6.830 2009 Lecture 16: Parallel Databases

today's topic: **parallel databases**
how to get more performance than is possible w/ one computer

--------- MOTIVATION ----------
why might one CPU/one disk not yield enough performance?
can only scan big tables at 50 MB/sec
can only read random rows, or probe indices, 100 times/second
CPU needed to push data around, hash, lock, &c

who might have workloads too big for one CPU/disk?
data-warehouse giant scans/joins/aggregations
ebay-style OLTP, maybe 100s of updates/sec, 1000s of queries/sec

--------- 3 STRATEGIES ----------

solution 1: **SMP hardware**
[diagram: CPUs, sys bus, RAM, many disks]
each client gets a thread
shared memory for lock table, buffer cache

SMP is very convenient for the software
if you already used threads and latches, you're practically there
mysql, postgresql, simpledb
any transaction can use any disk
see each others' updates and locks in shared mem

what kinds of workloads will benefit from SMP hardware?
OLTP if many concurrent clients, not many locking conflicts
keep different disks and different CPUs busy
single big join?
if tables striped over disks
if we write specialized parallel join algorithms

performance goals:
what might we expect from our money?
e.g. if we spend 10x as much money on a fancy SMP server
1. **speedup**: make existing workload take 1/10th as much time
usually no speedup for single OLTP transaction
can get speedup for big joins by splitting work over servers
2. **scaleup**: run 10x larger job in same time
could expect to handle more OLTP xactions concurrently on more servers
can join bigger tables in same time
we really want *linear* speedup and/or scaleup
don't want to pay 10x money, get only 2x improvement

why isn't SMP hardware the final answer?
shared memory and I/O very hard for more than a few dozen CPUs
want any CPU to r/w any disk buffer or lock
want any CPU to r/w any disk at 50 MB/sec
hard to build the interconnect for this!
16 CPU SMP currently cheap (i.e. about 16x one CPU)
256 CPU SMP (really NUMA) possible but MUCH more expensive per CPU

solution 2: **shared disk**
get rid of shared memory!
buy network-attached disks -- a "**SAN**"
and lots of independent DB servers attached to the SAN
network allows any DB server to r/w any disk
you only have to send disk I/O over interconnect
not CPU memory traffic as well, as in SMP

shared disk pros:
cheaper per CPU than SMP for many CPUs
no expensive interconnect for shared memory
can use commodity server boxes
s/w partially the same as for single server
any CPU can r/w any disk
can act like solitary srvr w/ many disks

shared disk cons:
the **network may be expensive** -- 50 MB/sec per disk?
the **disks may be expensive -- need special SAN interface**

**** how to deal with locks?**
server 1 and server 2 want to update the same row
central lock manager?
stripe locks over servers?
**** how to deal with dirty buffers?**
server 1 writes a page
server 2 wants to read that page
can't read its own cached copy
can't read from the disk!
server 1 must invalidate or update other cached copies
server 2 must know to ask server 1 for latest data
lots of inter-server lock and buffer chatter!
in practice **cannot scale shared disk beyond dozens of CPUs**
only somewhat better than SMP
(Oracle promises infinite scalability, but anecdotal evidence
seems to say typical 2 nodes no more than 6-8 nodes seen in
practice, and often used mostly for redundancy)

how to do better than SMP or shared disk?
the **real problem is not the hardware, but the DB software's**
desire to r/w any disk from any CPU

solution 3: **shared nothing**
DON'T let s/w r/w any disk from any CPU!
each CPU has a disk (or a few disks)
each CPU only allowed to directly use data on its own disks
so: commodity server/disk boxes, LAN connecting them, clever s/w
all the cool kids are playing this game
vertica, DB2 parallel, netezza, teradata, ad-hoc setups at facebook &c

**** shared nothing questions:**
how to partition data over servers
how to run big join and aggregate queries
how to run OLTP queries
how to ensure load balance
how to deal with locks and recovery

---------- OLTP ----------

how to partition data for OLTP
suppose you are ebay

item(item_id, title, &c)
bid(bid_id, amt, time, user_id, item_id)

SELECT * FROM bid, item WHERE item_id = 3; to display an item page w/ bids
how about this partition plan?
items on server 1, bids on server 2
how about this partition?
even item_ids on server 1, odd on server 2
even bid_ids on server 1, odd on server 2
a good partition plan:
assign item rows to servers with hash(item_id)
assign bid rows to servers with hash(item_id) (same hash function)
then displaying item page only hits one server
can display many diff item pages in parallel w/ many servers
and adding a bid hits only one server / one disk arm
suppose you want to display some user info for each bid too? e.g. user name
can't partition users by item_id!
hash by user_id, hit on server per user
or duplicate some user info in bid table
usually can't find one partition plan that makes every query go to one server!
so duplicate data, or cache, or pre-compute periodically, or buy lots of servers

suppose we spend money to increase from 10 to 100 servers
are we going to see speedup of one query?
are we going to be able to process a larger # of queries in one second?

