CS647 Modern Storage Systems

- Methodology
  - Workload Characteristics
  - Benchmarking

CS647 Modern Storage Systems, Fall 2023
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

How do you evaluate a (storage) system?

- In all cases you need (a) some system + (b) some input
  - Calculate and output statistics
- Many combinations but usually three options
  - Model it – Build it – (Simulate it)
  - What does each of these mean?
    - Let's limit discussion to storage systems for now

Option 1: Modelling = model of system + input

- System model = set of components that provide most important functionality and their main costs
  - E.g. device access latency, etc; storage cache latency, etc.
  - We can consider many factors/aspects/costs of the system
  - Can be very useful (and fast) to abstract a system in a set of operations and their cost
  - How close to reality? E.g. How do you model runtime decisions?
- Input model = description of workload parameters
  - E.g. uniform distribution of key and value sizes
  - Usually a coarse abstraction but no need to run applications
  - How close to reality?
- Create a set of equations that compute statistics
  - Based on how the components behave for the specific input parameters
  - E.g Traffic(S) = L*f*(2*S), where S dataset size, f growth factor, L # of levels, etc.
- Plot curves with results
  - E.g. for model-based analysis of LSM KV stores check [Batsaras et al.]
    - N. Batsaras, G. Saloustros, A. Papagiannis, P. Fatourou, and A. Bilas. VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. February 2020. arXIV 2003.00103. arxiv.org/abs/2003.00103v1
- Usually, system or input model not representative (often both)
  - Not easy for storage systems due to system complexity and dynamic behaviour
  - Our understanding of KV stores would really benefit from more work on modelling for certain aspects, e.g. tail latency

Option 2: Actual system + Actual input (apps)

- Build a system prototype and run the actual apps
  - Higher degree of certainty
  - A form of workload-driven evaluation
- In storage we tend to build actual system prototypes today
  - A lot of work to be realistic, takes time, need to deal with side issues
    - It has to work, never underestimate this for complex systems
  - **Actual system prototype can be close to reality**
    - Is it? Is there any difference between a research prototype and a product?
    - Should they be different? By necessity or by design?
  - Could also build a simulator-based prototype; some form of system implementation
    - A realistic simulator is also a lot of work, especially to run actual applications
    - Even then, it misses a lot of important elements of a real system
    - **How do you know the simulator does the right thing?**
      - You try to calibrate + verify, but you always doubt
      - Personal resolution → If possible, do the actual system
      - Use simulation only if prototyping not possible
      - (Do both: usually not enough resources/time to do both well)
    - **In this area, system simulation not used much**
- <u>However:</u> Generally, not easy to run actual applications
  - Cloud applications can be complex services
  - Real application input and user data may not be available
- We have a system – but what do we run on it?

Option 3: Prototype + workload generator

- Workload generator
  - A program that generates actual requests based on a model
  - E.g. read requests with uniform size over a range of sizes
- What is a ”good” workload generator?
  - “good” = representative of actual applications
  - Not necessarily, but we know of no other better approach
  - Running an actual system with the wrong workload will not lead anywhere…

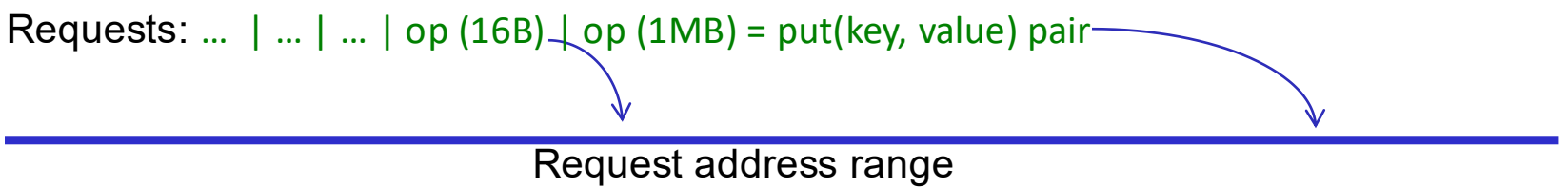
KVS evaluation today

- Dominates by far: Prototype implementation + Synthetic workload
  - Reasonably accurate (we hope)
  - Reasonably fast/convenient (not always)
- It would be fantastic to make progress on ”system modelling”
  - Generate quickly asymptotic behavior and trends
  - Better system models (e.g. dynamic behavior), even with simple input models
  - Better input models, even with simple system models
  - Improving both seems rather futuristic at this point

How can we generate synthetic workloads?

