Computer - A programmable electronic device that can store, retrieve, and process digital data
Hardware - The electronic machinery (wires, circuits, etc.)
Software - Programs (instructions) and data
Key Parts of Computer Hardware:
Processor - hardware that executes instructions
Main Memory (DRAM)- hardware that stores data and programs, byte-level addressing
Disk - Similar to memory, but persistent, slower, and higher capacity
Network interface controller(NIC) - sends and retrieves data over the network
Key Aspects of Software:
Instruction - command understood by the hardware
Program - A collection of instructions for hardware and to execute
Programming language - A human-readable formal language to write program
Application Programming Interface(API) - set of programs for use
Data - Digital representation of information that is stored, processed
Main Kinds of Software:
Firmware - Read-only programs that offer basic hardware control
Operating System(OS) - A collection of interrelated programs that enable application/software to use hardware easily. Ex. Windows, MacOS, Linux
Application Software - A program to manipulate data (Excel, Chrome, PostgreSQL)
Data Systems Infrastructure:
Data acquisition/preparation: Python, scikit-learn, R
Feature Engineering Training & Inference Model Selection: TensorFlow, PyTorch
Serving/Monitoring: Dask, Spark, AWS
Data Representation of Data:
Bits: All digital data are sequences of 0 & 1 (binary digits)
Amenable to high-low/off-on electromagnetism
Layers of abstraction to interpret bit sequences
Given k bits, we can represent 2^k unique data items
Data type: First layer of abstraction to interpret a bit sequence with a human-understandable category of information(common: Boolean, Byte, Integer)
Example: Boolean, Byte, Integer, "floating point" number (Float), Character, and String
Data structure: A second layer of abstraction to organize multiple instances of data types as a more complex object with specified properties
Examples: Array, Linked list, Tuple, Graph, etc.
A Byte(8 bits): the basic unit of data types
Represents 2^8 unique data items
$Ceil(Log_2(k))$ bits needed to distinguish k data items

Boolean:
E.g: Y/N or T/F responses
Just 1 bit needed
Actual size is almost always 1B, i.e., 7 bits are wasted!
Extra 7 bits for accessing information
Integer:
E.g: # of friends, age, # of likes
Typically 4 bytes; Many variants (short, unsigned, etc.)
Java int can represent -231 to (231 - 1)
C unsigned int can represent 0 to (232 - 1)
Python3 int is effectively unlimited length (PL magic!)
Hexadecimal representation: more succinct and readable
Base 16 instead of base 2 cuts display length by ~4x
Digits are 0, 1, ... 9, A (10_{10}), B, … F (15_{10})
Each hexadecimal digit represents 4 bits
From Hexadecimal to binary: 2F → 0010 1111

Decimal	Binary	Hexadecimal
5_{10}	101_2	5_{16}
47_{10}	$10\ 1111_2$	$2F_{16}$
163_{10}	$1010\ 0011_2$	$A3_{16}$
16_{10}	$1\ 0000_2$	10_{16}

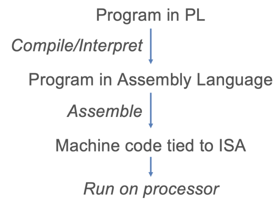
Alternative notations
0xA3 or $A3_H$

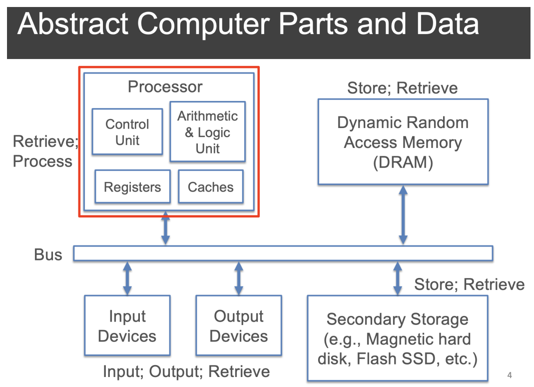
Floats: half(2B); single(4B); double(8B)
E.g: salary, scores, model weights
Single-precision: 4B long; ~8 decimal digits (Java, C)
Double-precision: 8B long; ~16 decimal digits (Python)
Floating point arithmetic (addition and multiplication) is not associative
Exponent 0xFF and fraction 0 is +/- "Infinity"
Exponent 0xFF and fraction <> 0 is "NaN"
Character(Char):
E.g: Represents letters, numerals, punctuations, etc
1 Byte in C, 2 Byte in Java
Python does not have a char type, use str or bytes
String: variable sized array of char
Digital object: Collections of basic data types(string, array, set, bytes, integers, floats, and characters)

- SQL dates/timestamp: string (w/ known format)
- ML feature vector: *array* of floats (w/ known length)
- Neural network weights: *set* of multi-dimensional *arrays* (matrices or tensors) of floats (w/ known dimensions)
- Graph: an *abstract data type* (ADT) with *set* of vertices (say, integers) and *set* of edges (*pair* of integers)
- Program in PL, SQL query: string (w/ grammar)
- DRAM addresses: *array* of bytes (w/ known length)
- Instruction in machine code: *array* of bytes (w/ ISA)

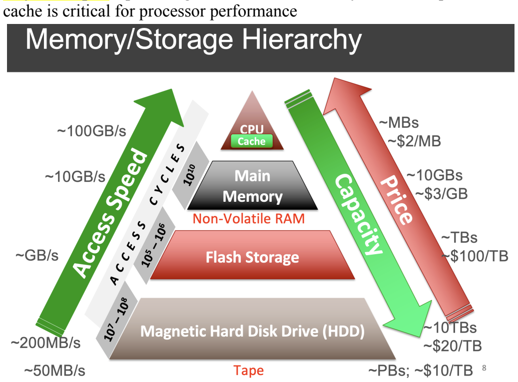
Serialization: The process of converting a data structure (or program objects in general) into a neat sequence of bytes that can be exactly recovered
Serializing bytes and characters/strings is trivial
2 alternatives for serializing integers/floats:
As byte stream (aka "binary type" in SQL)
As string, e.g., 4B integer 5 -> 2B string as "5"
String ser. common in data science (CSV, TSV, etc.)

We often convert a trained model into a format that can be stored or transmitted. This involves transforming it into a sequence of bytes that can be written to disk or sent over network (i.e. we have to serialize it)
We can serialize any other ML related artifacts like transformers, data, metadata, etc.
Deserialization: the process of converting a serialized model back to its original data structure to be used for inference.
We load it back into memory for inference or evaluation purposes
Can be implemented in various formats, such as JSON, protocol buffers, or Apache Avro.
Basics of Processors:
Processor: Hardware to orchestrate and execute instructions to manipulate data as specified by a program
Examples: CPU, GPU, FPGA, TPU, embedded, etc.
Instruction Set Architecture (ISA):
The vocabulary of commands of a processor
Specifies bit length/format of machine code commands
Has several commands to manipulate register contents

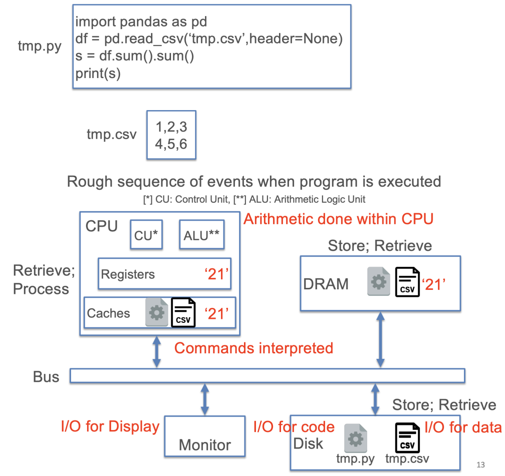




Load-store architecture - How processor executes machine code
Register: Tiny local memory ("scratch space") on processors into which instructions and data are copied
Caches: Small local memory to buffer instructions/data
Types of ISA commands to manipulate register contents:
Memory access: load (copy bytes from DRAM address to register); store (reverse); put constant
Arithmetic & logic on data items in registers (ALU): add/multiply/etc.; bitwise ops; compare, etc.
Control flow (branch, call, etc.)
Processor Performance
Modern CPUs can run millions of instructions per second
ISA influences #clock cycles each instruction needs
CPU's clock rate lets us convert that to runtime (ns)
Most programs do not keep the CPU always busy
Memory access commands stall the processor
Worse, data may not be in DRAM—wait for disk I/O!
Actual execution runtime of program may be orders of magnitude **higher** than what clock rate calculation suggests
The arithmetic & Logic Unit and Control Unit are idle during memory-register transfer
Key Principle: Optimizing access to main memory and use of processor cache is critical for processor performance



→ Due to **OOM access latency differences across memory hierarchy**, optimizing access to lower levels and careful use of higher levels is critical for overall system performance!
Locality of Reference: Many programs tend to access memory locations in a somewhat predictable manner
Spatial: Nearby locations will be accessed soon
Temporal: Same locations accessed again soon
Locality can be exploited to reduce runtimes using caching and/or prefetching across all levels in the hierarchy
Concepts of Memory Management
Caching: Buffering a copy of bytes from lower level at higher level to exploit locality
Prefetching: Preemptively retrieving bytes (typically data) from addresses not explicitly asked yet by program
Spill/Miss/Fault: Data needed for program is not yet available at a higher level; need to get it from lower level
Register Spill(register to cache);
Cache Miss(cache to main memory)
"Page" Fault (main memory to disk)
Hit: Data needed is already available at higher level
Cache Replacement Policy: Policies of when new data needs to be loaded to a higher level, which old data to evict to make room? Many policies exist with different properties
Memory Hierarchy in Action



Locality of Reference in Data Science:
Data Layout: The order in which data items of a complex data structure or an abstract data type (ADT) are laid out in memory/disk
Data Access Pattern (of a program on a data object): The order in which a program has to access items of a complex data structure in memory
Hardware Efficiency (of a program):
How close actual execution runtime is to best possible runtime given the CPU clock rate and ISA
Improved with careful data layout of all data objects used by a program based on its data access patterns
Key Principle: Raise cache hits; reduce memory stalls!

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

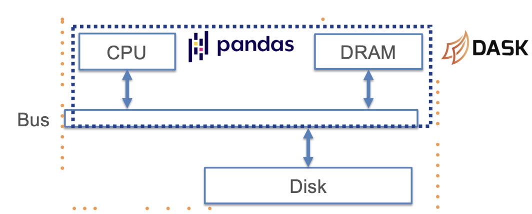
```
for i = 1 to n
  for j = 1 to m
    for k = 1 to p
      C[i][j] += A[i][k] * B[k][j]
```

Rewrite

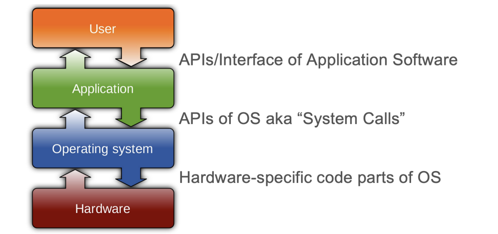
```
for i = 1 to n
  for k = 1 to p
    for j = 1 to m
      C[i][j] += A[i][k] * B[k][j]
```

- Although the math is the same and gives the same results ("logically equivalent"), the physical properties of program execution are vastly different
- Commonly used in *compiler optimization* and later on, also in *query optimization*

The first one B[k][j] misses; each * op is a stall!
Matrices/tensors are ubiquitous in statistics/ML/DL programs
Decades of optimized hardware-efficient libraries exist for matrix/tensor arithmetic (linear algebra) that reduce memory stalls and increase parallelism (more on parallelism later)
Multi-core CPUs: BLAS/LA PACK (C), Eigen (C++), la4j (Java), NumPy/SciPy (Python; can wrap BLAS)
GPUs: cuBLAS, cuSPARSE, cuDNN, cuDF, cuGraph
Memory Hierarchy in PA0
Pandas DataFrame needs data to fit entirely in DRAM
Dask DataFrame automatically manages Disk vs DRAM for u
Full data sits on Disk, brought to DRAM upon compute()
Dask stages out computations using Pandas
Tradeoff: Dask may throw memory configuration issues



Role of an OS in a Computer
An OS is a large set of interrelated programs that make it easier for applications and user-written programs to use computer hardware effectively, efficiently, and securely
Without OS, computer users must speak machine code
2 key principles in OS (any system) design & implementation:
Modularity: Divide system into functionally cohesive components that each do their jobs well
Orchestra example: Consider a conductor orchestrating different sections
Abstraction: Layers of functionalities from low-level (close to hardware) to high level (close to user)
Car example: A pedal to transmission to engine to wheels

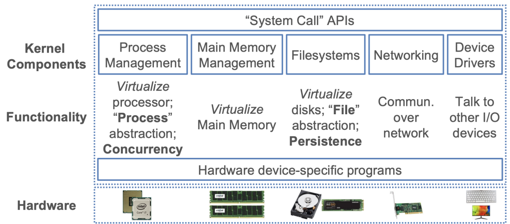


"Application Software" notion is now more complex due to multiple tiers of abstraction; "Platform Software" or "Software Framework" is a new tier between "Application" and OS

Key Components of OS
API of OS called "System Call"

Kernel: The core of an OS with modules to abstract the hardware and APIs for programs to use

Auxiliary parts of OS include shell/terminal, file browser for usability, extra programs installed by I/O devices, etc.



