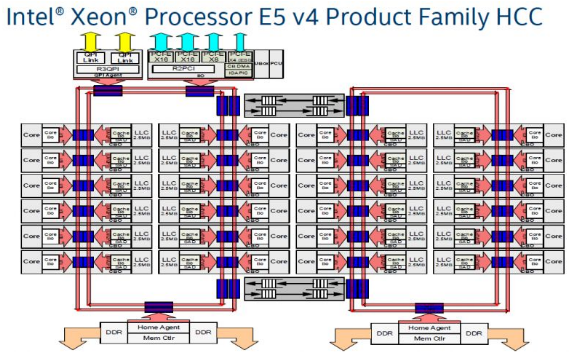
Challenges in Thread-per-core Implementations

Avi Kivity (@AviKivity)

NGCC, Sep 2017

Background



- Ever-increasing core counts
  - Intel: 28c56t per socket
  - AMD: 32c64t per socket
  - Qualcomm aarch64: 48c per socket
  - IBM POWER9: 24c96t per socket
  - Cloud: AWS: i3.16xlarge, 2s32c64t
- Utilizing high core count machines is increasingly harder
  - Lock contention / coherency costs
  - NUMA/NUCA optimization
  - Asymmetric architecture -> over- or under- utilization
- But desirable
  - Increase MTBF
  - Reduce management burden
  - Less space / switch ports / etc.
  - High-density storage
  - Reduce cost

Problems with asymmetric architecture

- Too few active threads - not all cores are utilized
  - Low throughput but machine is underloaded
- Too many active threads - contention
  - Runnable threads wait for a core, adding latency
  - Threads move from core to core, increasing coherency costs
  - Lock contention

- But architecture *is* symmetric:
  - All nodes have the same roles
  - Data partitioned by token
  - Good partitioning required for inter-node scaling

Thread-per-core

- Partition data among logical cores* in a node in the same way that data is partitioned among nodes in a cluster; call this a *shard*
- Assign one thread to a logical core/shard
    - Never have too many or too few threads
    - No lock contention - threads don’t share data
        - If done well, no locks at all
    - Good distribution a given
    - Go home early
- Everything except data is replicated on shards
    - Metadata, schema
    - Cluster topology
- Per-shard mutable metadata
    - Cache LRU
    - Statistics

Initial approach

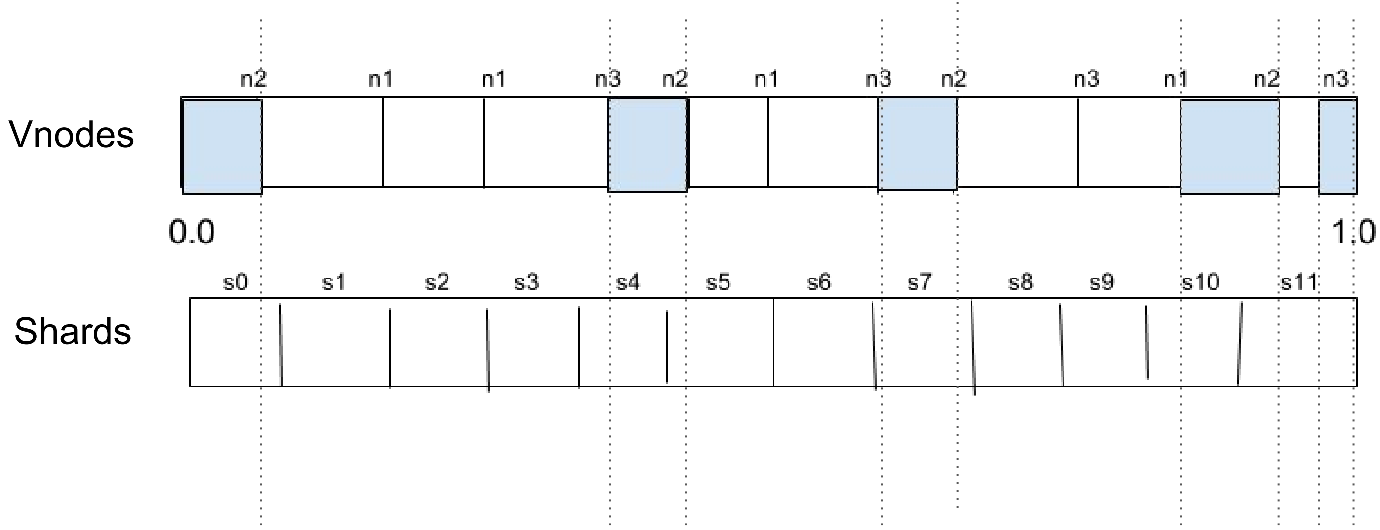
- Divide token space equally among *shards*:

  shard = floor(nr_shards * (token - min_token) / (max_token - min_token))

- Message receiver on replica becomes “replica coordinator”
  - Single-partition operations are forwarded to “replica shard” via Local Procedure Call (LPC)
  - Scans require multiple LPCs over affected shards
- Each shard owns set of private sstables

Aliasing

When number of vnodes “is similar” to the number of shards, some shards don’t get any data



Aliasing (cont'd)

- Aliasing can be eliminated with much larger number of Vnodes or Shards
- Vnodes are fixed for compatibility reasons, so…

  normalized_token = (token - min_token) / (max_token - min_token)

  shard = floor(normalized_token * 4096 * nr_shards) % nr_shards

- Each shard gets 4096 equally distributed token ranges
- Have (4096 * nr_shards) ranges per node (Snodes)
- All cores now busy! http://github.com/avikivity/shardsim
- (drawing omitted)

