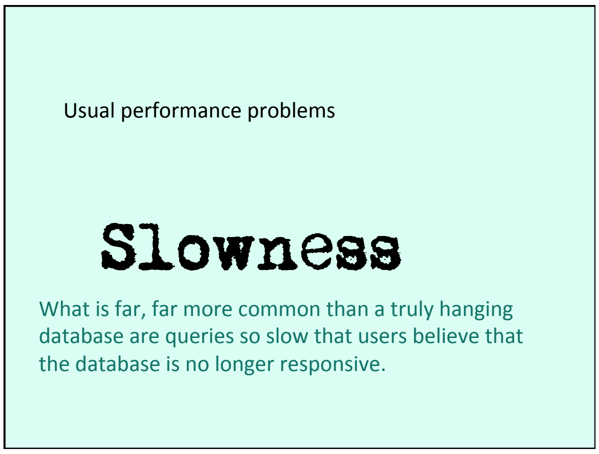


One such case may happen with Oracle when log files have to be archived before being overwritten (a common setting) and the destination is full.



Check log files

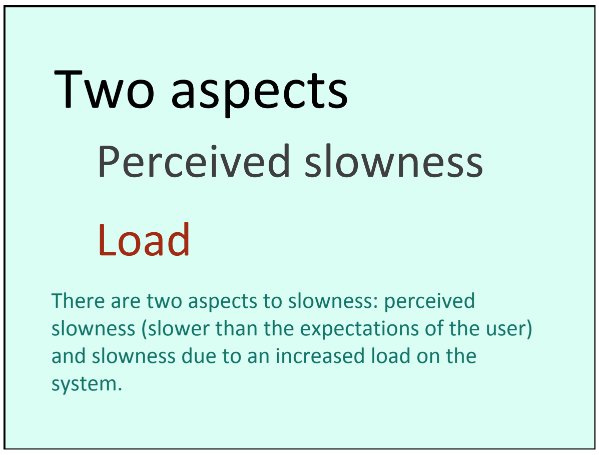
These problems usually show up in various error log files (assuming they aren't written to the full directory), but checking these files isn't always the first thing you think of.



Usual performance problems

Slowness

What is far, far more common than a truly hanging database are queries so slow that users believe that the database is no longer responsive.

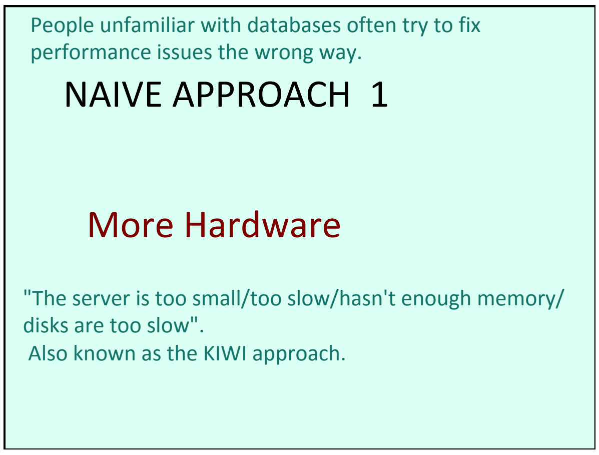


Two aspects

Perceived slowness

Load

There are two aspects to slowness: perceived slowness (slower than the expectations of the user) and slowness due to an increased load on the system.



People unfamiliar with databases often try to fix performance issues the wrong way.

NAIVE APPROACH 1

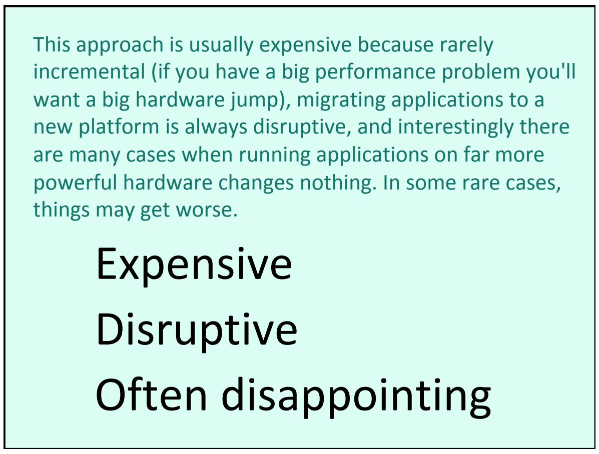
More Hardware

"The server is too small/too slow/hasn't enough memory/ disks are too slow".
Also known as the KIWI approach.

KIWI

Not this.

Kill It With Iron



This approach is usually expensive because rarely incremental (if you have a big performance problem you'll want a big hardware jump), migrating applications to a new platform is always disruptive, and interestingly there are many cases when running applications on far more powerful hardware changes nothing. In some rare cases, things may get worse.

Expensive

Disruptive

Often disappointing

A fast car isn't the only requirement for driving fast.

http://www.carthrottle.com/supercars-in-cities/

NAIVE APPROACH 2

Adding indexes everywhere

Another common problem with people who have very vague ideas about database is the deeply ingrained belief that an index is always the answer to a query speed problem.

Useless indexes are far more common than missing indexes.

NAIVE APPROACH 3

Can we tune the database?

When you come as THE expert, there is often this hope that you will know the obscure parameter setting that fixes all problems.

Parameters.

ORACLE > 300

On an oracle mailing list, a beginner once asked about which parameter to change to make SQL run faster. Someone jokingly mentioned the undocumented parameter `_run_sql_faster` (undocumented parameters in Oracle start with _) that had to be set to TRUE, as, he explained, Oracle was introducing on purpose loops in their code to slow everything down.

Magic Parameter Dance

The real fun was a few minutes later when someone (not the original poster) asked:
What is your version of Oracle? I tried on mine and it said "invalid parameter".

NAIVE APPROACH 4

What are the slow queries?

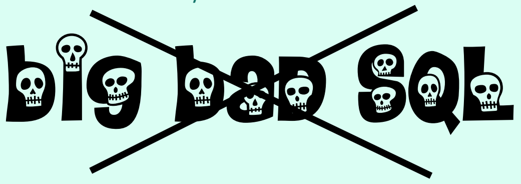
Another poor approach to performance issues is to try to track down the killer query that drags everything down.

You find this approach in the (optional) "slow query" MySQl log file.

MySQL "slow query log"

$$\left.\begin{array}{l} > 10\text{ s} \\ >= 0\text{ rows examined} \\ \text{Only queries that use indexes} \end{array}\right\}\text{default}$$

Although you can find some pretty slow queries, in most cases what really loads a DBMS server is a bunch of moderate queries, and sometimes very fast ones, that are executed far too many times.



More about this in the developer approach to performance.

First of all expectations should be realistic.

Slowness
is a problem
only when
it could and
should be fast.

What matters isn't the number of rows that you return: it's the number of rows that you have to inspect.

