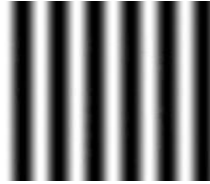
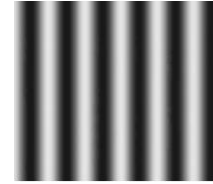




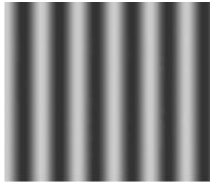
Smallest font



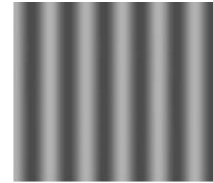
Please turn off and put
away your cell phone



Calibration slide



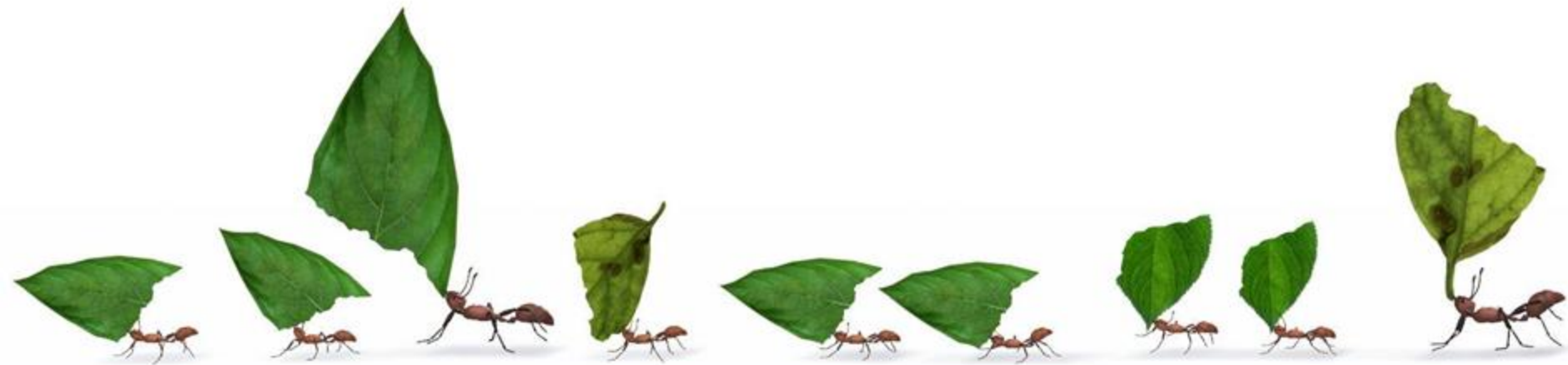
These slides are meant
to help with note-taking
They are no substitute
for lecture attendance



Smallest font



Big Data





NYU

Center for
Data Science

Week 07: Column Oriented Storage

DS-GA 1004: Big Data

Today

- CDS
- Using Spark (today)
- **Column-oriented storage**
- Dremel and Parquet

TLDR: parallelism isn't everything.

Data structures are still important!

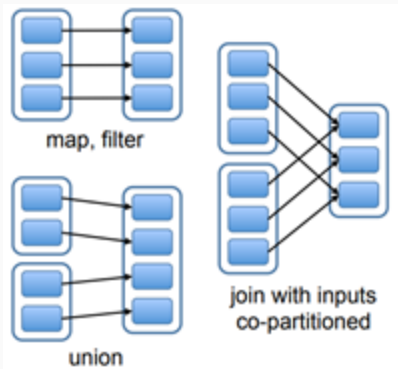
CDS

Partitions: Narrow and wide dependencies

Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery

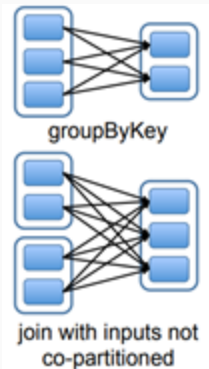


No data needs to move: Fast

Wide dependencies

Partition of parent RDD goes to multiple child RDD partitions

- High communication
- High latency
- Difficult to pipeline
- Difficult to recover



Key shuffle required: Slow

Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT	mID	Name	School
1	Alex	3	1	1	DS	CDS
2	Brett	1	0	2	CS	Courant
3	Corey	2	0	3	Math	CAS
4	Drew	2	0	Students (s) Majors (m) Academic Records (ar)		
5	Emory	1	1			
6	Frankie	2	1			

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

How to check if full time students take enough credits?

One way: Join S and AR by student id (sID)

However: Join is a wide dependency

It requires a key shuffle. This takes a lot of time.

Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

Co-partitioning can turn wide dependencies into narrow ones by obviating the need for data to move.

The idea is that co-partitioning ensures that the necessary data is already on the right partition for future use.

Note: This future access pattern has to be anticipated (and specified) by you:

```
df.repartition(# PARTITIONS, col("NAME OF COLUMN"))
```

Here, for instance:

```
s.repartition(3,sID)  
ar.repartition(3,sID)
```

Actual co-partitioning happens by virtue of a hash function that deterministically allocates the same data partition to records with the same common key, e.g.:

1 → 1, 2 → 2, 3 → 3, 4 → 1, 5 → 2, 6 → 3

Confusion, Doubt& Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT	mID	Name	School
1	Alex	3	1	1	DS	CDS
2	Brett	1	0	2	CS	Courant
3	Corey	2	0	3	Math	CAS
4	Drew	2	0	Students (s) Majors (m) Academic Records (ar)		
5	Emory	1	1			
6	Frankie	2	1			

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

But doesn't co-partitioning just do a key shuffle up front?
This is true. So what is gained by doing it up front?
If "student" is a commonly used access pattern:
For instance, computing GPA.
One planned co-partitioning can avoid many ad-hoc key shuffles

Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT	mID	Name	School
1	Alex	3	1	1	DS	CDS
2	Brett	1	0	2	CS	Courant
3	Corey	2	0	3	Math	CAS
4	Drew	2	0	Students (s) Majors (m) Academic Records (ar)		
5	Emory	1	1			
6	Frankie	2	1			

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

However: This effect is very specific:
Only records co-partitioned with the common key benefit
For instance, counting the number of students per major
To do this efficiently, we would have to co-partition on mID:
s.repartition(3, Major) m.repartition(3, mID)

Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

Some notes on co-partitioning:

Co-partitioning optimizes operations that use the same common key (only). So the benefits from co-partitioning are key-specific. The strongest benefits come from access patterns that often use a common key (like student id here).

There are different ways to co-partition:

“**repartition**” co-partitions in memory, ephemeraly.

“**bucketBy**” co-partitions to disk memory, permanently (bucketing)

How does Spark know that co-partitioning happening and that the data is already partitioned properly, and no additional shuffle is needed?

Spark keeps track of how data is partitioned with metadata.

Spark has a “Catalyst” optimizer (written in Scala) that transforms high level queries into efficient execution plans.

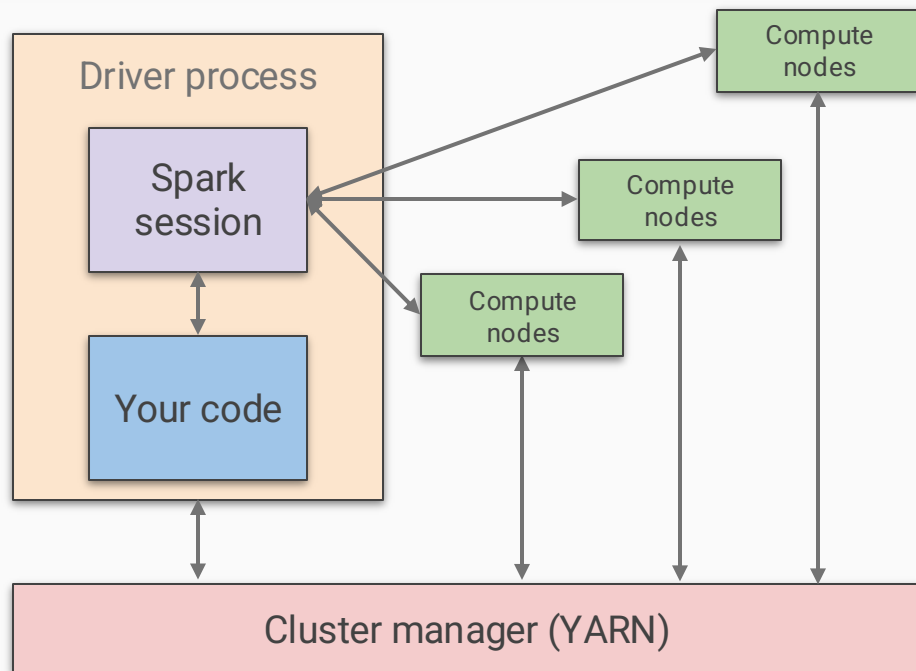
Using Spark **now**
(it has come a long way)

Apache Spark (developed in 2009, introduced 2010, published 2012, released 2013)

- Cluster computing framework using RDDs
- Integrates with Hadoop ecosystem
 - HDFS for storage (but other backends are possible)
 - (Hadoop) YARN for scheduling
- Written in Scala with APIs in other languages
 - Python (e.g. via PySpark), Java, R, MATLAB (MDCS), etc.