Handling small tables

- Fix aliasing -> timeouts during initialization
    - Full scan of a table -> (4096 * nr_shards) LPCs
    - Kills even mid-size nodes
- Adjust behavior of replica-side coordinator during range scans
    - Start by assuming a large table and scan one Snode
    - If we reach the end of the Snode, add more Snodes
        - Exponentially: 1, 2, 4, 8, …
        - Results from different shards are concatenated
    - Eventually, we have two or more Snodes from a single shard per iteration
        - Results now have to be merge-sorted
    - Extra complications from Thrift wraparound queries
- Reduces small-table scan times to a reasonable level

Hundreds of thousands of ranges

- Range proliferation means that a lot of code needs to work in terms of range iterators
    - Or latency goes to hell

Persistent state for sharding

- With naive algorithm, min/max partition in sstable tells you which shard it belongs to
- With new algorithm, need new sstable component to store disjoint partition ranges
  - During startup, examine component to see which shards this sstable intersects with

Resharding

- Goal is to have each sstable serve one shard, but cannot always achieve
  - Can restart server with different number of shards
  - `nodetool refresh`
  - Algorithm changes (naive -> new)
- SSTable will start shared
- Resharding job (similar to `nodetool cleanup`) will automatically split it among shards

Resharding (cont'd)

- Naive approach to sharing sstables (open each sstable independently on each shard) fails:
  - Multiple copies of CompressionInfo/Summary/Filter -> OOM
  - Multiple file handles -> out of file handles
  - Very slow startup -> systemd kills
  - 48-shard to 64-shard nodes very typical
- Adjust sstable implementation to allow true sharing of large const data and fds, duplicate everything else
- "Guess" correct shard and open sstable on that shard
  - If guessed wrong, share the sstable with relevant shards

Resharding (cont'd)

- Naive resharding job based on compaction
  - Read one sstable
  - Throw away data not relevant to shard
  - Write one sstable
- Problems
  - Each shard reads shared sstables - huge read amplification
  - Lots of tiny sstables generated (looking at you LCS)
    - Out of file descriptors (again)
    - Horrible read performance
    - Huge compaction backlog
  - Need to coordinate sstable deletion among shards
    - Avoid data resurrection

Resharding (cont’d)

- New resharding algorithm
  - One shard generates sstables on behalf of shards sharing that sstable
  - Read data, write to correct sstable
  - No read amplification
  - For LCS, read nr_shards sstables, so no huge number of small sstables
  - Transfer resharded sstables to new owners at end
  - Disallow compaction of shared sstables
  - Prioritize resharding over compaction

Multiplexing work

- Each thread now does all kinds of work
  - Foreground: cql processing, replica-side request processing
  - Background: memtable flush, compaction, repair, streaming
- Need to switch back and forth to provide good latency
  - All potentially long-running code is interruptible
    - At row granularity, sometimes finer
    - But not too often
  - Checks whether internal time-slice exceeded
    - Saves state and falls back to user-level scheduler
- Alternative: two threads per core
  - One thread for foreground work, one for background
  - Can work well with Hyperthreading
  - Not explored

The return of SEDA

- Multiplexing unrelated work -> bad instruction cache hit rates -> low IPC (instructions per clock)
- Collect similar work into micro-batches, issue together
    - CQL processing, replica cache access, etc.
    - About 2X IPC improvement, latency not harmed

Coordinator issues

- Not enough connections
  - Mostly a benchmark problem
    - Usually, real life apps have enough connections
  - Have request load balancer, but never used
- Bad connection distribution
  - Some shards have too many, some too few
  - Added connection-level load balancer

Streaming, repair

- Range selection must be aware of sharding algorithm to load all cores in parallel
- “repair/streaming coordinator”

Operating System

- Use hwloc library to examine hardware
- Each shard’s thread pinned to logical core
- Each shard’s memory pinned to NUMA node

Operating System (cont’d)

- TCP processing not balanced among shards
  - Scripts to examine/configure NIC
  - Sufficient hardware queues -> one queue per shard
  - Fewer -> one queue per shard group, rps/xps
  - Even fewer -> dedicate core(s) to TCP processing
- Balance storage interrupts (mostly NVMe)

Operating system (cont’d)

- Blocking system calls
  - Destroy performance
  - With some work, can find conditions for io_submit() to work
  - Some ops (open() / fsync() etc) go to auxiliary thread

Docker

- hwloc describes the host instead of the container
  - Worst problem is memory
- Add tunables for explicit memory, shard count configuration
- “Overprovisioned” tunable for less aggressive polling
- Wonderful hardware tuning scripts don’t work unless container is privileged, or run outside container

I/O Coordination

- Some disks too slow if I/O is issued in parallel from all cores
  - Latency explodes
- Designate a subset of cores as I/O coordinators and route I/O through them
- Not a problem with modern NVMe disks

Monitoring

- Expose per-shard metrics
  - Aggregate on the client side
- Critical to expose problems in the sharding logic
  - A misbehaving shard contributes just 1-2% to aggregated metric, so invisible unless per-shard metrics are provided
- Results in huge amounts of metrics exposed

Possible future directions

- Eliminate replica-side coordinator for single-partition access
  - Send RPC directly to correct shard
  - Need to avoid explosion in number of connections
- Driver involvement
  - Token-aware driver sends reqs to correct shard, not just correct node
  - Need to avoid explosion in number of connections
- Asymmetric placement for small tables

Summary

- Hard problem, but inevitable
- Huge throughput gains
  - Will become even more important with time
- Plenty of surprises, but all solvable