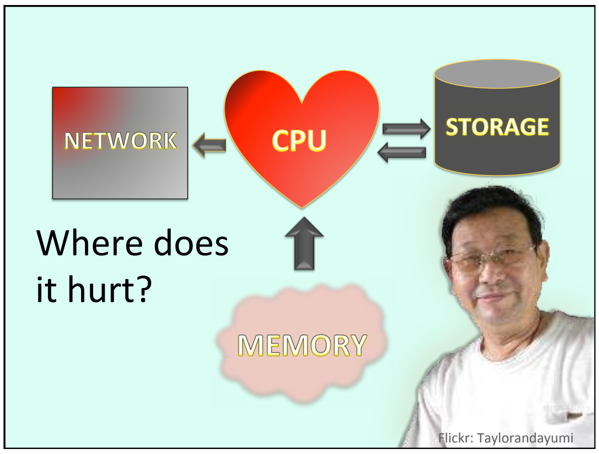
You
CANNOT
aggregate 5 billion rows fast.

Modeling?

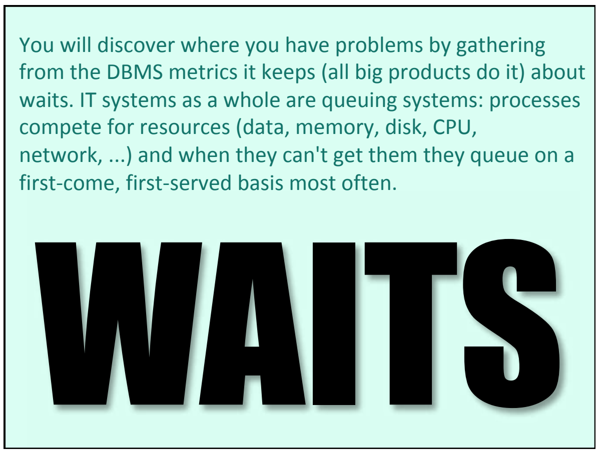
If you spend your time aggegating data it’s perhaps that your database model isn’t suitable for what you want to do.
Database design is very important for performance (although when you are asked to fixed an urgent production performance problem, re-designing the database isn't an option). Badly designed databases lead almost invariably to performance issues when the number of rows takes off.

Correct starting-point

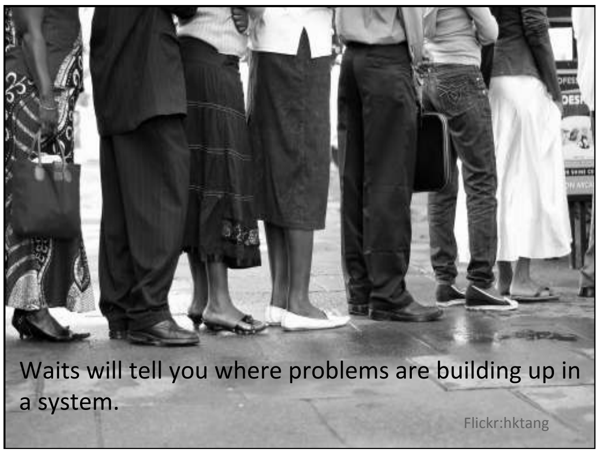
After having listed a number of naive approaches to solving a performance problem, pehraps it's time to say, once again from a DBA standpoint, what is the correct approach.
The correct approach is to find out WHERE you have a problem; which is the resource that fails you.



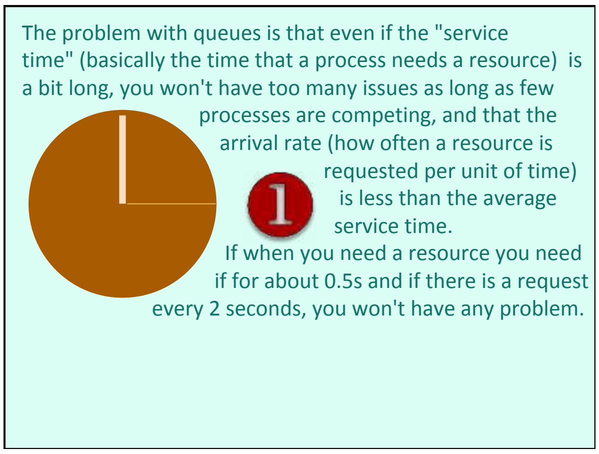
Where does it hurt?


You will discover where you have problems by gathering from the DBMS metrics it keeps (all big products do it) about waits. IT systems as a whole are queuing systems: processes compete for resources (data, memory, disk, CPU, network, ...) and when they can't get them they queue on a first-come, first-served basis most often.

WAITS

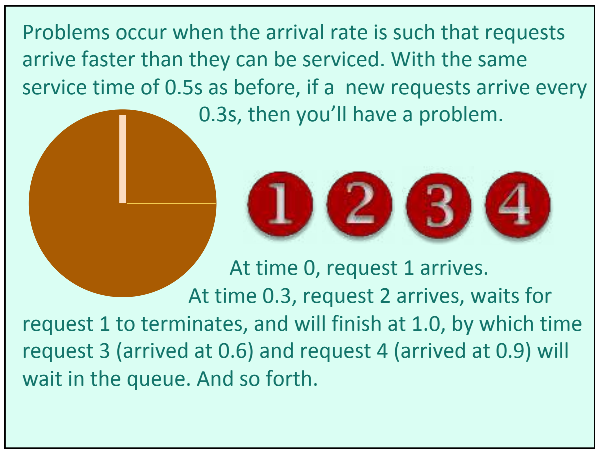


Waits will tell you where problems are building up in a system.



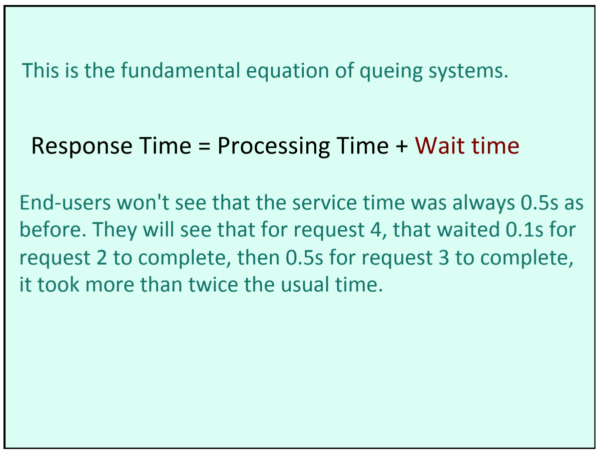
The problem with queues is that even if the "service time" (basically the time that a process needs a resource) is a bit long, you won't have too many issues as long as few processes are competing, and that the arrival rate (how often a resource is requested per unit of time) is less than the average service time.

If when you need a resource you need if for about 0.5s and if there is a request every 2 seconds, you won't have any problem.



Problems occur when the arrival rate is such that requests arrive faster than they can be serviced. With the same service time of 0.5s as before, if a new requests arrive every 0.3s, then you'll have a problem.

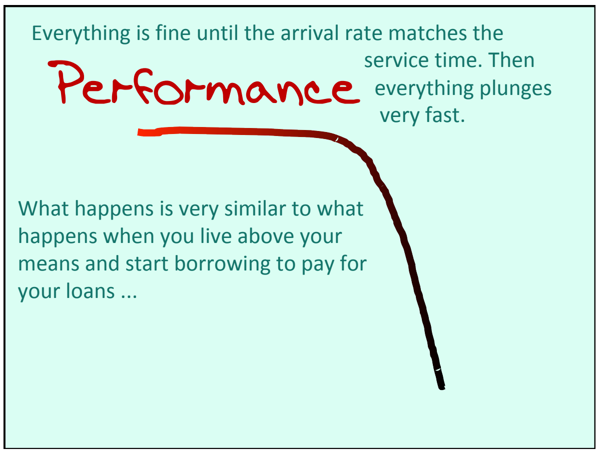
At time 0, request 1 arrives. At time 0.3, request 2 arrives, waits for request 1 to terminates, and will finish at 1.0, by which time request 3 (arrived at 0.6) and request 4 (arrived at 0.9) will wait in the queue. And so forth.