The Abstraction of a Process
Process Management: Virtualize processor 'process abstraction; concurrency'

Main Memory Management: virtualize main memory

Filesystems: virtualize disk; "file" abstraction

Networking: Communication over network

Device Drivers: Talk to other I/O devices

Process: A running program, the central abstraction in OS

Started by OS when a program is executed by user

OS keeps inventory of "alive" processes (Process List) and handles apportioning of hardware among processes

A **query** is a program that becomes a process

A data system typically abstracts away process management because user specifies the queries/processes in system's API

High-level steps OS takes to get a process going:

1. Create a process (get Process ID; add to Process List)
2. Assign part of DRAM to process, aka its Address Space
3. Load code and static data (if applicable) to that space
4. Set up the inputs needed to run program's main()
5. Update process' State to Ready
6. When the process is scheduled (Running), the OS temporarily hands off control to the process to run the show!
7. Eventually, process finishes or run Destroy

Virtualization of Hardware Resources
OS has **mechanisms** and policies to regain control

Virtualization: Each hardware resource is treated as a virtual entity that OS can divide up and share among processes in a controlled way

Limited Direct Execution:

OS mechanism to time-share CPU and preempt a process to run a different one, aka "context switch"

A Scheduling policy tells OS what time-sharing to use

Processes also must transfer control to OS for "privileged" operations (e.g., I/O); System Calls API

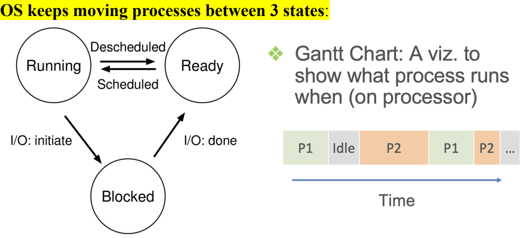
Virtualization of Processors:

Virtualization of processor enables process isolation (i.e., each process given an "illusion" that it alone runs)

Inter-process communication possible in System Calls API

Later: Generalize to Thread abstraction for concurrency

Process Management by OS
OS keeps moving processes between 3 states:



Sometimes, if a process gets "stuck" and the OS does not schedule something else, the system hangs; it needs to reboot!

Scheduling Policies/Algorithms

Schedule: Record of what process runs on each CPU & when Policy controls how OS time-shares CPUs among processes

Key terms for a process (aka job):

Arrival Time: Time when process gets created

Job Length: Duration of time needed for process

Start Time: Times when process first starts on processor

Completion Time: Time when process finishes/killed

Response Time = [Start Time] – [Arrival Time]

Turnaround Time = [Completion Time] – [Arrival Time]

Workload: Set of processes, arrival times, and job lengths that OS Scheduler has to handle

In general, the OS may not know all Arrival Times and Job Lengths beforehand! But **preemption** is possible

Key Principle: Inherent tension in scheduling between overall workload performance and allocation fairness

Performance metric is usually Average Turnaround Time

Fairness: Many metrics exist (e.g., Jain's fairness index)

100s of scheduling policies studied!

We will be overviewing some well-known ones:

FIFO (First-In-First-Out)

SJF (Shortest Job First)

SCTF (Shortest Completion Time First)

Round Robin

Random, etc.

Different criteria for ranking; preemptive vs not

Complex "multi-level feedback queue" schedulers

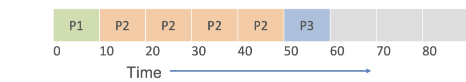
ML-based schedulers are "hot" nowadays!

First-In-First-Out aka First-Come-First-Served (FCFS)

Ranking criterion: Arrival Time; no preemption allowed

Main con: Short jobs may wait a lot, aka "Convoy Effect"

Example: P1, P2, P3 of lengths 10,40,10 units arrive closely in that order



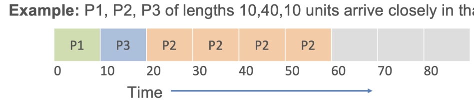
Process	Arrival t1	Start t2	Completion t3	Response t4 = t2-t1	Turnaround t5 = t3-t1
P1	0	0	10	0	10
P2	0	10	50	10	50
P3	0	50	60	50	60
			Avg:	20	40

Shortest Job First (SJF):

Ranking criterion: Job Length; no preemption allowed

Main con: Not all Job lengths might be unknown beforehand.

Example: P1, P2, P3 of lengths 10,40,10 units arrive closely in that order



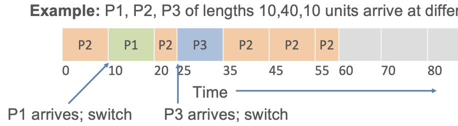
Process	Arrival t1	Start t2	Completion t3	Response t4 = t2-t1	Turnaround t5 = t3-t1
P1	0	0	10	0	10
P2	0	20	60	20	60
P3	0	10	20	10	20
(FIFO Avg: 20 and 40)			Avg:	10	30

Shortest Completion Time First (SCFT):

Ranking criterion: Jobs might not all arrive at same time; preemption possible

Main con same as SJF: Job lengths might be unknown beforehand

Example: P1, P2, P3 of lengths 10,40,10 units arrive at different times



Process	Arrival t1	Start t2	Completion t3	Response t4 = t2-t1	Turnaround t5 = t3-t1
P1	10	10	20	0	10
P2	0	0	60	0	60
P3	25	25	35	0	10
(SJF Avg: 10 and 30)			Avg:	0	26.7

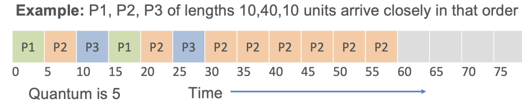
Round Robin:

In Round Robin job lengths need not be known

Ranking criterion: Fixed time quantum given to each job; cycle through jobs

Main con: RR is often very fair, **but** Avg Turnaround Time goes up

Example: P1, P2, P3 of lengths 10,40,10 units arrive closely in that order



Process	Arrival t1	Start t2	Completion t3	Response t4 = t2-t1	Turnaround t5 = t3-t1
P1	0	0	20	0	20
P2	0	5	60	5	60
P3	0	10	30	10	30
(SJF Avg: 10 & 30; SCTF Avg: 0 & 26.7)			Avg:	5	36.7

DRAM vs. Disk

DRAM is much faster, DRAM is volatile while disk is not, DRAM has less capacity. DRAM is more expensive.

Concurrency:

Modern computers often have multiple processors and multiple cores per processor

Concurrency: Multiple processors/cores run different/same set of instructions simultaneously on different/shared data

New levels of shared caches are added

Multiprocessing: Different processes run on different cores (or entire CPUs) simultaneously

Thread: Generalization of OS's Process abstraction

A program spawns many threads; each run parts of the program's computations simultaneously

Multithreading: Same core used by many threads

Issues in dealing with multithreaded programs that write shared data:

Cache coherence

Locking; deadlocks

Complex scheduling

Scheduling for multiprocessing/multicore is more complex

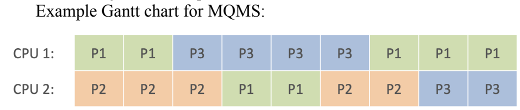
Load Balancing: Ensuring different cores/proc. are kept roughly equally busy, i.e., reduce idle times

Multi-queue multiprocessor scheduling (MQMS) is common

Each processor/core has its own job queue

OS moves jobs across queues based on load

Example Gantt chart for MQMS:



Thankfully, most data-intensive computations in data science do not need concurrent writes on shared data! Although we often need concurrent reads

Concurrent low-level ops abstracted away by libraries/APIs

Partitioning / replication of data simplifies concurrency

Later topic (Parallelism Paradigms) will cover parallelism in depth:

Multi-core, multi-node, etc.

Task parallelism, Partitioned data parallelism, etc.

File and Directory:
File: A persistent sequence of bytes that stores a logically coherent digital object for an application

File Format: An application-specific standard that dictates how to interpret and process a file's bytes

1000s of file formats exist (e.g., TXT, DOC, GIF, MPEG); varying data models/types, domain-specific, etc.

Metadata: Summary or organizing info. about file content(aka payload)stored with file itself; format-dependent

Directory: A cataloging structure with a list of references to files and/or (recursively) other directories

Typically treated as a special kind of file.

Sub-dir., Parent dir., Root dir.

Filesystem
Filesystem: The part of OS that helps programs create, manage, and delete files on disk (secondary storage)

Roughly split into **logical level and physical level:**

Logical level exposes file and directory abstractions and offers System Call APIs for file handling

Physical level works with disk firmware and moves bytes to/from disk to DRAM

Dozens of filesystems exist, e.g., ext2, ext3, NTFS, etc.

Differ on:

how they layer file and directory abstractions as bytes, what metadata is stored, etc.

how data integrity/reliability is assured, support for editing/resizing, compression/encryption, etc.

Some can work with (can be "mounted" by) multiple OSs.

OS abstracts a file on disk as a virtual object for processes

File Descriptor: An OS-assigned positive integer identifier/ reference for a file's virtual object that a process can use:

0/1/2 reserved for STDIN/STDOUT/STDERR

File Handle: A PL's abstraction on top of a file descriptor (fd)

System Call API for File Handling:

open(): Create a file; assign fd; optionally overwrite

read(): Copy file's bytes on disk to in-mem. buffer

write(): Copy bytes from in-mem. buffer to file on disk

fsync(): "Flush" (force write) "dirty" data to disk

close(): Free up the fd and other OS state info on it

lseek(): Position offset in file's fd (for random read/write later)

Dozens more (rename, mkdir, chmod, etc.)

Files v.s Databases: Data Mode
Database: An organized collection of interrelated data

Data Model: An abstract model to define organization of data in a formal (mathematically precise) way

E.g., Relations, XML, Matrices, DataFrames

Every database is just an abstraction on top of data files:

Logical level: Data model for higher-level reasoning

Physical level: How bytes are layered on top of files

All data systems (RDBMSs, Dask, Spark, PyTorch, etc.) are application/platform software that use OS System Call API for handling data files

Data as File: Structured
Structured Data: A form of data with regular substructure

Relational Database, Matrix, Tensor, DataFrame, sequence:

Matrix and DF have row/col numbers, relation is orderless (TSV, CSV)

Transpose support only by Matrix, DF

Most RDBMSs and Spark serialize a relation as binary file(s), often compressed

Different RDBMSs and Spark/HDFS-based tools serialize relation/tabular data in different binary formats, often compressed:

One file per relation; row vs columnar (e.g., ORC, Parquet) vs hybrid formats

RDBMS vendor-specific vs open Apache

Parquet becoming especially popular

Comparing Structured Data Models
Ordering: Matrix and DataFrame have row/col numbers; Relation is orderless on both axes!

Schema Flexibility: Matrix cells are numbers. Relation tuples conform to pre-defined schema. DataFrame has no pre-defined schema but all rows/cols can have names; col cells can be mixed types!

Transpose: Supported by Matrix & DataFrame, not Relation

Semistructured Data: A form of data with less regular / more flexible substructure than structured data

Tree-Structured:

Typically serialized as a restricted ASCII text file (extensions XML, JSON, YML, etc.)

