CS307

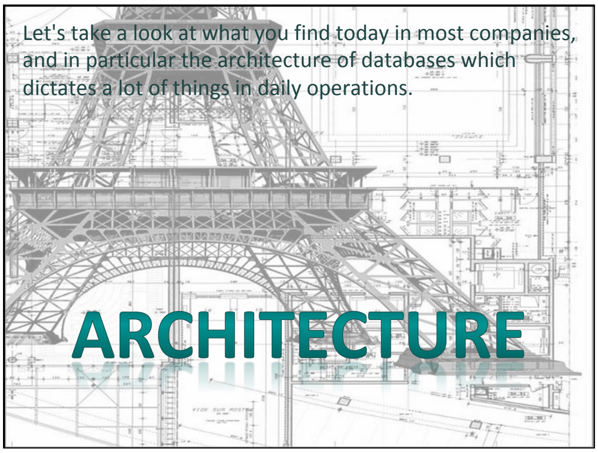
Database Principles

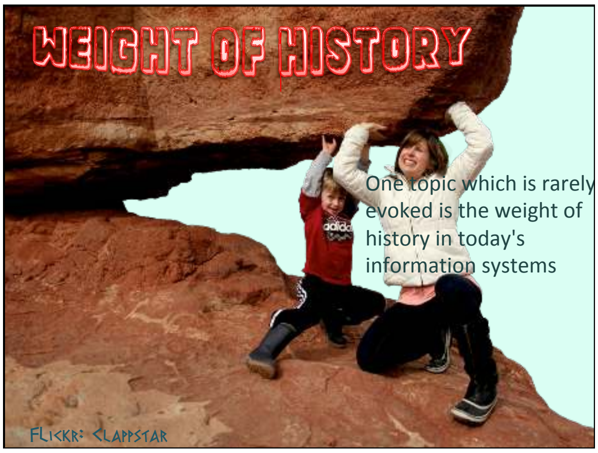
Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

NEXT WEEK !

Guy Harrison





ORACLE
1978 Version 1
1985 Version 5
1988 Version 6

Major architecture change

INGRES
(1970s)

SYBASE
1987 Version 1
1988

PostgreSQL
1988 Prototype

SQL Server

MySQL
1994 First release

ALL major DBMS products were designed in the 1980s (MySQL is more recent but was based on SQL Server). They CANNOT change their architecture because it would be suicidal for them to require dumping and reloading today's massive databases for a migration. Only incremental improvements are possible.

Mid 1980s

This is what a $M 1 machine was looking like in the 1980s.

VAX 8600 (high-end) digital

- 32-bit architecture
- Processor, ~10 to 20MHz clock
- 4 to 256 Mb of memory
- I/Os ~10 to 30 Mb/s

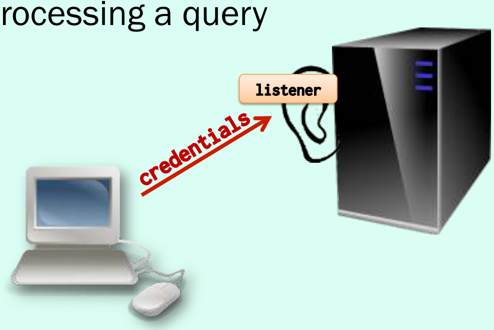
Keep in mind that the big DBMS products were designed for this.

Dumb hardware

Beginning of multi-processor computers

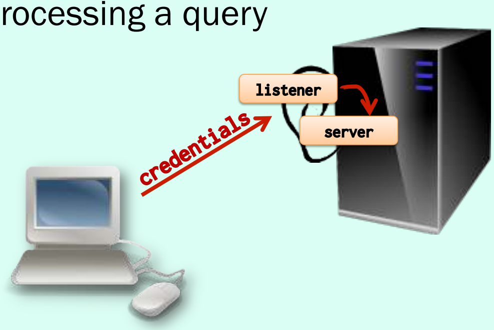
OK, how are we going to process a query? First we must send our credentials to the DBMS server and establish a session.

Processing a query

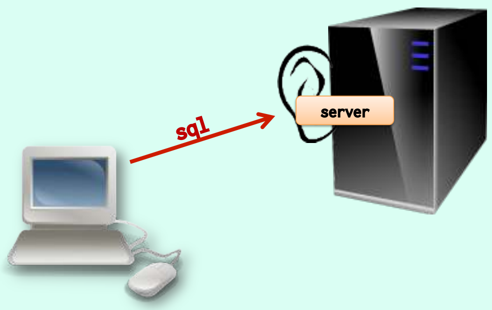


The listener may fork a process that will become our personal server and execute queries for us.