This is the fundamental equation of queing systems.

Response Time = Processing Time + Wait time

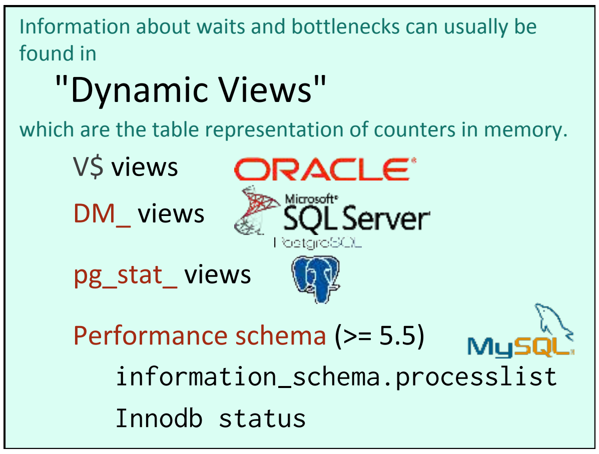
End-users won't see that the service time was always 0.5s as before. They will see that for request 4, that waited 0.1s for request 2 to complete, then 0.5s for request 3 to complete, it took more than twice the usual time.



Everything is fine until the arrival rate matches the service time. Then everything plunges very fast.

Performance

What happens is very similar to what happens when you live above your means and start borrowing to pay for your loans ...



Information about waits and bottlenecks can usually be found in

"Dynamic Views"

which are the table representation of counters in memory.

V$ views ORACLE

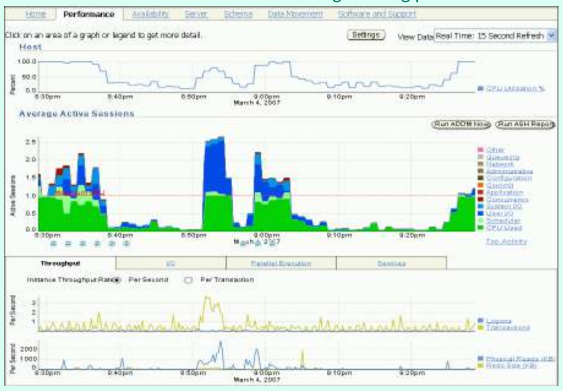
DM_ views Microsoft SQL Server

pg_stat_ views PostgreSQL

Performance schema (>= 5.5) MySQL

```
information_schema.processlist
Innodb status
```

Tools are available for real-time monitoring with big products.



Graphics and color are nice ...

BUT?

There are several problems with these tools. Very often they are far too technical and if you aren't a specialist of a DBMS you won't understand much. They will also show you many things that you can't act upon anyway. Finally they may show symptoms but they perform no diagnosis.

Meaning of averages?

Additionally, they will show you averages over sometimes thousands of active sessions, and averages aren't always very meaningful.

www.old-picture.com

Tracing

hard to do on a production database

Configuration mistakes

Not enough server processes (pooling)

Bad network configuration

Buffers far too small

Journal files on slow or busy disks

There are indeed from time to time big database configuration mistakes. You are more likely to encounter these problems with a low-end DBMS such as MySQL than a high-end one such as Oracle that requires seasoned DBAs

Database queries

But most performance issues are traceable to database queries. Once again, not THE big bad query that drags everything down. Mostly queries that are fast enough when executed alone, and become terrible when repeated.

An empirical law commonly applied in IT is Pareto's law, that states that 80% of the load comes from 20% of processes. More like 90/10 with databases.

Vilfredo Pareto
1848-1923

80

20

Query execution

Physical reads	Read pages from disk (I/Os)
Logical reads	Inspect pages in memory

When a query executes, it can access pages on disk (physical I/Os) or in memory (logical I/Os). For many years people thought that as long as you had 10 LIOs for 1 PIO, everything was fine. Forget ratios.

The number of logical reads is the indicator that really counts. Logical reads are CPU-intensive. If the number of memory blocks accessed times the size of the memory block is far greater than the size of the tables involved in the query, you have a problem. It happens. A lot of programs access repeatedly the same data, and will query 1,000,000 times per day a value, such as an exchange rate, that is updated only once a day.

Logical reads

CPU intensive

How to optimize queries that cannot be changed?

When you are a DBA, though, even if everybody seems to think that if the database is slow you are the only one able to fix the problem, you cannot really cure the problem. You can only try to make the problem less painful.

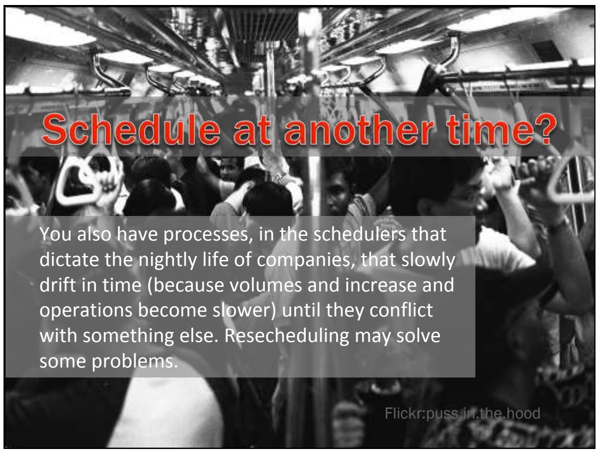
Try the simplest.

是故百戰百勝，非善之善者也；
不戰而屈人之兵，善之善者也。

You sometimes encounter processes that run every night for no good reasons. Stuff that used to be there for a good reason that is long gone.

Don't do what isn't necessary

Remember that information systems often have a long past. Some "data check" programs that scan everything may once have been useful, no longer be, and still be running. Reports may be generated that nobody reads. In every big company you discover somewhere machines that are running, for which you are paying software licenses, and the users of which are unknown (a popular tactic is unplugging the machine and waiting for complaints). The same happens with programs.



Schedule at another time?

You also have processes, in the schedulers that dictate the nightly life of companies, that slowly drift in time (because volumes and increase and operations become slower) until they conflict with something else. Resecheduling may solve some problems.

Flickr:puss in the hood

Parsing issues

Once you are certain that everything that runs is necessary and couldn't run at a better time, you have to try harder.
A common issue is parsing, which is a CPU intensive operation, and that can sometimes be limited. When people hard-code search criteria in queries (concatenating constants in queries instead of using prepared statements with parameters), not only do they open the door to SQL injection but every query for which only constants are different appears as a new query that has to be parsed.

