Smallest font

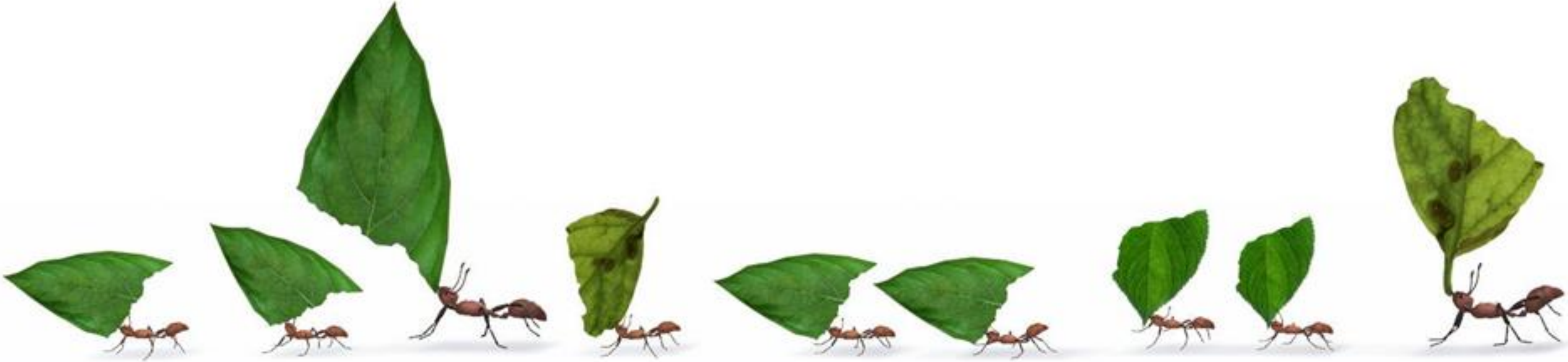
Please turn off and put away your cell phone

Calibration slide

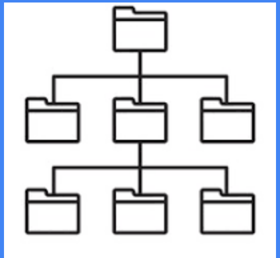
These slides are meant to help with note-taking They are no substitute for lecture attendance

Smallest font

Big Data



Week 02: Centralized systems
File systems
Relational databases



DS-GA 1004: Big Data

Announcements

- This week: Lab 2 (SQL)
- The office hour schedule of all teaching staff is now in effect
- Download the sittyba today for an updated version (all office hours, accurate due dates, working links, etc.)
- **HW 1 is due next Monday (02/10)**

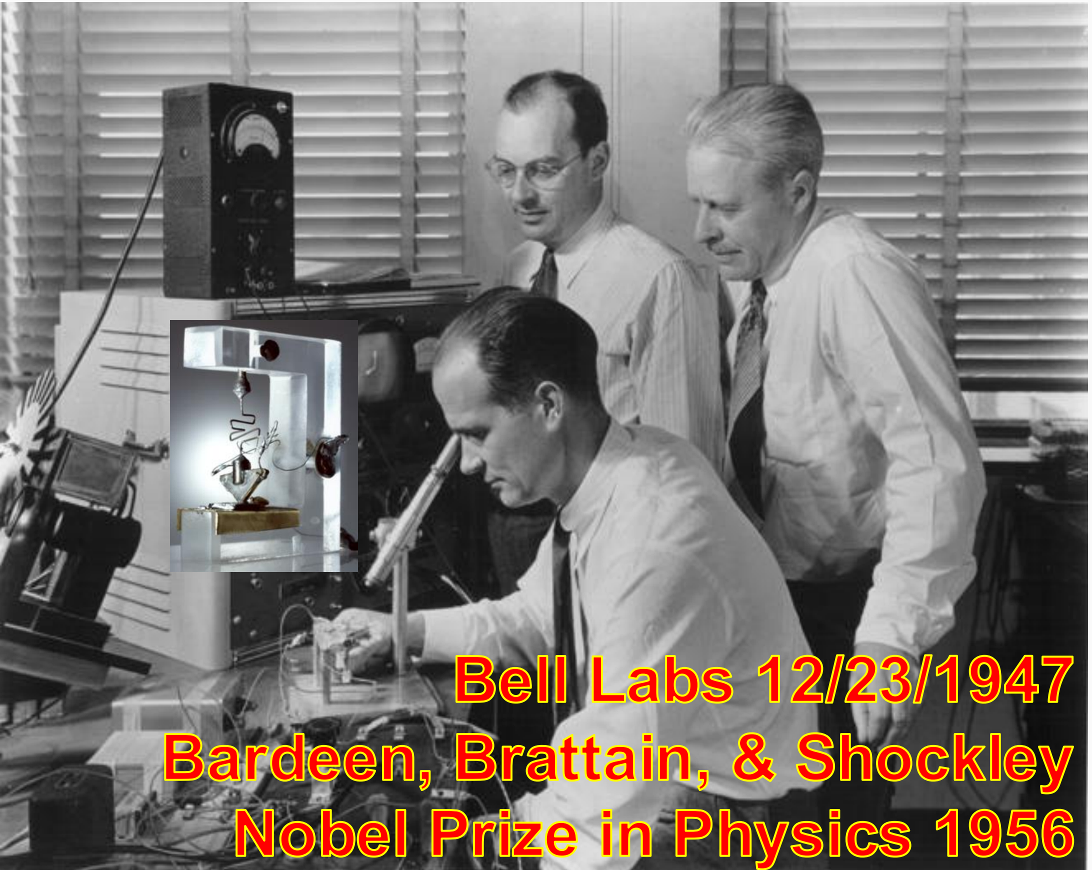
Confusion, Doubt, & Struggle (CDS)

1) Is a large array that came from a random number generator “data” (for the purpose of this class)?

