Zoolander: Efficient Latency Management in NoSQL Stores

Aniket Chakrabarti and Christopher Stewart
The Ohio State University

Daiyi Yang
StumbleUpon.com

Rean Griffith
VMWARE

Extended Abstract

NoSQL stores expose narrow APIs for data access, e.g., get(key) or put(key,val). While these APIs often give up strong consistency and transactions, they can scale throughput under intense workloads. Widely used stores, e.g., Apache Zookeeper, Cassandra, and Memcached, have been shown to achieve 10^9 accesses per day in the face of workload shifts, software faults, and performance bugs. However providing low latency for every access remains challenging. Latency, unlike throughput, quickly yields diminishing returns under scale out approaches, making it important to choose the most efficient approach. Further, DNS timeouts, GC, and other rare system events can hold resources from time to time [3]. These events hardly impact throughput, but they can increase latency for some accesses a lot.

Internet services that access a lot of data under tight response time demands need NoSQL stores that provide low latency all the time. Figure 1 depicts such services and their demands. We describe them below.

1. Old-school services, such as e-commerce websites, are increasingly using NoSQL instead of databases. In these embarrassingly parallel services, end-user requests access data independently but each request must complete quickly. Slow accesses translate to unhappy end users.

2. Map reduce services spawn parallel worker nodes that compute local results and forward them to reducers to produce the final output. Here, the term service reflects a growing trend where jobs are expected to complete within response time targets. We note two reasons for this trend. First, jobs that run on pay-as-you-go clouds cost less if they finish within 1-hour leasing intervals. Second, jobs that finish quickly offer qualitative business advantages, e.g., real-time Twitter analysis.

3. Scientific computing as a service is an emerging workload on public clouds [2]. In the past, these workloads ran only on private, custom hardware but public clouds can offer performance-to-cost efficiency. The challenge is matching the absolute performance of private hardware. These workloads use barriers and synchronization heavily. One slow data access can delay the completion of a barrier and ultimately delay the entire workload.

Our Contribution: Zoolander is a transparent middleware for managing the latency of NoSQL stores. Here, we use the term transparent because Zoolander can control latency for many underlying key value stores, including the stores listed above. Zoolander makes two important contributions. First, it supports replication for predictability as a first class approach for latency management. Replication for predictability copies data to multiple nodes, sends every access to all nodes, and uses the result from the access that finishes first. At first glance, this may seem wasteful. Why should two nodes compete to provide fastest response time instead of dividing the work? We have found that NoSQL stores exhibit heavy-tailed service times. Sometimes, data access can be 10^5 times slower than the mean service time. By taking the fastest response, replication for predictability reduces the effect of these slow accesses on end users. Sometimes, it is more efficient to manage latency by reducing heavy tails than by dividing work.

Zoolander uses replication for predictability to meet SLOs more efficiently, i.e., using fewer resources. Our second contribution is a set of analytic models provide the expected SLO under a workload and replication strategy. Managers can use our models to quickly iterate over a wide range of replication strategies that mix divide-the-work and reduce-the-tail approaches. The input to our models capture key workload features that often change within and across Internet services. These practical inputs allow managers to ask long term questions, e.g., “If we push a prototyped feature into production, how many additional cloud instances would the backend NoSQL store need to maintain the same SLO?” Also, managers can ask questions that can be used for immediate feedback control, e.g., “If the request arrival rate doubles, what will NoSQL store’s SLO become?”

This abstract provides brief results to motivate these contributions. First, we highlight a heavy tail in a popular key value store, explaining its effect on end users. Then we describe a concrete example where replication for predictability could be used in practice. Note, that Zoolander will soon be released as an open-source middleware platform.

Replication for predictability: A dumb idea whose time has come. Replication for predictability is not a new idea. RAID 0 and, more recently, the SCADS manager [5] have used this approach to lower perceived latency. Nonetheless, it is rarely used in existing NoSQL stores which
have focused on high throughput rather low latency. However, as the workloads described in Figure 1 become more prominent, NoSQL stores will need to provide low latency and high throughput—meaning the time to use replication for predictability along with existing approaches has come. When data is replicated for predictability across $N$ nodes, the fastest node masks the $N-1$ slowest nodes. A key insight in NoSQL stores is that the difference between the fastest and second fastest response may be significant. We deployed Zookeeper [1] on our local 32 node Dell cluster. Each node has 2.66 GHz bandwidth, 3MB L2 cache, 2GB of DRAM memory. Zookeeper used 3 nodes, reflecting a typical configuration for fault tolerance. We ran a client that sequentially issued 100 write accesses on a separate node. Figure 2 groups the writes according to the client perceived response time and shows the total number of seconds used by each group. There is a heavy tail—accesses above the 50th percentile account for time than accesses below the 50th percentile. The slowest 10 percent account for 20% of the total time that the client spent waiting.

We reran this experiment several times, issuing the same sequence of accesses. While the heavy tail distribution persisted across runs, the accesses that fell in the slowest 10 percent changed from run to run implying low correlation between service time and issue-order. This motivates replication for predictability as there is high probability that an anomalously slow access on a node can be masked by a fast response from another node. Zoolander provides a low overhead, software-based implementation this approach. Zoolander on Realistic Workloads. Zoolander uses analytic models to guide its replication strategy. Without diving into details, we can describe the conditions where Zoolander prefers specific approaches. Under heavy request arrivals, Zoolander’s models prefer to divide the work, using data partitioning. This approach exploits the lack of transaction support in NoSQL stores. Under heavy request arrivals and small partitions, Zoolander’s models still prefer to divide the work, using data replication. This approach comes with consistency costs. As a middleware, Zoolander supports any underlying consistency model exposed by the underlying NoSQL store. Finally, under moderate to light workloads, Zoolander’s models prefer replication for predictability. The intuition is that if queuing delay is not large then divide the work approaches provide diminishing returns in terms of latency. The analysis above described the most efficient technique to improve latency across request arrivals. It leaves out an important question, is it profitable to improve latency?

Applications should consider scaling out under light to moderate loads if the costs of scale out are low or if the gains from low latency are very high. For example, NoSQL stores at Facebook exhibit diurnal patterns where the arrival rate can drop by 50% at night. Since Facebook owns its own datacenters, the cost of scale out at night is the price of energy. For instance, a 2GHz CPU with 2MB cache will likely consume 100W making its cost as low as $0.003 per hour. If replication for predictability can complete 100 additional Facebook end-user requests within SLO, then the cost per request would be only $0.00003, far below the typical yield per page view ($0.00058 [4]). Conversely, consider a social networking site that uses only cloud instances from Amazon at the cost of $0.085 per hour. Here, the cost of would be prohibitive. Zoolander’s models also consider the end to end value of these approaches.

1. REFERENCES