Some data systems also offer binary file formats

Can layer on Relations too

Graph-Structured:

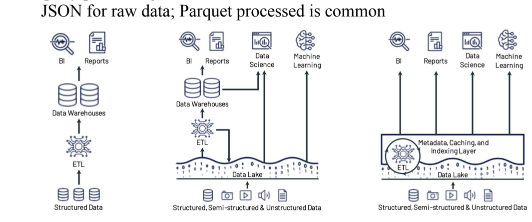
Typically serialized with JSON or similar textual formats

Some data systems also offer binary file formats

Again, can layer on Relations too

Unstructured: Data Files on Data "Lakes"
Data "Lake": Loose coupling of data file format for storage and data/query processing stack

JSON for raw data; Parquet processed is common



ETL: Extract, Transform, Load

Tradeoffs: Pros and cons of Parquet vs text-based files (CSV, JSON, etc.):
Less storage: Parquet stores in compressed form; can be much smaller (even 10x); less I/O to read
Column pruning: Enables app to read only columns needed to DRAM; even less I/O now!
Schema on file: Rich metadata, stats inside format itself
Complex types: Can store them in a column
Human-readability: Cannot open with text apps directly
Mutability: Parquet is immutable/read-only; no in-place edits
Decompression/Deserialization overhead: Depends on application tool
Adoption in practice: CSV/JSON support more pervasive but Parquet is catching up, especially in enterprise "big data" situations

Data as File: Other Common Formats
Machine Perception data layer on tensors and/or time-series
Myriad binary formats, typically with (lossy) compression, e.g., WAV for audio, MP4 for video, etc.
Text File (aka plaintext): Human-readable ASCII characters
Docs/Multimodal File: Myriad app-specific rich binary formats

Virtualization of GRAM with Pages
Page: An abstraction of fixed size chunks of memory/storage
Makes it easier to virtualize and manage DRAM
Page Frame: Virtual slot in DRAM to hold a page's content
Page size is usually an OS configuration parameter
E.g., 4KB to 16KB
OS Memory Management has mechanisms to:
Identify pages uniquely (page frame 0 for OS)
Read/write page from/to disk when requested by a process

Apportioning of DRAM: Elements
A process's Address Space:
Slice of virtualize DRAM assigned to it alone!
OS "translates" DRAM vs disk address
Page Replacement Policy:
When DRAM fills up, which cached page to evict?
Many policies in OS literature
Memory Leaks:
Process forgot to "free" pages used a while ago
Wastes DRAM and slows down system
Garbage Collection:
Some PL implementations can auto-reclaim some wasted memory

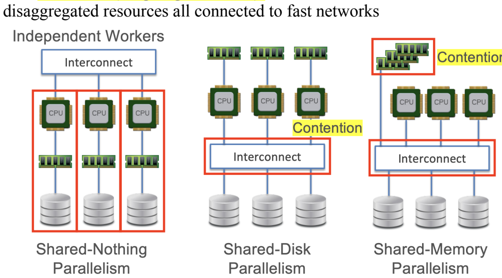
Storing Data In Memory
Any data structure in memory is overlaid on pages
Process can ask OS for more memory in System Call API
If OS denies, process may crash
Apache Arrow:
Emerging standard for columnar in-memory data layout
Compatible with Pandas, (Py)Spark, Parquet, etc.
Persistent Data Storage
Hard Disk, CD, SSDs
SSDs has a key latency dichotomy for random vs. sequential data
Volatile Memory: A data storage device that needs power/electricity to store bits; e.g., DRAM, CPU caches (SRAM)
Persistence: Program state/data is available intact even after process finishes
Non-Volatile or Persistent memory/storage: A data storage device that retains bits intact after power cycling
E.g., all levels below DRAM in memory hierarchy
"Persistent Memory (PMEM)": Marketing term for large DRAM that is backed up by battery power!
Non-Volatile RAM (NVRAM): Popular term for DRAM-like device that is genuinely non-volatile (no battery)
Note: PMEM and NVRAM are typically used in high-performance servers and storage systems where fast, reliable access to data is critical.

Disk and Data Organization on Disk
Disk: Aka secondary storage; likely holds the vast majority of the world's day-to-day business-critical data!
Data storage/retrieval units: disk blocks or pages
Unlike RAM, different disk pages have different retrieval times based on location:
Need to optimize layout of data on disk pages
Orders of magnitude performance gaps possible
Disk space is organized into **files**
Files are made up of disk pages aka blocks(basic unit)
Typical disk block/page size: 4KB or 8KB:
Basic unit of reads/writes for a disk
OS/RAM page is not the same as disk page!
Typically, [OS/RAM page size] = [Disk page size] but not always; disk page can be a multiple, e.g., 1MB
File data (de-)allocated in increments of disk pages

Magnetic Hard Disks
Key Principle: Sequential vs. Random Access Dichotomy
Accessing disk pages in sequential order gives higher throughput
Random reads/writes are OOM slower!
Need to carefully lay out data pages on disk, not the case for DRAM
Abstracted away by data systems: Dask, Spark, RDBMSs, etc.
Flash SSD vs. Magnetic Hard Disks
Random reads/writes are not much worse
Different locality of reference for data/file layout
But still block-addressable like HDDs
Data access latency: 100x faster! (Note: Access ~ Lookup)
Data transfer throughput: Also 10-100x higher (Note: Access ~ Read/Write
Parallel read/write more feasible
Cost per GB is 5-15x higher!
Read-write impact asymmetry; much lower lifetimes
NVRAM vs. Magnetic Hard Disks
NVRAM is like a non-volatile form of DRAM, but with similar capacity as SSDs
Random R/W with less to no SSD-style wear and tear
Byte-addressability (not blocks like SSDs/HDDs)
Spatial locality of reference like DRAM; radical change!
Latency, throughput, parallelism, etc. similar to DRAM
Alas, limited to HPC and enterprise environments

Cloud computing
Cloud: shared-Disk, shared-memory, [shared nothing]
Compute, storage, memory, networking, etc. are virtualized and exist on remote servers; rented by application users
Main pros of cloud vs on-premise clusters:
Manageability: Managing hardware is not user's problem
Pay-as-you-go: Fine-grained pricing economics based on actual usage (granularity: seconds to years!)
Elasticity: Can dynamically add or reduce capacity based on actual workload's demand
Infrastructure-as-a-Service (IaaS) (IT Administrators):
Compute:
Elastic Compute Cloud (EC2) (PA)
Elastic Container Service (ECS)
Serverless compute engines:
Fargate (serverless containers),
Lambda (serverless functions)
Storage:
Simple storage service (S3)
Elastic Block Store (EBS)
Elastic File System (EFS)
Glacier (storage classes)
Networking:
CloudFront (low latency content delivery)
Virtual Private Cloud (VPC)
Platform-as-a-Service (PaaS) (Software Developer):
Database/Analytics Systems:
Aurora, Redshift, Neptune, ElastiCache, DynamoDB, Timestream, EMR, Athena
Blockchain: QLDB
IoT: Greengrass
ML/AI: SageMaker* (both Paas and SaaS)
Software-as-a-Service (SaaS) (End-user):
ML/AI: SageMaker*, Elastic Inference, Lex, Polly, Translate, Transcribe, Textract, Rekognition, Ground Truth
Business Apps: Chime, WorkDocs, WorkMail

Evolution of Cloud Infrastructure:
Data Center: Physical space from which a cloud is operated
3 generations of data centers/clouds:
Cloud 1.0 (Past): Networked servers; user rents servers (timesliced access) needed for data/software
Cloud 2.0 (Current): "Virtualization" of networked servers; user rents amount of resource capacity; cloud provider has a lot more flexibility on provisioning (multi-tenancy, load balancing, more elasticity, etc.)
Cloud 3.0 (Ongoing Research): "Serverless" and disaggregated resources all connected to fast networks



Most parallel RDBMSs (Teradata, Greenplum, Redshift), Hadoop, and Spark use shared-nothing parallelism

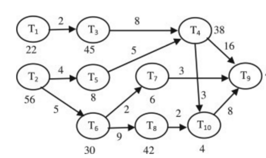
Modern networks in data centers have become much faster:
In terms of gigabit Ethernet connection speeds, one can find speeds in the order of magnitude 100GbE to even TbE!
Decoupling of compute+memory from storage is common in cloud
Hybrids of shared-disk parallelism+shared-nothing parallelism
E.g, store datasets on S3 and read as needed to local EBS

New Cloud Renting Paradigms
Cloud 2.0's flexibility enables radically different paradigms
AWS example below; Azure and GCP have similar gradations
Such bundling means some applications might under-utilize some resources!
Serverless paradigm gaining traction for some applications, e.g., online ML prediction serving on websites
User gives a program (function) to run and specifies CPU and DRAM needed
Cloud provider abstracts away all resource provisioning entirely
Higher resource efficiency; much cheaper, often by 10x vs Spot instances
Aka Function-as-a-Service (FaaS)
Logical next step in serverless direction: full resource disaggregation! That is, compute, memory, storage, etc. are all network-attached and elastically added/removed
Is all this complexity worth it?:
Depends on the user's/application's **Pareto tradeoffs!**
On-premise cluster are still common in large enterprises, healthcare, and academia; "hybrid clouds" too
Recall main pros of cloud: manageability, cost, and elasticity
Some **main cons of cloud (vs on-premise)**:
Complexity of composing cloud APIs and licenses; data scientists must keep relearning; "CloudOps" teams
Cost over time can crossover and make it costlier!
Easier to **waste money** accidentally on the fly
"Lock-in" by cloud vendor
Privacy, security, and governance concerns
Internet disruption or unplanned downtime, e.g., AWS outage in 2015 made Netflix, Tinder, etc. unavailable!
Layers of typical cloud: Compute, Storage, Networking
Spot vs On-Demand:
On-demand has static price
Need manual launch request
You can determine when to interrupt instance

Bias-Variance Tradeoff of ML
When prediction target complexity is high, more training data coupled with more complex models yield higher accuracy as number of training examples grows
High Bias: Roughly, model is not rich enough to represent data
High Variance: Model overfits to given data; poor generalization
Large-scale training data lowers variance and raises accuracy!
Why Large-Scale Data?
Large-scale data is a game changer in data science:
Enables study of granular phenomena in sciences, businesses, etc. not possible before
Enables new applications and personalization/customization
Enables more complex ML prediction targets and mitigates variance to offer high accuracy
Hardware has kept pace to power the above:
Storage capacity has exploded (PB clusters)
Compute capacity has grown (multi-core, GPUs, etc.)
DRAM capacity has grown (10GBs to TBs)
Cloud computing is "democratizing" access to hardware;SaaS

Big Data
Big Data Typical characterization by 3 Vs:
Volume: larger than single-node DRAM
Variety: relations, docs, tweets, multimedia, etc.
Velocity: high generation rate, e.g., sensors, surveillance

Parallel Data Processing
Basic Idea: Split up workload across processors and perhaps also across machines/workers (aka "Divide and Conquer")
Common in parallel data processing: **"threads"**
Generalization of process abstraction of OS
A program/process can spawn many threads
Each runs its part of program's computations simultaneously
All threads share address space (so, data too)
In multi-core CPUs, a thread uses up 1 core
"Hyper-threading": Virtualizes a core to run 2 threads!
Common in parallel data processing: **"Dataflow Graph"**:
A directed graph representation of a program with vertices being abstract operations from a restricted set of computational primitives:
Extended relational dataflows: RDBMS, Pandas, Modin
Matrix/tensor dataflows: NumPy, PyTorch, TensorFlow
Enables us to reason about data-intensive programs at a higher level (logical level?)
Task Graph: Similar but coarse-grained; vertex is a process
Logical Query Plan: Relational Algebra Gate Graph
Neural Computational Graph: Neural Network Graph

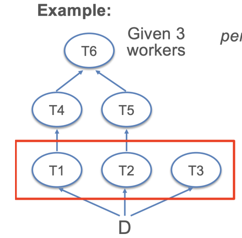


- More coarse-grained than operator-level dataflows
- Vertex: A full task/process
- Edge: A dependency between tasks
- Directed Acyclic Graph model (DAG) common
- Data may not be shown

Task Parallelism
Topological sort of tasks in task graph for scheduling
Notion of a "worker" can be at processor/core level, not just at node/server level
Thread-level parallelism possible instead of process-level
E.g., Dask: 4 worker nodes x 4 cores = 16 workers total
Main pros of task parallelism:
Simple to understand; easy to implement
Independence of workers => low software complexity
Main cons of task parallelism:
Data replication across nodes; wastes memory/storage
Idle times possible on workers

Degree of Parallelism
The largest amount of concurrency possible in the task graph, i.e., how many task can be run simultaneously
Possible Bottlenecks in Dask:
Memory errors, poor optimization, huge task graphs take too long to serialize, poor scalability of scheduler, machine failures.

Example:



Q: How do we quantify the runtime performance benefits of task parallelism?

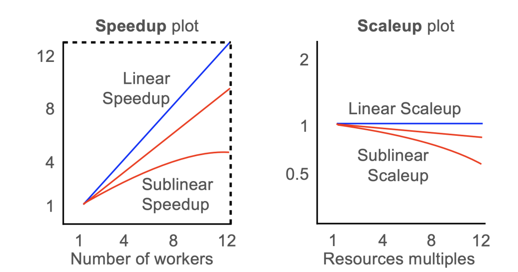
But over time, degree of parallelism keeps dropping in this example

Degree of parallelism is only 3

So, more than 3 workers is not useful for this workload!

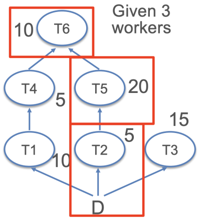
Scaleup refers to the ability of a system to retain the same performance ratio of tasks-per-resources when both the tasks and the resources increase at same rate

$$\textbf{Speedup} = \frac{\text{Completion time given only 1 worker}}{\text{Completion time given n (>1) workers}}$$



Most commonly, scaling does not demonstrate ideal linear behavior.