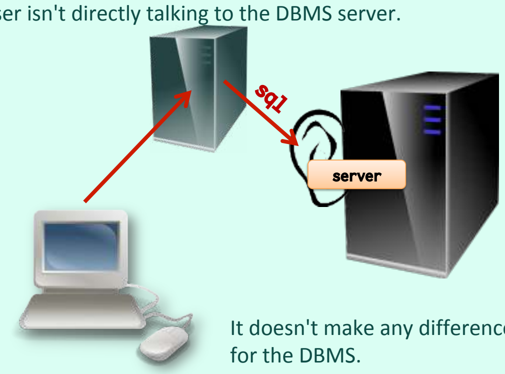
Processing a query



Later, SQL queries will be directly sent to this server. The listener is only here for new connections.



In many cases (HTTP server, application server) the end-user isn't directly talking to the DBMS server.



It doesn't make any difference for the DBMS.

```
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = 'Tim'
  and p.surname = 'Burton'
  and c.credited_as = 'D'
```

Syntax ✓

- Do tables exist? ✓
- Right to access? ✓
- Do columns exist? ✓
- Indexes we can use? ✓
- Best way to access data? ✓

queries on the data dictionary

execute

When the server receives the text of a query, a lot of operations precede the execution of the query proper.

This phase is known as "parsing" and is similar to a dynamic compilation of a query. It's usually pretty CPU-intensive, and making it efficient is crucial.

PARSING

~ dynamic compilation

```
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = 'Tim'
  and p.surname = 'Burton'
  and c.credited_as = 'D'
```

- Syntax ✓
- Do tables exist? ✓
- Right to access? ✓
- Do columns exist? ✓
- Indexes we can use? ✓
- Best way to access data? ✓

One way to improve efficiency is to keep data dictionary information (meta-data) in a shared cache to avoid additional queries.

meta data

Kept in shared memory

```
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = 'Tim'
  and p.surname = 'Burton'
  and c.credited_as = 'D'
```

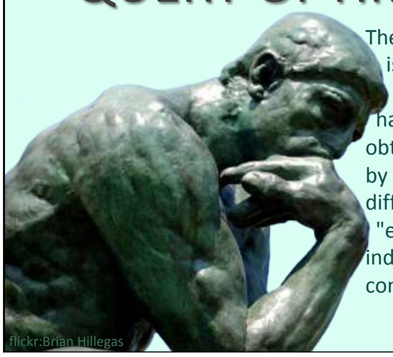
- Syntax ✓
- Do tables exist? ✓
- Right to access? ✓
- Do columns exist? ✓
- Indexes we can use? ✓
- Best way to access data? ✓

Another crucial phase is the one when the optimizer tries to determine the most efficient way to access data.

meta data

Kept in shared memory

QUERY OPTIMIZER



flickr:Brian Hillegas

The query optimizer is a vital element in a DBMS product. You have seen that we can obtain the same result by writing the query in different ways, but that "entry points" and indexes used can be completely different.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
INNER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

TEXTBOOK

The problem is that if SQL queries are simple in textbooks, they are far more complicated in real life and, like in a chess game, you cannot explore all possibilities. Query optimizing is part of the overall response time.

```
SELECT V_RCGLOBAL_V2.SERNOM1, V_RCGLOBAL_V2.EQP_EQPNOM,
       CLNVCDSLAM.CLNNUM, V_RCGLOBAL_V2.CLNNUMDSLAM,
       V_RCGLOBAL_V2.PLQCODE,
       V_RCGLOBAL_V2.EQP_EQPNOM2,
       CLNVCNTUBAS.CLNNUM,
       V_RCGLOBAL_V2.CLNNUMNTUBAS,
       V_RCGLOBAL_V2.CLN[illegible],
       V_RCGLOBAL_V2.CLN[illegible]
FROM   V_RCGLOBAL_V2 V_RCGLOBAL_V2,
    B_CONXLIEN CLNVCDSLAM,
    B_CONXLIEN CLNVCNTUBAS
where  V_RCGLOBAL_V2.VLNIDGENEDSLAM = CLNVCDSLAM.VLNIDSUPPORT
and    V_RCGLOBAL_V2.VLNIDGENENTUBAS = CLNVCNTUBAS.VLNIDSUPPORT
and    CLNVCDSLAM.CLNVTYPE = 'VP'
and    CLNVCNTUBAS.CLNVTYPE = 'VP'
and    V_RCGLOBAL_V2.CLNVTYPEDSLAM = 'VC'
and    V_RCGLOBAL_V2.CLNVTYPENTUBAS = 'VC'
and    ( V_RCGLOBAL_V2.PLQ_PLQCODE = 'SE3')
and    (V_RCGLOBAL_V2.EQP_EQPNOM= 'DSMAD201')
ORDER BY 2,3,4
```

REAL LIFE

Complex view

This is a pretty average query in the corporate world.

