

CS290f - Lecture 18 Scaling the database

Scalable Internet Services, Fall 2006

Thorsten von Eicken
Department of Computer Science
University of California at Santa Barbara

*Some database material from Cal Henderson's
"Building Scalable Web Sites"*

DB performance bottlenecks

- **Disk seeks**
 - time for the disk to find a piece of data
 - improves slowly with new disks, very hard to optimize for a single table
 - distribute the data onto more than one disk
- **Disk reading and writing**
 - bandwidth/throughput to/from disk
 - more disks (read/write in parallel)
- **CPU cycles**
 - process data in memory
 - sorting/scanning large tables
 - keep tables small, keep operations simple
 - trade hard-to-scale DB cpu with easy-to-scale app server cpu
- **Memory size**
 - working set exceeds memory size
 - disk write-mostly workload becomes read-mostly
 - buy more memory, partition database, archive old data

DB performance: find faster

- **Indexes**
 - Contains subset of table columns and pointer to full record
 - Optimized for fast searching by specific column(s)
 - ◆ typically using balanced tree data structure
 - InnoDB tables are stored in primary key order: no index needed for that
 - Unique key indexes vs. non-unique indexes
 - Use EXPLAIN statement to see whether/how operation uses indexes
 - **Downside:** keeping indexes up to date
 - ◆ Easy to overlook and can negate impact of index
 - ◆ Ask: is the table mostly read or mostly written?
- **Increasing selectivity of queries**
 - Select ... where type = 'employee'
 - ◆ ... probably selects large fraction of table => low selectivity
 - Select ... where owner = current_user
 - ◆ ... probably selects small fraction of table => high selectivity
 - Check query execution plan to ensure that high selectivity conditions get evaluated first

DB performance: less data

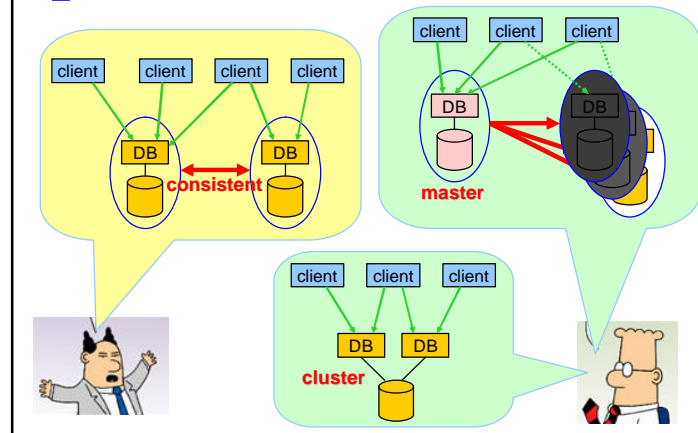
- **Reduce number of tables joined**
 - In complex schemas there are often multiple join options
 - Optimize for smaller tables, better indexes, higher selectivity
- **Reducing size of tables**
 - Delete or archive old data
 - Use table partitioning (not available in mysql)
 - ◆ Internally divide table into N "sub-tables" based on condition
 - ◆ Choose condition such that queries run in single partition
 - ◆ Dropping a partition is fast
 - ? example: partition by month to drop/archive old data
- **Increasing # of disk spindles**
 - Separating log disks from data disks (usefulness depends on RAID system)

DB performance: tweak schema

Denormalizing data

- Duplicating frequently needed information
 - DVD ratings
 - ◆ Compute average over all individual ratings vs. store average
- Book reviews
 - ◆ Store reviewer name in review (in addition to key to reviewer record)
- Downsides: inconsistent data
 - ◆ Put code into model or other DB layer
 - ◆ Put code into database triggers and stored procedures
 - ◆ Run periodic integrity checks
 - ◆ Consider adding a "data management" tier to your architecture

Database Replication



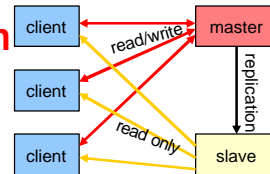
MySQL Replication

logical replication

- Slave connects to master
- Fetches transaction log
- Replays transactions locally

problems

- Slave often slower than master
 - ◆ Single thread replays log!
- Slave may become inconsistent
 - ◆ Non-deterministic updates
 - ? e.g. "insert into t select from u", where t has auto-increment key
 - ◆ Be sure to have integrity constraints!
- Slave lags master => not consistent with master!



Replication trees

Replication to multiple slaves!

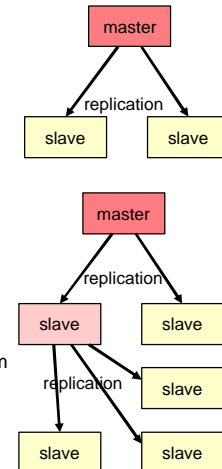
- Scales read bandwidth
- Does not scale write bandwidth

Tree replication

- Saves threads, connections, and bandwidth on master
- Increases latency
- Useful for remote datacenters

More replication tricks

- Exclude databases & tables by name from replication
- Replicate into different table type
 - ◆ InnoDB -> MyISAM for full-text search
- Add special indexes on replicas



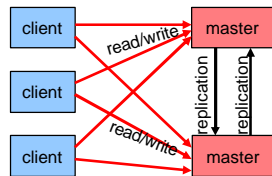
Master-master replication

“Works”

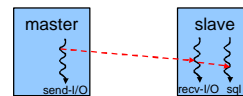
- The mechanics work
- Best: write tables or DB portions to specific server

