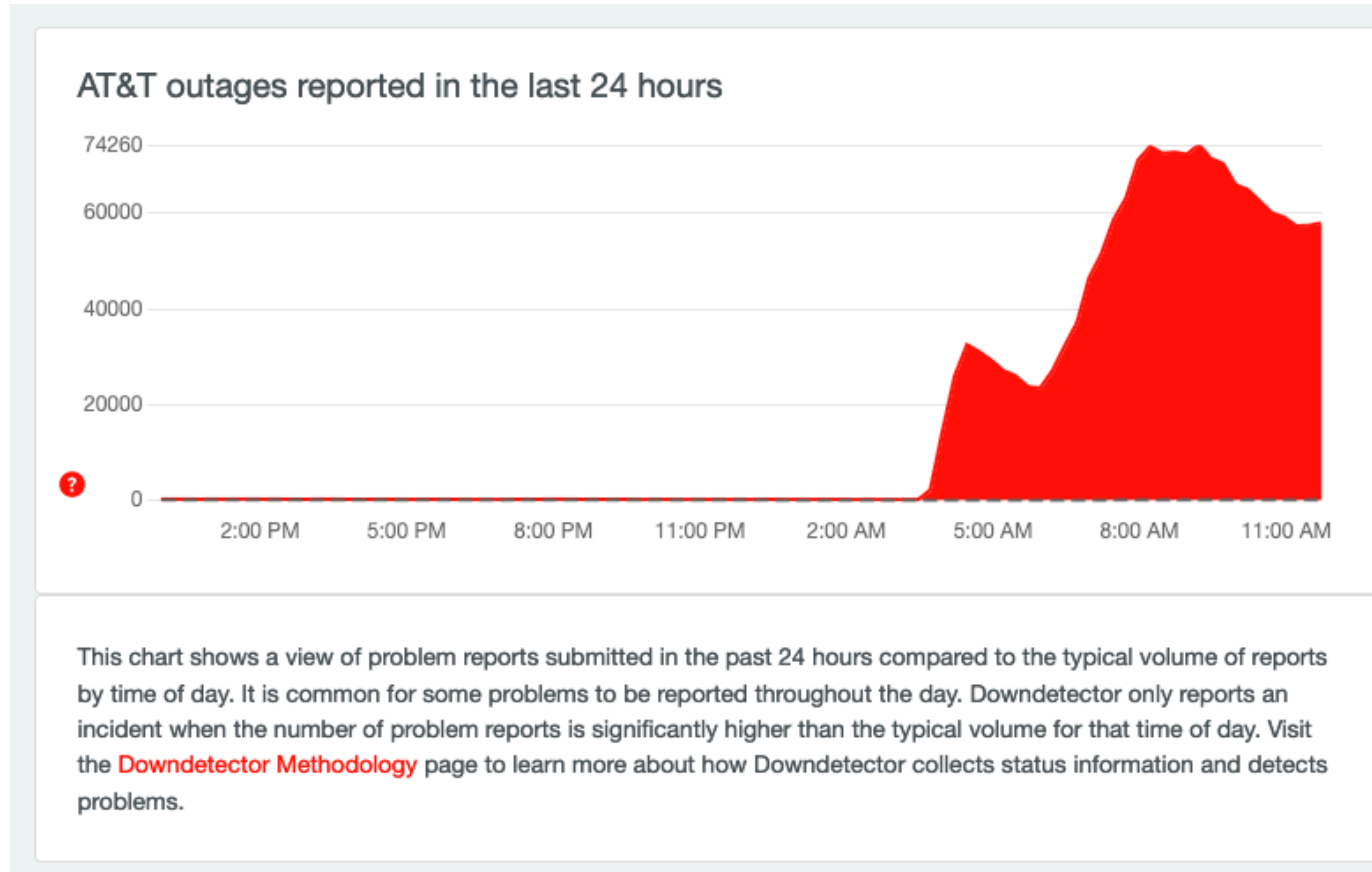
6.1800 Spring 2024

Lecture #7: Performance + Other Concerns

performance, with a deep dive into storage and filesystems

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

6.1800 in the news



https://downdetector.com/status/att/

6.1800 in the news

Thank you for reaching us. We understand how important it is to enjoy the most of our services. I would like to thank you for your patience while our systems are undergoing planned maintenance/being optimized for performance.
^DanielG

Network Update

Last updated February 22, 2024, 6:46 p.m. CT

Based on our initial review, we believe that today's outage was caused by the application and execution of an incorrect process used as we were expanding our network, not a cyber attack. We are continuing our assessment of today's outage to ensure we keep delivering the service that our customers deserve.

sometimes performance issues are a result of **human error**

https://about.att.com/pages/network-update

6.1800 in the news

Many 911 centers in the state are getting flooded w/ calls from people trying to see if 911 works from their cell phone. Please do not do this. If you can successfully place a non-emergency call to another number via your cell service then your 911 service will also work. #outage

9:08 AM · Feb 22, 2024 · **224.1K** Views

failures in one system can impact people who don't use that service at all

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**

2. programs should be able to **communicate** with each other → **bounded buffers** (virtualize communication links)

3. programs should be able to **share a CPU** without one program halting the progress of the others → **threads** (virtualize processors)

you've also seen virtualization as a technique for running multiple operating systems on the same physical hardware

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** with each other → **bounded buffers** (virtualize communication links)
3. programs should be able to **share a CPU** without one program halting the progress of the others → **threads** (virtualize processors)

today: performance more generally, with a focus on storage, and how the abstractions that an operating system provides impact our systems

performance issues have influenced a lot of the system designs you've seen so far