Tom Kyte's dance floor analogy

Tom Kyte, one of the rock stars in the Oracle world, used a very good analogy to explain that the best way to run an SQL join depends on the number of rows, in the same way that the best way to pair dancers depends on the affluence on the dance floor.

QUERY OPTIMIZER

- Logical transforms
- Indexes
- Volumes
- Storage
- Hardware performance
- System load
- Settings

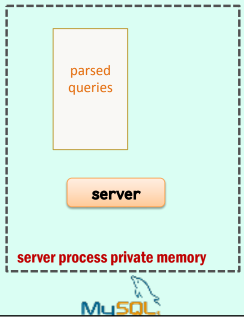
The optimizer has to (or can) take into account a lot of factors.

As a result

PARSING takes time

Let's put it another way: we'd rather not parse exactly the same query many times.

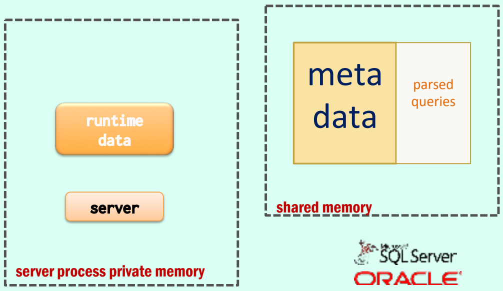
Keep parsed queries in memory



As most applications run exactly the same SQL statements again and again, a DBMS will cache a parsed query for reuse. For MySQL, it will be cached for a session.

For Oracle and SQL Server, cached parsed queries will be shared between sessions.

Keep parsed queries in memory



Query cache management

LRU

Least Recently Used

Of course we cannot hold in cache zillions of parsed queries. We need to manage the cache, and replace queries that haven't been executed in a while with new ones.

```sql
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = 'Tim'
  and p.surname = 'Burton'
  and c.credited_as = 'D'
```

Checksum

We primarily recognize identical queries by computing a text checksum.

+ check tables are same and context identical

```sql
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = 'Howard'
  and p.surname = 'Hawks'
  and c.credited_as = 'D'
```

Basically the same query

DIFFERENT CHECKSUM!

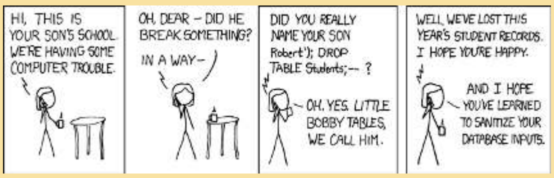
And here we have a problem because this looks like a different query from the preceding one, yet there is no reason why the execution plan should be any different.

```sql
select m.title, m.year_released
from movies m
     inner join credits c
     on c.movieid = m.movieid
     inner join people p
     on p.peopleid = c.peopleid
where p.first_name = :first_name
  and p.surname = :surname
  and c.credited_as = 'D'
```

"variable binding"

This is the main reason why we should use parameters and "bind" them at run time. The text of the query is unchanged whichever name we attach to it. Note that I keep 'D' because, as we have more actors, the plan might be different for them.

(Parenthesis)



http://xkcd.com/327/

Additionally, of course, variable binding protects against SQL injection.

Some issues:

~~select …~~
~~from :my_table~~

~~select …~~
~~from …~~
~~order by :user_choice~~

~~select …~~
~~from …~~
~~where col in (:user_list)~~

And unfortunately you cannot bind a table or column name, nor (which would be quite useful) a whole list of values.

Some issues:

Selectivity?

There is another problem with parameters: the choice of using or not using an index depends mostly on data selectivity. The optimizer will base its decision on the values passed when the query is first parsed. Nothing guarantees that the decision will still be adequate with other parameter values.

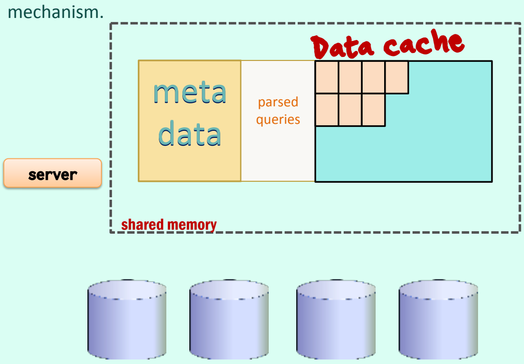
More radical

Cache Query Result

Read Only/Slowly Changing

Some products can go further than caching parsed queries and can even cache results associated with a query and its parameters. It's OK if the database is a read-only database or changes only very slowly (for instance, for a query such as retrieving the last articles in a blog).