Doesn't work

- Conflicts bound to occur...
- No consistent view of entire DB



Replication lag



Time lag between master and slave updates

Monitoring

- Seconds_Behind_Master: time between I/O thread on slave receiving log entry and SQL thread executing entry
- Alert if lag becomes “too large”

```
mysql> SHOW SLAVE STATUS\G
***** 1. row *****
Slave_IO_State: Waiting for master
to send event
Master_Host: localhost
Master_User: root
Master_Port: 3306
Connect_Retry: 3
Master_Log_File: gbichot-bin.005
Read_Master_Log_Pos: 79
Relay_Log_File: gbichot-relay-bin.005
Relay_Log_Pos: 548
Relay_Master_Log_File: gbichot-bin.005
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Last_Errno: 0
Last_Error:
Skip_Counter: 0
Exec_Master_Log_Pos: 79
Relay_Log_Space: 552
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Seconds_Behind_Master: 8
```

PostgreSQL replication

Multiple replication packages

- No clear standard solution

PGCluster

- Multi-master no delay synchronous replication for load sharing or HA

Slony-I

- Master to multi-slave cascading asynchronous replication using triggers
- Designed for data centers to keep hot-backups of the data servers
- Does not support hot-failover but does support planned switch over

pgpool

- CConnection pooling front end with synchronous replication
- Supports up two PostgreSQL servers in synchronous replication
- Data must be manually synchronized after a failure.

pg_comparator

- Pg_comparator is effectively rsync for PostgreSQL
- Only one table can be synchronized at a time
- Nice package for moving development code into test, or test to live, or to recover from a Slony failure

Pgcluster

Replication

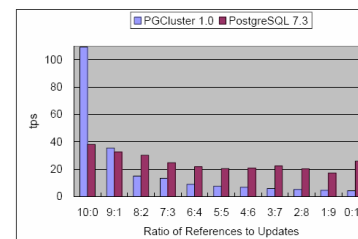
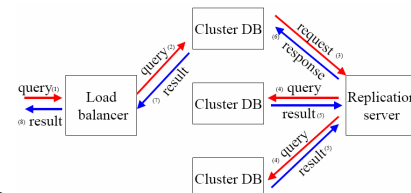
- All nodes perform updates

Synchronous

- Gated by slowest node

Overall status

- Active development
- ... by one guy in Japan



Oracle HA (High Availability)

Shared disk subsystem

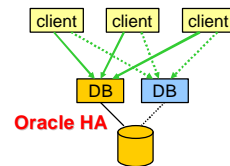
- Both Oracle nodes can mount disks
- Use SCSI locks to arbitrate access
- Only one node mounts disks at a time

Fail-over

- Notice that master node is down
- Acquire SCSI locks
- Mount disks/filesystems
- Recover DB from redo logs
- Clients fail-over

Downsides

- Does not scale performance
- Only single copy of data
- Will it work...?



13

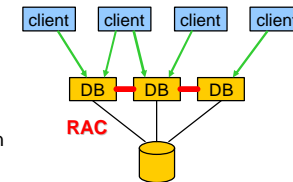
Oracle RAC

"Real Application Cluster"

- Shared disks as in HA
- Distributed buffer manager
- Distributed lock manager
- Lots of node-node communication

Pros & cons

- Scales performance
- Redundant DB nodes
- Communication overhead
- LAN only
- Single copy of data



14

"Vertical partitioning"

Partition by entire tables

- (terminology non-standardized)
- "Clusters of tables"

Can't transact on tables in multiple partitions

- Need to ensure application is aware of the partitioning
- Tension between scalability and ease of coding

Often can also partition application

- Use web services APIs to glue partitions back together

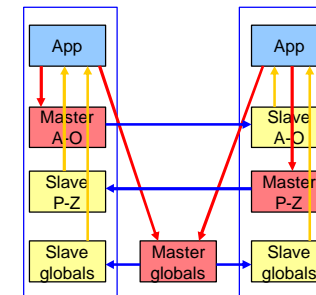


15

By-user partitioning

Idea

- For many large web sites each user's data is largely independent of other users' data
- Partition database by user
- Global data resides in separate database
- All partitions may have read-access to all other partitions

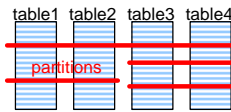


16

"Horizontal partitioning"

Other names:

- "Federation" of databases
- Database "shards" or "cells"



Idea

- Divide tables into chunks that can be distributed across many machines
- To scale, increase the number of chunks
 - ◆ Presumably there are more chunks, so the chunk size may remain constant
- Oracle RAC does this

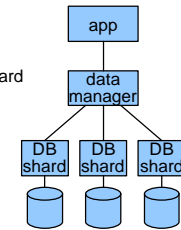
17

Implementing Shards

General case horribly complicated...

Alternative

- Organize data such that joins operate within one shard
- Replicate data (denormalize) when shard structure does not suit needs
- Add "data manager" layer to abstract shards



Denorm example

- Each user has a blog
 - ◆ user_id => shard
- Each blog entry has tags
 - ◆ Entries stored in user's shard
- Display related blog entries (of other users)
 - ◆ Naïve: search for tags in each shard
 - ◆ Better: create new table
 - ? map tag -> <shard_id, blog_entry_id>
 - ◆ Partition new table by tag_id

18

Summary

In the end, the bottleneck will be in the ACID

- Concurrency control (Consistency & Isolation)
- Logging/persistence (Atomicity & Durability)

... and the replication

- Synchronization & communication

In the end, this will mean

- Time, headaches, programmers, DBAs, hardware, \$\$\$, ...

Putting the bottleneck in the DB

- Well understood field, lots of solutions, ...
- Reduce problem to subset of data that matters
- Expensive, complex, slow

Putting the bottleneck into distributed memory

- Potentially faster, thanks to no-SQL and no-disk
- Less well standardized, often performance at expense of semantics
- Still need a DB for persistence: often end up with two big problems

19