**Chapter 38: Redundant Arrays of Inexpensive Disks (RAIDs)**

RAIDs is a technique to use multiple disks in concert to build a faster, bigger and more reliable disk system.

Externally, a RAID looks like a disk: a group of blocks one can read or write. Internally, it is very complex, consisting of multiple disks, memory and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks.

RAIDs offer **performance** over a single disk. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is **capacity**. Large data sets demand large disks. Finally, RAIDs can improve **reliability**. Spreading data across multiple disks makes the data vulnerable to the loss of a single disk. With some form of **redundancy**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

RAIDs provide these advantages **transparently** to the systems that use them.

**38.1 Interface And RAID Internals**

When a file system issues a logical I/O request to the RAID, the RAID internally must calculate which disks to access in order to complete the request, and then issue one or more physical I/Os to do so. For example, consider a RAID that keeps two copies of each block. When writing to such a mirrored RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued.

A RAID system is often built as a separate hardware box, with a standard connection to a host. Internally, RAIDs are complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases, non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations.

At high level, RAID is very much a specialized computer system as it has processor, memory and disks. The difference is that instead of running applications, it runs specialized software designed to operate the RAID.

**38.2 Fault Model**

RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design.

The first fault model we will assume is simple, and it is called the **fail-stop** fault model. In this model, a disk can be in exactly one of two states: **working** or **failed**. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost. The critical aspect is that this model assumes about fault detection. Thus, for now, we do not have to worry about more complex “silent” failures such as disk corruption.

**38.3 How To Evaluate A RAID**

1. **Capacity**: given a set of N disks, each with B blocks, how much useful capacity is available to clients of the RAID. It should be N\*B, but if we have a system that keeps two copies of each block, the capacity is N\*B/2.
2. **Reliability**: how many disk faults can the given design tolerate.
3. **Performance**:

There are three important RAID designs: Level 0 (striping), level 1 (mirroring) and level 4/5(parity-based redundancy).

**38.4 RAID Level 0: Striping**

This level serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will stripe blocks across the disks of the system as follows (assume here a 4-disk array):

Table

Description automatically generated

The idea is simple as we just need to spread the blocks of the array across the disks in round-robin fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array. We call the blocks in the same row a **stripe**. For example, blocks 0, 1, 2 and 3 are in the same stripe.

We also assume that only 1 block (4KB) is placed on each disk before moving on to the next. We can make the **chunk size** to be two as follows:

Table

Description automatically generated

We can place two blocks on each disk before moving on to the next disk. In this case, the chunk size is 2\*4 = 8KB. And a stripe is 4 chunks or 32KB.

**Chunk Sizes**

Chunk size mostly affects performance of the array. For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file. However, the positioning time to access blocks across multiple disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size reduces such intra-file parallelism and relies on multiple concurrent requests to achieve high throughput. However, large chunk sizes reduce positioning time. For example, a single file fits within a chunk and thus is placed on a single disk, the positioning time incurred while accessing it will just be the positioning time of a single disk.

Choosing the best chunk size is hard to do since it requires knowledge about the workload presented to the disk system. We will assume that we will use a block size of 4KB.

**Back To RAID-0 Analysis**

Consider N disks and each of size B blocks, stripping delivers N\*B blocks of useful capacity. A thing to consider here is that any disk failure will lead to data loss. In addition, all disks are utilized, often in parallel, to service user I/O requests.

**Evaluating RAID Performance**

We will consider two metrics: **single-request latency** and **steady-state throughput**. The first reveals how much parallelism can exist during a single logical I/O operation and the second metric is also critical.

In addition, there are two types of workloads: **sequential** and **random**. With a sequential workload, we assume that requests to the array come in large contiguous chunks. For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. We only consider one of the two workloads, but in reality, it is a mix.

With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. For random access, it spends most time seeking and waiting for rotation and little time transferring data.

To capture this difference in our analysis, we will assume that a disk can transfer data at S MB/s under a sequential workload, and R MB/s when under a random workload (S >> R).

Assume a sequential transfer of size 10MB on average and a random transfer of 10KB on average. We also assume the average seek time is 7ms, rotational delay is 3ms and transfer rate of disk is 50MB/s.

For sequential, the time to transfer is 10MB/50MB/s = 200ms, so the total time to complete a request is 200 + 7 + 3 = 210. Therefore, S = 10MB/210ms = 47.62MB/s.

For random transfer, the time to transfer is 10KB/50MB/s = 0.195ms, so the total time to complete a request is 10.195ms. Thus, R = 10KB/10.195ms = 0.982MB/s.

**Back To RAID-0 Analysis, Again**

From a latency perspective, for example, the latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.

From the perspective of steady-state sequential throughput, we’d expect to get the full bandwidth of the system. Thus, throughput equals N (the number of disks) multiplied by S (the sequential bandwidth of a single disk).