Data is also cached. Tables (and index) are divided into pages (typically 8K, 16K or 32K) which are brought in memory through a paging mechanism.



Now what happens when data is changed?

SEVERAL problems

Atomicity	Transactions (rollback)
Durability	Persistence
	Consistency
Isolation	Concurrency

32

The ACID acronym predates relational database and symbolizes everything that is desirable to have with a transactional databases.

ACID

The dogma that every database management system should be "ACID" has been questioned by the NOSQL movement (more about this later)

33

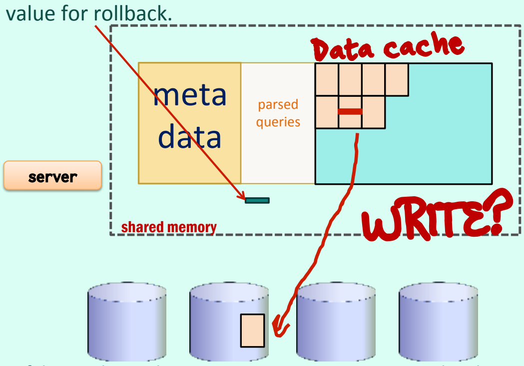
Atomicity — Atomicity is related to a transaction: seen as a single unit.

Consistency — Data should always be seen as consistent (constraints)

Isolation — When one user changes data, it should not make it inconsistent to others.

Durability — When something is saved to a database, it should become persistent, whatever happens.

For instance, before a change we must save the previous value for rollback.



If data is changed in memory, are we going to immediately write it to file for durability?

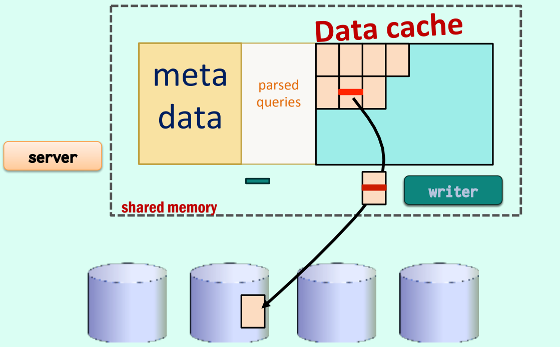
35

In fact, this would be a massive performance concern (especially in the 1980s, when there was no cache in disk units). With a massive update (massive load, for instance) we would just keep on waiting for data to be written to disk, and it would become a massive bottleneck.

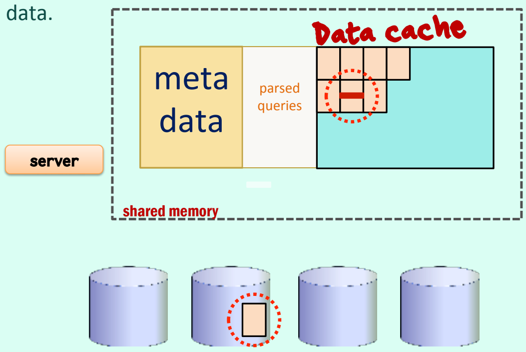
MASSIVE UPDATE — I/O STORM

36

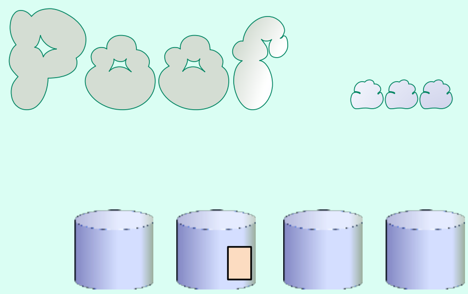
What will happen with a product such as Oracle is that a dedicated writer process will, ASYNCHRONOUSLY, write "dirty" pages to files from time to time.



So, at any given time, memory (representing the true database state) and files will not exactly contain the same data.



What happens if the system crashes and everything that was in memory simply vanishes?



39

Not committed?

If the change wasn't committed, then losing it is just what we would expect: a crash should rollback any on-going transaction.

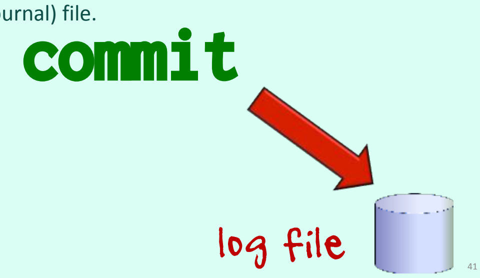
Committed?