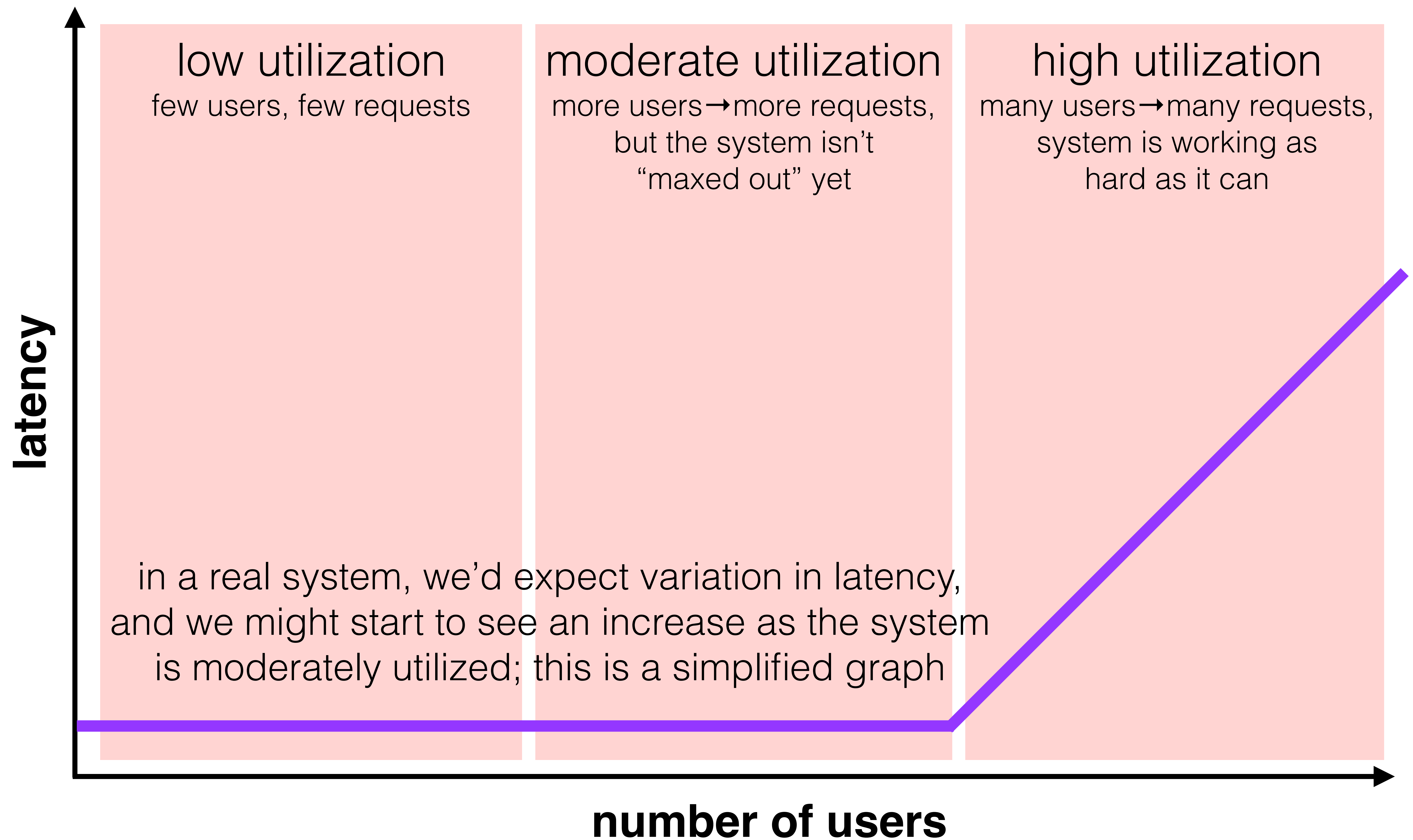
latency: how long does it take to complete a single request?

example: how long does it take to retrieve a particular piece of data in an OS?

throughput: how many requests per unit of time?

example: how many reads or writes can a system do to a disk at once?

utilization: what fraction of resources are being utilized? this puts our performance measurements in context



question: how do you expect latency to change as the system progresses through these stages?

performance issues have influenced a lot of the system designs you've seen so far

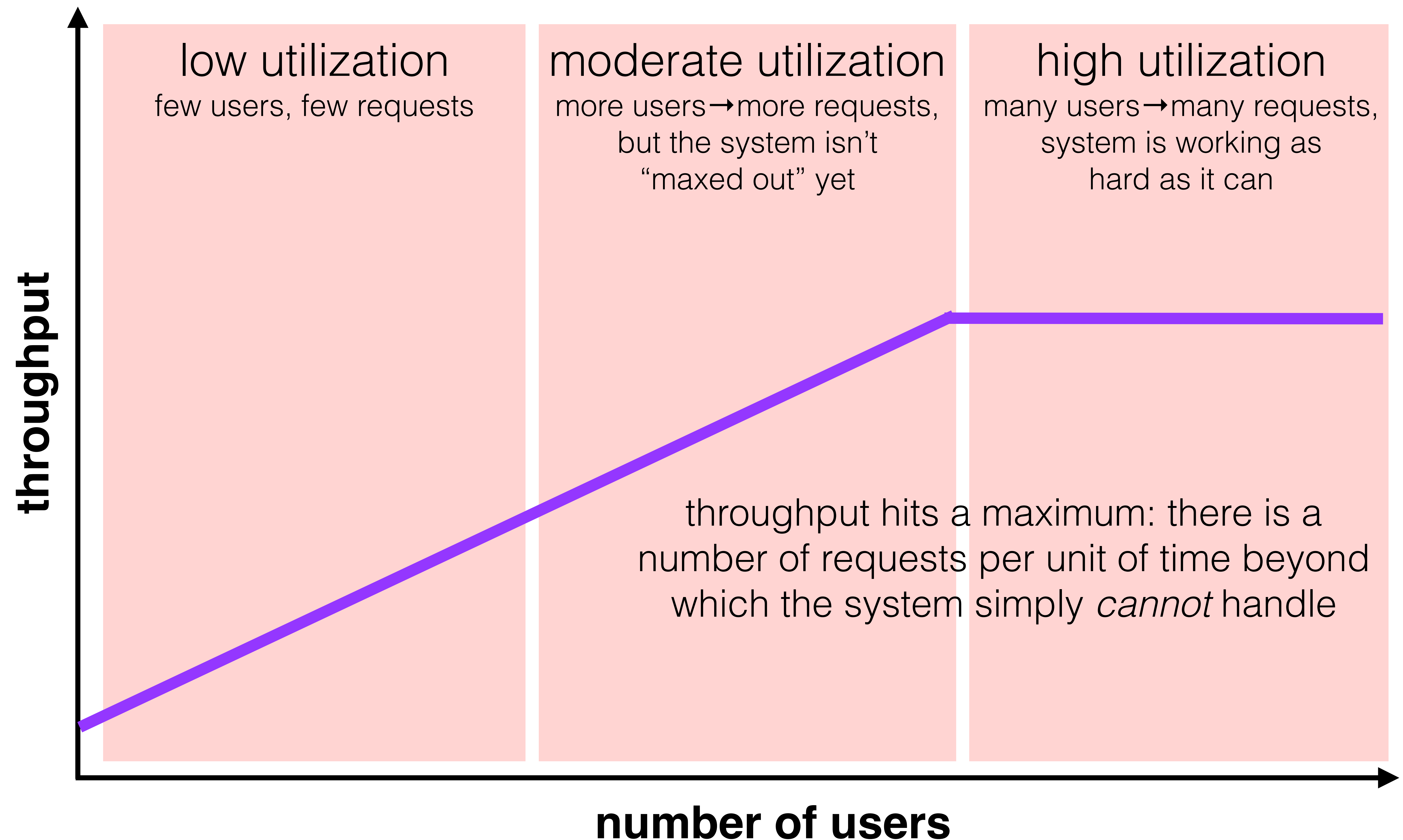
latency: how long does it take to complete a single request?