Force substitution of constants by variables

ORACLE®

Microsoft® SQL Server®

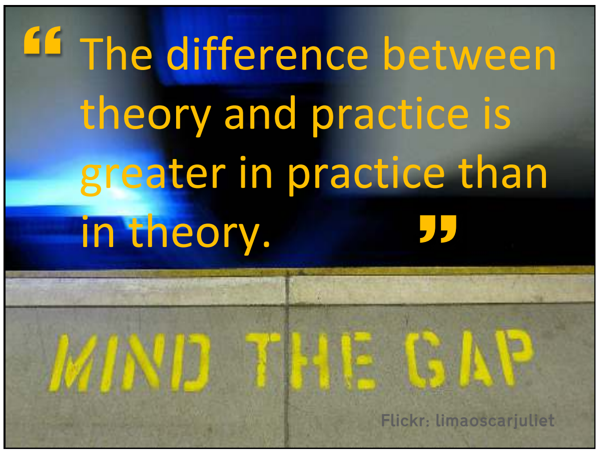
To avoid this problem, both Oracle and SQL server implement modes in which, before queries are passed to the optimizer, constants are extracted on the server side and turned into parameters.

Because SQL is declarative, the optimizer, which decides on how queries should be executed, has a very important role.

SQL = Declarative Language

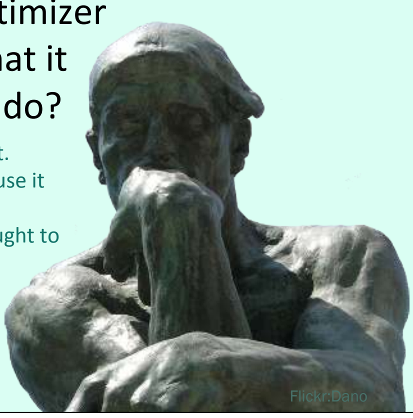
You STATE what you want, the system is clever enough to find it as fast as possible.

That's the theory.


“The difference between theory and practice is greater in practice than in theory.”

Is the optimizer doing what it ought to do?

Sometimes, not.
But often because it doesn't know everything it ought to know.



When the optimizer finds a criterion such as
where column_name = 'value'
it needs to check a few things about the column.

Is there an index?

The optimizer won't create indexes for you.

Can it be used?

If the index is on (other_column, column_name) and there is no condition on other_column ...

Is it worth using?

If 60% of rows contain 'value' in column_name, probably not.

Execution plan

The execution plan is the output of the optimizer: hopefully the fastest way to return the result set. Optimizers often perform a good job, and as always it's the cases when they fail that everybody notices.

explain

We have seen that "explain" can tell you about index usage.

Index used?

Once again, the optimizer won't be able to create a good execution plan and decide about using or not using indexes (few rows, indexes good, many rows, indexes bad) if it hasn't statistics. But what kind of statistics?

Ideally an index on a column in which one value dominates should NOT be used for the most common value, and possibly used for a rare value. The choice of the optimizer will be based on what it knows of distribution, therefore value statistics.

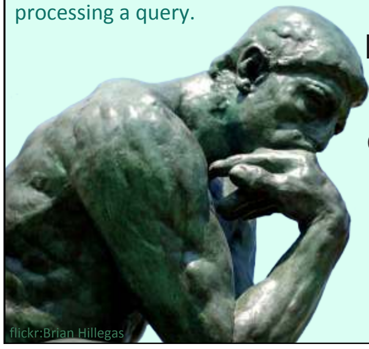
Statistics ?

Those statistics must also be up-to-date (at least not diverge too much from reality when the query is executed)

Up-to-date ?

The main problem of the optimizer is getting an estimate of how many rows are returned at each step when processing a query.

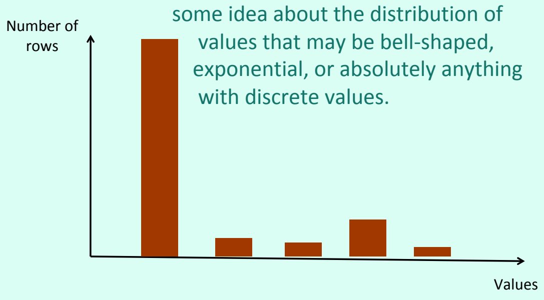
Selectivity?



How many rows returned at each step?

(This is known as cardinality estimate. Remember, *cardinality* just means *count*)

To do a good job, an optimizer needs some idea about the distribution of values that may be bell-shaped, exponential, or absolutely anything with discrete values.



People

🇨🇳	1,300,000,000
🇫🇷	60,000,000

In terms of population size, there is no contest between China and France.

Family Names

🇨🇳	3,000
🇫🇷	1,300,000

BUT when it comes to surnames, it's the opposite. France and Italy are the countries with the most different surnames. It's common for French schoolboys (more than girls) to call each other by their surnames, it sometimes happens in companies too.

One Family Name

🇨🇳 400,000

🇫🇷 46

It means that on average, there are almost half a million Chinese per surname, and under 50 French.
But we have already said it, averages are misleading.
Some names are common in France too.

Basic statistics on columns are composed of minimum and maximum values, as well as the number of distinct values; but as we have just seen with surnames, dividing the number of rows in a table by the number of distinct values in a column (assuming uniform distribution) gives a pretty wrong idea of how many rows will be associated with one value.

Minimum value

Maximum value

Number of distinct values

To know better about the distribution of values, DBMS compute (often on indexed columns that aren't unique but sometimes on other columns as well) frequency histograms that roughly say how many times one value occurs.

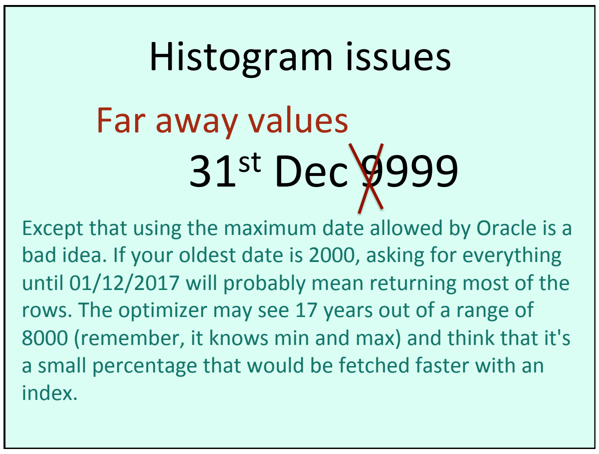
It's easy when there is a small number of distinct values. When there is a large number of distinct values, the DBMS uses a small sample and tries to guess from there. Sometimes it guesses wrong.

Histogram issues

Far away values

Oracle doesn't index nulls

A problem with Oracle is linked to something which is common with dates, dummy values. As nulls may not be stored in Oracle and may have no address, they cannot be indexed (SQL Server can index nulls). Columns "valid_until" are frequent, and rather than having a null to indicate current values, people often prefer using a dummy, far away date that can be indexed.