If the change was committed it's a different story. The D in ACID means that we expect any committed change to survive a system crash.

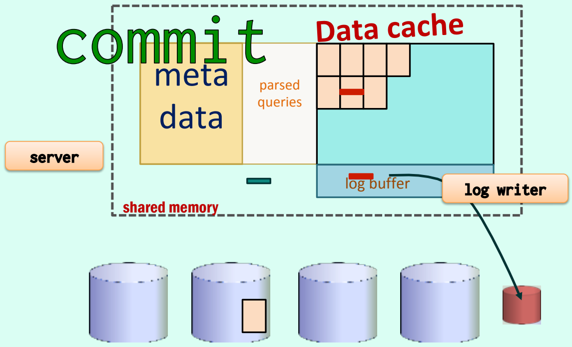
DURABILITY

40

Practically, it means that every COMMIT should result into a SYNCHRONOUS file write: the commit call only returns when changes are safely written to disk. We aren't going to update all the blocks affected by the transaction, which would take too much time, but only save the new values to an additional log (journal) file.

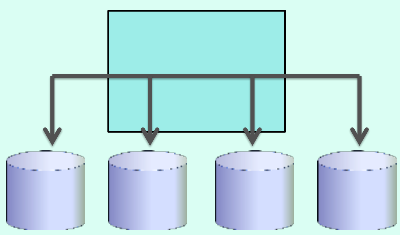


In Oracle, every pending change (new values, as opposed to old values kept for rollback) will also be saved to a "log buffer" that will be flushed to a log file on commit (or when dangerously full)



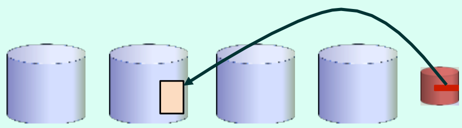
When you shutdown the database cleanly, all the buffers are flushed to files, which represent the last state in memory.

Clean shutdown :
Flush buffers to files



When restarting after a crash, the log is scanned for all committed changes, and they are applied if need be to the file blocks, until the files contain the exact image of the database at the last COMMIT before the crash.

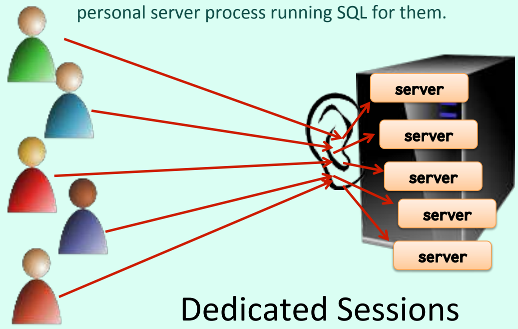
RESTART AFTER CRASH



What about several sessions?

So far we have, kind of selfishly, considered only the interaction of our session with the database. One of the reasons why databases were created was to share data, which of course implies many concurrent sessions and many interesting problems.

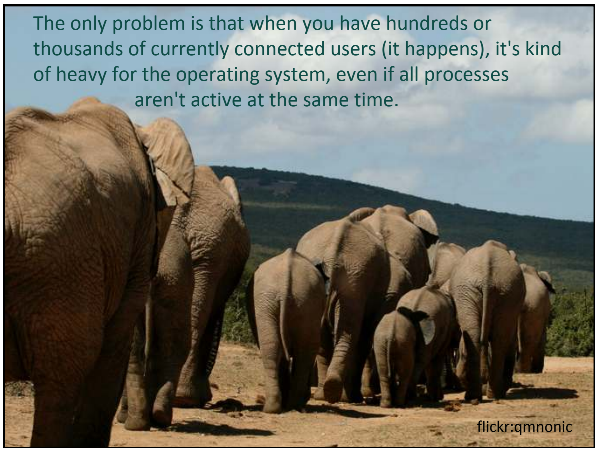
First of all, we may assign to all connected users their own personal server process running SQL for them.



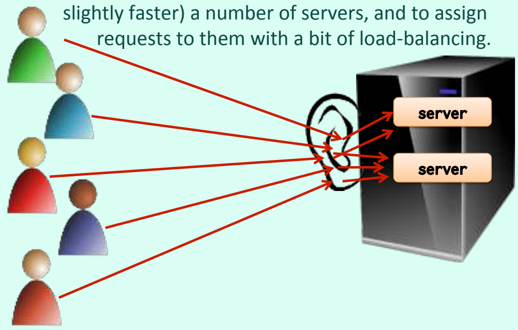
Dedicated Sessions

It works well, it has been the Oracle standard for years.

The only problem is that when you have hundreds or thousands of currently connected users (it happens), it's kind of heavy for the operating system, even if all processes aren't active at the same time.