example: how long does it take to retrieve a particular piece of data in an OS?

throughput: how many requests per unit of time?

example: how many reads or writes can a system do to a disk at once?

utilization: what fraction of resources are being utilized? this puts our performance measurements in context



question: how do you expect throughput to change as the system progresses through these stages?

performance issues have influenced a lot of the system designs you've seen so far

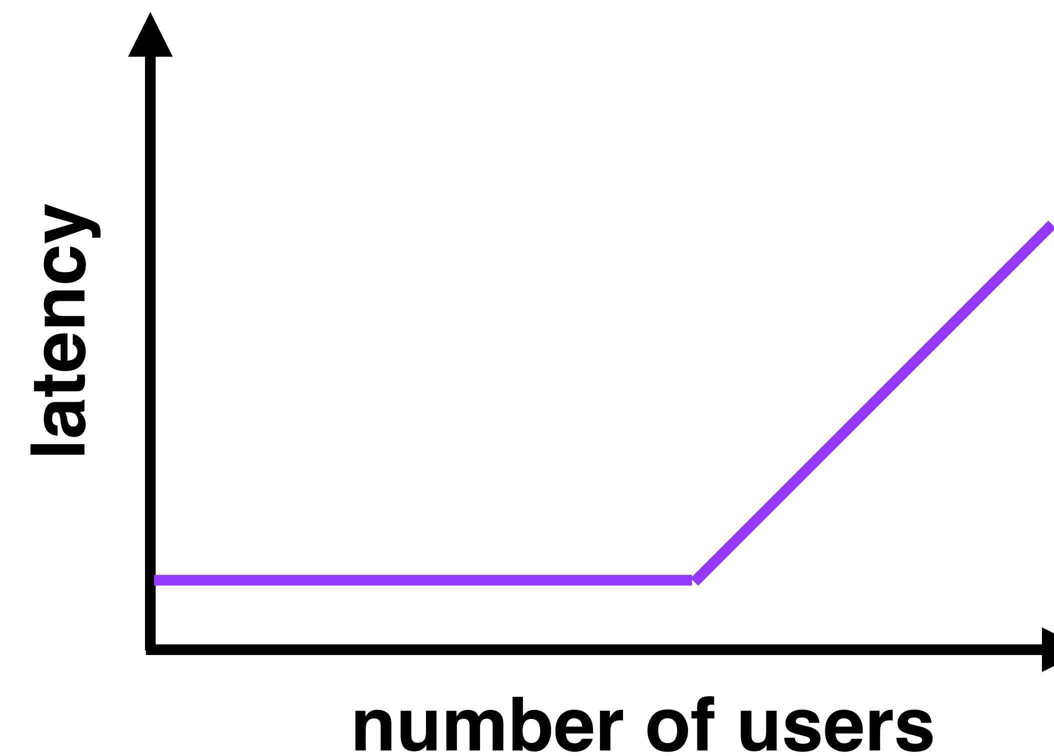
latency: how long does it take to complete a single request?

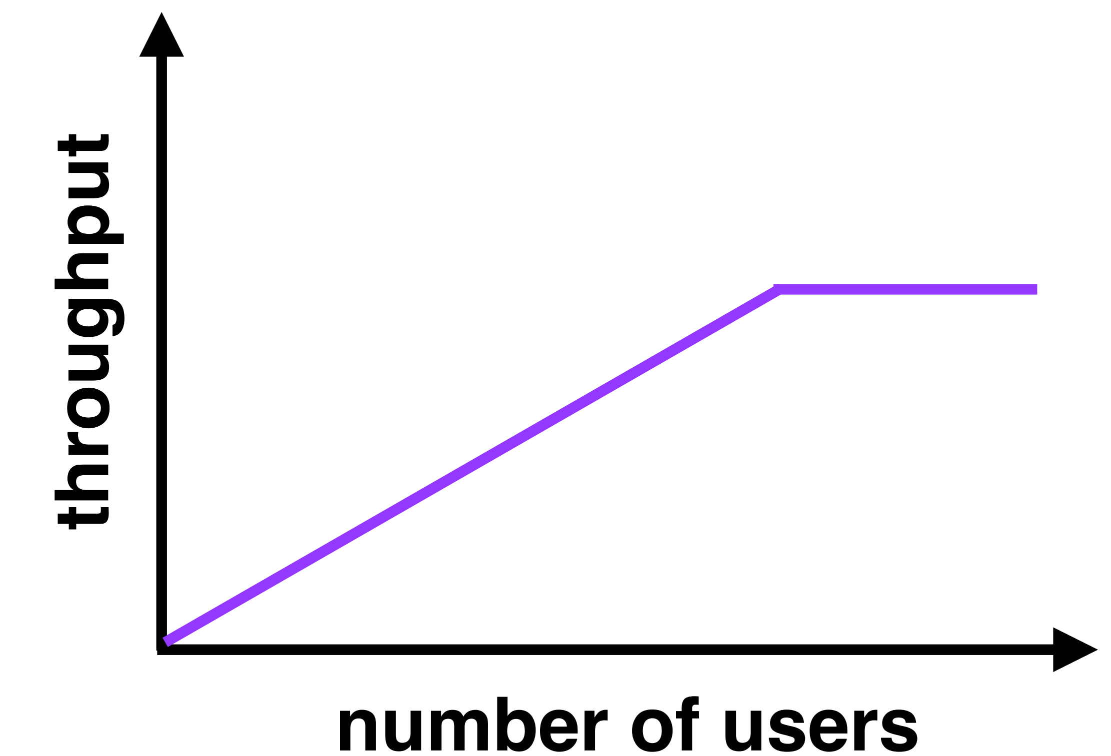
example: how long does it take to retrieve a particular piece of data in an OS?