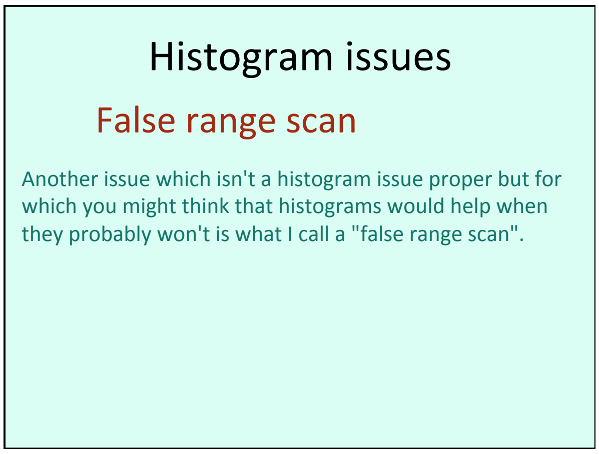


Histogram issues

Far away values

31st Dec ~~9~~999

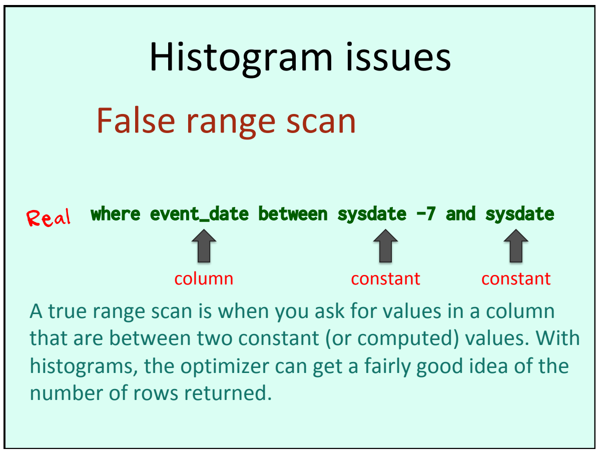
Except that using the maximum date allowed by Oracle is a bad idea. If your oldest date is 2000, asking for everything until 01/12/2017 will probably mean returning most of the rows. The optimizer may see 17 years out of a range of 8000 (remember, it knows min and max) and think that it's a small percentage that would be fetched faster with an index.



Histogram issues

False range scan

Another issue which isn't a histogram issue proper but for which you might think that histograms would help when they probably won't is what I call a "false range scan".



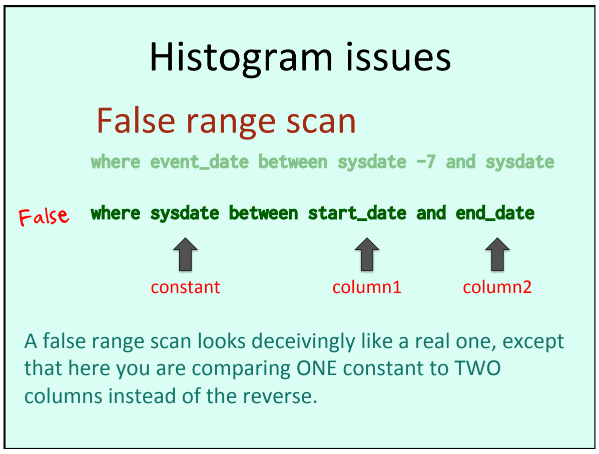
Histogram issues

False range scan

Real `where event_date between sysdate -7 and sysdate`

column
constant
constant

A true range scan is when you ask for values in a column that are between two constant (or computed) values. With histograms, the optimizer can get a fairly good idea of the number of rows returned.



Histogram issues

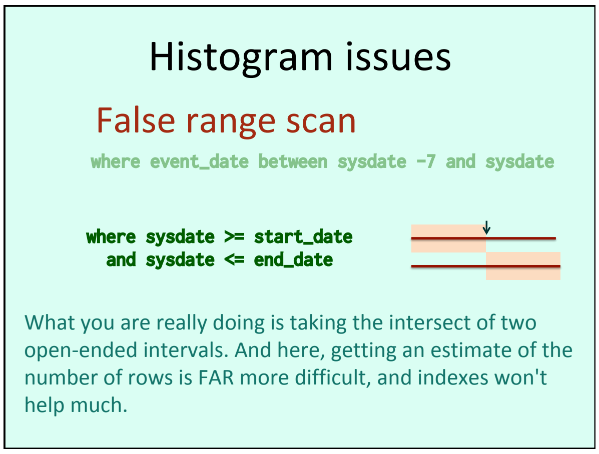
False range scan

`where event_date between sysdate -7 and sysdate`

False `where sysdate between start_date and end_date`

constant
column1
column2

A false range scan looks deceivingly like a real one, except that here you are comparing ONE constant to TWO columns instead of the reverse.



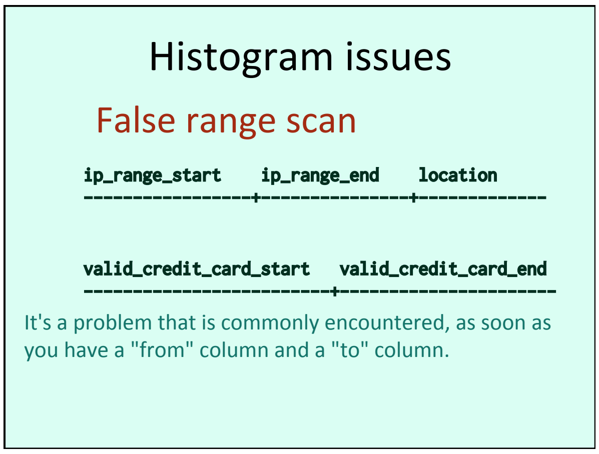
Histogram issues

False range scan

```
where event_date between sysdate -7 and sysdate
```

```
where sysdate >= start_date
  and sysdate <= end_date
```

What you are really doing is taking the intersect of two open-ended intervals. And here, getting an estimate of the number of rows is FAR more difficult, and indexes won't help much.



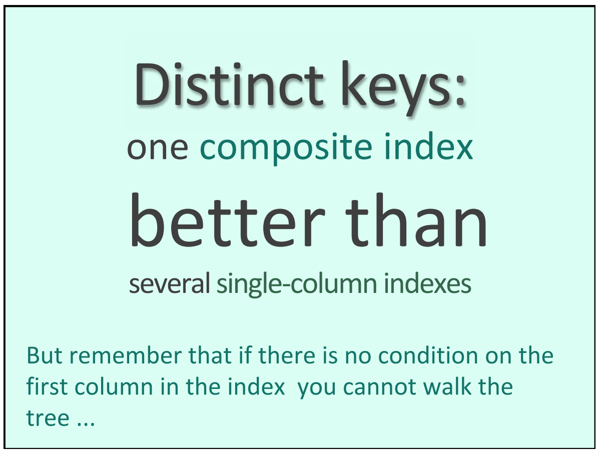
Histogram issues

False range scan

```
ip_range_start    ip_range_end    location
-----------------+---------------+-------------

valid_credit_card_start   valid_credit_card_end
-------------------------+-----------------------
```

It's a problem that is commonly encountered, as soon as you have a "from" column and a "to" column.



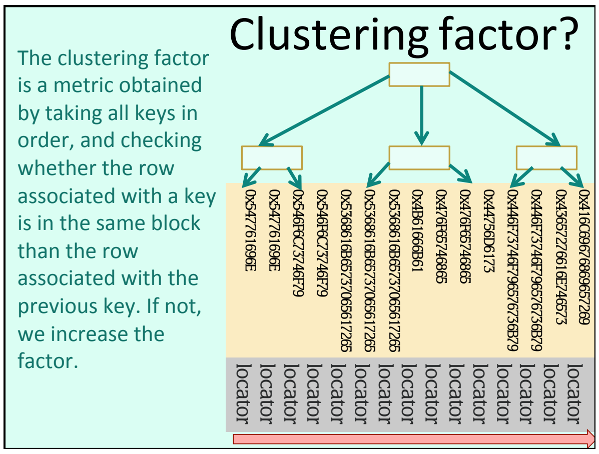
Distinct keys:

one composite index

better than

several single-column indexes

But remember that if there is no condition on the first column in the index you cannot walk the tree ...