when might we get linear 10x increase in performance?

what might prevent us from getting 10x?
bad load balance! or "**skew**"
1 server with lots of work, 99 servers idle, no speedup at all
some items much more popular than others
accidentally put too many popular items on same server
or maybe **partition plan forces us to hit more servers** as # servers increases
e.g. if bids for an item were spread over many servers

what are the options for partitioning?
usually "**horizontal**": partition the rows (not the columns)
each server responsible for some set of rows
1. hash on some column to get server #
2. assign ranges of keys in some column to servers
3. round-robin inserts (essentially random)
you can also throw in "**replication**" for read-mostly data, that
are needed everywhere

when to use each kind of partitioning?
load balance? (data vs accesses)
round-robin is perfect
hash might work well, might not

range might or might not
associative searches want all of same key together, i.e. bids by item_id
hash and range work well
round-robin does not
sequential access, e.g. all sales between May 10 and May 15
hash and round-robin: must talk to all servers
range: maybe talk to just one server
programmer convenience?
range requires thought, maybe re-partitioning as data grows
hash and round-robin are automatic

---------- OLAP ----------

what about big scan/filter/join/aggregate, for data warehousing?
on shared-nothing parallel DB

Sam already anticipate a bit of this... let's see more...

SELECT name WHERE eyes = 'blue' and hair = 'red';
if partitioned on eyes or hair?
otherwise?
(advantage in touching many machines? depends on intra-query parallelism)

select type, avg(price) ... group by type;
if partitioned by type?
otherwise?
local partial partition
report type, sum, n to querying machine
or if many types, machine chosen by hash(type)

select ... from big, small where big.x = small.y;
equi-join big table against small table
send copy of small to each server
do a hash join on each server against its partition of big table

select ... from big1, big2 where big1.x = big2.y;
equi-join two big tables
if big1 partitioned on x, and big2 partitioned on y, join locally

Otherwise? I make sure I will get that property by re-partitioning tables!!
let's re-partition both, on big1.x and big2.y
scan big1, send each row to server[hash(x)]
to temporary storage: memory or disk
scan big2, send each row to server[hash(y)]
same hash!
we sent each entire table across the LAN
but now every pair of rows that might join are on same server
each server now separately uses a standard join algorithm

Even better:
- I could apply filters first..
- I could use semi-join: send only the join column from one table, perform the semi-join, send

the
results to the other tables, and finish the join... this might move less data.

big tree of joins?
re-partition after each join, send over network

There is a non trivial query optimization problem here:
- multi-site execution (computation might need to change)
- cost of network vs cpu vs disk

it's one more dimension than local query optimization...

how fast does the network have to be?
each host needs about as fast as its disk
entire LAN needs n*disk total throughput
but only 1*disk to any one host
can we buy such LANs?
you can get 10 gbit host interfaces (i.e. 10x faster than one disk)
you can get 10 gbit switches with modest #s of ports
how to build a switch that can handle n*host total traffic?
[4x4 crossbar]
but **you can't build very big+fast crossbars**, maybe 16 ports
if more? multiple levels, needs to be fat in the middle
can't just have one switch in the middle unless it has faster links than hosts
usually a low limit to how fast you can make the links
so instead use more links
C1 C2 C3 C4
E1 E2 E3 E4
each Ex has four hosts
each Ex has link to each of Cx
need to spread load from each Ex over all Cxs

suppose we spend money to increase from 10 to 100 servers
are we going to see speedup of one big query?
are we going to be able to join larger tables in same time?

are we going to get a 10x increase in performance?
when might we get 10x?
what might prevent us from getting 10x?

what about indices?
might be able to get away with local indices
each server just indexes its own partition
and only uses local indices
might help with e.g. filtering scans
what if you are looking for one item?
1. ask all servers to look up in local index
2. arrange to only look for keys in partition columns, then only ask one server
3. painfully maintain global index

how to cope with locking in a shared-nothing DB?
do we need a global lock server?
if a row/page is read/written, initially done on home server
so each server can keep lock table for its own data
don't need to lock another server's data

no lock server required

--------- CLOSURE ----------

let's step back for a minute

parallel programming is usually viewed as very difficult

hard to reason about parallel algorithms
hard to get good speedup
hard to avoid races, maintain correctness

has turned out to be relatively easy to use parallel DBs -- why?

1. **relational model abstracts from physical layout**
   can partition &c w/o changing apps
2. **transactions and locking take care of parallel correctness**
   once you're willing to program in transaction model
3. **applications often have lots of inherent parallelism**
   OLTP queries on different data -- due to many independent users
   scan &c on different parts of same table

6.830 / 6.814 Database Systems

Fall 2010