throughput: how many requests per unit of time?

example: how many reads or writes can a system do to a disk at once?

utilization: what fraction of resources are being utilized? this puts our performance measurements in context

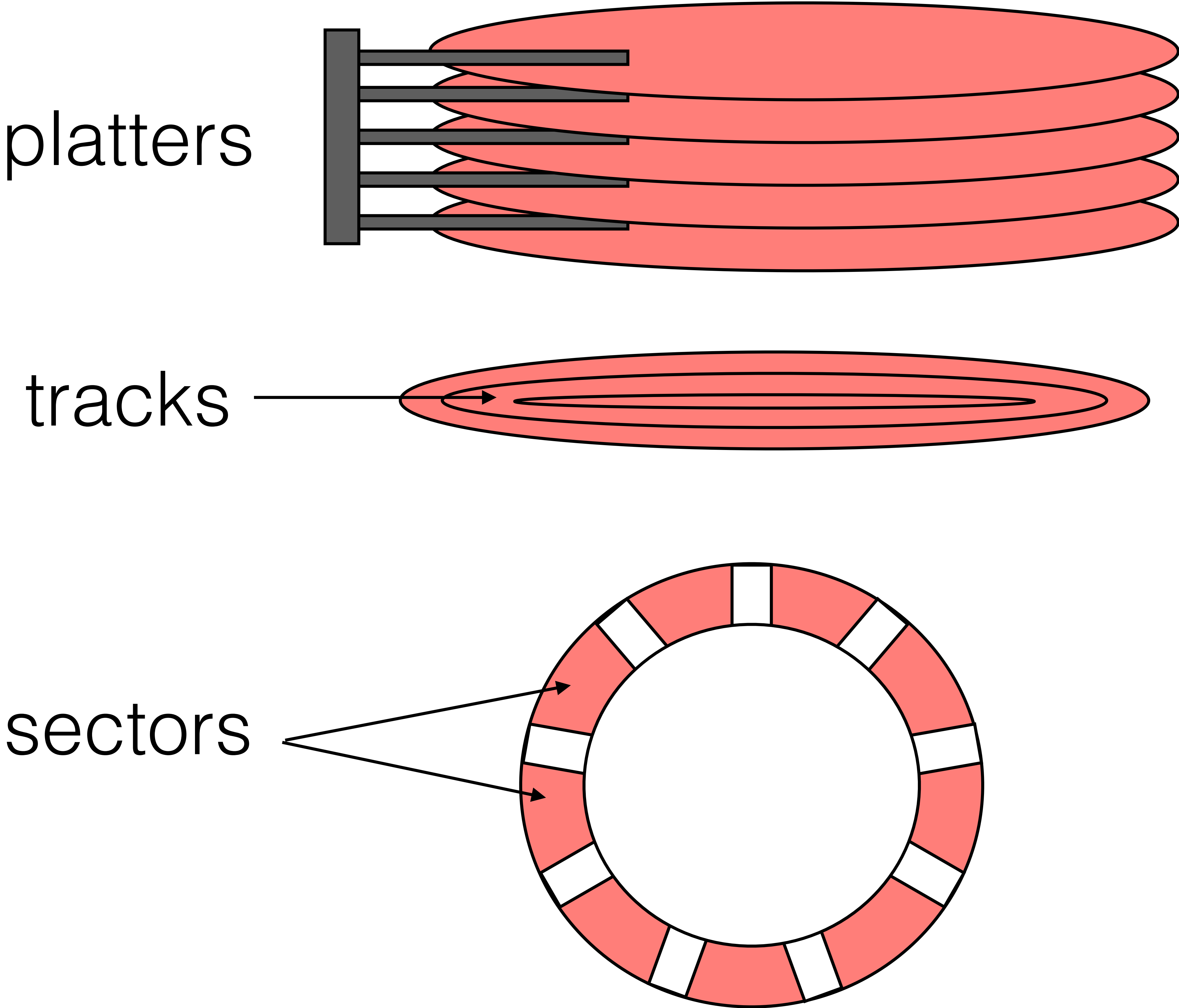




our general approach to improving performance is to measure our systems to find a **bottleneck**, and then to relax the bottleneck with general techniques such as caching, parallelism, etc.