221	46	168	42	36	165	40	207	83	116	119	103	63	197	127	96	18	227	86	208	5	200	34
37	65	175	35	48	228	157	44	76	133	76	254	39	3	208	51	226	165	25	55	166	140	129
241	155	4	188	254	12	232	206	168	176	211	148	42	124	97	201	251	187	104	55	240	150	140
18	142	58	115	236	82	195	70	177	186	174	222	172	230	131	127	162	39	79	3	131	145	108
41	29	10	77	43	115	45	192	173	208	112	252	52	1	62	13	60	55	198	206	255	110	14
89	79	193	76	181	99	161	247	248	67	16	120	3	121	203	70	111	200	44	48	203	143	204
39	113	0	122	30	136	75	37	33	232	181	4	52	201	212	246	22	90	28	133	51	145	142
181	118	76	217	39	220	176	148	152	184	226	123	110	199	193	247	16	12	228	177	85	99	169
30	178	154	88	89	7	25	221	21	184	96	156	216	237	38	125	20	178	96	91	18	47	242
7	216	71	28	223	201	223	119	208	212	167	81	113	155	137	71	243	36	241	55	216	18	219
185	86	233	199	77	37	149	88	248	139	104	157	125	251	19	250	58	118	21	2	212	171	71
152	186	153	50	238	92	33	150	182	209	202	132	227	4	117	59	203	131	185	195	110	12	181
57	20	111	211	189	20	224	12	212	155	48	65	138	202	207	36	32	54	155	138	110	231	126
129	2	225	103	71	255	155	181	72	165	249	31	123	79	32	242	62	236	254	207	229	66	72
37	234	192	171	161	93	109	122	166	250	149	42	128	105	213	193	90	58	127	114	206	17	169
107	33	102	206	197	69	217	150	169	33	181	252	219	67	193	201	80	83	81	146	54	225	241
101	184	6	195	192	47	210	65	218	138	171	16	63	71	89	210	139	70	158	99	63	174	129
238	52	4	244	180	146	193	170	46	117	130	118	9	183	233	242	19	162	245	178	238	120	1
210	131	245	166	240	162	15	197	228	54	170	195	126	213	195	26	31	141	102	122	69	36	71
146	207	204	121	64	181	122	228	239	163	205	161	12	216	15	226	102	156	85	243	175	225	28
124	103	215	34	97	61	156	3	93	14	162	10	169	37	213	137	87	9	36	161	93	143	150
99	8	238	208	169	203	82	184	231	112	210	42	10	103	102	45	217	154	72	72	14	242	78
104	96	91	45	203	20	162	68	145	190	119	198	172	254	146	68	130	176	69	202	14	34	224
162	246	111	88	130	149	89	34	226	185	121	171	224	204	139	192	121	70	85	187	44	104	37
163	230	133	87	54	208	148	70	116	192	28	60	206	105	233	92	197	211	240	143	12	248	144
7	181	227	145	18	76	130	46	94	102	221	90	113	23	125	17	43	164	227	179	92	93	233
158	68	51	150	45	42	199	232	230	179	212	6	175	80	55	17	243	219	214	215	135	135	231
41	123	86	29	10	41	49	229	204	10	153	133	122	217	248	214	98	216	21	225	141	239	158
122	61	191	200	53	111	107	159	190	140	46	233	43	0	244	67	47	137	177	47	178	188	61
123	202	32	15	239	70	81	189	213	53	88	195	81	171	223	3	64	123	16	110	97	64	186
37	213	24	182	117	94	55	144	226	32	219	163	211	106	141	236	62	11	192	63	171	51	215
92	111	0	77	139	226	216	126	114	128	156	216	40	125	40	178	87	209	158	192	1	117	146
202	141	123	64	7	36	254	43	195	46	225	27	157	243	116	101	94	232	180	199	121	234	132
164	127	94	40	58	106	90	138	160	193	165	241	118	175	159	224	31	209	164	227	147	57	110
32	223	182	83	180	43	106	34	149	7	34	247	7	208	110	226	216	225	150	251	171	198	101
232	113	212	86	142	57	99	213	194	179	85	16	231	4	161	69	17	105	78	245	8	193	215
187	125	154	10	215	176	97	101	164	85	145	143	139	209	19	106	183	164	104	34	136	78	195
7	83	106	153	167	164	145	241	108	127	159	19	5	123	200	77	72	75	24	232	253	88	213
254	230	83	212	6	201	100	196	53	76	130	71	234	165	14	41	73	196	148	119	190	56	193
41	45	234	96	140	229	111	189	1	144	127	85	250	247	190	42	173	174	54	117	58	187	100
131	14	44	224	99	238	48	215	147	217	110	181	114	98	111	67	220	91	207	187	116	81	118
19	13	250	31	103	220	158	214	130	182	179	214	199	214	207	217	221	249	58	98	176	241	206

Confusion, Doubt, & Struggle (CDS)

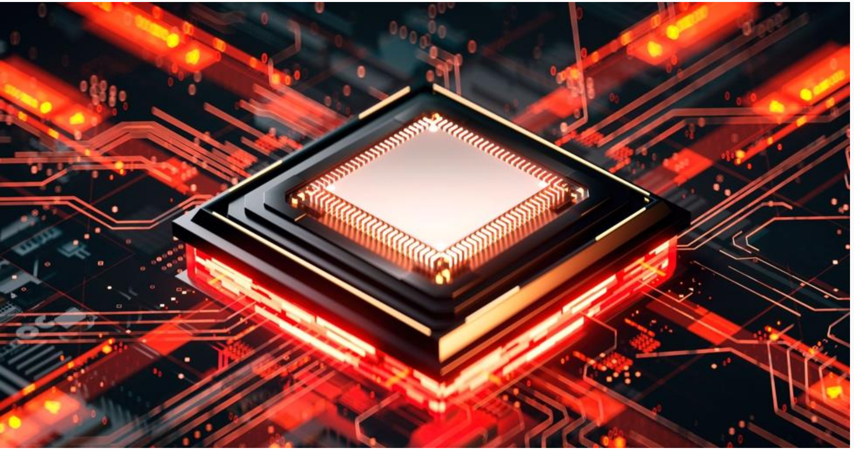
2) What is a transistor and why does it matter how many of them there are on a chip?



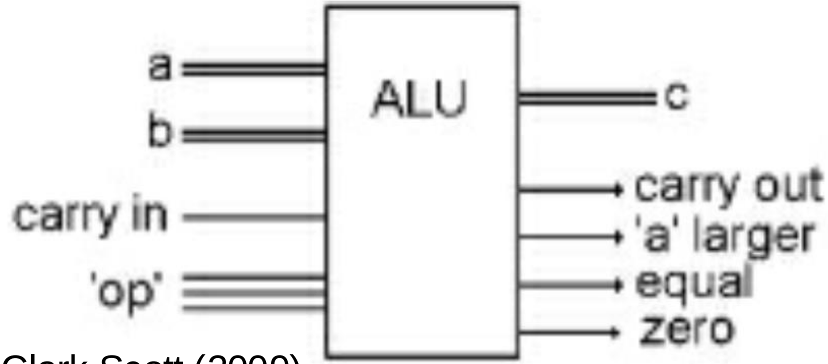
- Transistors are very fast and reliable electronic switches.
- They allow currents to either flow in a wire or not, implementing binary logic.
- Since their invention, it has been possible to miniaturize them by several orders of magnitude (modern transistors = single digit nm), allowing to pack billions of them on a chip.
- The physical implementation is beyond this class, but concisely – when a voltage is applied to a “doped” semiconductor (usually germanium or silicone) in a transistor, it effectively opens a gate that allows current to flow (otherwise it remains closed).

Confusion, Doubt, & Struggle (CDS)

3) What is a CPU (central processing unit) aka "the processor"?