- Unfortunately, not easy to answer
  - Representative cloud workloads too complex and often not available
- Does "good" or "bad" workload matter? → Clearly
  - Large requests have different behavior from small requests
  - Mixes of operations vastly different
- Today
  - Rely on application statistics from real environments
  - Generate synthetic workloads that are useful
- **How?**
  - This is our discussion for the rest of this class

Parameters for KVS input



- Key distribution: Where does a request fall in the storage address space?
  - Uniform, concentrated, …
- Size: Size of operations / requests (keys, values)
- Content (value): e.g. to evaluate impact of compression
- Type of ops: How many of read, write/insert, update, scan, delete
- Sequence of operations: e.g. one insert then one read then...
- Arrival time of each operation? Constant? Bursty?
  - Operation rate: 1 req/s or 1M req/s
  - Number of threads issuing requests or outstanding requests (queue depth)
- Key range (address space)
  - One (shared) for everyone or one for each application?
  - E.g. how many independently-accessed devices
- How do these relate to M|M|c and similar queuing models?
  - IN: arrival rate, service time, #servers, max queue sz/inf, #jobs/inf, queuing policy/FIFO
  - OUT: throughput, latency, (queue) wait time, server busy time, average queue size

How many config points for KVS input?

- Key access frequency (hotness): **3 configurations**
  - Spread out → uniform
  - Concentrate to hot ranges (more than one) → zipfian
  - Concentrate to most recent values → latest (special case for zipfian where hot = latest)
- (Hot) Key location in key space: does it matter? (rocksdb/db_bench says yes)
  - YCSB picks hot keys in a single manner, spreading them throughout the key space
  - Affects significantly cache behavior and read/write amplification
- Size of keys, values: **5-10 (x%, y%, z%) configurations**
  - Fixed: small, medium, large ← ok but looks simplistic
  - Better: Distribution for key size, value size - too many distributions - What is representative of real applications?
  - Settle: x% <= 32B, 32B < y% <=128B, 128B < z% <= MAX → multinomial distribution
  - [(100,0,0), (0,100,0), (0,0,100), (33,33,33), (80,10,10), (10,80,10), (10,10,80), etc.]
- Operation type distribution: **5-10 configurations**
  - Similar to key size, settle for: (Read, Insert, Update, Scan, Delete) = (R%, I%, U%, S%, D%),
- Operation arrival time distribution: **5-10 configurations (2 distributions x 2-5 rates = 5-10)**
  - Equidistant → Uniform interarrival time = constant rate
  - Bursty → some form of bell shaped distribution (let's say normal)
  - Depending on what we measure, it makes sense to also vary arrival/burst rate (e.g. light, medium, heavy)
- Number of key ranges (devices, stores, etc): **1-5 configurations**
  - From 1 to 100 (for concurrency)
- Operation type sequence → too much, pick randomly
- Value distribution → skip it, already too much
- Usually we want to examine a few system parameters too: **5-10 configurations**
  - Number of threads (ordering), amount of available memory, number of storage devices
- **# configs to examine the impact of a new system parameter or technique**
  - **Conservative: 3 x 5 x 5 x 5 x 2 = 750 runs to do a "sparse" pass over**
  - **More aggressive: 3 x 10 x 10 x 10 x 5 = 15000 config points**
  - **Are these configurations points are close to reality (=applications users care about) ?**

What do we typically measure in each run?