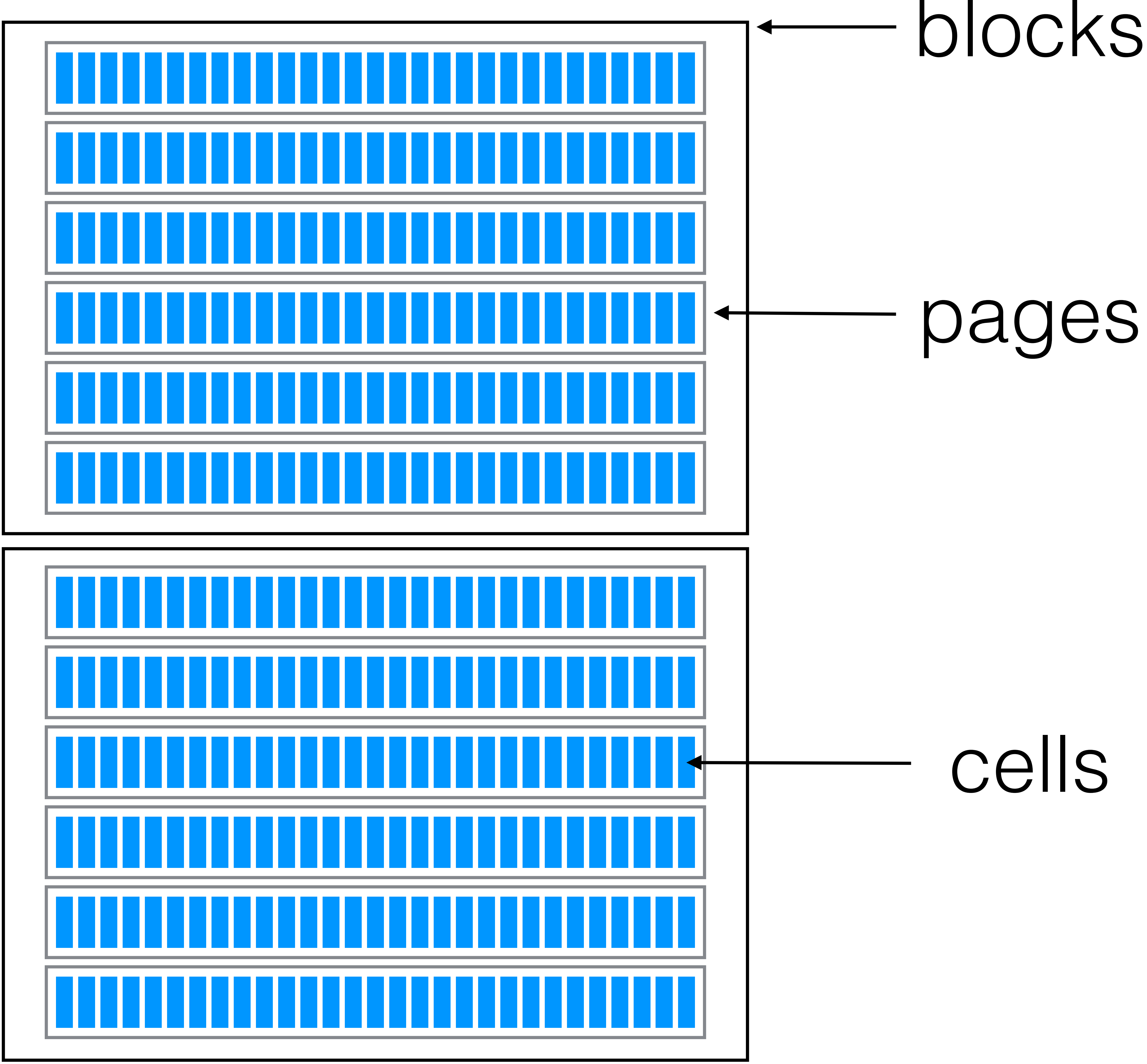
we'll make this concrete with an example: performance in reading/writing to a file

the **disk** is often the main bottleneck in reading/writing stored data



hard disk drives (HDDs)

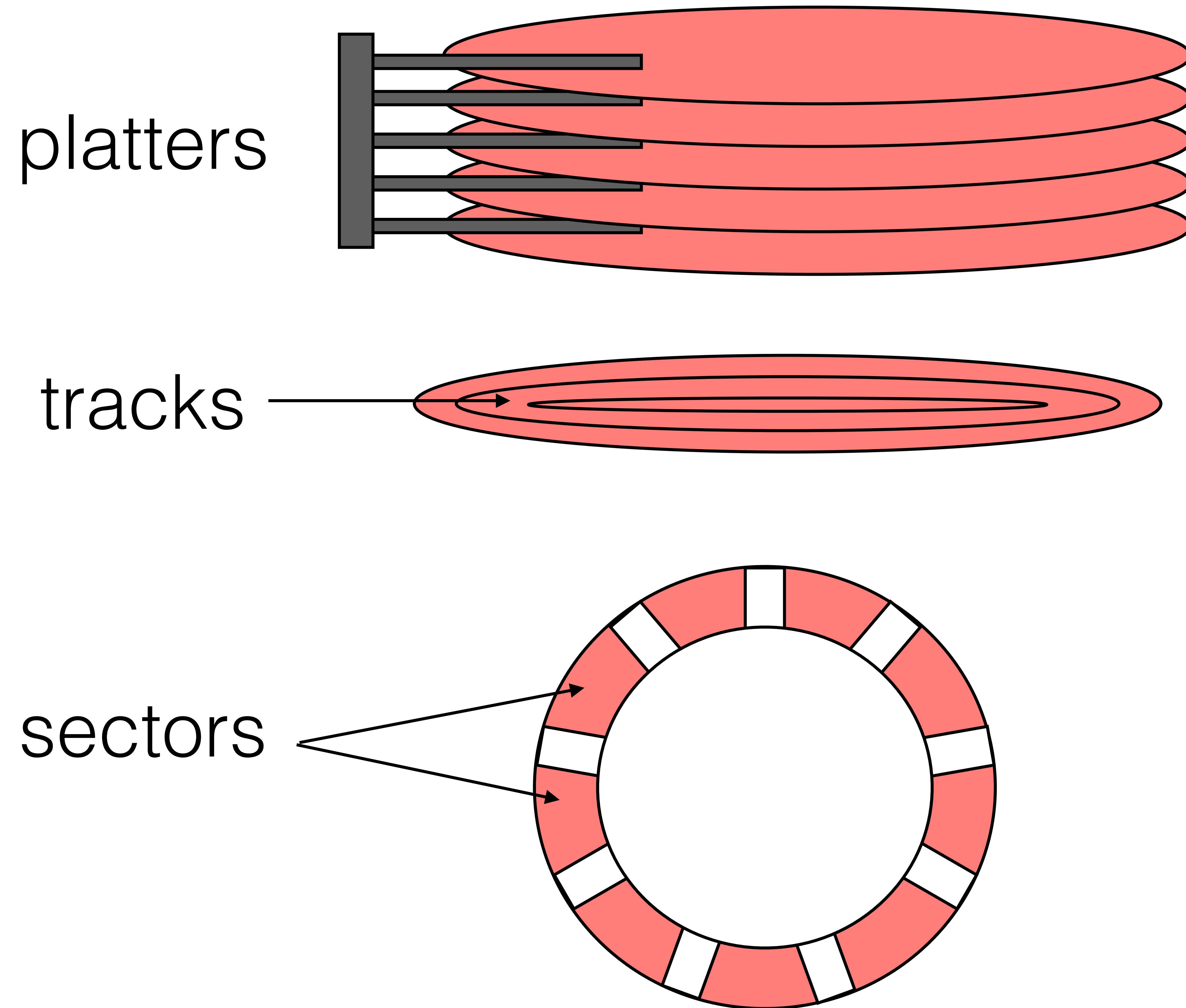
common in datacenters



solid state drives (SSDs)

common in personal computers

the **disk** is often the main bottleneck in reading/writing stored data



hard disk drives (HDDs)

common in datacenters

example HDD specs (Hitachi 7K400)