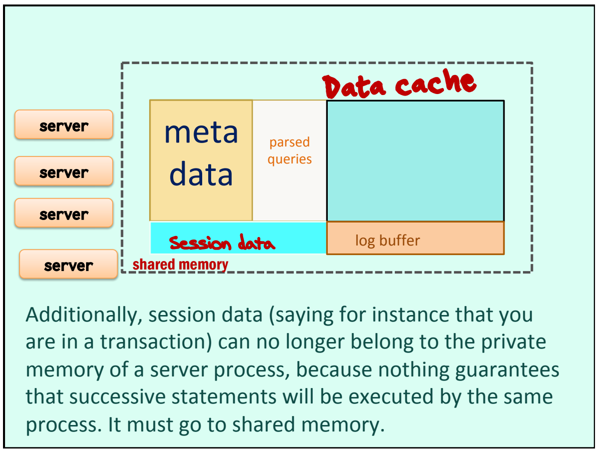


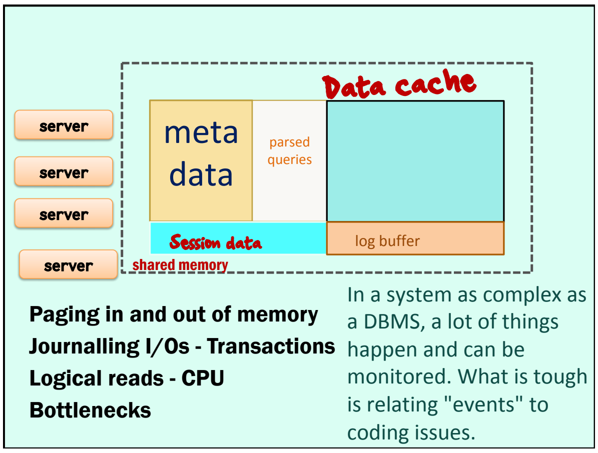
An alternative is to pre-start (which makes connection slightly faster) a number of servers, and to assign requests to them with a bit of load-balancing.

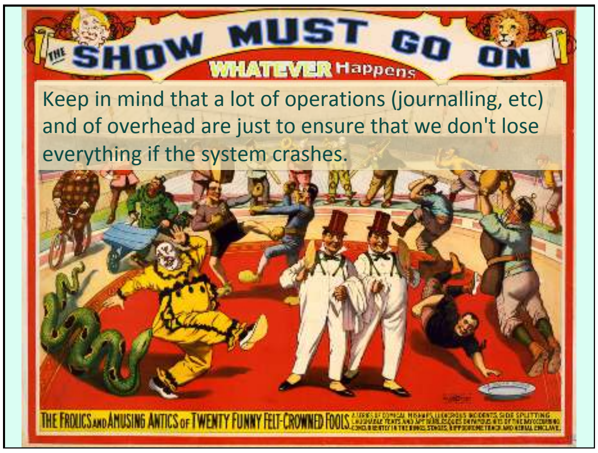


Session Pooling


The problem then is that if your query is unlucky it may end up in the wrong queue.
flickr: oatsie40


Data cache
server
server
server
server
meta data
parsed queries
Session data
log buffer
shared memory
Additionally, session data (saying for instance that you are in a transaction) can no longer belong to the private memory of a server process, because nothing guarantees that successive statements will be executed by the same process. It must go to shared memory.


Data cache
server
server
server
server
meta data
parsed queries
Session data
log buffer
shared memory
Paging in and out of memory
Journalling I/Os - Transactions
Logical reads - CPU
Bottlenecks
In a system as complex as a DBMS, a lot of things happen and can be monitored. What is tough is relating "events" to coding issues.


THE SHOW MUST GO ON
WHATEVER Happens
Keep in mind that a lot of operations (journalling, etc) and of overhead are just to ensure that we don't lose everything if the system crashes.
THE FROLICS AND AMUSING ANTICS OF TWENTY FUNNY FELT-CROWNED FOOLS

The real fun, though, begins when concurrent sessions start modifying data required by several of them.

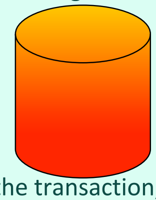
What about a session querying data being changed by another session?

ISOLATION

53

Data Change

The main problem is that a transaction can last "a certain time". You may need to run a lot of code before deciding on commit or rollback.



Begin Transaction

Commit

During the transaction, the database will be in a kind of transient state of which you cannot say whether it will become permanent.