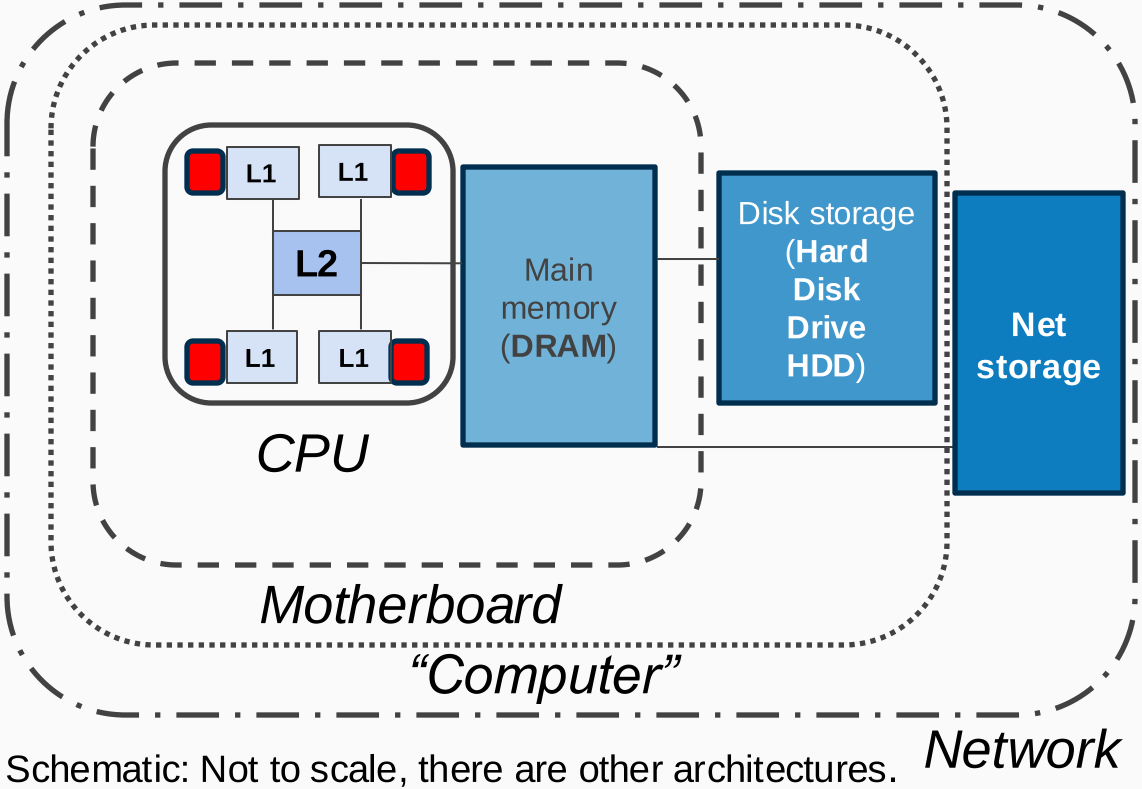
- The CPU is the "brain" of the computer.
- It consists of several components, including many transistors that are arranged in an "integrated circuit".
- CPUs store a little bit of data locally (in the L1/L2 caches) and – together with the instructions (from programs) process them, which results in outputs.
- The clock of the CPU generates a signal that synchronizes CPU operations (several billion times a second in modern CPUs).
- Each clock cycle, a CPU (core) can perform one operation, e.g. reading data from memory, executing instructions or writing data to memory.



Clark Scott (2009)

Confusion, Doubt, & Struggle (CDS)

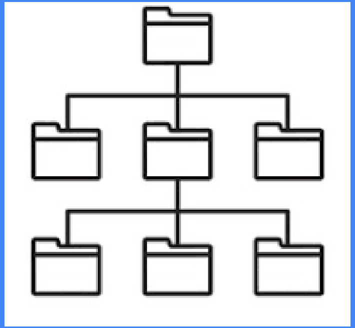
4) How does a (modern) computer work?



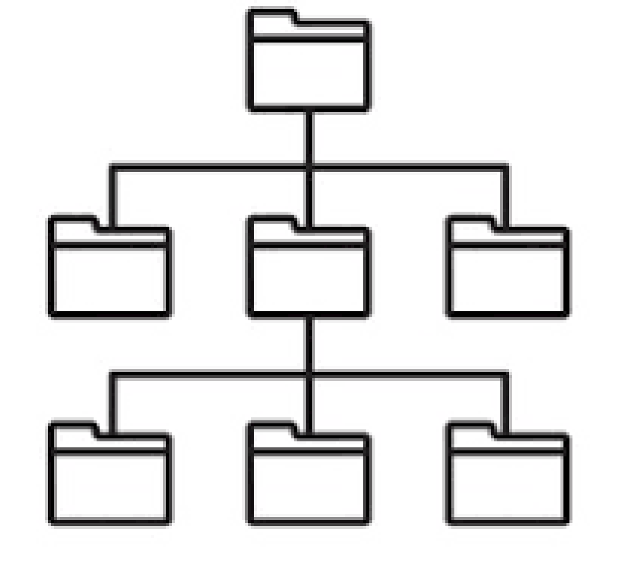
Schematic: Not to scale, there are other architectures.
Not depicted: Non class-relevant components (e.g. fan, power,…)

- Why does this matter?
- In a computer, electrical signals travel down wires at a substantial fraction (50-75%, depending on material and temperature) of the speed of light in vacuum.
- The speed of light in vacuum corresponds to a distance just shy of a foot (11.8” = 30 cm) per ns.
- As signals go back and forth to implement the computation, this can add up.
- Less distance = faster execution.
- Distance is time

Today: 4 parts



- File systems
- Relational databases
- SQL
- Transaction integrity



File systems!

- To understand modern distributed tools, it helps to understand the solutions of previous generations.

- What problems did people face in designing:
    - Relational databases?
    - Map-reduce?
    - Spark?
    - Dask?

File systems

003_Mozart_RequiemInDMinor_Section4_15.mp4
003_Mozart_RequiemInDMinor.mp3
004_Beethoven_FurElise_Section1_05.mp4
004_Beethoven_FurElise_Section1_10.mp4
004_Beethoven_FurElise_Section1_15.mp4
004_Beethoven_FurElise_Section2_05.mp4
004_Beethoven_FurElise_Section2_10.mp4
004_Beethoven_FurElise_Section2_15.mp4
004_Beethoven_FurElise_Section3_05.mp4
004_Beethoven_FurElise_Section3_10.mp4
004_Beethoven_FurElise_Section3_15.mp4
004_Beethoven_FurElise_Section4_05.mp4
004_Beethoven_FurElise_Section4_10.mp4
004_Beethoven_FurElise_Section4_15.mp4
004_Beethoven_FurElise.mp3
005_Debussy_ClaireDeLune_Section1_05.mp4
005_Debussy_ClaireDeLune_Section1_10.mp4
005_Debussy_ClaireDeLune_Section1_15.mp4
005_Debussy_ClaireDeLune_Section2_05.mp4
005_Debussy_ClaireDeLune_Section2_10.mp4
005_Debussy_ClaireDeLune_Section2_15.mp4
005_Debussy_ClaireDeLune_Section3_05.mp4
005_Debussy_ClaireDeLune_Section3_10.mp4
005_Debussy_ClaireDeLune_Section3_15.mp4
005_Debussy_ClaireDeLune_Section4_05.mp4
005_Debussy_ClaireDeLune_Section4_10.mp4
005_Debussy_ClaireDeLune_Section4_15.mp4
005_Debussy_ClaireDeLune.mp3

- You should all (?) be intuitively familiar with these
- Use directories to organize your data
- Structured data can be stored as files