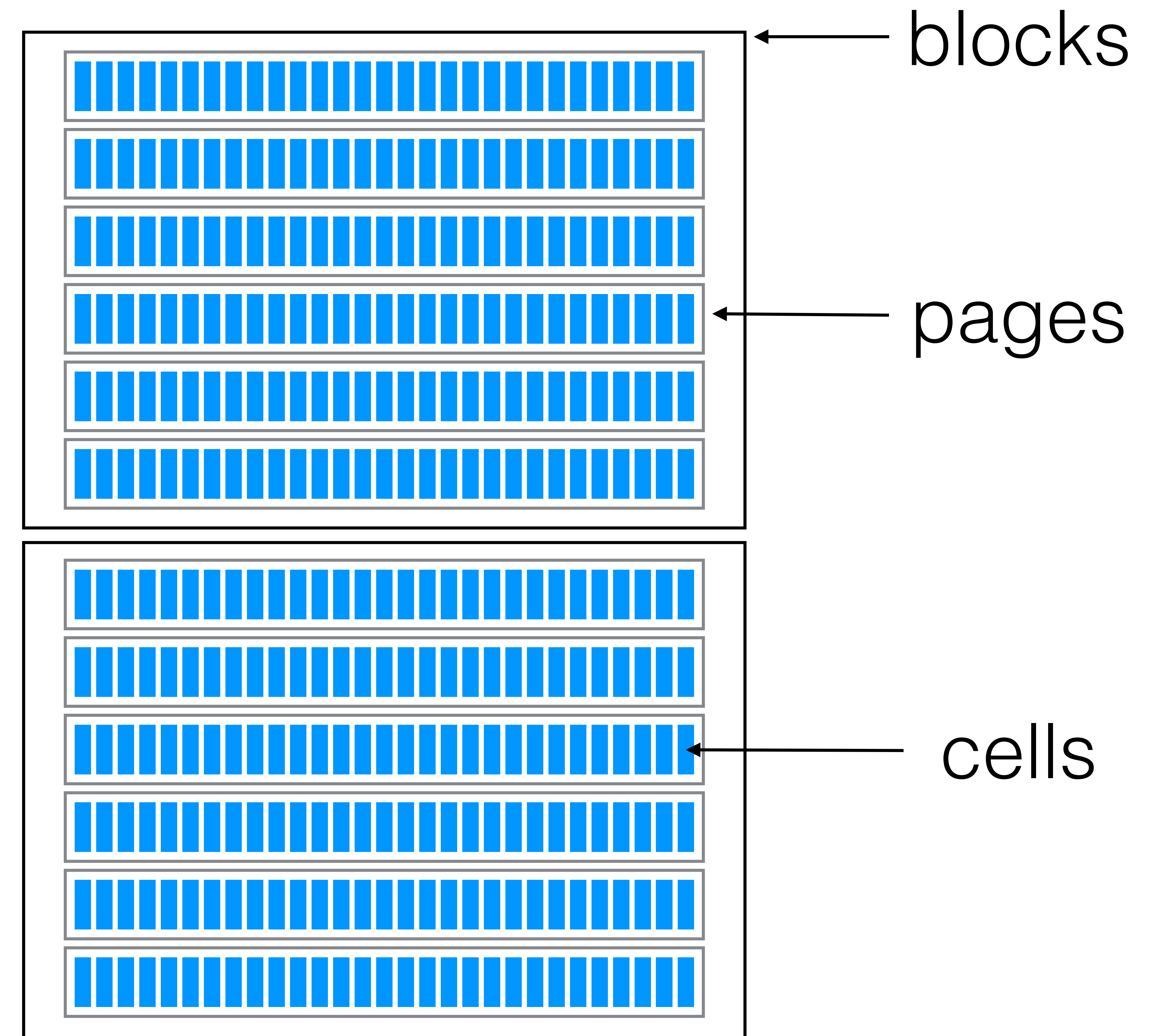
```
capacity: 400GB
number of platters: 5
number of heads: 10
number of sectors per track: 567-1170
number of bytes per sector: 512
time for one revolution: 8.3ms
average read seek time: 8.2ms
average write seek time: 9.2ms
```

since so much time of reading/writing is spent seeking, avoiding random access can improve performance

the **disk** is often the main bottleneck in reading/writing stored data

since SSDs don't involve moving parts, disk seeks are not a concern (this is one of the reasons SSDs are so much faster than HDDs)