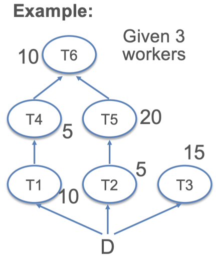
Example:



Given 3 workers

- In general, overall workload's completion time on task-parallel setup is always *lower bounded* by the **longest path** in the task graph
- Possibility: A task-parallel scheduler can "release" a worker if it knows that will be idle till the end
  - Can saves costs in cloud
  - Implemented as autoscaling in Kubernetes, can be custom implementation on EC2s or VMs.

Example:



Given 3 workers

Completion time with 1 worker: 10+5+15+5+20+10 = 65

Parallel completion time: 35

Speedup = 65/35 = 1.9x

Ideal/linear speedup is 3x

***Q:** Why is it only 1.9x?*

"Dask is a flexible library for parallel computing in Python"

2 key components:

APIs for data science ops on large data

Dynamic task scheduling on multi-core/multi-node

Design desirables:

Pythonic: Stay within PyData stack (e.g., no JVM)

Familiarity: Retain APIs of NumPy, Pandas, etc.

Scaling Up: Seamlessly exploit all cores

Scaling Out: Easily exploit cluster (needs setup)

Flexibility: Can schedule custom tasks too

Fast?: "Optimized" implementations under APIs

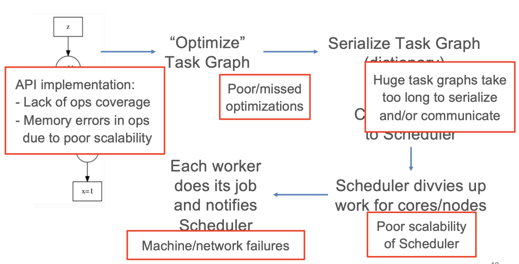
"Lazy Evaluation":

Ops on data structures are NOT executed immediately

Triggered manually, e.g., compute()

Dataflow graph / task graph is built under the hood

Possible Issue in Dask:



Dask: Task-Parallelism best Practices:

Data Partition sizes:

Avoid too few chunks (low degree of par.)

Avoid too many chunks (task graph overhead)

Be mindful of available DRAM

Rough guidelines they give:

\# data chunks ~ 3x-10x # cores, but

\# cores x chunk size must be < machine DRAM, but

chunk size shouldn't be too small (~1 GB is OK)

Q: Do you tune any of these when using an RDBMS?

Dask still lacks "physical data independence"!

Use the Diagnostics dashboard:

Monitor # tasks, core/node usage, task completion

Task Graph sizes:

Too large:

Bottlenecks (serialization / communication / scheduling)

Too small: Under-utilization of cores/nodes

Rough guidelines:

Tune data chunk size to adjust # tasks (see previous point)

Break up a task/computation

Fuse tasks/computations aka "batching", or in other cases break jobs apart into distinct stages.

Execution Optimization Tradeoffs

Be judicious in tuning data chunk sizes

Be judicious in batching vs breaking up tasks

Speedup is a function of the above factors

Single-Instruction Multiple-Data (SIMD)

A fundamental form of parallel processing in which different chunks of data are processed by the "same" set of instructions shared by multiple processing units (PUs)

Aka "vectorized" instruction processing (vs "scalar")

Data science workloads are very amenable to SIMD

Note: no "master" scheduler in this scenario

Single-Instruction Multiple Thread (SIMT): Generalizes notion of SIMD to different threads concurrently doing so

Each thread may be assigned a core or a whole PU

Single-Program Multiple Data (SPMD): A higher level of abstraction generalizing SIMD operations or programs

Under the hood, may use multiple processes or threads

Each chunk of data processed by one core/PU

Applicable to any CPU, not just vectorized PUs

Most common form of parallel programming

In this case, work is distributed from a central scheduler or orchestrator.

In data science computations, an often useful surrogate for completion time is the instruction throughput FLOP/s, i.e., number of floating point operations per second

Modern data processing programs, especially deep learning (DL) may have billions of FLOPs aka GFLOPs!

Amdahl's Law: Formula to upper bound possible speedup

A program has 2 parts: one that benefits from multi-core parallelism and one that does not

Non-parallel part could be for control, memory stalls, traversing a linked list

1 core: $T_{yes} \longrightarrow$ n cores: T_{yes}/n

1 core: $T_{no} \longrightarrow$ n cores: T_{no}

$$\text{Speedup} = \frac{T_{yes} + T_{no}}{T_{yes}/n + T_{no}} = \frac{n(1+f)}{n+f}$$

Denote $T_{yes}/T_{no} = f$

Moore's Law: The number of transistors in a dense integrated circuit doubles

	Multi-core CPU	GPU	FPGA	ASICs (e.g., TPUs)
Peak FLOPS/s	Moderate	High	High	Very High
Power Consumption	High	Very High	Very Low	Low-Very Low
Cost	Low	High	Very High	Highest
Generality / Flexibility	Highest	Medium	Very High	Lowest
"Fitness" for DL Training?	Poor Fit	Best Fit	Low Fit	Potential exists but yet unrealized
"Fitness" for DL Inference?	Moderate	Moderate	Good Fit	Best Fit
Cloud Vendor Support	All	All	AWS, Azure	AWS, GCP

Practices:

A. Databases are typically stored as files. — A. T

B. An OS typically has mechanisms to wrest control of hardware back from a user process. — B. T

C. More accurate ML models are always larger in the number of bytes of memory they need than less accurate ones. — C. F

D. SCTF is the fairest scheduling policy we discussed in class. — D. F

E. A filesystem is a specific format of serializing data files. — E. F

F. CPU caches are usually cheaper per MB than Flash SSD. — F. F

G. If DRAM is infinite Pareto frontiers are irrelevant in ML practice. — G. F

H. DataFrame and Relation are equivalent data models. — H. F

Linear?

10. Suppose an SQL query takes 20min to run on a single worker node and *x* min when run on 5 worker nodes. What is the speedup for the given value of *x*? Is the speedup linear, sublinear, or superlinear?

A. x = 7min

B. x = 4min

C. x = 3min

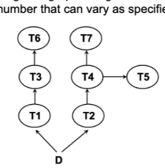
A. Speedup = 20/7 = 2.86x; < 5x, sublinear

B. Speedup = 20/4 = 5x; = 5x, linear

C. Speedup = 20/3 = 6.67x; > 5x, superlinear

Task Graph

Consider the following task graph and given task lengths (in time units). The value of x is a non-negative number that can vary as specified in the following questions.



Task	Length
T1	15
T2	5
T3	10
T4	x
T5	10
T6	10
T7	10

1. What is the degree of parallelism of this workload for task-parallel execution?
Answer: 3 (The figure shows that tasks T5, T6, and T7 can be executed independently).

2. Assume x=20. Suppose the workload is executed in a task-parallel manner for the lowest possible completion time on 3 workers. All workers are on until the last task finishes. What is the total idle time (add across workers)?
Answer: If we have 3 workers: T1-T3-T6 and T2-T4-T7 can run in parallel on 2 workers, followed by T5 running on the 3rd worker. T5 needs to wait for 25 units to begin, and so do T6 and T7. T5, T6, and T7 finish at the same time. So, total idle time is the wait time for T1-T3 or T2-T4, which is 25.

3. Assume x=20. What is the lowest possible completion time of this workload with task-parallelism?
Answer: We need to find the longest single path for a worker. In the present scenario, it will take for all workers 35 units to complete their tasks.

4. Assume x=5. What is the value of the lowest possible completion time with task-parallelism when given only 2 workers?
Answer: If we only have 2 workers, then the path T1-T3-T6 takes 35 units for one worker; the path T2-T4-T7 (or T2-T4-T5) takes 20 units for the second worker. The second worker will finish earlier and can do the remaining task T5 (or T7) within 10 more units, thus totaling 30 units. As a result, the lowest possible completion given 2 workers will take 35 units, because this is the longest path needed to be followed (the one done by the first worker).

DRAM is the level of memory that has the lowest latency to read data from

Dennard Scaling: As transistors get smaller, their power density stays constant, so that the power use stays in proportion with area.

Takeaway from hardware trends: it is hard for general-purpose CPUs to **sustain FLOP-heavy programs** like deep nets

Motivated the rise of "accelerators" for some classes of programs

Graphics Processing Unit (GPU): Tailored for matrix/tensor ops

Basic idea: use tons of ALUs; massive data parallelism (SIMD on steroids); Titan X offers ~11 TFLOP/s!

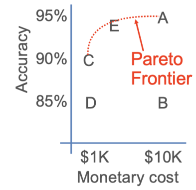
Tensor Processing Unit (TPU): Even more specialized tensor ops in DL inference; ~45 TFLOP/s!

Field-Programmable Gate Array (FPGA): Configurable for any class of programs; ~0.5-3 TFLOP/s

Real-World ML: Pareto Surfaces

***Q:** Suppose you are given ad click-through prediction models A, B, C, and D with accuracies of 95%, 85%, 90%, and 85%, respectively. Which one will you pick?*

***Q:** What about now?*



- Real-world ML users must grapple with multi-dimensional *Pareto surfaces*: accuracy, monetary cost, training time, scalability, inference latency, tool availability, interpretability, fairness, etc.
- *Multi-objective optimization* criteria set by application needs / business policies.

(1):Data processing programs need to go through the OS System Call API to read text files but can typically bypass that API if they want to read binary file: **FALSE**

(2):Which of the following properties of data processing programs is sometimes exploited to help reduce runtimes?: Spatial locality of reference;Temporal locality of reference ;Parallelism in computations

(Post Midterm:)

How much DRAM might a machine have?

Common DRAM configs:

- Average Laptop: 16GB
- t2.xlarge EC2 instance: 16GB (at $0.19/hour)
- 2023 MacBook Pro: 32GB-96GB
- Consumer Deep Learning / Gaming PC: 128GB ($288 fixed)
- r7g.metal EC2 instance: 512GB (at $3.43/hour)
- hpc6id.32xlarge EC2 instance: 1024GB (at $5.70/hour)

Less common: u-24tb1.112xlarge: 24TB (at $218.40/hour)

Scalable Data Access

Central Issue: Large data file does not fit entirely in DRAM

Basic Idea: Divide-and-conquer again.

"Split" a data file (virtually or physically) and stage reads of its pages from disk to DRAM; vice versa for writes.

Single-node disk: Paged access from file on local disk

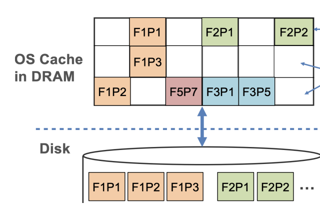
Remote read: Paged access from disk(s) over a network

Distributed memory: Data fits on a cluster's total DRAM

Distributed disk: Use entire memory hierarchy of cluster

Paged Data Access to DRAM

Basic Idea: "Split" data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)



- Recall that files are already virtually split and stored as *pages* on both disk and DRAM

Page Management in DRAM Cache

Caching: Retaining pages read from disk in DRAM

Eviction: Removing a page frame's content in DRAM

Spilling: Writing out pages from DRAM to disk

- If a page in DRAM is "dirty" (i.e., some bytes were written but not backed up on disk), eviction requires a spill.
- The set of DRAM-resident pages typically changes over the lifetime of a process

Cache Replacement Policy: The algorithm that chooses which page frame(s) to evict when a new page has to be cached but the OS cache in DRAM is full

- Popular policies include Least Recently Used, Most Recently Used, etc. (more shortly)

Quantifying I/O: Disk, Network

Page reads/writes to/from DRAM from/to disk incur latency

Disk I/O Cost: Abstract counting of number of page I/Os; can map to bytes given page size

Sometimes, programs read/write data over network

Communication/Network I/O Cost: Abstract counting of number of pages/bytes sent/received over network

I/O cost is abstract; mapping to latency is hardware-specific

Example: Suppose a data file is 40GB; page size is 4KB

I/O cost to read file = 10 million page I/Os

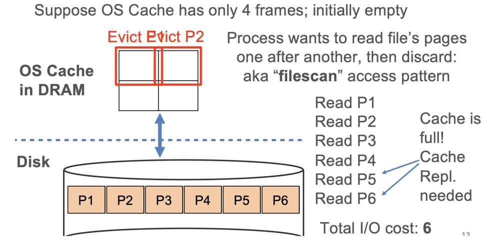
Disk with I/O throughput: 800 MB/s → 40GB/800MBps = 50s

Network with speed: 200 MB/s → 40GB/200MBps = 200s

Scaling to (Local) Disk

Suppose OS Cache has only 4 frames; initially empty

Process wants to read file's pages one after another, then discard: aka "**filescan**" access pattern



In general, scalable programs stage access to pages of file on disk and efficiently use available DRAM

- Recall that typically DRAM size << Disk size

Modern DRAM sizes can be 10s of GBs; so we read a

"chunk"/"block" of file at a time (say, 1000s of pages)
❖ On magnetic hard disks, such chunking leads to more sequential I/Os, raising throughput and lowering latency!
❖ Similarly, write a chunk of dirtied pages at a time

Generic Cache Replacement Policies

What to do if number of page frames is too few for file?