⇒ data persists across application runs

File systems

003_Mozart_RequiemInDMinor_Section4_15.mp4
003_Mozart_RequiemInDMinor.mp3
004_Beethoven_FurElise_Section1_05.mp4
004_Beethoven_FurElise_Section1_10.mp4
004_Beethoven_FurElise_Section1_15.mp4
004_Beethoven_FurElise_Section2_05.mp4
004_Beethoven_FurElise_Section2_10.mp4
004_Beethoven_FurElise_Section2_15.mp4
004_Beethoven_FurElise_Section3_05.mp4
004_Beethoven_FurElise_Section3_10.mp4
004_Beethoven_FurElise_Section3_15.mp4
004_Beethoven_FurElise_Section4_05.mp4
004_Beethoven_FurElise_Section4_10.mp4
004_Beethoven_FurElise_Section4_15.mp4
004_Beethoven_FurElise.mp3
005_Debussy_ClaireDeLune_Section1_05.mp4
005_Debussy_ClaireDeLune_Section1_10.mp4
005_Debussy_ClaireDeLune_Section1_15.mp4
005_Debussy_ClaireDeLune_Section2_05.mp4
005_Debussy_ClaireDeLune_Section2_10.mp4
005_Debussy_ClaireDeLune_Section2_15.mp4
005_Debussy_ClaireDeLune_Section3_05.mp4
005_Debussy_ClaireDeLune_Section3_10.mp4
005_Debussy_ClaireDeLune_Section3_15.mp4
005_Debussy_ClaireDeLune_Section4_05.mp4
005_Debussy_ClaireDeLune_Section4_10.mp4
005_Debussy_ClaireDeLune_Section4_15.mp4
005_Debussy_ClaireDeLune.mp3

- You should all (?) be intuitively familiar with these
- Use directories to organize your data
- Structured data can be stored as files

  ⇒ data persists across application runs

- Some great properties:
  - **Easy** to implement
  - **Data** does not vanish
  - **Inherently** organized (tree structure)
  - **Portable** across systems

File systems can be awesome!

So why are we not always using them (exclusively)?

Reasons not to rely (only) on file systems

- Does not expose or exploit the structure of data
  - What if I want to search by file contents? Better options than brute force?
- Each query / analysis required writing a new program
  - Little re-usability between similar analyses
  - Or the same analysis with slightly different data structures
- Directory hierarchies may be too restrictive

When are file systems not awesome?

- When your data is structured along multiple axes
- When data have complex interactions
- When your analyses are complex
- **Relational databases to the rescue!**

Databases did *not* replace file systems

- File archives are still the most common way to share large datasets
  - But we usually include some metadata/indexing structure as well
- As we'll see soon, **Hadoop** relies on a **distributed** file system
  - DB **abstractions** can be built on top
  - But this comes with **restrictions** on file contents and structure

File-based storage

- Often, data of lives (permanently) **on disk / in the file-system**
- If it's small, we can load it into **main memory:**
  - df ← read_csv('my_data.csv')
  - analyze(df)
- If it's **too big**, we have several options:
  - **Sampling** / approximate computation
  - **Stream processing** (one or few records at a time)
  - **Data structures** / index structures
  - **Parallel computation**
  - ...
  - Buy more memory

Good solutions often combine two or more of these strategies!

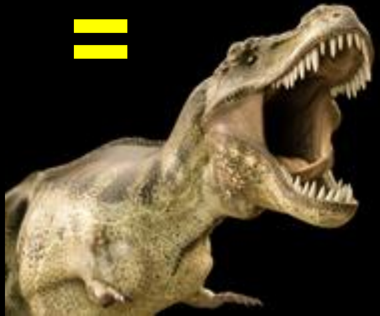
We'll see that often this semester.

Where we’re going next

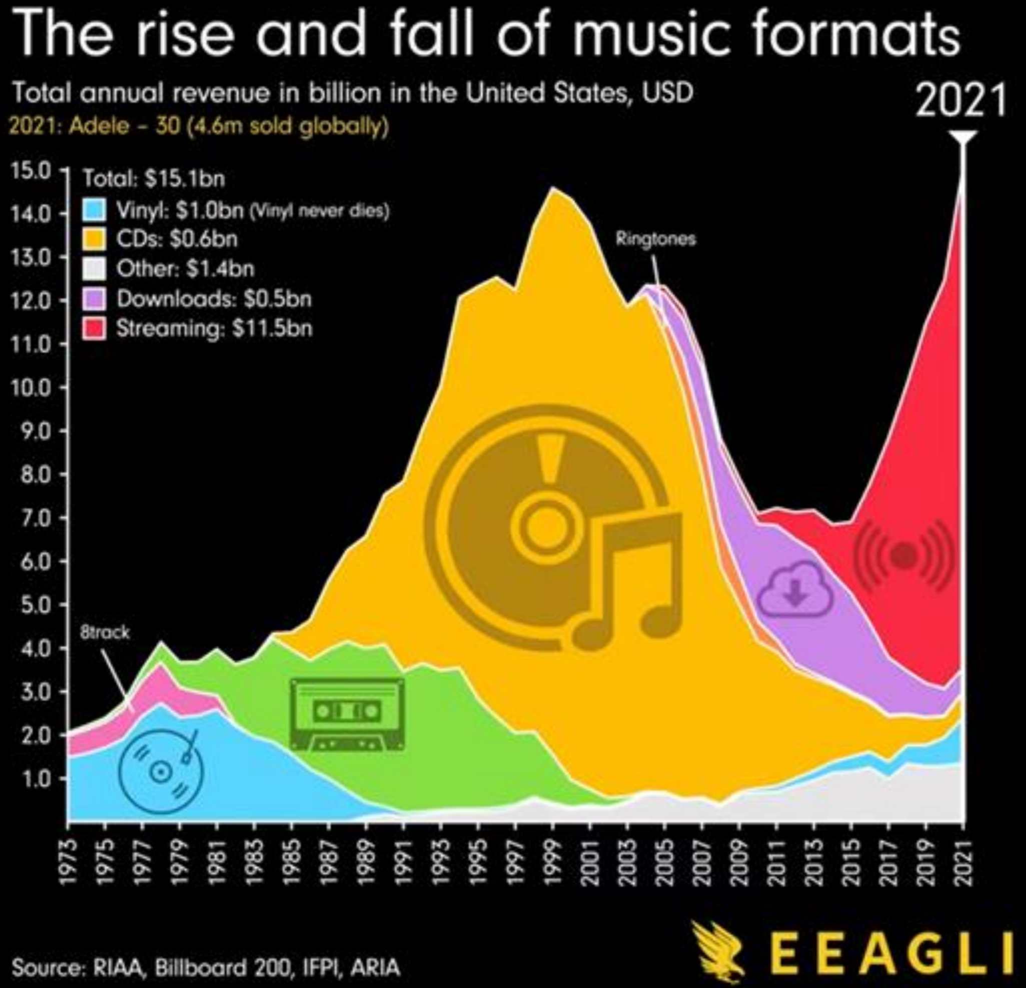
- **Database management systems** (DBMS)
  - Provide a standardized interface to store, load, and process data
- The **relational model** imposes constraints on how data is organized
  - i.e., tables / spreadsheets / dataframes
- Putting these two ideas together = **RDBMS**!