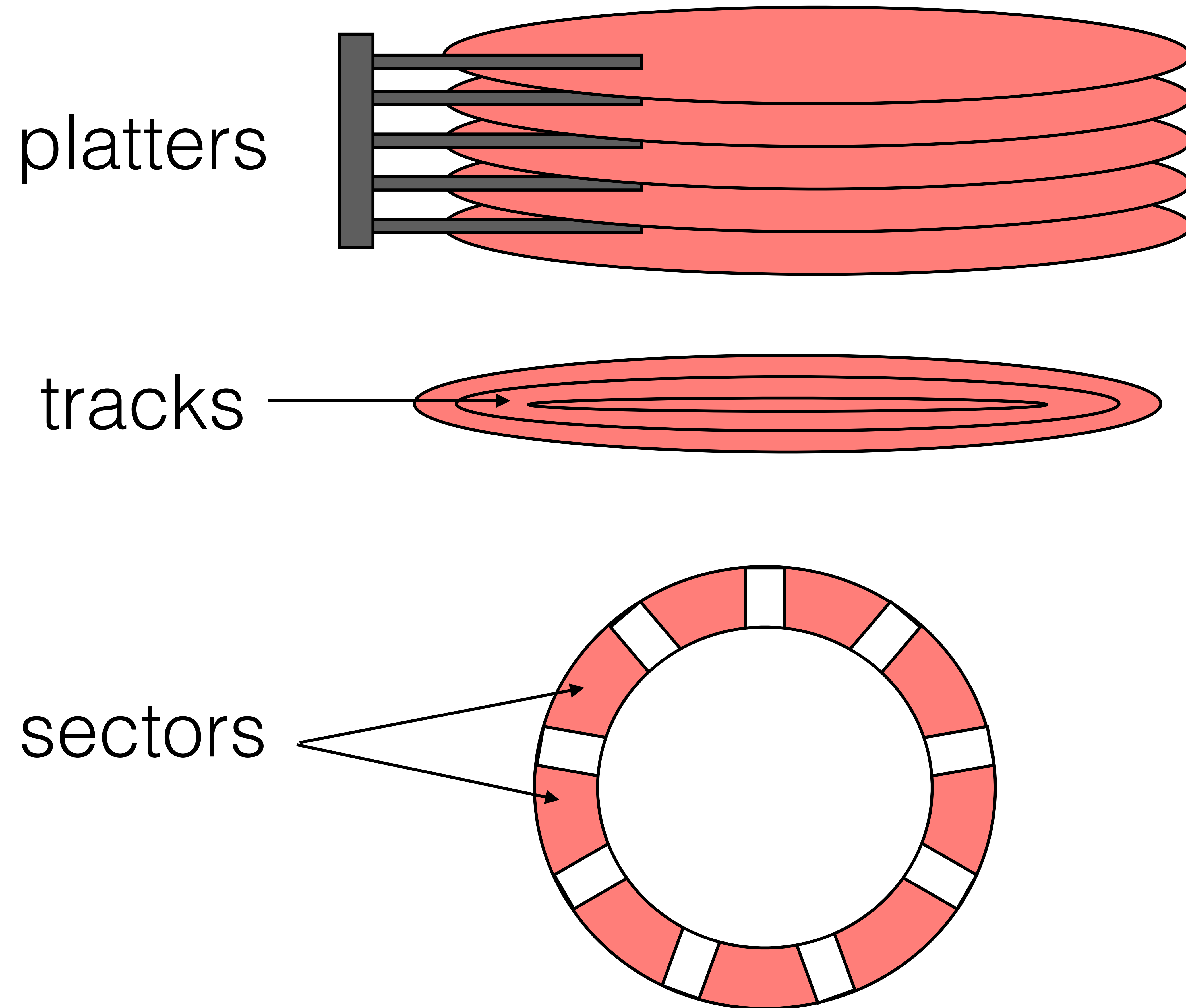
however, because of how writes are done, the SSD controller is careful about how it writes new data and makes changes to existing data



solid state drives (SSDs)

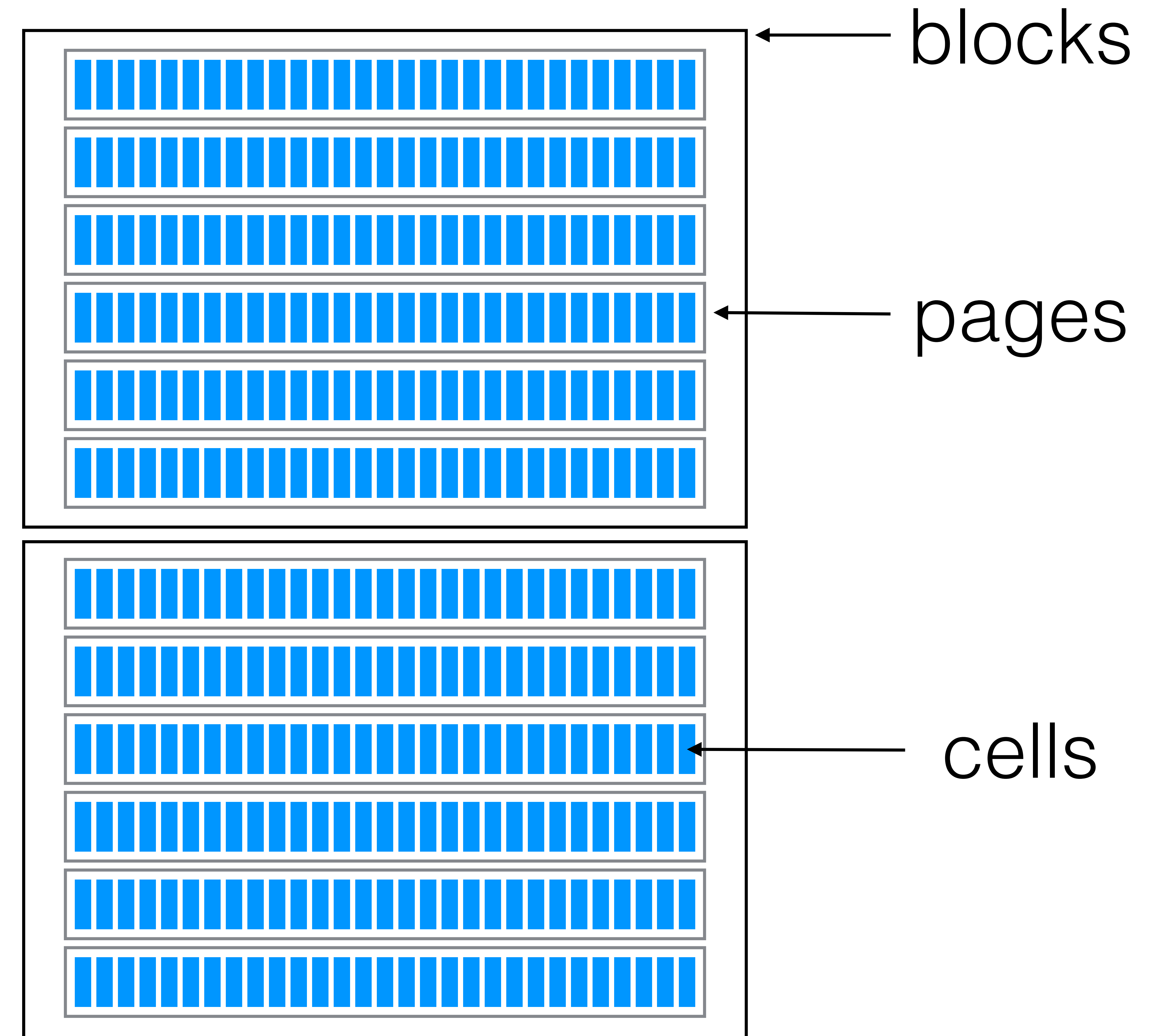
common in personal computers

so far, we have always imagined reading/writing data via the abstraction of a **filesystem**.
does that abstraction ever get in the way?



hard disk drives (HDDs)