CONSISTENT STATE

CONSISTENT STATE

ATOMICITY

4 levels for ISOLATION

~~dirty reads~~

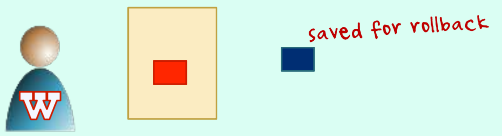
read committed

repeatable read

serialization

The SQL standard defines four isolation levels, from no isolation at all to paranoid. Some products let you set it, others impose it.

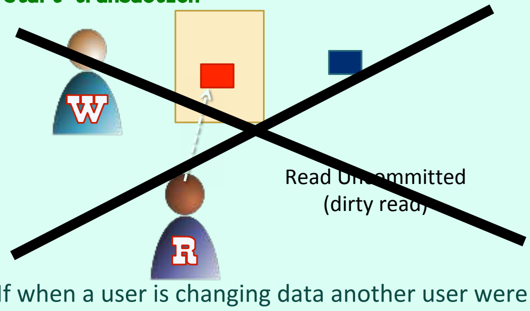
start transaction



commit

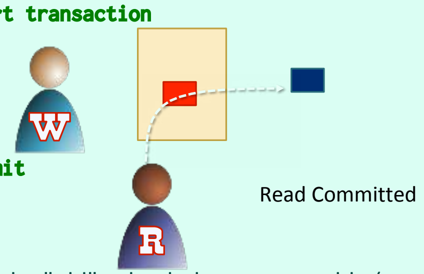
Let's see once again how a transaction works. You start it. Before applying any change the DBMS saves the value from before your transaction, in case you'd want to rollback. Then you commit (or rollback) and the value previously saved for rollback (in memory or on file) can be marked as disposable.

start transaction



If when a user is changing data another user were able to read the new value before being sure that there will be no rollback, we would run into problems pretty soon.

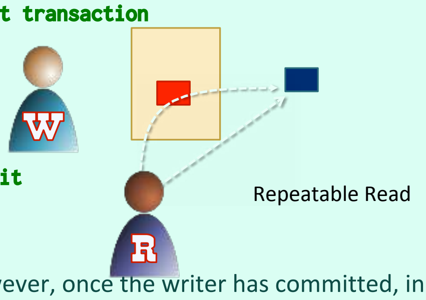
start transaction



commit

But the "old" value belongs to a stable (committed) state of the database. What most DBMS products will do is that they will "serve" to the reader this value, at least known to be consistent and to have been once "official current value".

start transaction



commit

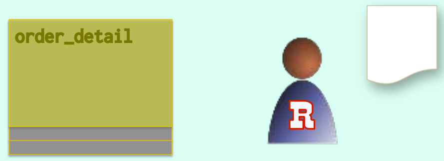
However, once the writer has committed, in the read committed model whe should read the new, now official, value. It may be inconsistent with a previous read and we may favor consistency for the reader over timeliness.

In practice, the problem is more a problem of data consistency of foreign keys when we are scanning big related tables. A single SELECT will usually be consistent (a product such as Oracle ignores any change having happened since the start of the SELECT, even if it was committed). But if we SELECT twice (two different queries) from two tables with an FK relationship, changes that may have happened (and have been committed) between the time when we started reading the first table and the time when we were reading the second table may lead to problems such as orphaned rows.

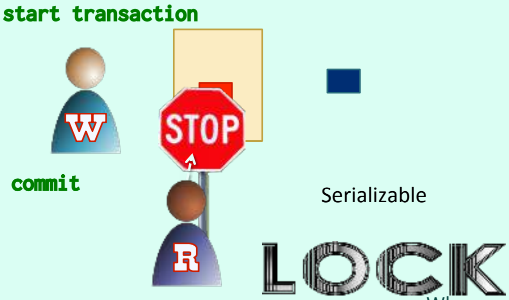
Repeatable read Point-in-time read

order

If I haven't read an uncommitted order when reading ORDER, then I should ignore rows refering to this order in ORDER_DETAIL even if they have been committed by the time I reach them.



start transaction



The last level is to block readers when data is being modified. Guaranteed to be consistent but bad for performances.

Table-level

Row-level

When you have two writers you cannot do otherwise than locking.

BACKUP ISSUES

All of this, the fact that files and memory may not been completely in synch, plus the fact that people may be modifying the data, leads to most interesting issues when you want to backup a database without stopping it.

Backing up something consistent?

Files image of memory?

No file being written while copied?

Need to prevent changes?

- Communication Manager
- Query Processor
- Transaction & Buffer Manager
- Storage Manager
- Process Manager

It all explains why an enterprise-grade DBMS is a complicated beast, with a lot of subcomponents taking some special tasks in charge, embodied by several background processes that must be coordinated.