**38.5 RAID Level 1: Mirroring**

With a mirrored system, we simply make more than one copy of each block in the system. Each copy should be placed on a separate disk. By doing so, we can tolerate disk failures.

Table

Description automatically generated

The arrangement above is a common one and is sometimes called **RAID-10** or **RAID 1+0, stripe of mirrors**. Another common arrangement is **RAID-01** or **RAID 0+1, mirror of stripes**, which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them.

When reading a block from a mirrored array, the RAID can either read one or another.

**RAID-1 Analysis**

Let us assess RAID-1. From a capacity standpoint, RAID-1 is expensive; with the mirroring level = 2, we only obtain half of our peak useful capacity. With N disks of B blocks, RAID-1 useful capacity is (N\*B)/2.

From a reliability standpoint, RAID-1 does well. It can tolerate the failure of any one disk.

Finally, we analyze performance. From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write. However, because the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay of the two requests and thus will be higher than a write to a single disk.

To analyze steady-state throughput, let us start with the sequential workload. The maximum bandwidth obtained for sequential write is N\*S/2 (half of the peak bandwidth) as for each logical write, we have to perform 2 physical writes.

We also obtain the same performance during a sequential read because it skips through one disk every read since the files are written twice.

Random reads are the best case for a mirrored RAID. In this case, we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers N\*R MB/s.

Random writes perform N\*R/2 MB/s since each logical writes is equal to two physical writes.

**38.6 RAID Level 4: Saving Space With Parity**

We now present a different method of adding redundancy to a disk array known as **parity**. Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. However, it costs performance.

Table

Description automatically generated

Consider five-disk RAID-4 system. For each stripe of data, we added a single **parity** block that stores the redundant information for that stripe of block. For example, parity block P1 has redundant information that it calculated from blocks 4, 5, 6, and 7.

To compute parity, we need to use a mathematical function that enables us to withstand the loss of any one block from our stripe. It turns out the simple function **XOR** does the trick quite nicely.

A screenshot of a computer

Description automatically generated with low confidence

XOR is 1 if there are odd number of ones, 0 otherwise. the number of 1s in any row, including the parity bit, must be an even (not odd) number; that is the invariant that the RAID must maintain in order for parity to be correct. Thus, we can use this information to reconstruct missing values. For example, if in the first row, C2 is missing, we can use the other four information to know what it was.

In data blocks, we simply do XOR operation across each bit of the data blocks.

A picture containing table

Description automatically generated

**RAID-4 Analysis**

From a capacity standpoint, RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is (N−1)\*B.

RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

Regarding performance, sequential read can utilize all the disks except the parity disk. Thus, the peak effective bandwidth is (N-1)\*S MB/s.

The sequential write also has the bandwidth of (N-1)\*S MB/s since RAID-4 performs an optimization known as **full-stripe write** that calculate the parity in parallel as they are written.

Random read also has the bandwidth of (N-1)\*R MB/s as a set of 1-block random reads will be spread across the data disks of the system but not the parity disk.

Regarding random writes, imagine we wish to overwrite block 1 in the example above. We could just go ahead and overwrite it, but that would leave us with a problem: the parity block P0 would no longer accurately reflect the correct parity value of the stripe; in this example, P0 must also be updated. There are two methods: **additive parity** that read the other data blocks in parallel and XOR with the new block and the results are also written in parallel and **subtractive parity** that we only update the parity when there is a change in bits. The former method takes a lot of time as it scales with the number of disks. The latter method achieves the bandwidth of R/2 MB/s. (We called this **small-write problem**)

**38.7 RAID Level 5: Rotating Parity**

RAID-5 is similar to RAID-4, accepts that it rotates the parity block across drives:

Graphical user interface, application, table

Description automatically generated

**RAID-5 Analysis**

The performance is the same, except that the bandwidth for random writes is N/4 MB/s. The factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations, which is simply the cost of using parity-based RAID.

RAID-5 almost completely replaced RAID-4. The only place where it has not is in systems that know they will never perform anything other than a large write, thus avoiding the small-write problem altogether. In those cases, RAID-4 is sometimes used as it is slightly simpler to build.

**38.8 RAID Comparison: A Summary**

Table

Description automatically generated

**38.9 Other Interesting RAID Issues**

There are many other RAID designs, including Levels 2 and 3 from the original taxonomy, and Level 6 to tolerate multiple disk faults.

There is also what the RAID does when a disk fails. Sometimes it has a **hot spare** sitting around to fill in for the failed disk. There are also more realistic fault models, to take into account latent **sector errors** or **block corruption**. Finally, we can even build RAID as a software layer: such **software RAID** systems are cheaper but have other problems, including the consistent-update problem.