Relational databases are the Vynil of data storage and management

=



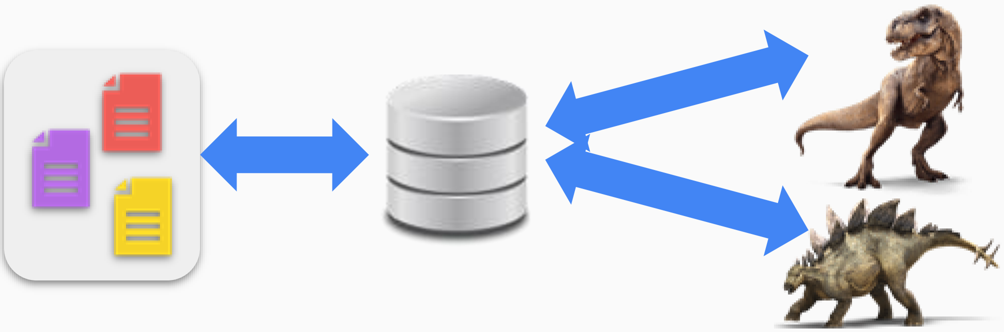
Source



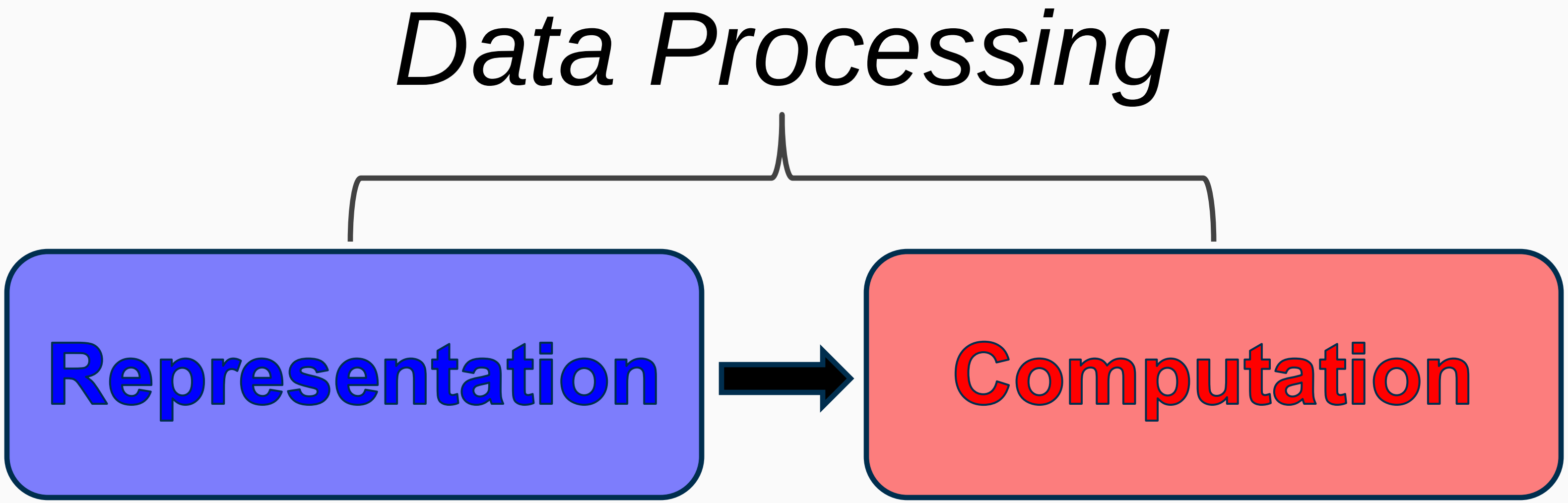
But why?

Database management systems (DBMS)

- DBMS's job is to provide
  - Data integrity / consistency
  - Concurrent access
  - Efficient storage and access
  - Standardized format / administration
  - Standardized query interface (language)
- DBMS come in many flavors
  - **Relational (RDBMS)**
  - Semi-structured (e.g., XML)
  - Object-oriented
  - Object-relational
  - ...



The philosophy of the relational model



The relational model constrains how the data is represented (through the use of schemas)

The relational model

- **High-level**: tables of data that you're probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.

- Each column of a table represents a **set** of possible values (numbers, strings, etc.)

The relational model

- **High-level**: tables of data that you're probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.

- Each column of a table represents a **set** of possible values (numbers, strings, etc.)

- A ***relation*** over sets $A_1, A_2, A_3, \ldots, A_n$ is a **subset** of their cartesian product
  - $R \subseteq A_1 \times A_2 \times A_3 \times \ldots \times A_n$
  - The **rows** of the table are elements of R, also known as ***tuples***
  - $(a_1, a_2, \ldots, a_n) \in R \Rightarrow a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$

Example: pachyderms

- $A_1 = \{s \mid s \text{ is a string}\}$
  A_2 = {"Miocene", "Pliocene", "Pleistocene", "Holocene", …}
  A_3 = {"Carnivore", "Herbivore", "Omnivore", …}
  A_4 = {False, True}

- Any A_i could be finite or infinite

- $R \subseteq A_1 \times A_2 \times A_3 \times A_4$ need not contain all combinations!

Species	Era	Diet	Extinct
Elephant	Holocene	Herbivore	False
Woolly Mammoth	Pleistocene	Carnivore	True
Mastodon	Pliocene	Herbivore	True

Aside: why "relations" and not "tables"?

- Relations are the abstract model of data
- **Table** refers to an **explicitly** constructed relation
  - I.E., records you've observed / collected
- Other relations in a DB:
  - **view**: a relation defined **implicitly**, and constructed **dynamically** at run-time
  - **temporary table**: the output of a **query**

Exercise: counting relations
Of a set A with 5 elements and a set B with 3 elements (this is not trivial)

- Count the combinations:

  $|A| = 5, |B| = 3 \Rightarrow |A \times B| = 15$

- # Relations = # Possible subsets of the combinations

  $|2^{A \times B}| = 2^{15}$

- Order matters! $A \times B \neq B \times A$. We need to count both!

  $2^{15} + 2^{15} = 2^{16}$

- But now we've double-counted the empty relation $\emptyset \in 2^{A \times B}$ and $\emptyset \in 2^{B \times A}$

  $\Rightarrow 2^{15} + 2^{15} - 1 = 2^{16} - 1 = 65535$

Properties of relations

- $R \subseteq A_1 \times A_2 \times A_3 \times \ldots \times A_n$ is a set
  - The tuples (rows) of R are **unordered**
  - Tuples are **unique** ⇒ no duplicates!
  - Relations over common domains (columns) can be combined by set operations

Species	Era	Diet	Extinct
Elephant	Holocene	Herbivore	False
Woolly Mammoth	Pleistocene	Carnivore	True
Mastodon	Pliocene	Herbivore	True

Properties of relations

- $R \subseteq A_1 \times A_2 \times A_3 \times \ldots \times A_n$ is a set
  - The tuples (rows) of R are **unordered**
  - Tuples are **unique** $\Rightarrow$ no duplicates!
  - Relations over common domains (columns) can be combined by set operations
- In practice, add a column (e.g., A_0) with **identifiers** to force uniqueness
  - This is not (usually) part of the data, but is generated automatically by the DBMS
  - ID fields are often used as **primary keys**, and give a default order to rows

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

Schemas

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Datacene	Data	False

- A relation is defined by a **schema:**

Pachyderms(id: int, Species: string, Era: string, Diet: string, Extinct: boolean)

- **Any** tuple (int, string, string, string, boolean) is valid under this schema
  - ⇒ Schemas enforce type (syntax), not semantics!

Schemas can be hard to design!

- Imagine making a schema to track customers

    Customer(id: int, lastname: string, firstname: string)

- **Are all strings valid as names**?
- **What constraints could/should you add to the name field of our customer database to ensure data integrity?**

This can go wrong very easily...