Loosely inspired by Hellerstein/Stonebraker/Hamilton *Architecture of a Database System*

STARTUP

Something as "simple" as starting a database requires multiple steps, checking that the database was cleanly shut down, and starting a recovery process if it wasn't.

Read configuration file

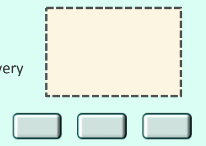
Allocate shared memory

Start processes

Identify database files

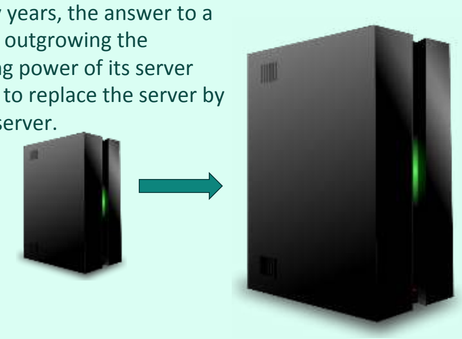
Check for recovery

Accept connections



SCALING UP

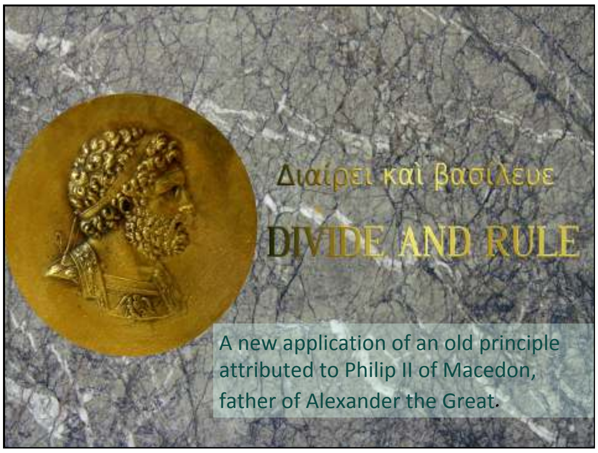
For many years, the answer to a database outgrowing the processing power of its server has been to replace the server by a bigger server.



Having a single server is taking a risk if it breaks, and additionally migrating to another server is disruptive.

SCALING OUT

This is why people quickly thought of an alternative, adding more servers and making them share the load.

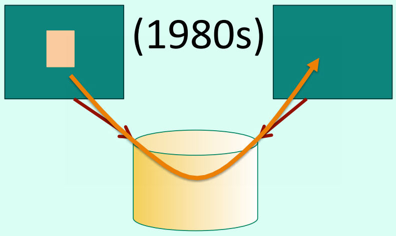


Distributed Systems

But here everything becomes more complicated all of a sudden.

In the 1980s, Oracle tried connecting multiple servers to a single database. Complete disaster when the two servers wanted to modify the same data (or simply when one wanted to see what had just been modified by the other), data had to transit via the files.

Oracle Parallel Server (1980s)



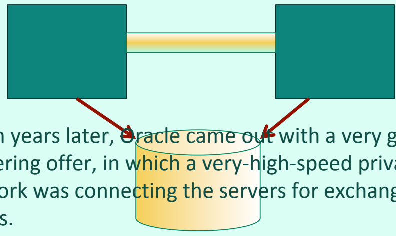
In 1994 Oracle bought from Digital Equipement (which they swallowed whole later), RDB, a respected relational database management system that only worked on Digital systems (SW/HW).

1994 RDB acquired by Oracle

One of the strengths of RDB was working with clusters of machine; it was the only product doing it well. All of a sudden, Oracle Development gained strong competencies in this area.

Oracle RAC (2001)

Real Application Clusters



Seven years later, Oracle came out with a very good clustering offer, in which a very-high-speed private network was connecting the servers for exchanging data blocks.

Coordinator node

Node "owns" blocks

Owner can change

RAC implements a complicated "cache-fusion" algorithm in which a coordinator nodes always knows who is working on what , and blocks can be exchanged quickly. Of course it works better when the workload is fairly different for all servers, but when different servers can work in parallel on disjoint data, it can be extremely efficient and it handles conflicts well.

Neil Gunther's

Universal Law of Computational Scalability

www.perfdynamics.com

Relative capacity C(*N*) of a computational platform:

$$C(N) = \frac{N}{1 + \alpha\,(N-1) + \beta\, N(N-1)}$$

$0 <= \alpha, \beta < 1$

α Level of contention

β Coherency delay (latency for data to become consistent)

The problem with clustering is that it follows a law of diminishing returns: adding a second server will less than double your capacity, and in practice people have clusters of 2, 3 or 4 machines at most (Neil Gunther is a famous Australian consultant/academic specializing on performance)