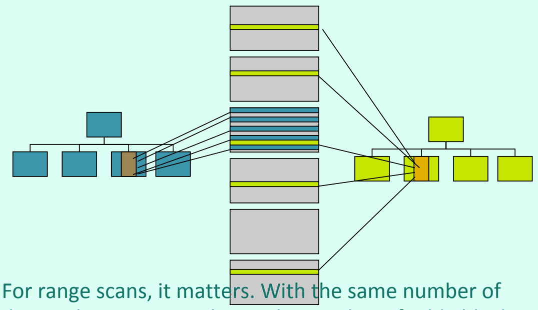
Clustering factor?

The clustering factor is a metric obtained by taking all keys in order, and checking whether the row associated with a key is in the same block than the row associated with the previous key. If not, we increase the factor.

Relationship between **order of rows** and **order of keys**

What is the result? If the clustering factor is close to the number of rows in the table, that means that two successive keys are generally in different blocks. If it's close to the number of blocks in the table, it means that the rows are more or less in the same order as the keys.

Order of Data



For range scans, it matters. With the same number of distinct keys in two indexes, the number of table blocks to inspect may be very different.

Lower clustering factor

=

Fewer blocks to fetch and inspect

```
select …
from …
where datecol between
    cast(:date_min as date)
and cast(:date_max as date)
and status = :status
```

?

But when a status value is passed as a parameter, either because the developer coded the query as such or because the replacement of constants by parameters was forced on the server side, we may have a problem.

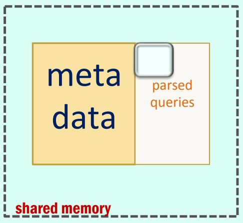
Check parameters when parsing?

The obvious idea is that when we parse the query, we should peek at the values and take decisions such as to use or not to use an index based on these values. The problem is that parameters are meant to change; and if the execution path selected was probably good for the values passed when the query was analyzed, nothing guarantees that it will remain so when the query will be executed again with different parameters.

We may end up with a query that is parsed for a rare value, will decide that the index is beneficial, and will use the index even for common values.

```
select …
from …
where datecol between
    cast(:date_min as date)
and cast(:date_max as date)
and status = :status
```

ERROR

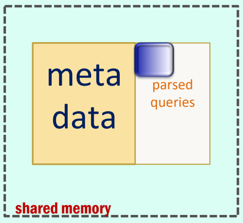


Rare: use index

What may happen is that if the query isn't executed for a while, the parsed query will age out of the cache, and the query may be reexecuted later in the day with a completely different plan.

```
select …
from …
where datecol between
    cast(:date_min as date)
and cast(:date_max as date)
and status = :status
```

COMPLETED



Common: don't use index

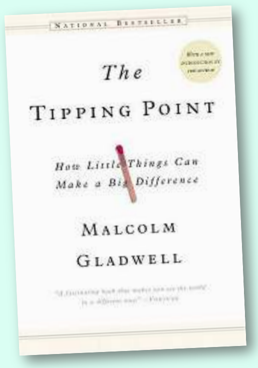
Users will run exactly the same query, with exactly the same values, are different times of the day, and get performance that may widely vary. Usually it drives them crazy.

Plan Instability

People prefer predictable mediocrity to something that is sometimes blazingly fast and at other times hopelessly slow.
A product such as Oracle tries to remedy to this problem by keeping watch over queries that exhibit instability and reparsing them more often, but it's a remedial process after the problem occurred, not a preventive one.

This phrase by Blaise Pascal could illustrate that some small changes may have big effects. If Cleopatra had been a little less attractive, the history of Europe (and possibly the world) would have been different.

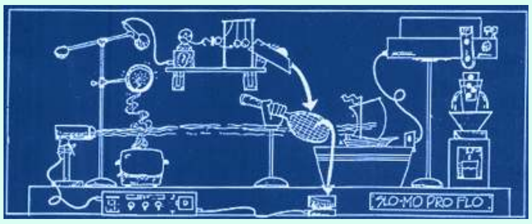
Cleopatra's nose: had it been shorter, the whole face of the world would have been changed.



Gladwell expressed the same idea more recently.
The problem is that optimizer choices are subject to threshold values, that a plan can change all of a sudden because one table has grown a bit faster than others, and sometimes it can make a big difference in performance.

Some DBAs try to fix big issues by forcing queries that have a given signature (checksum) to apply an execution plan known to work decently.

"Attach" execution plan to query



It's not a long-term solution.

Cure the illness
not the symptom.

Most performance issues come from poor or careless (or both) development. It's extremely hard to try to improve a bad program by only playing with the DBMS server. We shall see later the developer side.
What the DBA should do, though, is whatever is possible to help the optimizer do a good job.

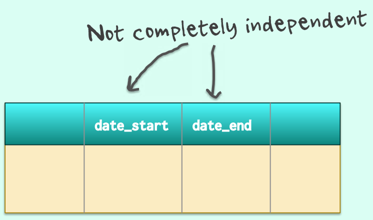
For instance, the optimizer may far overestimate or underestimate a number of rows returned because of correlations between columns.

Correlation

The optimizer assumes 3rd Normal Form

... attributes only depend on the key and are independent

However, saying "independent" is often a bit far fetched.



If I tell you "how many people where born after 1980 and died before 1940" you'll tell me immediately 0. For the optimizer, if 20% of people were born after 1980 and 20% died before 1940, it will be 20%x20% = 4% of the total number of rows. You can ask Oracle to compute stats on groups of columns.

Design and Statistics

"Artificial" Correlations

Knowing which statistics to collect and, from there, trying to estimate how many rows should be returned by such or such parameter value (a crucial step in the choice of an execution plan) is hard enough for the DBMS system. Poor design choices make it far more difficult for it and increase significantly the chances of its taking a bad decision.

Let's take for example the "Entity/attribute/Value" model so loved of people who start coding without really knowing which attributes their data will ultimately hold (more about it later)

PEOPLE

entity	attribute	value
567	first_name	Gene
567	surname	Tierney
567	birth	1920
567	death	1991
567	gender	F

What should be horizontal is vertical. Not only can you forget about foreign keys, but the optimizer will struggle.

468 rows
346 different first names (456 people have one)
418 different surnames
106 different birth years
71 different death years (for 237 people)
2 genders

I have computed a few statistics of my own on a relatively small sample of actors and directors. First I stored my data in a traditional way (id, first name, surname, born, died, gender)

Entity/Attribute/Values

5 different attributes – almost uniform
(null won't appear)

Rows = surname + born + gender for everybody	1404
first_name for almost everybody	456
death year for half the people	237
Total	2097 rows

With the EAV model, the same data becomes a table with half the number of columns and more than three times the number of rows. Only known values are stored.

With a standard model, how many rows will be returned by a condition on gender is easy to compute.

```
gender = 'F'
```
~50%, ~230 rows

With an EAV model, because the attribute name is now part of the data, assuming data independence is wrong.

`attr_name = 'Gender'`	1/5	20%
`and attr_value = 'F'`	230/2097	11%

20% * 11% ?