- Length constraints are problematic in both directions
- So are character set constraints (accents, spaces, etc.)
- Search and linkage are difficult if the data must be modified to fit schema
- **Be careful!**

Relational databases

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

id	Name	Species	Movie
1	Jumbo Jr.	Elephant	Dumbo
2	Ellie	Woolly Mammoth	Ice Age
3	Doug	Mastodon	One million BC

- A relational database consists of one or more relational schemas
- Structured data can be encoded by **joining** on **shared attributes**
- The collection of schemas defines your **data model**

Keys

id	First Name	Last Name	Age
1	Homer	Simpson	39
2	Marge	Simpson	39
3	Bart	Simpson	10
4	Homer	Thompson	39
5	Homer	Simpson	28

- Keys are what determine the **identity** of a row
- Keys can be simple (single column) or **compound** (two or more columns)
  - Example: (First Name, Last Name)
  - This prevents two rows with the same combination of first and last name
- You can have **primary** and **alternate keys**
  - Usually a good idea to keep a primary numeric key as well as others you may want...

Foreign Keys

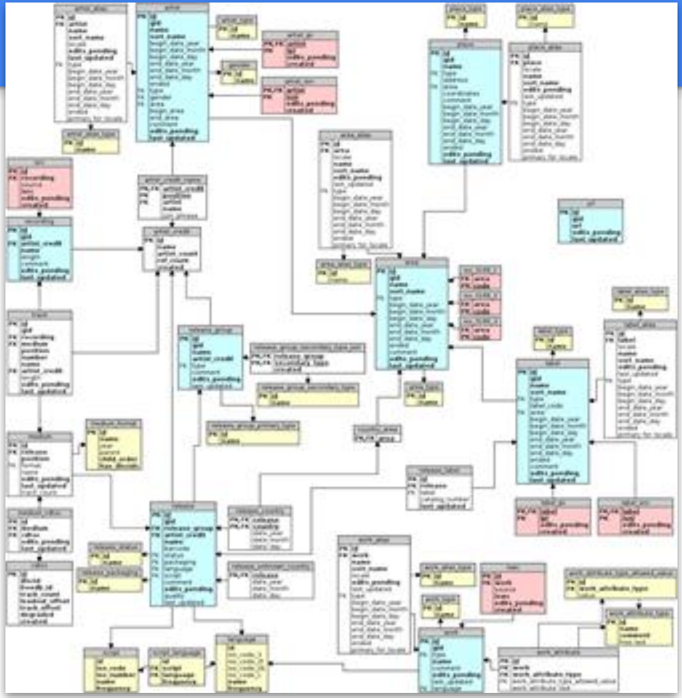
id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

- A **key** from one relation can be a **column** in another
  - This is called a **FOREIGN KEY** constraint
- This can be used to establish a link **between** the table in different tables/relations
- **This is not automatic**: must be included in the **schema** definition!

id	Name	pachydermID	Movie
1	Rocky	25	Jungle Book
2	Jumbo Jr.	1	Dumbo
3	Doug	3	One million BC

Normalization

- A database schema is **normal** if data is not redundantly stored
  - Use **identifiers**, not **values**, to link between relations
- Modifying a record is easy if it exists in exactly one place
- But it can also be difficult
  - Reading complex data can be cumbersome
  - Multiple levels of indirection



https://musicbrainz.org/doc/MusicBrainz_Database/Schema

Summary

Relations are powerful!

- We use relational data every day without thinking of it
- Databases consist of one or more relations
- Putting data into a relational model can make it easier to work with
- Schemas provide some degree of safety and validation

Structured Query Language (SQL)

- SQL is the language we use to talk to databases
    - Not a procedural language like Python or C
    - **Declarative**: state what you want, not how to compute it
- Think of it more like a **protocol** than a programming language
- SQL is an ANSI standard, but different implementations each have quirks
    - **MySQL** vs **Postgres** vs **SQLite** vs **MSSQL** vs **Oracle SQL**...

SELECTing data

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

- Get all rows: **SELECT** * **FROM** Pachyderms

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

SELECTing data

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

- Get all rows: **SELECT** * **FROM** Pachyderms

- Get some rows: **SELECT** * **FROM** Pachyderms **WHERE** Extinct = True

id	Species	Era	Diet	Extinct
1	Woolly Mammoth	Pleistocene	Carnivore	True
2	Mastodon	Pliocene	Herbivore	True

SELECTing data

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

- Get all rows: **SELECT** * **FROM** Pachyderms
- Get some rows: **SELECT** * **FROM** Pachyderms **WHERE** Awesome = True
- Get columns: **SELECT** Era, Species **FROM** Pachyderms **WHERE** id > 2

Era	Species
Pliocene	Mastodon
Miocene	Dataphant

Selection

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

- Remove tuples by filtering (WHERE …)
- And remove / rename / reorder columns
- Result of SELECT is always another relation
- You typically iterate over rows produced by SELECT in your host language

```
for row in db.execute('SELECT * FROM Pachyderms):
    print(row)
```

Python + sqlite3 example

Joining relations

id	Species	Era	Diet	Extinct
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

id	Name	Species	Movie
1	Jumbo Jr.	Elephant	Dumbo
2	Ellie	Woolly Mammoth	Ice Age
3	Doug	Mastodon	One million BC

- Data is typically structured across multiple relations
- We can combine relations by **JOIN**ing
- **SELECT** * from Pachyderms **JOIN** Character

A [?] JOIN B

Least specific

Most specific

CROSS JOIN	**All combinations** of rows (r_1, r_2) $r_1 \in A, r_2 \in B \Rightarrow A \times B$ (no matching condition)
[LEFT/RIGHT/FULL] OUTER JOIN	**All rows are retained** from A (**LEFT**) or B (**RIGHT**), even if no match is found. Fill missing data with **NULL**
INNER JOIN	**Only matching rows** are retained (Like **OUTER** but without NULLs)
NATURAL JOIN	Rows must match on **all shared columns** (Special case of **INNER**)

A [?] JOIN B

INNER and **OUTER** joins are most common

Least specific

Most specific

CROSS JOIN	**All combinations** of rows (r_1, r_2) $r_1 \in A,\ r_2 \in B \ \Rightarrow A \times B$ (no matching condition)
[LEFT/RIGHT/FULL] OUTER JOIN	**All rows are retained** from A (**LEFT**) or B (**RIGHT**), even if no match is found. Fill missing data with **NULL**
INNER JOIN	**Only matching rows** are retained (Like **OUTER** but without NULLs)
NATURAL JOIN	Rows must match on **all shared columns** (Special case of **INNER**)

Modifying data

INSERT INTO table (column1, column2, …)
VALUES (value1, value2, …),
[(row_2_value1, row_2_value2, …), ...]

UPDATE table
SET column1 = value1, column2 = value2, …
WHERE [some condition]

CAUTION:

Use your RDBMS library to sanitize queries

db.execute("SELECT * FROM Pachyderms WHERE species = '%s'" % name)

name variable becomes **part of** the query code!

db.execute(" SELECT * FROM Pachyderms WHERE species = '?'", name)

name variable is a **parameter** to the query code!