Cache Replacement Policy: Algorithm to decide which page frame(s) to evict to make space

Typical frame ranking criteria:
❖ recency of use
❖ frequency of use
❖ number of processes reading it

Typical optimization goal: Reduce total page I/O costs

A few well-known policies:
❖ **Least Recently Used (LRU):** Evict page that was used longest ago
❖ **Most Recently Used (MRU):** (Opposite of LRU)
❖ ML-based caching policies are "hot" nowadays!

Data Layouts and Access Patterns
❖ Recall that data layouts and data access patterns affect what data subset gets cached in higher level of memory hierarchy
  ❖ Recall matrix multiplication example and CPU caches
❖ **Key Principle:** Optimizing layout of data file on disk based on data access pattern can help reduce I/O costs
  ❖ Applies to both magnetic hard disk and flash SSDs
    ❖ But especially critical for magnetic hard disks due to vast differences in latency of random vs sequential access!

Row-store vs Column-store Layouts
❖ A common dichotomy when serializing 2-D structured data (relations, matrices, DataFrames) to file on disk
❖ Based on data access pattern of program, I/O costs with row- vs col-store can be orders of magnitude apart!
❖ **With row-store:** need to fetch all pages; I/O cost: 6 pages
❖ **With col-store:** need to fetch only B's pages; I/O cost: 2 pages

This difference generalizes to higher dimensions for tensors

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

Say, a page can fit only 4 cell values

Row-store: [1a,1b,1c,1d] [2a,2b,2c,2d] [3a,3b,3c,3d]

Col-store: [1a,2a,3a,4a] [5a,6a] [1b,2b,3b,4b]

Hybrid/Tiled/"Blocked" Layouts

Sometimes, it is beneficial to do a hybrid, especially for analytical RDBMSs and matrix/tensor processing systems

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

Say, a page can fit only 4 cell values

Hybrid stores with 2x2 tiled layout:

[1a,1b, 2a,2b] [1c,1d, 2c,2d] [3a,3b, 4a,4b] …

[1a, 2a, 1b, 2b] [1c, 2c, 1d, 2d] [3a, 4a, 2b, 3b] …

Key Principle: What data layout will yield lower I/O costs (row vs col vs tiled) depends on data access pattern of the program!

Dask's DataFrame

Basic Idea: Split data file (virtually or physically) and stage reads of its pages from disk to DRAM (vice versa for writes)
❖ Dask DF scales to disk-resident data via a row-store
❖ "Virtual" split: each split is a Pandas DF under the hood
❖ Dask API is a "wrapper" around Pandas API to scale ops to splits and put all results together
❖ If file is too large for DRAM, need manual repartition() to get physically smaller splits (< ~1GB)

Modin's DataFrame

Basic Idea: Split data file (virtually or physically) and stage reads of its pages from disk to DRAM (vice versa for writes)
❖ Modin's DF aims to scale to diskresident data via a tiled store
❖ Enables seamless scaling along both dimensions
❖ Easier use of multi-core parallelism
→ Many in-memory RDBMSs had this, e.g., SAP HANA, Oracle TimesTen
→ ScaLAPACK had this for matrices

Scaling with Remote Reads

Basic Idea: Split data file (virtually or physically) and stage reads of its pages from disk to DRAM (vice versa for writes)
❖ Similar to scaling to local disk but not "local":
❖ Stage page reads from remote disk/disks over the network (e.g., from S3)
❖ More restrictive than scaling with local disk, since spilling is not possible or requires costly network I/Os
  ❖ OK for a one-shot filescan access pattern
  ❖ Use DRAM to cache; repl. policies
  ❖ Can also use smaller local disk as cache

Scaling to Disk: Non-dedup. Project

R

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

```
SELECT C FROM R
```

Row-store: [1a,1b,1c,1d] [2a,2b,2c,2d] [3a,3b,3c,3d] [4a,4b,4c,4d] [5a,5b,5c,5d] [6a,6b,6c,6d]

❖ Straightforward **filescan** data access pattern
  ❖ Read one page at a time into DRAM; may need cache repl
  ❖ Drop unneeded columns from tuples on the fly
❖ I/O cost: 6 (read) + output # pages (write)

R

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

```
SELECT C FROM R
```

Col-store: [1a,2a,3a,4a] [5a,6a] [1b,2b,3b,4b] [5b,6b] [1c,2c,3c,4c] [5c,6c] [1d,2d,3d,4d] [5d,6d]

❖ Since we only need col C, no need to read other pages!
❖ I/O cost: 2 (read) + output # pages (write)
❖ Big advantage for col-stores over row-stores for SQL analytics queries (projects, aggregates, etc.); popular in online analytical processing ("OLAP")
  ❖ Rationale for col-store RDBMS (e.g., Vertica) and Parquet

Scaling to Disk: Simple Aggregates:

Similar behavior with Non-dedup

Scaling to Disk: Group By Aggregate

R

A	B	C	D
a1	1b	1c	4
a2	2b	2c	3
a1	3b	3c	5
a3	4b	4c	1
a2	5b	5c	10
a1	6b	6c	8

```
SELECT A, SUM(D)
FROM R
GROUP BY A
```

❖ Now it is not straightforward due to the GROUP BY!
❖ Need to "collect" all tuples in a group and apply aggregation function to each
❖ Typically done with a **hash table** maintained in DRAM
  ❖ Has 1 record per group and maintains "running information" for that group's aggregation function
  ❖ Built on the fly during filescan of R; holds the output in the end

Hash table (output)

A	Running Info.
a1	17
a2	13
a3	1

R

A	B	C	D
a1	1b	1c	4
a2	2b	2c	3
a1	3b	3c	5
a3	4b	4c	1
a2	5b	5c	10
a1	6b	6c	8

```
SELECT A, SUM(D)
FROM R
GROUP BY A
```

Row-store: [a1,1b,1c,4] [a2,2b,2c,3] [a1,3b,3c,5] [a3,4b,4c,1] [a2,5b,5c,10] [a1,6b,6c,8]

Hash table in DRAM

A	Running Info.
a1	4 -> 9 -> 17
a2	3 -> 13
a3	1

❖ Note that the sum for each group is constructed *incrementally*
❖ I/O cost: 6 (read) + output # pages (write); just one filescan again!

Q: But what if hash table > DRAM size?!

Q: But what if hash table > DRAM size?

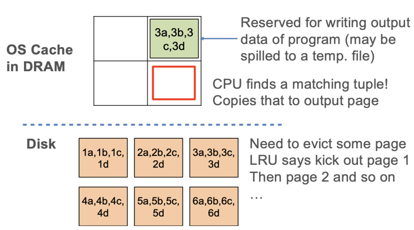
Program might crash depending on backend implementation. OS may keep swapping pages of hash table to/from disk; aka "thrashing"

Q: How to scale to large number of groups?
❖ Divide and conquer! Split up R based on values of A
❖ HT for each split may fit in DRAM alone
❖ Reduce running info. size if possible

Scaling to Disk: Relational Select
❖ Straightforward filescan data access pattern
❖ Read pages/chunks from disk to DRAM one by one
❖ CPU applies predicate to tuples in pages in DRAM
❖ Copy satisfying tuples to temporary output pages
❖ Use LRU for cache replacement, if needed
❖ I/O cost: 6 (read) + output # pages (write)

Scaling to Disk: Relational Select

$$\sigma_{B=\text{"3b"}}(R)$$



Scaling to Disk: Matrix Sum of Squares

M_{6x4}

2	1	0	0
2	1	0	0
0	1	0	2
0	0	1	2
3	0	1	0
3	0	1	0

Row-store: [2,1, 0,0] [2,1 0,0] [0,1, 0,2] [0,0, 1,2] [3,0, 1,0] [3,0, 1,0]

❖ Again, straightforward **filescan** data access pattern
  ❖ Very similar to relational simple aggregate
  ❖ Running info. in DRAM for sum of squares of cells
    ❖ 0 -> 5 -> 10 -> 15 -> 20 -> 30 -> 40
❖ I/O cost: 6 (read) + output # pages (write)

Scalable Matrix/Tensor Algebra:
❖ In general, tiled partitioning is more common for matrix/tensor ops
❖ DRAM-to-disk scaling:
  ❖ pBDR, SystemDS, and Dask Arrays for matrices
  ❖ SciDB, Xarray for n-d arrays
❖ CUDA for DRAM-GPU caches scaling of matrix/tensor ops

Numerical Optimization in ML:

Many regression and classification models in ML are formulated as a (constrained) minimization problem
  ❖ E.g., logistic and linear regression, linear SVM, DL classification and regression.
  ❖ Aka "Empirical Risk Minimization" (ERM) approach
  ❖ Computes "loss" of predictions over labeled examples
❖ Hyperplane-based models aka Generalized Linear Models (GLMs) use f() that is a scalar function of distances: w^{T}x_{i}

Batch Gradient Descent for ML
❖ Learning rate is a hyper-parameter selected by user or "AutoML" tuning procedures
❖ Number of epochs (iterations) of BGD also hyper-parameter

Data Access Pattern of BGD at Scale
❖ The data-intensive computation in BGD is the gradient
❖ In scalable ML, dataset D may not fit in DRAM
❖ Model w is typically (but not always) small and DRAM-resident
❖ Gradient is like SQL SUM over vectors (one per example)
❖ At each epoch, 1 filescan over D to get gradient
❖ Update of w happens normally in DRAM
❖ Monitoring across epochs (or iterations) for convergence needed
❖ Loss function L() is also just a SUM in a similar manner

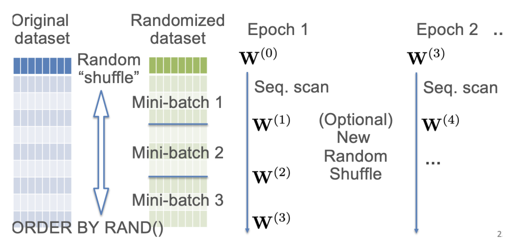
I/O Cost of Scalable BGD
❖ Straightforward filescan data access pattern for SUM
❖ Similar I/O behavior as non-dedup. project and simple SQL aggregates
❖ I/O cost: 6 (read) + output # pages (write for final w)

Stochastic Gradient Descent for ML
❖ Two key cons of BGD:
❖ Often, too many epochs to reach optimal
❖ Each update of w needs full scan: costly I/Os, full design matrix in memory
❖ Stochastic GD (SGD) mitigates both cons
❖ Basic Idea: Use a sample (mini-batch) of D to approximate gradient instead of "full batch" gradient
❖ Done without replacement
❖ Randomly reorder/shuffle D before every epoch
❖ Sequential pass: sequence of mini-batches
❖ Another big pro of SGD: works better for non-convex loss too, especially DL
❖ SGD often called the "workhorse" of modern ML/DL

Access Pattern of Scalable SGD:

$$\mathbf{W}^{(t+1)} \leftarrow \mathbf{W}^{(t)} - \eta\nabla\tilde{L}(\mathbf{W}^{(t)}) \qquad \nabla\tilde{L}(\mathbf{W}) = \sum_{i\in B}\nabla l(y_i, f(\mathbf{W}, x_i))$$

Sample mini-batch from dataset without replacement



I/O Cost of (Very) Scalable SGD:
❖ I/O cost of random shuffle is non-trivial; need so-called "external merge sort" (skipped in this course)
❖ Typically amounts to 1 or 2 passes over file
❖ Mini-batch gradient computations: 1 filescan per epoch:
❖ As filescan proceeds, count # examples seen, accumulate perexample gra
❖ Typical mini-batch sizes: 10s to 1000s... or 1 if transformer model and limited resources...
❖ Orders of magnitude more model updates than BGD!
❖ Total I/O cost per epoch: 1 shuffle cost + 1 filescan cost
❖ Often, shuffling only once upfront suffices
❖ Loss function L() computation is same as before (for BGD)

Too Big To Fit, scale-up vs.scale-out

When an application becomes too big or too complex to run efficiently on a single server, there are some options:
1:migrate to a larger server, and buy bigger licenses–**vertical scale up**
2:distribute data+compute across multiple servers–**horizontal scale out**
The histories of MPI, Hadoop, Spark, Dask, etc., represent generations of scale-out, which imply **trade-offs** both for the risks as well as the Inherent overhead costs

Why Ray:

Machine learning is pervasive / Distributed computing is a necessity
Python is the default language for DS/ML

What is Ray?