Assuming that columns are independent, the optimizer will believe that only 2% of the 2,000 rows will be returned, and will be wrong by an order of magnitude.

High odds to pick the wrong algorithm

As, if you remember Kyte's allegory, the suitability of an algorithm depends on data volumes, an EAV model will underestimate volumes and pick on indexes and algorithms that use indexes.

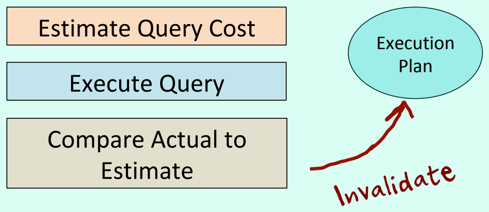
"Extended statistics" on several columns

Hence the benefit of "extended statistics" (which only the top products such as Oracle allow), not only to detect genuine correlation in the data (such as between "born" and "died") but also to try to correct artificial correlation introduced by the data model (because any value cannot be associated with any attribute).

What can the optimizer do?

Let's see how DBMS vendors try to address the problem of performance instability

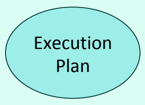
One way is to use a "feedback loop" and compare actual to predicted performance.



Ooooooops

One problem of course is this is a reactive, and not proactive, plan improvement (it means that you know of the problem only when it has occurred).

Variant A : OK



Variant B : KO

And the same execution plan may, depending on parameters associated with a query, be very good in one case and very bad in another case.

Oracle >= 11

Adaptative Cursor Sharing

Bind sensitiveness → Criterion Histogram UNDER SCRUTINY

Bind awareness → Oooops!

Oracle has introduced a mechanism in version 11. If a parameter is bound as a search criterion and there is a histogram that indicates possible problem, a query is checked. If a problem occurs, the plan will be re-evaluated when parameters change.

SQL Server

"Recompile" directive in SQL statement

SQL Server has a slightly cruder mechanism, a directive that you can add to a problem query that asks for it to be recompiled at each execution. Note that it must be in the code, so the DBA cannot do much there.

ORACLE

Dynamic sampling 0 ... **2** ... 10

Occurs during hard-parse

When set to 3 and above tries to validate guesses

An interesting, and often effective, feature in Oracle is "dynamic sampling" which basically asks the optimizer to guess less and check data a bit more.

A Developer perspective

Full control of the application

Let's now see performance from the completely different perspective of the developer. The developer CAN change the application. One problem is that, if DBAs are often seasoned professionals, in many cases developers are the cheapest beginners that could be found.

This is code I have found for real.

CONTEXT

Production code

"Reference database"

One of the top 20 investment banks worldwide

```
-- ----------------------------------------------------------------
-- # NAME : F_RCP_NBRDET_CVNBDR
-- # COM  : Compute the number of details in TDETCVNBDR table
-- #        for a given convention and type I, E or P.
-- ----------------------------------------------------------------

FUNCTION F_RCP_NBRDET_CVNBDR(
          p_codcvn        tcvnbdr.codcvn%TYPE,
          p_typdet        tdetcvnbdr.typdet%TYPE) RETURN NUMBER
IS
  CURSOR c_det
  IS
  SELECT codint
  FROM tdetcvnbdr
  WHERE typdet = p_typdet AND
        codcvn = p_codcvn;
  t_count         NUMBER(10) := 0;
BEGIN
  FOR rec IN c_det LOOP
     t_count := t_count + 1;
  END LOOP;
  RETURN t_count;
EXCEPTION
  WHEN OTHERS THEN
       RETURN NULL;
END;
```

Never heard about count(*)?

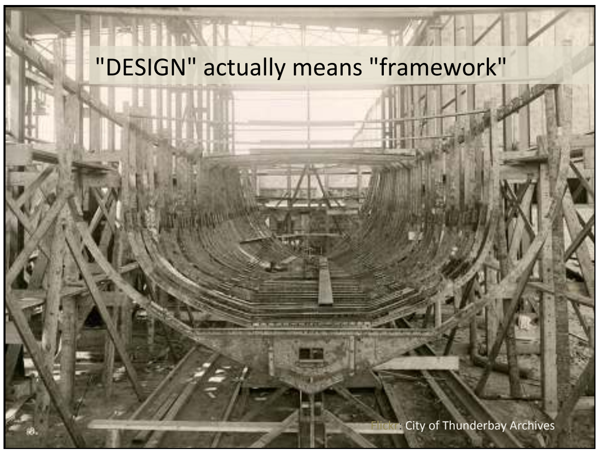
Any idea of what this would cost in a query, compared to a simple join?

FUNDAMENTALS

Database Design

Many people forget that performance is something that comes straight from design.

"DESIGN" actually means "framework"



There is a very clear link between bad design and bad performance. Bad design makes everything more complicated. Things that could be done automatically in the database need to be done in programs, slowing them significantly.

bad design

bad performance

SQL against BAD DESIGN

Querying reasonably efficiently badly designed tables often requires acrobatic SQL that few people are able to write.

Flickr: suran2007

First example

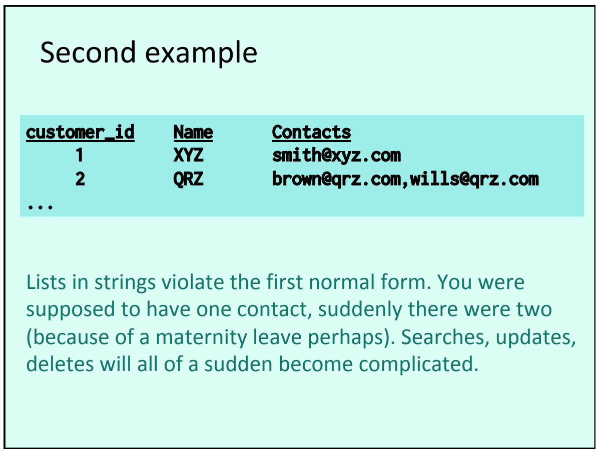
(Real life example)

Loc_id	Name	Name_fr	Name_it	Name_de
1	London	Londres	Londra	London
2	Edinburgh	Edimbourg	Edimburgo	Edinburgh
3	Rome	Rome	Roma	Rom
4	Paris	Paris	Parigi	Paris
5	Munich	Munich	Monaco	München
6	Copenhagen	Copenhague	Copenaghen	Kopenhagen
...		...	...	...

When people enter a name, you have to look for it into all columns or, if you know the language, have one query per language. If you ever add a new language, you'll have to modify all your queries.

Loc_id	Lang	Name
1	en	London
1	fr	Londres
1	it	Londra
2	en	Edinburgh
2	fr	Edimbourg
3	en	Rome
4	en	Paris
5	en	Munich
6	en	Copenhagen
...		

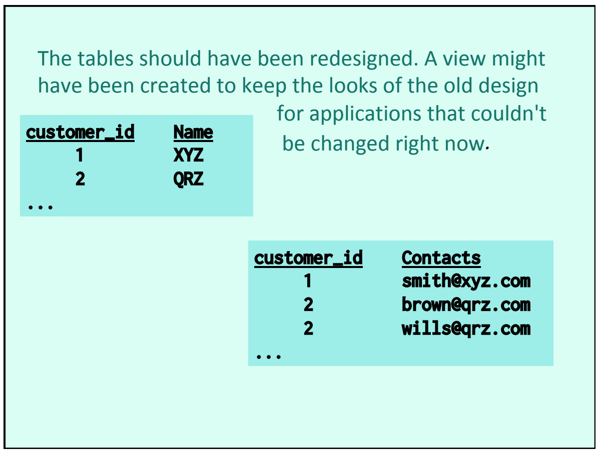
Primary key includes the language. Additonally index on name and language if you have many places. Adding new languages will not impact the code.