- Application metrics
  - Usual: Throughput, latency, tail latency
  - Recovery time, time to reach stable performance under elasticity, etc.
- System metrics
  - Utilization of different resources
  - Throughput, latency at many different points in the system
  - CPU overhead (cycles) for different operations
  - CPU time breakdown, queue sizes, many more…
- How do you show results?
  - Typical graphs for latency, throughput, CPU breakdowns, etc.
- Do not rely (only) on recipes
  - Think what you need to see in order to understand your system
  - Show high-level (what a user would see) and low-level (what happens in the system) statistics and behavior

Workload Generators: Tool vs. Model

- YCSB, fio, etc. are used to generate I/O requests
  - There are two aspects to all of these
- 1. The framework/tool
  - Parameters for each input aspect
  - Can generate requests based on these parameters
  - fio has been extremely useful as a tool
    - Flexible I/O Tester, https://fio.readthedocs.io/en/latest/fio_doc.html
- 2. The models it assumes/uses
  - The model essentially is the set of values for each parameter
  - Models require knowledge/understanding of applications
  - They are hard to come by - This is what is needed most
  - YCSB (next) provides models based on application behavior
    - Lasting impact
- Do not confuse (1) and (2)

YCSB: Yahoo Cloud Serving Benchmark

- ACM SoCC'2010 [Best paper + 10 years test-of-time award]
- 1. Picks a few configurations for the synthetic workload
  - Based on experience, observations, NO real analysis
  - Intention to model operation of cloud systems
    - Scale-out, Elasticity, High availability
  - Two types of benchmarking (tiers)
    - (1) Performance of the system: achieved performance on fixed # servers
    - (2) Scaling of the system
      - Performance: Servers increase while the system is running
      - Capacity + Performance: Data and servers increase over the lifetime
    - Post-mortem: Tier-1 has been by far the most popular use for YCSB
- 2. Implements distributions carefully
  - And simplify other aspects
- 3. Puts everything in a workload generator (KV client/tool)
- Similar work with RocksDB – Usenix FAST'2020
  - Read the paper – we will discuss next time

YCSB: Types of Target Systems (cloud)

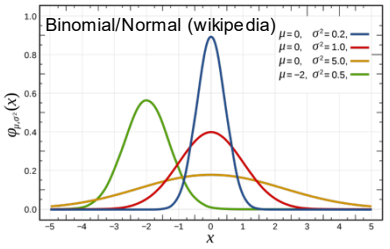
System (Yahoo! Production)	Read/Write optimized	Latency/ Durability	Sync/Async replication	Row/column
PNUTS	Read	Durability	Async	Row
BigTable	Write	Durability	Sync	Column
HBase	Write	Latency	Async	Column
Cassandra	Write	Latency	Tunable	Column
Sharded MySQL	Read	Latency	Async	Row

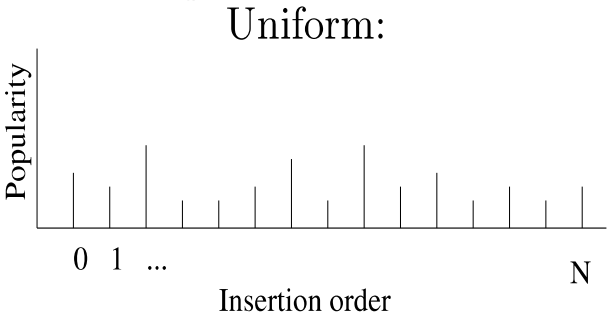
- Each is some sort of KV store
- No access to actual applications/data creating requests for these
- How do we evaluate them?