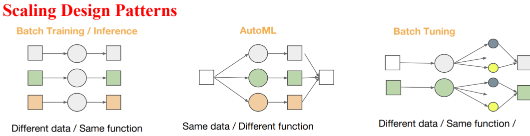
A **simple/general-purpose** library for distributed computing
● An ecosystem of Python libraries (for scaling ML and more)
● Runs on laptop, public cloud, K8s, on-premise
A layered cake of functionality and capability for scaling ML workloads

Ray Core: Tasks / Actors / Objects

Ray AI Runtime is a scalable runtime/toolkit for end-to-end ML applications.

Ray Basic Design Patterns
- Ray Parallel Tasks
  - Functions as stateless units of execution
  - Functions distributed across the cluster as tasks
- Ray Objects as Futures
  - Distributed (immutable objects) store in the cluster
  - Fetched when materialized
  - Enable massive asynchronous parallelism
- Ray Actors
  - Stateful service on a cluster
  - Enable Message passing

Scaling Design Patterns



(Circle: Compute; Square: Data)

Ray Task: A function remotely executed in a cluster

Python → Ray APIs:

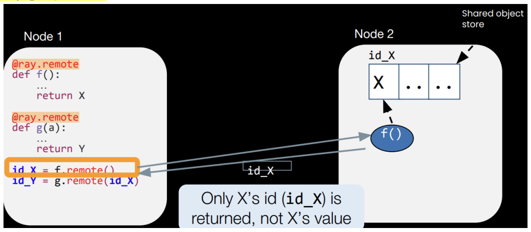
Ray Task: A function remotely executed in a cluster

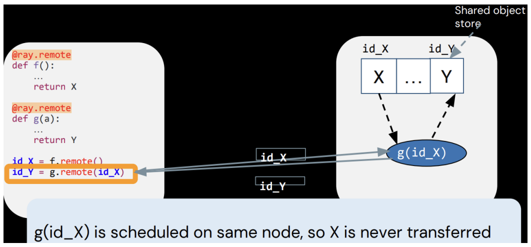
@ray.remote(num_cpus=2)
Def f(a,b):
  Return a+b
f.remote(1,2)

Ray Actor: A class remotely executed in a cluster

```
@ray.remote(num_gpus=4)
Class HostActor:
  Def __init__(self):
    Self.num_devices = os.environ["CUDA_VISIBLE_DEVICES"]
  Def f(self, output):
    Return f"{output} {self.num_devices}"
Actor = HostActor.remote()
actor.f.remote("hi")
```

Dynamic task graph: build at runtime
ray.get() block: until result available





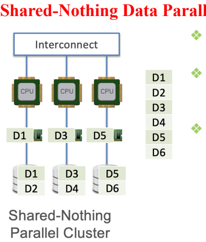
Distributed Applications with Ray:
ML Libraries (All using Ray core APIs & patterns)
● Ray AI Runtime ● Distributed scikit-learn/Joblib
● Distributed XGBoost on Ray ● Ray Multiprocess Pool
Ray provides generic platform for LLMs
Simplify orchestration and scaling:
● Spot instance support for data parallel training
● Easily spin up and run distributed workloads on any cloud
● Optimize CPUs/GPUs by pipelining w/ Ray Data
Inference and serving:
● Ability to support complex pipelines integrating business logic
● Ability to support multiple node serving
Training
● Integrates distributed training with distributed hyperparameter tuning w/ ML frameworks

Ray Key Takeaways
● Distributed computing is a necessity & norm
● Ray's vision: make distributed computing simple
○ Don't have to be distributed programming expert
● Build your own disruptive apps & libraries with Ray
● Scale your ML workloads with Ray libraries (Ray AIR)
● Ray offers the compute substrate for Generative AI workloads

Introducing Data Parallelism
Basic Idea of Scalability: Split data file (virtually or physically) and stage reads/writes of its pages between disk and DRAM
Data Parallelism: Partition large data file physically across nodes/workers; within worker: DRAM-based or disk-based
❖ The most common approach to marrying parallelism and scalability in data systems
❖ Generalization of SIMD and SPMD idea from parallel processors to large-scale data and multi-worker/multi-node setting
❖ Distributed-memory vs Distributed-disk

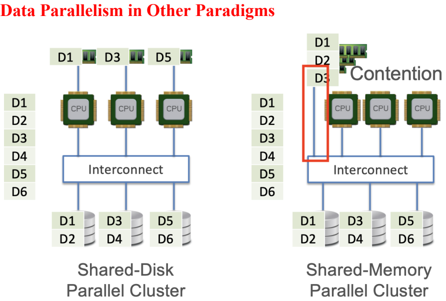
3 Paradigms of Multi-Node Parallelism
Data parallelism is technically orthogonal to these 3 paradigms but **most commonly paired with shared-nothing**

Shared-Nothing Data Parallelism



❖ Partitioning a data file across nodes is aka **sharding**
❖ Part of a stage in data processing workflows called **Extract-Transform-Load** (ETL)
❖ ETL is an umbrella term for all kinds of processing done to the data file before it is ready for users to query, analyze, etc.
  ❖ Sharding, compression, file format conversions, etc.

Data Parallelism in Other Paradigms



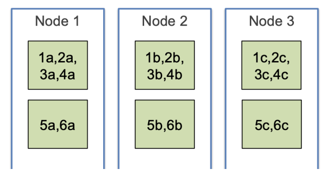
Data Partitioning Strategies
❖ Row-wise/horizontal partitioning is most common (sharding)
❖ 3 common schemes (given k nodes):
  ❖ **Round-robin:** assign tuple i to node i MOD k
  ❖ **Hashing-based:** needs hash partitioning attribute(s)
  ❖ **Range-based:** needs ordinal partitioning attribute(s)
❖ **Tradeoffs:**
  ❖ For Relational Algebra (RA) and SQL:
    ❖ Hashing-based most common in practice for RA/SQL
    ❖ Range-based often good for range predicates in RA/SQL
  ❖ But all 3 are often OK for many ML workloads (why?)
❖ **Replication** of partition across nodes (e.g., 3x) is common to enable "fault tolerance" and better parallel runtime performance

Other Forms of Data Partitioning
❖ Just like with disk-aware data layout on single-node, we can partition a large data file across workers in other ways too:

Columnar Partitioning

R

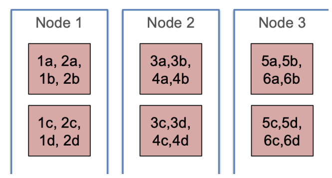
A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d



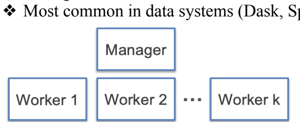
Hybrid/Tiled Partitioning

R

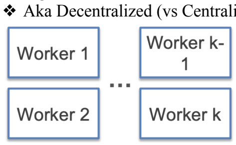
A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d



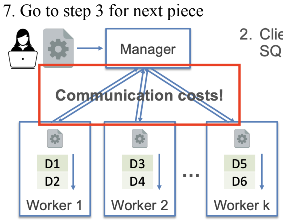
Cluster Architectures:
Manager-Worker Architecture:
❖ 1 (or few) special node called Manager (aka "Server" or archaic "Master"); 1 or more Workers
❖ Manager tells workers what to do and when to talk to other nodes
❖ Most common in data systems (Dask, Spark, par. RDBMS, etc.)



Peer-to-Peer Architecture
❖ No special manager
❖ Workers talk to each other directly
❖ E.g., Horovod
❖ Aka Decentralized (vs Centralized)



Bulk Synchronous Parallelism (BSP)
❖ Most common protocol of data parallelism in data systems (e.g., in parallel RDBMSs, Hadoop, Spark)
❖ Shared-nothing sharding + manager-worker architecture
1. Sharded data file on workers
2. Client gives program to manager (SQL query, ML training, etc.)
3. Manager divides first piece of work among workers
4. Workers work independently on self's data partition (cross-talk can happen if Manager asks)
5. Worker sends partial results to Manager
6. Manager waits till all k done
7. Go to step 3 for next piece



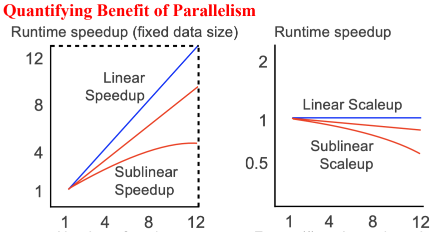
Aka **(Barrier) Synchronization**

Speedup Analysis/Limits of of BSP

$$\textbf{Speedup} = \frac{\text{Completion time given only 1 worker}}{\text{Completion time given k (>1) workers}}$$

❖ Cluster overhead factors that hurt speedup:
❖ **Per-worker**: startup cost; tear-down cost
❖ **On manager:** dividing up the work; collecting/unifying partial results from workers
❖ **Communication costs**: talk between manager-worker and across workers (when asked by manager)
❖ Barrier synchronization suffers from "stragglers" (workers that fall behind) due to skews in shard sizes and/or worker capacities

Quantifying Benefit of Parallelism



Speedup plot / Strong scaling   **Scaleup** plot / Weak scaling

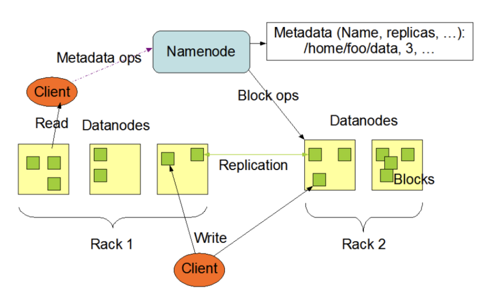
Distributed Filesystems
❖ Recall definition of file; distributed file generalizes it to a cluster of networked disks and OSs
❖ **Distributed filesystem (DFS)** is a cluster-resident filesystem to manage distributed files
  ❖ A layer of abstraction on top of local filesystems
  ❖ Nodes manage local data as if they are local files
  ❖ Illusion of a one global file: DFS APIs let nodes access data sitting on other nodes
  ❖ 2 main variants: Remote DFS vs In-Situ DFS
    ❖ **Remote DFS**: Files reside elsewhere and read/written on demand by workers
    ❖ In-Situ DFS: Files resides on cluster where workers exist

Network Filesystem (NFS)
❖ An old remote DFS (c. 1980s) with simple client-server architecture for replicating files over the network
Network Filesystem (NFS)
❖ **Main pro:** simplicity of setup and usage

❖ But many cons:
  ❖ Not scalable to very large files
  ❖ Full data replication
  ❖ High contention for concurrent reads/writes
  ❖ Single-point of failure

Hadoop Distributed File System (HDFS)
❖ Most popular in-situ DFS (c. late 2000s); part of Hadoop; open source spinoff of Google File system (GFS)
❖ Highly scalable; scales to 10s of 1000s of nodes, PB files
❖ Designed for clusters of cheap commodity nodes
❖ Parallel reads/writes of sharded data "blocks"
❖ Replication of blocks to improve fault tolerance
❖ Cons: Read-only + batchappend (no fine-grained updates/writes)

HDFS Architecture

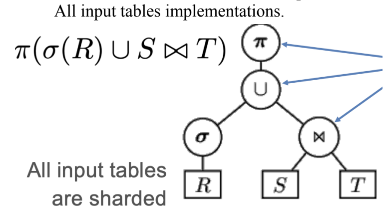


❖ NameNode's roster maps data blocks to DataNodes/IPs
❖ A distributed file on HDFS is just a directory (!) with individual filenames for each data block and metadata files
❖ HDFS has configurable parameters:

Parameter name	**Purpose**	**Default value**
Data block size	Splitting data into chunks	128 MB
Replication factor	Ensure data availability	3x

Data-Parallel Dataflow/Workflow
❖ **Data-Parallel Dataflow:** A dataflow graph with ops wherein each operation is executed in a data-parallel manner
❖ **Data-Parallel Workflow:** A generalization; each vertex a whole task/process that is run in a data-parallel manner
Note: In parallel environments like parallel RDBMSs and Spark:
Each of these extended relational ops have scalable data-parallel
All input tables implementations.

$$\pi(\sigma(R) \cup S \bowtie T)$$



Distributed Computing Paradigms
Different paradigms and models used in distributed computing:
Batch processing: Breaking tasks into smaller sub-tasks that can be processed independently.
Message passing: Communication between nodes through message passing protocols like MPI.
Shared memory: Multiple nodes accessing a common memory space.
MapReduce: A programming model for processing large datasets in a distributed manner.
Stream processing: Real-time processing of continuous data streams.

Distributed File Systems → like HDFS (Hadoop)
Fault Tolerance: With HDFS, the company stores multiple replicas of the data across different nodes. If a node fails, the data is still accessible from other replicas, ensuring fault tolerance and preventing data loss.
Scalability: As the company's data grows, they can add more nodes to the Hadoop cluster and distribute the data across these nodes. HDFS scales horizontally, allowing the company to accommodate the increasing volume of data without compromising performance.
Data Locality: When processing the customer data and performing analytics, HDFS ensures data locality by storing the data on the same nodes where the computation is performed. This reduces data transfer over the network and improves overall processing efficiency.