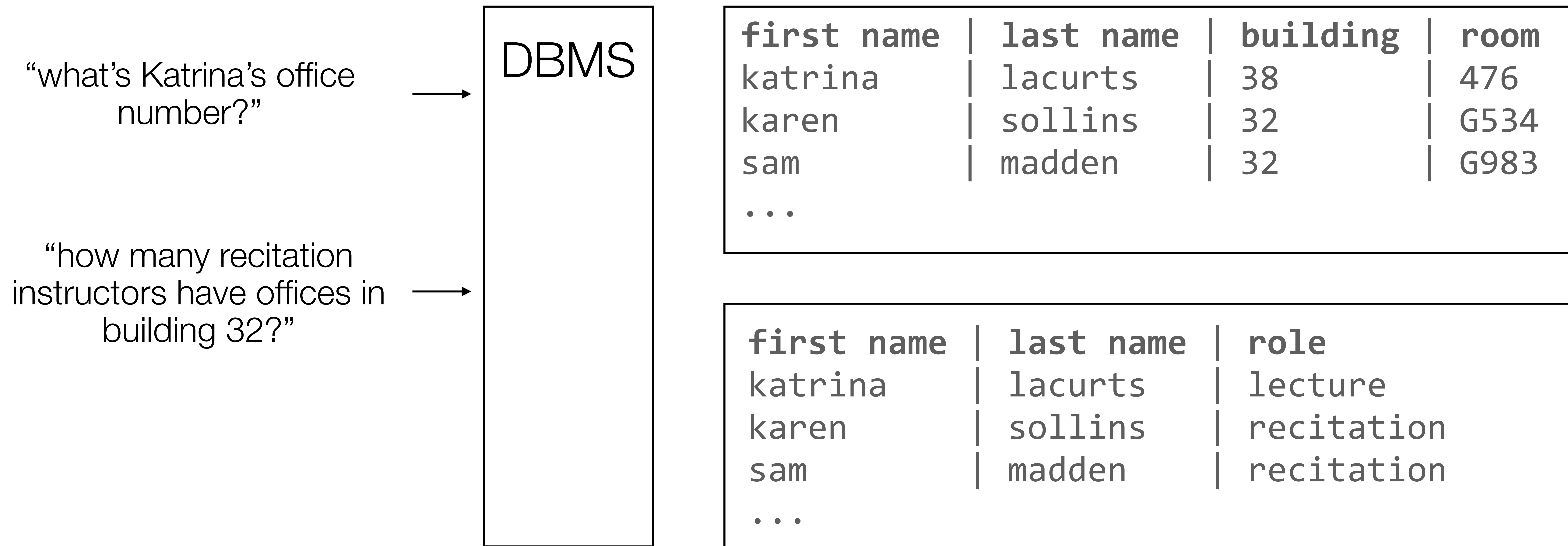
common in datacenters



solid state drives (SSDs)

common in personal computers

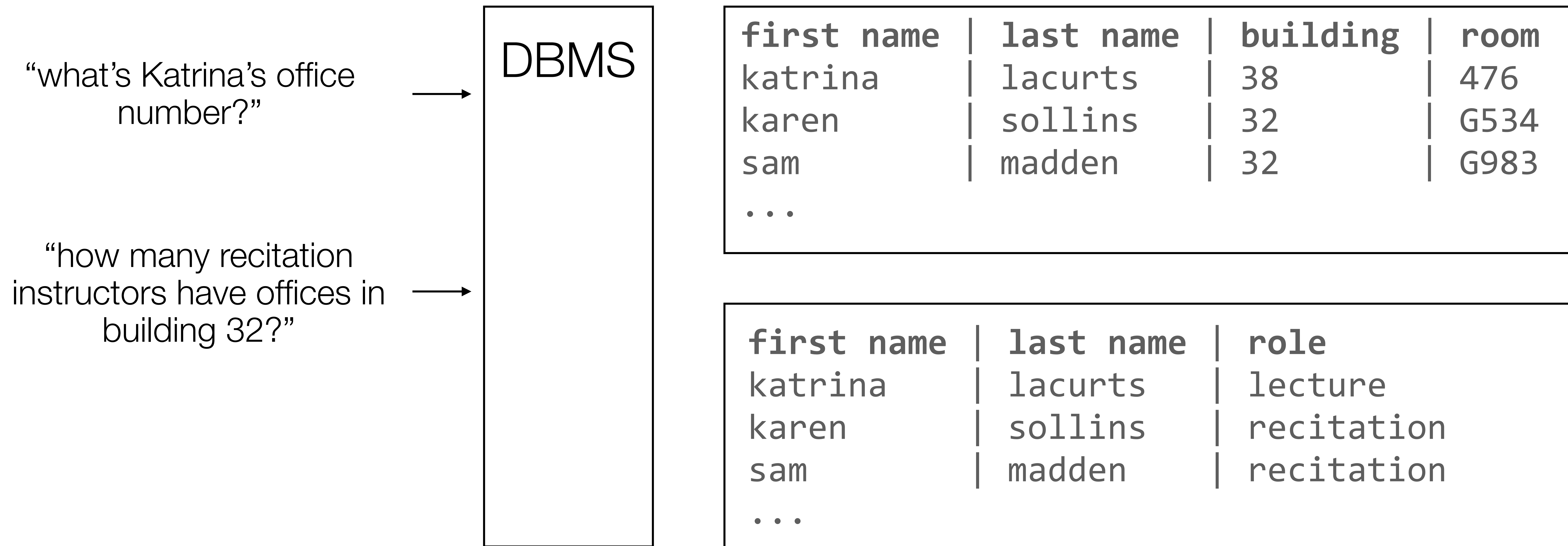
so far, we have always imagined reading/writing data via the abstraction of a **filesystem**.
does that abstraction ever get in the way?



how should the data be stored as files?

one file for everything? one file per table? per row? per column? per cell?

so far, we have always imagined reading/writing data via the abstraction of a **filesystem**. does that abstraction ever get in the way?



the DBMS knows so much about the data and related queries that it can do a very good job at predicting which byte it needs next

it's in a good position to exploit block-level control over loading or evicting data to memory

performance is important throughout systems. we often measure throughput, latency, and utilization, and use techniques such as caching and batching to improve performance

in reading/writing files, the **disk** is often the bottleneck. performance changes dramatically depending on the pattern of reads/writes (e.g., random vs. sequential access)

abstractions such as the **filesystem** work well in many places, but sometimes get in the way, especially when it comes to performance. **block-level control** can make sense for certain applications, such as **databases**

block-level control isn't perfect for *every* type of database; some do just fine with filesystems