Aggregation

Aggregation queries

- Aggregation lets us summarize multiple tuples into a single result
- **Example**: find the average height of people within a zip code

SELECT Zip, AVG(Height) **FROM** Residents **GROUP BY** Zip

Zip	AVG(Height)
10003	2.68
10011	2

id	Species	Height	Address	Zip
1	Elephant	3.66	6 Washington Place	10003
2	Woolly Mammoth	2	60 Fifth Ave	10011
3	Mastodon	1.7	411 Lafayette St.	10003

Some useful aggregators

- **AVG, SUM, MIN, MAX** ⇐ what you see is what you get
- **COUNT**(**DISTINCT** x) ⇐ # of unique values of column x
- **COUNT**(*) vs **COUNT**(x) ⇐ # rows vs # non-nulls of a column
- **GROUP_CONCAT**(x) ⇐ concatenate (string) values
  GROUP_CONCAT(x, y) ⇐ same, but join with string y

Aggregation conditions

- **SELECT** … **WHERE** [condition] **GROUP BY** [fields]
- **WHERE** clause applies to input, not **output**
- What if you only want to keep certain groups (e.g., sum > 10)?
  - … **HAVING** [group condition]
  - **SELECT** sum(Height) **FROM** TallPachys **GROUP BY** zip **HAVING** sum(Height) > 10

Indexing

What is an index?

Why do we (sometimes) use an index when using databases?

Logical and Physical storage

- Relational schemas provide one view of the data
  - Set or list of **tuples**
- This may not be the best way to organize the data internally
  - Organizing by column can be much more efficient!
  - And what data structure do you use for each type? Hash tables? Trees?
- RDBMS abstract these decisions away from you (the user)
  - But sometimes you can help it out, if you know how data will be used

Indexing

id	Name	Continent	Street	State
1	Elephant	Asia	Z1402 Krishi Bank Road	Matlab
2	Woolly Mammoth	Europe	Friedhofstr. 18	Ba-Wü
3	Mastodon	North America	60 Fifth Avenue	New York

- An **index** is a data structure over one or more columns that can accelerate queries
- Example:
  - A table that has a few distinct values repeated millions of times
  - And you frequently want all rows with exactly one given value
  - It might be faster to store a mapping **value → rows** than to search each row independently

Drawbacks of indices

- They take time and **space** to construct
- **Composite indices** (multiple columns) are particularly costly
- **Updates** become slower
- No guarantee that they will help in all queries

When to index?

- When data is **read more often** than written
- When queries are **predictable**
- When queries rely on a **small number of attributes**
- **Remember**: you can always add or delete indices later

Summary

SQL is magic!

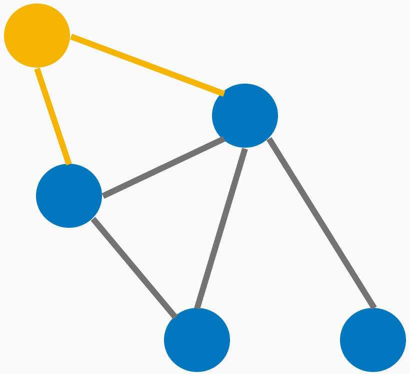
- SQL provides a standard interface to relational databases
- Modern frameworks often provide SQL-style interfaces
    - Pandas .groupBy(), .merge(), etc…
- Use indices to organize your data ahead of time

Databases (can) ensure the integrity of transactions

More file system drawbacks: consistency!

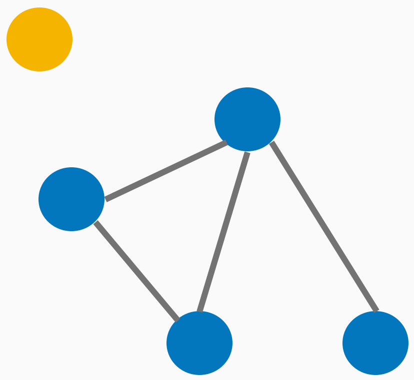
- What if you want to impose **constraints** on your data?

- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?



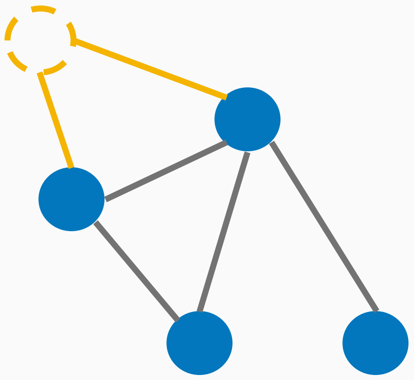
More file system drawbacks: consistency!

- What if you want to impose **constraints** on your data?
- Example:
    - Guaranteeing that a graph is **connected**
    - Vertices stored in **nodes.dat**
    - Edges stored in **edges.dat**
    - What if you want to add a **vertex** and two **edges**?
- Add vertex first: graph is **disconnected**



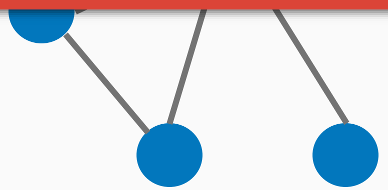
More file system drawbacks: consistency!

- What if you want to impose **constraints** on your data?
- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?
- Add edges first: **edges** are **invalid**



More file system drawbacks: consistency!

- What if you want to impose
- Example:
  - Guaranteeing that a graph is
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?
- Add edges first: **edges** are **invalid**



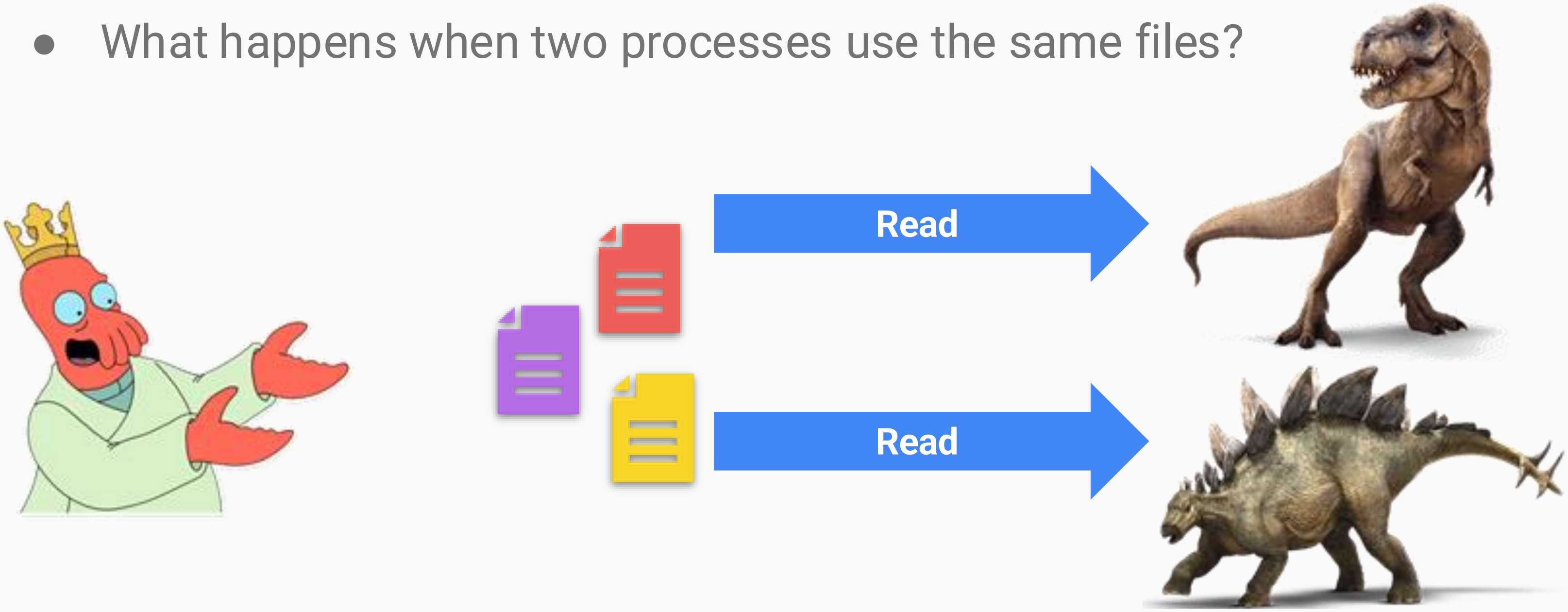
Either operation by itself can render the data **inconsistent.**

Operations need to be performed simultaneously, but file systems do not generally provide this functionality.