Challenges & considerations in distributed analysis
While dealing with large amounts of data the primary challenge is that it cannot fit on a single machine.
Storage Tradeoff: Storing data entirely in memory yields better performance but is expensive. Disk storage is cheaper but results in lower performance.
Hybrid Caching: Combination of SSD flash disks and hard disks for storing data subsets. Placement of data on appropriate storage medium is crucial.
Distributing Data: Root-leaf approach for distributing data across thousands of machines. Each leaf machine holds a portion of the data, results merged at the root.

Latency Impact: Latency from the slowest machine affects overall performance. Mitigating latency through optimization techniques is essential.
Overhead in Data Transfer: Serialization, compression, and encryption introduce overhead. File format overhead, decryption, and decompression impact performance.
Hardware Support: Encryption at rest and in motion requires hardware support. Hardware advancements crucial for efficient distributed analysis.
Serialization and Interpretation: Data structures are serialized for transmission over a wire. Receiving machine must interpret the serialized data correctly.

Distributed Collaborative filtering
In the diagram, the process of making collaborative filtering distributed is illustrated with two nodes (Node 1 and Node 2) as an example. Here's a breakdown of the components:
1. User-Item Data: Represents the initial user-item interaction data used for collaborative filtering.
2. Data Partitioning: The data is partitioned into subsets and distributed across multiple nodes.
3. Local Similarity Computation: Each node independently computes local similarities (e.g., cosine similarity) based on the user-item interactions available on that node.
4. Data Exchange and Aggregation: The computed similarities are exchanged and aggregated across the nodes to generate a global similarity matrix.
5. Recommendation Generation: Each node utilizes the global similarity matrix and the locally available user-item interactions to generate personalized recommendations for its subset of users.
6. Result Integration and Final Recommendations: The recommendations generated by each node are integrated to produce the final distributed recommendations.

Language Models and Challenges in Distributed Training
1. **Computational Resources:** Large language models require immense computational power, memory, and storage. Training and inference across distributed systems necessitate significant hardware resources
2. **Communication Overhead:** In distributed training, coordinating updates across multiple nodes introduces communication overhead. Efficient communication protocols and optimized data exchange mechanisms are essential.
3. **Data Synchronization:** Ensuring consistent model parameters and synchronization of large amounts of data across nodes is a challenge. In distributed inference, managing data consistency for parallel processing can be complex.
4. **Scalability:** Scaling distributed training and inference to accommodate growing model sizes and datasets is crucial. Load balancing and resource allocation need to be optimized for efficient scalability.

How to Parallelize GPTs?
The parallelization of the GPT architecture can be achieved by utilizing techniques such as model parallelism and data parallelism. Let's disc
Model Parallelism: Model parallelism involves distributing the model across multiple devices or machines. In the case of GPT, where the model consists of stacked transformer layers, each layer can be allocated to different devices. This allows for parallel computation of different layers, reducing the overall training or inference time. Model parallelism can be particularly useful when dealing with very large models that cannot fit into a single device's memory.
Data Parallelism: Data parallelism involves dividing the data into multiple subsets and processing them simultaneously on different devices. In the context of GPT, the training data can be partitioned into smaller batches, and each batch is processed by a separate device or machine. The gradients calculated on each device are then synchronized and aggregated to update the model parameters. Data parallelism enables faster training by parallelizing the computation across multiple devices.

Benefits of Distributed Computing for Large Language Models
Scalability: Distributed computing enables efficient scaling of resources to handle large-scale training and inference workloads.
Speed: Parallel processing across multiple nodes reduces the time required for training and inference tasks.
Fault tolerance: Distributed systems provide resilience by replicating data and computations across multiple nodes, ensuring uninterrupted operation even in the face of failures.

Real-world Applications
Language translation: Distributed computing facilitates the training and serving of language translation models that can handle large volumes of text.
Content generation: Distributed language models enable the generation of coherent and contextually relevant content for various applications, such as chatbots or content personalization.
Sentiment analysis: Large language models distributed across multiple nodes can process and analyze vast amounts of text data to derive sentiment insights.

Considerations and Challenges
Data synchronization: Ensuring consistency and synchronization of data across distributed nodes.
Communication overhead: Efficient communication and coordination between nodes to minimize latency and optimize performance.
Resource management: Proper allocation and management of computational resources across the distributed system.

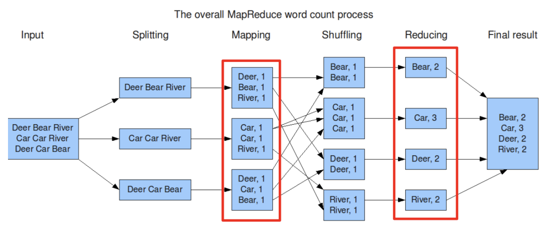
Parallel RDBMSs
- Parallel RDBMSs are highly successful and widely used
- Typically shared-nothing data parallelism
- Optimized runtime performance + enterprise-grade features:
- ANSI SQL & more
- Business Intelligence (BI) dashboards/APIs
- Transaction management; crash recovery
- Indexes, auto-tuning, etc.
- **4 new concerns of Web giants vs RDBMSs built for enterprises:**
  - **Developability:** Custom data models and computations hard to program on SQL/RDBMSs; need for simpler APIs
  - **Fault Tolerance:** Need to scale to 1000s of machines; need for graceful handling of worker failure
  - **Elasticity:** Need to be able to easily upsize or downsize cluster size based on workload
  - **Cost:** Commercial RDBMSs licenses too costly; hired own software engineers to build custom new systems

A new breed of parallel data systems called **Dataflow Systems** jolted the DB folks from being complacent!

What is MapReduce?
- A programming model for parallel programs on sharded data + **distributed system** architecture
- **Map** and **Reduce** are terms from functional PL; software/data/ML engineer implements logic of Map, Reduce
- System handles data distribution, parallelization, fault tolerance, etc. under the hood
- Created by Google to solve "simple" data workload: index, store, and search the Web!
- Google's engineers started with MySQL! Abandoned it due to reasons listed earlier (developability, fault tolerance, elasticity, etc.)
- **Standard example**: count word occurrences in a doc corpus
- **Input**: A set of text documents (say, webpages)
- **Output:** A dictionary of unique words and their counts

```
function map (String docname, String doctext) :
  for each word w in doctext :
    emit (w, 1)
function reduce (String word, Iterator partialCounts) :
  sum = 0
  for each pc in partialCounts :
    sum += pc
  emit (word, sum)     (red: Part of MapReduce API)
```

How MapReduce Works
Parallel flow of control and data during MapReduce execution:



Under the hood, each **Mapper** and **Reducer** is a separate process; Reducers face barrier synchronization (BSP)
Fault tolerance achieved using **data replication**
10

Benefits and Catch of MapReduce
- **Goal**: High-level functional ops to simplify data-intensive programs
- **Key Benefits:**
- Map() and Reduce() are highly general; any data types/structures; great for ETL, text/multimedia
- Native scalability, large cluster parallelism
- System handles fault tolerance automatically
- Decent FOSS stacks (Hadoop and later, Spark)
- **Catch:** Users must learn "art" of casting program as MapReduce
- Map operates record-wise; Reduce aggregates globally
- But MR libraries now available in many PLs: C/C++, Java, Python, R, Scala, etc.

Abstract Semantics of MapReduce
Map(): Process one "record" at a time independently
- A record can physically batch multiple data examples/tuples
- Dependencies across Mappers not allowed
- Emit 1 or more key-value pairs as output(s)
- Data types of input vs. output can be different
- **Reduce()**: Gather all Map outputs across workers sharing same key into an Iterator (list)
- Apply aggregation function on Iterator to get final output(s)
- **Input Split:**
- Physical-level shard to batch many records to one file "block" (HDFS default: 128MB?)
- User/application can create custom Input Splits
- **First step:** Transform text docs into relations and load: Part of the ETL stage

Suppose we pre-divide each doc into words w/ schema: DocWords (DocName, Word)
- **Second step:** a single, simple SQL query!

More MR Examples: Select Operation
- **Input Split:** Shard table tuple-wise
- **Map()**: On tuple, apply selection condition; if satisfies, emit key-value (KV) pair with dummy key, entire tuple as value
- **Reduce()**:
  - Not needed! No cross-shard aggregation here
- These kinds of MR jobs are called "Map-only" jobs

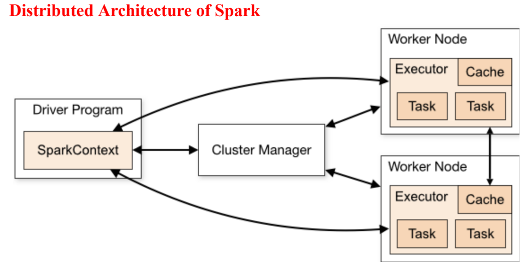
More MR Examples: Simple Agg
- Suppose it is algebraic aggregate (SUM, AVG, MAX, etc.)
- **Input Split:**
  - Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. Stats; emit pair with single global dummy key and incr. stats as value
- **Reduce()**:
  - Since only one global dummy key, Iterator has all sufficient stats to unify into global agg.

More MR Examples: GROUP BY Agg
- Assume it is algebraic aggregate (SUM, AVG, MAX, etc.)
- **Input Split**:
  - Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. Stats; emit pair with grouping attribute as key and stats as value
- **Reduce()**:
- Iterator has all suff. stats for a single group; unify those to get result for that group
- Different reducers will output different groups' results

More MR Examples: Matrix Sum of Squares
- Very similar to simple SQL aggregates
- **Input Split:**
- Shard table tuple-wise
- **Map():**
  - On agg. attribute, compute incr. stats; emit pair with single global dummy key and stats as value
- **Reduce():**
  - Since only one global dummy key, Iterator has all sufficient stats to unify into global agg.

What is Hadoop then?
- FOSS **system implementation** with
→ MapReduce as programming model, and
→ HDFS as filesystem
- MR user API; input splits, data distribution, shuffling, and fault tolerances handled by Hadoop under the hood
- Exploded in popularity in 2010s: 100s of papers, 10s of products
- A "revolution" in scalable+parallel data processing that took the DB world by surprise
- But nowadays Hadoop largely supplanted by Spark

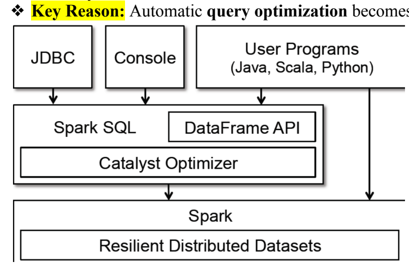
Apache Spark
- **Dataflow programming** model (subsumes most of Relational Algebra; MR)
- Inspired by Python Pandas style of chaining functions
- Unified storage of relations, text, etc.; custom programs
- Custom design (and redesign) from scratch
- Tons of sponsors, gazillion bucks, unbelievable hype!
- **Key idea vs Hadoop:** exploit distributed memory to cache data
- **Key novelty vs Hadoop:** lineage-based fault tolerance
- Open-sourced to Apache; commercialized as Databricks

Distributed Architecture of Spark


Resilient Distributed Datasets
Key concept in Spark.
- RDD has been the primary user-facing API in Spark since its inception. At the core an RDD is an immutable distributed collection of elements of your data,
  - partitioned across nodes in your cluster
  - that can be operated in parallel with a low-level API that offers transformations and actions.
- Good for dataset low-level transformation, actions and control.
- Good for unstructured data.
- Good for functional programming data manipulation.
- Not recommended for imposing a schema on your data.
- Lacks some optimization and performance benefits

Spark's Dataflow Programming Model
Transformations are relational ops, MR, etc. as functions
Actions are what force computation; aka **lazy evaluation**

Spark DF API and SparkSQL
- Databricks now recommends SparkSQL/DataFrame API; avoid RDD AP unless really needed!
- **Key Reason:** Automatic **query optimization** becomes more feasible

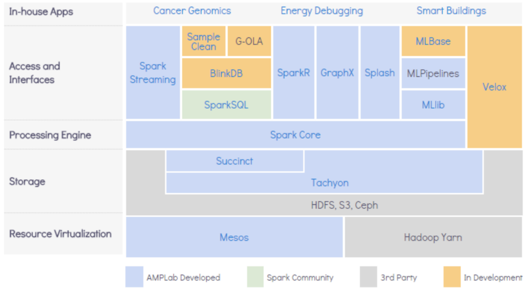


Query Optimization in Spark
- Common automatic query optimizations (from RDBMS world) are now performed in Spark's Catalyst optimizer:
- **Projection pushdown:** Drop unneeded columns early on
- **Selection pushdown:** Apply predicates close to base tables
- **Join order optimization:** Not all joins are equally costly
- Fusing of aggregates

Comparing Spark's APIs
A rough comparison of
RDD, DataFrames and Koalas (databricks pandas-like module)

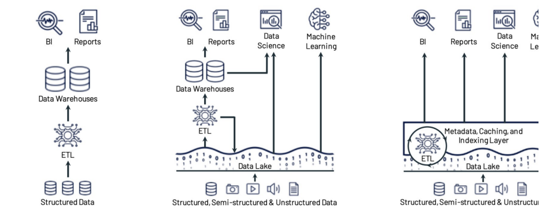
	RDD	DataFrame	Koalas
Abstraction Level	Low	High	High
Named Columns	No	Yes	Yes
Support for Query Optimization	No	Yes	Yes
Programming Mode	map-reduce	Dataflow, SQL	Pandas-like
Best suited for	Unstructured data Low-level ops Folks who like func. PLs and MapReduce	Structured data High-level ops Folks who know SQL, Python, R	Structured data Lower barrier to entry for folks who only know Pandas or Dask

Spark-based Ecosystem of Tools



New Paradigm of Data "Lakehouse"

❖ **Data "Lake":** Loose coupling of data file format and data/query processing stack (vs RDBMS's tight coupling); many frontends



(a) First-generation platforms. (b) Current two-tier architectures. (c) Lakehouse platforms.