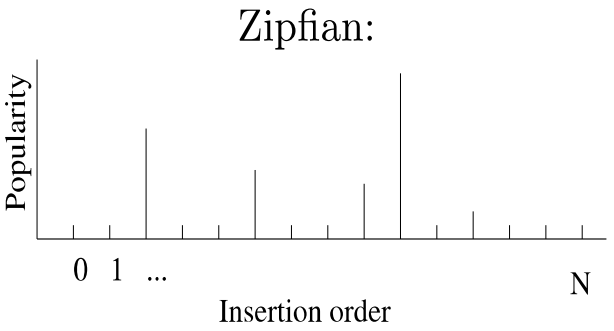
Distributions

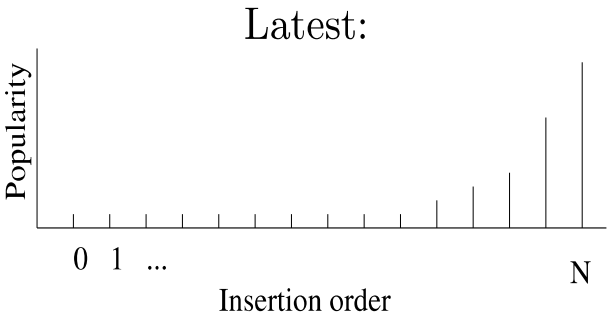
[Jim Gray, et al. Quickly generating billion-record synthetic databases. SIGMOD Rec. 23, 2 (June 1994), 243–252. https://doi.org/10.1145/191843.191886]

- Uniform (Random) → random in libc
- Normal & Multinomial → A number of categories with given probability
  - Uniform has equal probabilities for all values
  - If you have N values/categories, create a large range [0, MAX_INT]
  - Divide this range to N sub-ranges, one for each value
  - Each subrange has size equal to the probability of the respective value
  - Pick uniformly random numbers in the full range
  - Identify the subrange it belongs and pick the respective value
  - If you have a single peak, then Normal
- Zipfian = multinomial with a number of fixed popular areas
  - Popular items are anywhere in the range
  - They remain the same as items are inserted
- Latest
  - Similar to Zipfian but popular items change
  - As values are inserted popular are always the latest
- Implementation of distributions can be challenging
  - Ensure full range of items (keys) are generated
  - In latest, popularity needs to shift as new items arrive
  - Does the distribution still hold if
    - We look at a sub-range of the full range? (how big?)
    - We look at a smaller number of arriving requests? (how many?)
  - What is an appropriate duration of experiments to ensure these? 10M requests? 100M? 1B?









YCSB Configurations

- Parameters – YCSB simplifies evaluation space (non-trivially!)
  - Operation type sequence → too much, pick randomly
  - Value distribution → skip it, already too much
  - Size of keys, values: Allow for multinomial distribution – by default:
    - Use fixed sizes (default: K=20B, V=1000B)
    - In retrospective, a limitation
  - Operation interarrival time distribution: uniform, ~~normal~~
    - Use multiple threads
    - Each YCSB thread is synchronous (issue next when previous done)
    - In retrospective, ok – but not for modeling burstiness
  - Number of outstanding operations: fixed = number of threads
    - Allow users to change the number of YCSB threads in each run
    - In retrospective, ok – but not for modeling burstiness
  - Number of key ranges (regions): parameter
    - User can run multiple instances of the benchmark
    - In retrospective, ok
  - Key distribution: 3 options -- uniform, zipfian, latest
    - In retrospective, probably ok
  - Operation type distribution: multinomial
    - Can modify distribution parameters, but no one does
    - In retrospective, probably a limitation
- YCSB implements carefully distributions for the last two purposes
  - Random, Uniform+Multinomial (Normal), Zipfian, Latest

YCSB Synthetic Workloads

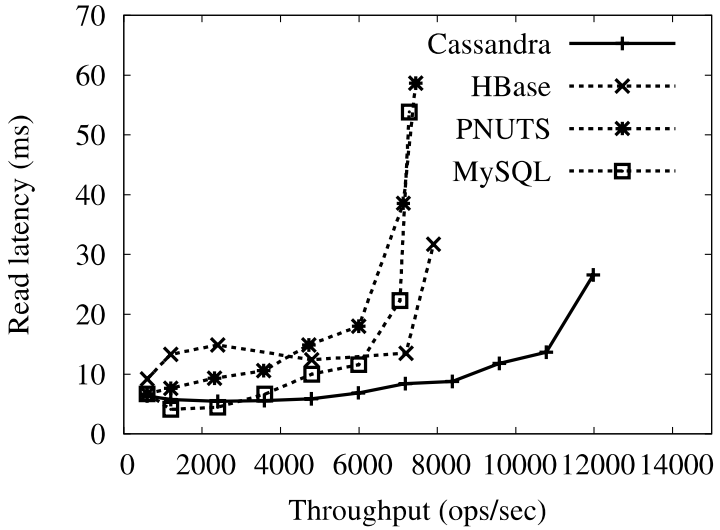
- Running YCSB
  - First load data [Load A] – Then run A-E in sequence [Run A-E] below