Second example

customer_id	Name	Contacts
1	XYZ	smith@xyz.com
2	QRZ	brown@qrz.com,wills@qrz.com
...		

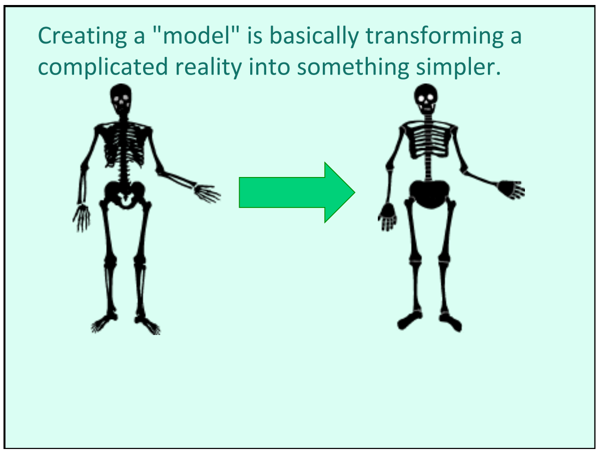
Lists in strings violate the first normal form. You were supposed to have one contact, suddenly there were two (because of a maternity leave perhaps). Searches, updates, deletes will all of a sudden become complicated.



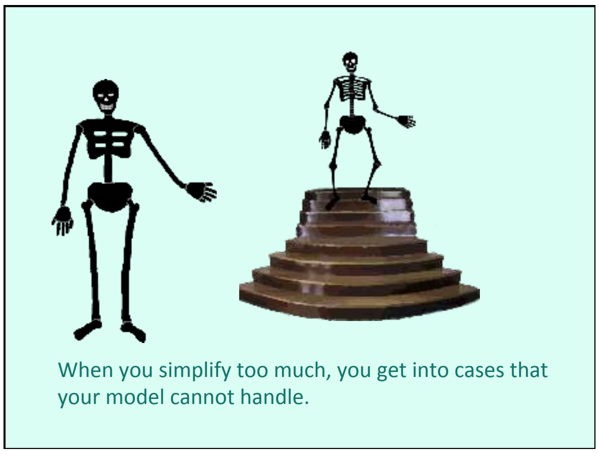
The tables should have been redesigned. A view might have been created to keep the looks of the old design for applications that couldn't be changed right now.

customer_id	Name
1	XYZ
2	QRZ
...	

customer_id	Contacts
1	smith@xyz.com
2	brown@qrz.com
2	wills@qrz.com
...	

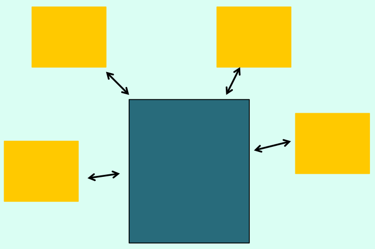


Creating a "model" is basically transforming a complicated reality into something simpler.

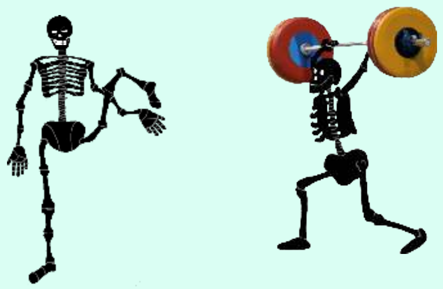


When you simplify too much, you get into cases that your model cannot handle.

Dimensional Model



In information systems, an over-simplified model is the "dimensional model" popular with decision support systems.



When you overdo everything and want a model that can handle anything, usually there is one thing that it doesn't handle well: load.

EAV Model

PEOPLE

peopleid	first_name	surname	birthdate	gender
567	Gene	Tierney	19-NOV-1920	F

PEOPLE

entity	attribute	value
567	first_name	Gene
567	surname	Tierney
567	birthdate	19-NOV-1920
567	gender	F

Such an example is the Entity/Attribute/ Value (or EAV) model, which is supposed to handle even the cases you hadn't thought of.

You have no nulls in such a model, and some people seem to be genuinely persuaded that its good relational design because of this.
If nulls are indeed better avoided, EAV fans seem to forget that they have no data types either, because all types of values are stored in to the same (text) "value" column, and that they cannot have foreign keys, that the only thing that can be unique is (entity, attribute), and so forth.

This model is terribly popular with software package vendors. Why? All companies like to think that they are doing things in a special way, that they have this "secret ingredient" that competitors haven't, and in an information system the "secret ingredient" might be a piece of data. How can I manage some non standard data? Simple, add attributes! No need to add columns to table, which would be (yek) programming! With the EAV model everything can be parameterized!

popular with software packages

Those vendors are selling the idea that their model is highly flexible. However, what they obtain isn't really the flexibility that you might think of.

FLEXIBILITY

Flickr: SD Dirk

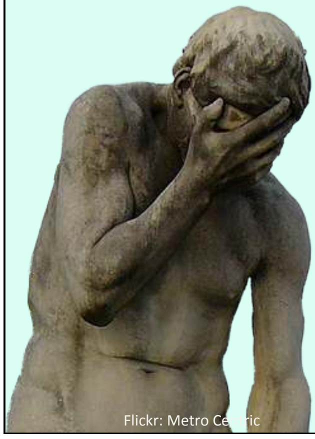
Unless you think that noodles are flexible.

Flickr: Jess Lander

an example

Real customer case, as always.

```
SELECT f.symbol
FROM string_attribute_values0 sav1,
     integer_attribute_values0 iav,
     string_attribute_values0 sav2,
     fundmstr f
WHERE f.ado_num = sav1.ado_num
 AND sav1.ado_num = iav.ado_num
 AND sav1.ado_num = sav2.ado_num
 AND sav1.attr_num=(select attr_num
                    from attribute_descriptions
                    where attr_id = :"SYS_B_00")
 AND sav1.attr_value = :"SYS_B_01"
 AND sav1.act_date <= SYSDATE and sav1.end_date > SYSDATE
 and iav.attr_num=(select attr_num
                   from attribute_descriptions
                   where attr_id = :"SYS_B_02")
 AND iav.attr_value != :"SYS_B_03"
 AND iav.act_date <= SYSDATE and iav.end_date > SYSDATE
 AND ((f.def_attr_templ = :"SYS_B_04"
       AND sav2.attr_num=(select attr_num
                          from attribute_descriptions
```



There were pages and pages like this, and performance was disastrous.
The EAV model is OK if you want to store parameters for your application, or perhaps for a small extension. Not if the main table in your application only has these three columns.

hit pages in memory

again

and again

and again

and again

An EAV table can only use indexes, even when scanning would be the right thing to do, and you will use tons of joins, very heavy on CPU. Performance plunges dramatically when volumes head up.

Some clever SQL rewriting sometimes helps (to a limited extent), but usually people have been so clever with their EAV design that when they come to writing SQL they have exhausted all their cleverness.