References and More Material

❖ **MapReduce/Hadoop:**
  ❖ MapReduce: Simplified Data Processing on Large Clusters.

❖ **Spark:**
  ❖ Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing.

Example: Batch Gradient Descent

❖ Very similar to algebraic SQL; vector addition
❖ **Input Split:** Shard table tuple-wise
❖ **Map():**
  ❖ On tuple, compute per-example gradient; add these across examples in shard; emit partial sum with single dummy key
❖ **Reduce():**
  ❖ Only one global dummy key, Iterator has partial gradients; just add all those to get full batch gradient.

Primer: K-Means Clustering

❖ **Basic Idea:** Identify clusters based on Euclidean distances; formulated as an optimization problem
❖ **Llyod's algorithm:** Most popular heuristic for K-Means
❖ **Input:** n x d examples/points
❖ **Output:** k clusters and their centroids
1. Initialize k centroid vectors and point-cluster ID assignment
2. Assignment step: Scan dataset and assign each point to a cluster ID based on which centroid is nearest
3. Update step: Given new assignment, scan dataset again to recompute centroids for all clusters
4. Repeat 2 and 3 until convergence or fixed # iterations

K-Means Clustering in MapReduce

❖ **Input Split:** Shard the table tuple-wise
  ❖ Assume each tuple/example/point has an ExampleID
❖ Need 2 jobs! 1 for Assignment step, 1 for Update step
❖ 2 external data structures needed for both jobs:
  ❖ Dense matrix A: k x d centroids; ultra-sparse matrix B: n x k assignments
  ❖ A and B first broadcast to all Mappers via HDFS; Mappers can read small data directly from HDFS files
❖ Job 1 read A and creates new B
❖ Job 2 reads B and creates new A

K-Means Clustering in MapReduce

❖ A: k x d centroid matrix; B: n x k assignment matrix
❖ **Job 1 Map():** Read A from HDFS; compute point's distance to all k centroids; get nearest centroid; emit new assignment as output pair (PointID, ClusterID)
❖ No Reduce() for Job 1; new B now available on HDFS
❖ **Job 2 Map():** Read B from HDFS; look into B and see which cluster point got assigned to; emit point as output pair (ClusterID, point vector)
❖ **Job 2 Reduce():** Iterator has all point vectors of a given ClusterID; add them up and divide by count; got new centroid; emit output pair as (ClusterID, centroid vector)

Building Stage of ML Lifecycle

❖ Perform **model selection**, i.e., convert prepared ML-ready data to **prediction function(s)** and/or other analytics outputs
❖ What makes model building challenging/time-consuming?
❖ **Heterogeneity** of data sources/formats/types
❖ **Configuration complexity** of ML models
❖ Large **scale** of data
❖ **Long training runtimes** of some models
❖ **Pareto optimization on criteria** for application
❖ **Evolution** of data-generating process/application
❖ Perform **model selection**, i.e., convert prepared ML-ready data to **prediction function(s)** and/or other analytics outputs
❖ Data scientist / ML engineer must steer 3 key activities that invoke ML **training** and **inference** as sub-routines:
  1. Feature Engineering (FE): Appropriate signals representation for domain of prediction function.
  2. Algorithm/Architecture Selection (AS): Choice of prediction functions class (incl. artificial neural networks (ANN) architecture).
3. Hyper-parameter Tuning (HT): Model improvement (accuracy, etc.) by configuring ML "knobs"

Model Selection Process

❖ Model selection is usually an iterative exploratory process with human making decisions on FE, AS, and/or HT
❖ Increasingly, automation of some or all parts possible: **AutoML**
❖ Decisions on FE, AS, HT guided by many constraints/metrics: prediction accuracy, data/feature types, interpretability, tool availability, scalability, runtimes, fairness, legal issues, etc.
❖ Decisions are typically application-specific and dataset-specific; recall Pareto surfaces and tradeoffs

Feature Engineering

❖ Converting prepared data into a feature vector representation for ML training and inference
  ❖ Aka feature extraction, representation extraction, etc.
❖ Umbrella term for many tasks dep. on type of ML model trained:
  1. Recoding and value conversions
  2. Joins and/or aggregates
  3. Feature interactions
  4. Feature selection
  5. Dimensionality reduction
  6. Temporal feature extraction
  7. Textual feature extraction and embeddings
  8. Learned feature extraction in deep learning

1. Recoding and value conversions

❖ Common on relational/tabular data
❖ Typically needs some global column stats + code to reconvert each tuple (example's feature values)

Example:
Decision trees can use categorical features directly but GLMs support only numeric features; need numerical vector such as **one-hot Encoded, weight of evidence / target encoding, integer encoding, embedding (via additional DL model), etc**

Example:
GLMs and ANNs need **standardization** (either mean/stdev or min/max based) and **decorrelation**
Scaling global stats: How to scale mean/stdev/max/min?
Reconversion: Tuple-level function to modify number using stats. How to scale?

Example:
Some models like Bayesian Networks or Markov Logic Networks benefit from (or even need) binning/discretization of numerics
Scaling global stats: How to scale histogram computations?
Reconversion: Tuple-level function to convert number to bin ID

2. Joins and Aggregates

❖ Common on relational/tabular data
❖ Most real-world relational datasets are multi-table; require key-foreign key joins, aggregation-and-key-key-joins, etc.

3. Polynomials and Feature Interactions

❖ Sometimes used on relational/tabular data, especially for high-bias models like GLMs
❖ Pairwise is common; ternary is not unheard of
❖ No global stats, just a tuple-level function
❖ Popularity of this has reduced due to GBMs popularity for tabular data, which encode nonlinearities and interactions as part of the learning process.

4. Feature Selection

❖ Often used on high dimensional relational/tabular data
❖ **Basic Idea**: Instead of using whole feature set, use a subset
❖ Formulated as a discrete optimization problem
  ❖ NP-Hard in #features in general
  ❖ Many heuristics exist in ML/data mining; typically rely on some information theoretic criteria
  ❖ Typically scaled as "outer loops" over training/inference
❖ Some ML users also prefer human-in-the-loop approach

5. Dimensionality Reduction

❖ Often used on relational/structured/tabular data
❖ **Basic Idea:** Transforms features to a different latent space
❖ Examples: Principal Component Analysis (PCA), Singular Value Decomposition (SVD), Linear Discriminant Analysis (LDA), Matrix factorization
❖ Feat. sel. preserves semantics of each feature but dim. red. typically does not—combines features in "nonsensical" ways
❖ Scaling this is non-trivial! Similar to scaling individual ML training algorithms (later)

6. Temporal Feature Extraction

❖ Many relational/tabular data have time/date
❖ Per-example reconversion to extract numerics/categoricals
❖ Sometimes global stats needed to calibrate time
❖ Complex temporal features studied in time series mining
Reconversion: Tuple-level function (many-to-one) to extract numbers/categories

7. Textual Feature Extraction

❖ Many relational/tabular data have text columns; in NLP, whole example is often just text
❖ Most classifiers cannot process text/strings directly
❖ Extracting numerics from text studied in text mining

Example:
Bag-of-words features: count number of times each word in a given vocabulary arises; need to know vocabulary first
Scaling global stats: How to get vocabulary?
Reconversion: Tuple-level function to count words; look up index
❖ **Knowledge Base-based**: Domain-specific knowledge bases like entity dictionaries (e.g., celebrity or chemical names) help extract domain-specific features
❖ **Embedding-based:**
  ❖ Numeric vector for a text token; popular in NLP
  ❖ Offline training of function from string to numeric vector in self-supervised way on large text corpus (e.g., Wikipedia); embedding dimensionality is a hyper-parameter
  ❖ Pre-trained word embeddings (Word2Vec and GloVe) and sentence embeddings (Doc2Vec) available off-the-shelf; to scale, just use a tuple-level conversion function

8. Learned Feature Extraction in DL

❖ A big win of Deep Learning (DL) is no manual feature engineering on unstructured data
❖ DL is not common on structured/tabular data, but growing in popularity. See: https://arxiv.org/pdf/2110.01889.pdf
❖ DL is very versatile: almost any data type as input and/or output:
❖ Convolutional NNs (CNNs) over image tensors
❖ Recurrent NNs (RNNs) and Transformers over text
❖ Graph NNs (GNNs) over graph-structured data
❖ Neural architecture specifies how to extract and transform features internally with weights that are learned
❖ **Software 2.0:** Buzzword for such "learned feature extraction" programs vs old hand-crafted feature engineering

Hyper-Parameter Tuning

❖ **Hyper-parameters:** Knobs for an ML model or training algorithm to control bias-variance tradeoff in a dataset-specific manner to make learning effective
❖ Examples:
  ❖ GLMs: L1 or L2 regularizer to constrain weights
  ❖ All gradient methods: learning rate
  ❖ Mini-batch Stochastic Gradient Descent: batch size
❖ HT is an "outer loop" around training/inference
❖ Most common approach: **grid search;** pick set of values for each hyperpa
❖ Also common: **random search** to subsample from grid
❖ Complex AutoML heuristics exist too for HT, e.g., Bayesian

Algorithm Selection in "classical" ML

❖ Not much to say; ML user typically picks models/algorithms in advance
❖ Best practice: first train more simple models (log. reg.) as baselines; then try more complex models (XGBoost)
❖ **Ensembles:** Build diverse models and aggregate predictions. Even for tabular data, ensembles yield better results and often win Kaggle comps with a few % boost in performance.
❖ More critical in DL; neural arch. is **inductive bias** in classical ML parlance; controls feature learning and bias-variance tradeoff
❖ Some applications: Many off-the-shelf pre-trained DL models to do "transfer learning," e.g., see models at HuggingFace.co
❖ Other applications: Swap pain of hand-crafted feature eng. for pain of neural arch. eng.! Neural arch probably a better interview skill

Automated Model Selection / AutoML

❖ It depends. HT and most of FE already automated mostly in practice; (neural) AS is often application-dictated
❖ AutoML tools/systems now aim to reduce data scientist's work; or even replace them?! ;)
Automated Model Selection / AutoML
Q: Can we automate the whole model selection process?
  ❖ **Pros:** Ease of use; lower human cost; easier to audit; improves ML accessibility
  ❖ **Cons:** Higher resource cost; less user control; may waste domain knowledge; may leave performance on the table
❖ Pareto-optima; hybrids possible

Major ML Model Families/Types

Generalized Linear Models (GLMs); from statistics
Bayesian Networks; inspired by causal reasoning
Decision Tree-based: CART, Random Forest, Gradient-Boosted Trees (GBT), etc.; inspired by symbolic logic
Support Vector Machines (SVMs); inspired by psychology
Artificial Neural Networks (ANNs): Multi-Layer Perceptrons (MLPs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Transformers, etc.; inspired by brain neuroscience
Unsupervised: Clustering (e.g., K-Means), Matrix Factorization, Latent Dirichlet Allocation (LDA), etc.

Scalable ML Training Systems

❖ Scaling ML training is involved and model type-dependent
❖ Orthogonal Dimensions of Categorization:
  1. Scalability: In-memory libraries vs Scalable ML system (works on larger-than-memory datasets)
  2. Target Workloads: General ML library vs Decision treeoriented vs De
  3. Implementation Reuse: Layered on top of scalable data system vs Custom from-scratch framework

Model Serving / Deployment

❖ A trained/learned ML model is just a prediction function: **f: Dx → Dy**
❖ A major consideration is, online/realtime vs. offline/batch.
❖ In the offline scenario, serving a model is more trivial where it is another processing function that we apply.
❖ In the online scenario, we become concerned with millisecond latency for responses, setting up APIs, load balancing, and monitoring.