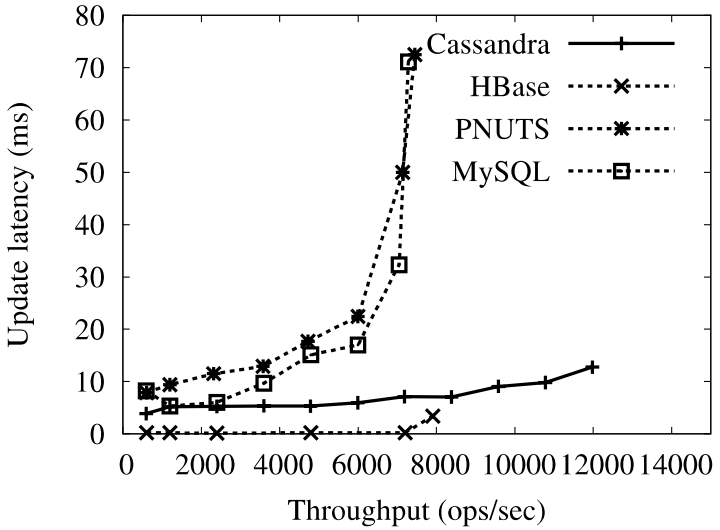
Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

- Run A-E chosen based on the experience of the authors
  - They have served us well (and continue to do so) for more than 10 years
- Specify number of threads, number of ranges
  - Key, value sizes default
- Can change / modify many parameters
  - But how? Easy to do, but what values to choose?
  - Difference between Tool vs. Model

Validation: How do you know results make sense?

- Show either of:
- 1. Synthetic workloads are close to actual workloads
- 2. When used to evaluate actual systems, it measures useful properties and shows differences of systems
- YCSB paper does (2)
  - YCSB workload A latency vs. throughput for read/write
  - e.g. From YCSB paper (Run A is update heavy)





How to Run YCSB

- Two versions
  - Java: https://github.com/brianfrankcooper/YCSB
  - C++: https://github.com/basicthinker/YCSB-C
  - We have an in-house modified version to reduce malloc overhead
  - Supports drivers for different KV stores (e.g. RocksDB)
  - Can build custom drivers for new KV stores
- Run (C++)
  - ./ycsb-ecdb -threads <#threads> –P <workload-file-path>
- Workload file with parameters – usually specify
  - recordcount: # of KV pairs to insert during load phase
  - operationcount: # of operations during run a-e phases
  - (read|update|scan|insert)portion: % of operations in run phases
- Run multiple instances – need to sync/summarize results

Reading

- Review
  - Slides, Your class notes
  - Benchmarking cloud serving systems with YCSB. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. In SoCC '10. https://dl.acm.org/doi/pdf/10.1145/1807128.1807152
- Read
  - Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. Z. Cao and S. Dong et al. 2020. In USENIX FAST'20. https://www.usenix.org/system/files/fast20-cao_zhichao.pdf
- Next class
  - Paper discussion for RocksDB paper above

Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. Z. Cao and S. Dong et al. 2020. In USENIX FAST'20. https://www.usenix.org/system/files/fast20-cao_zhichao.pdf

	UDB	**ZippyDB**	**UP2X**
Req rate	10.2B / 14d	420M / 24h	11M / 24h
avg	8400 req/s	4860 req/s	1284 req/s
peak	-	-	night: 1000 req/s day: 1500 req/s
ops	70% get	78% get	93% merge
	20% put	13% put	7% get
	3% scan		
hotness of keys	100% < 14 accesses	100% < 100 accesses	100% < 100 accesses
	70% < 2 gets	95% < 10	70% < 10
	35% < 1 get	80% < 2	20% < 1
	75% < 1 put	60% < 1	
	hot: ~5%	hot: ~5%	hot: ~30%
KV sizes (avg/std dev)	K: 27B/3B V: 126B/22B K+V: 150B	K: 48B/4B V: 43B/12B K+V: 80B	K: 10B/1B V: 47B/12B K+V: 60B