We need another layer of abstraction.

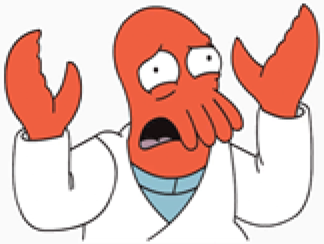
More drawbacks: concurrency!

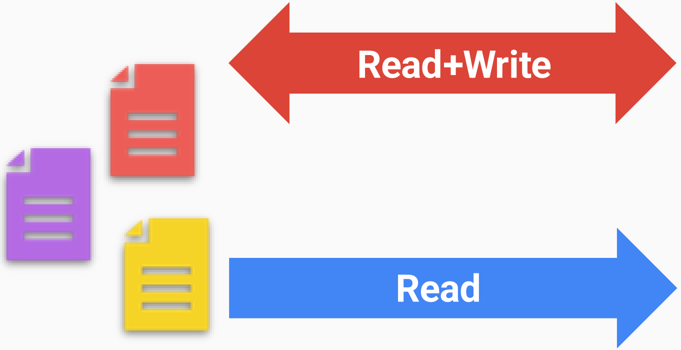
- What happens when two processes use the same files?



More drawbacks: concurrency!

- What happens when two processes use the same files?





More drawbacks: concurrency!

- What happens when two processes use the same files?

ACID principles to the rescue

Atomicity	Operations are all-or-nothing (No partial updates; operations bundled in **transactions**)
Consistency	Transactions move from one **valid** state to another (only!)
Isolation	Concurrent operations do not depend on **order** of execution
Durability	Completed transactions are **permanent** (usually implemented by flushing to disk before completion)

Atomicity in practice

- When modifying tables, wrap query statements in

  BEGIN TRANSACTION; [queries]; **COMMIT**;

  or

  BEGIN TRANSACTION; [queries]; **ROLLBACK**;

- Different DBMS use slightly different syntax (**START**, **BEGIN**)

- If a query **fails** mid-transaction, uncommitted changes will be **abandoned**
  - ⇒ DB is always left in a **valid state**

Aside: transactions example

- SQL transactions are kind of like “try … except” blocks in Python/Java/etc.
- It’s helpful to think of SQL interactions as a conversation or **session**
    - Each command executes and either succeeds or fails
- If you’re performing multiple queries, and one of them fails for some reason, you can **roll back** to the state of the database at the beginning
- If everything completes successfully, you can **commit** the transaction

The classic example

id	balance (≥ 0)
1	$10
2	$20

- Imagine transferring $20 from account id=2 to account id=1
    - “balance” column cannot go negative

- This is done with two update queries in a transaction:
    - BEGIN TRANSACTION
        - UPDATE account SET balance=balance-20 WHERE id=2
        - UPDATE account SET balance=balance+20 WHERE id=1
    - COMMIT

- These need to either both happen or neither happens

Adapted from https://www.sqlitetutorial.net/sqlite-transaction/

The classic example

id	balance (≥ 0)
1	$10
2	$20

- If we instead wanted to transfer $20 from id=1 to id=2, this would violate the schema constraint “balance >= 0”.
- This is done with two update queries:
  - BEGIN TRANSACTION
    - UPDATE account SET balance=balance-20 WHERE id=1 ← fails!
    - UPDATE account SET balance=balance+20 WHERE id=2
  - ROLLBACK
- So the second query is not committed - all changes are reverted!

Consistency in practice

- **Consistency** is maintained by **schema**
  - Schema can add basic value checks as well

```
CREATE TABLE Pachyderms (id INTEGER PRIMARY KEY,
                         species TEXT NOT NULL,
                         height NUMBER        CHECK (height >= 1.0))
```

- If data does not fit the **schema**, the operation **fails immediately**
  - ⇒ DB cannot enter an **invalid state**

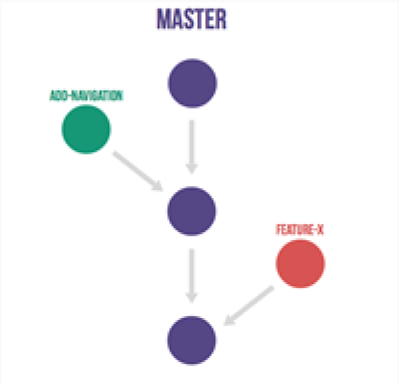
Isolation in practice

- Usually achieved by **locking** the database during modification
  - Only one transaction can hold the lock at a time
- This becomes a real problem for distributed databases!
  - Locking the entire DB would stall everyone
- As we'll see next week, **Map-Reduce** side-steps this problem

You've seen some of this already (In IDS – if not, you saw it in the lab last week)!

- **git** can be seen as a kind of distributed (non-relational) database

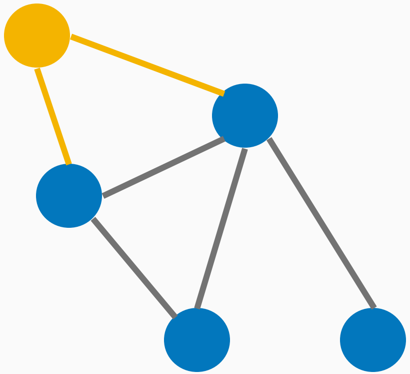
- **Atomicity**:
  - Changes are staged independently (**git add**) into a transaction
  - Transactions are finalized atomically (**git commit**)
- **Consistency**:
  - Conflicting changes are detected and forbidden (**merge conflict**)
- **Isolation**:
  - Different **branches** or **clones** can be modified independently
- **Durability**:
  - Objects are saved to the local repository



Revisiting the connected graph problem with properties of reliable transaction processing (ACID) in mind:

- What if you want to impose **constraints** on your data?

- Example:
    - Guaranteeing that a graph is **connected**
    - Vertices stored in **nodes.dat**
    - Edges stored in **edges.dat**
    - What if you want to add a **vertex** and two **edges**?



Which ACID property would be most useful for fixing our connected graph problem and why?

Summary

Relational database management systems

- The relational model is powerful!
    - Tables and joins are a simple, flexible model for many kinds of data!
    - SQL is a little strange, but powerful
- RDBMS provide
    - Data abstraction
    - A common language to interfacing with data (SQL)
    - Concurrent access

Next week

- Distributed computation with Map-Reduce
- Reading:
  - First: Dean & Ghemawat
  - Second: DeWitt & Stonebraker

Q&R