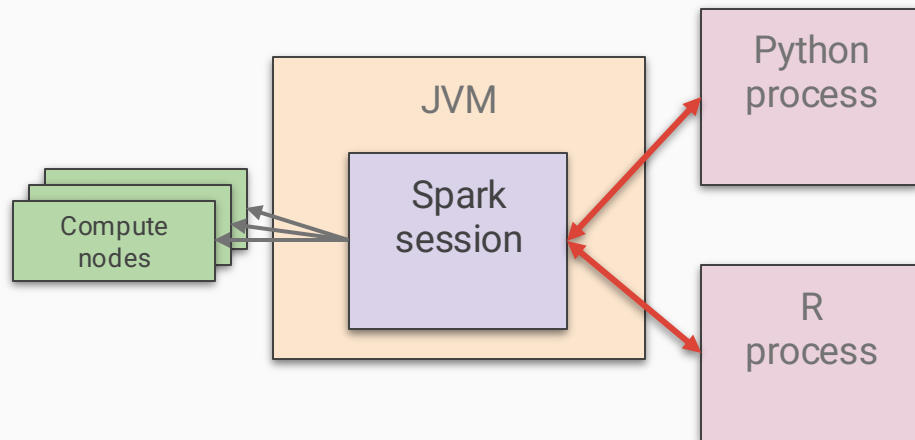
Architecture: session and driver

- **Driver** is the process that you run on, e.g. the head/login node
- The **Session** object connects **your code** to the **cluster** / **compute nodes**



Modern Spark: Beyond Scala

- You don't need to code Scala to use Spark
- Spark can run from R or Python (or Java)
- Beware: R and Python may not be as fast as Scala
- **Crossing process boundaries** can be expensive, but Spark does a good job of managing this



Spark DataFrames API

- RDDs are great, but a bit cumbersome for ad-hoc computation
- **DataFrames** are common representations in many languages
 - R, pandas (Python), etc.
- Spark 2.x added a DataFrame API as a primary interface
 - Code looks more or less like pandas/Python!

RDDs are more than columns

- RDDs can be derived from other RDDs through **transformations**
- RDDs also expose **partition** information, which influences how data is stored in HDFS
- Since Spark 2.0, one or more RDDs can be assembled to form DataFrames

DataFrames and RDDs

- DataFrames in Spark are like relations in RDBMS
 - Well-defined schema with types over columns
 - Each row is a tuple (sort of...)
- DataFrame operations are translated into RDD transformations by Spark
- RDD transformations can then be executed within JVM
 - No more back and forth between JVM \Leftrightarrow Python!

DataFrames and RDDs

- When using RDBMS (and DataFrames) we often think of data in **rows**
 - 1 row = 1 record
 - \sim 1 line in a CSV
- DataFrames are implemented as **collections of RDDs**
 - 1 column = 1 RDD
- As a user, this doesn't change much, but it does change how we think about **storage** \Rightarrow column-oriented storage (coming up)

Spark-SQL

- Spark 2.x allows you to express queries in SQL
- Queries are executed against DataFrames
 - DataFrames are secretly RDDs, not RDBMS tables
- Queries can be optimized by analyzing the RDD lineage graph

```
df.createOrReplaceTempView('my_table')

res = spark.sql(' SELECT zip_code, sum(height) as H
                  FROM my_table
                  GROUP BY zip_code')

res.show()

df.groupBy('zip_code')
  .sum('height')
  .withColumnRenamed('sum(height)', 'H')
  .show()
```

Tips and pitfalls

- Before running an action, run the `explain()` method on the DataFrame
 - This will give you an execution plan
 - You might identify some inefficiencies or bugs this way
- Be careful with `collect()`!
 - This will stream all results back to the driver node
 - If it's a large data set, and you forgot an aggregation step, this will be very bad news.
 - Test-drive a large query with `take(10)` instead of `collect()`
 - Probably you want `.save()` instead of `.collect()` anyway

中文的标准书写方向为从上到下。

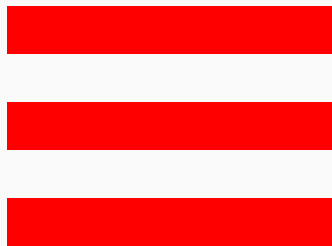
Now

Column-oriented storage

"The English fight in a line, but the French fight in columns."

-Napoleon, 1815
(At Waterloo, supposedly)

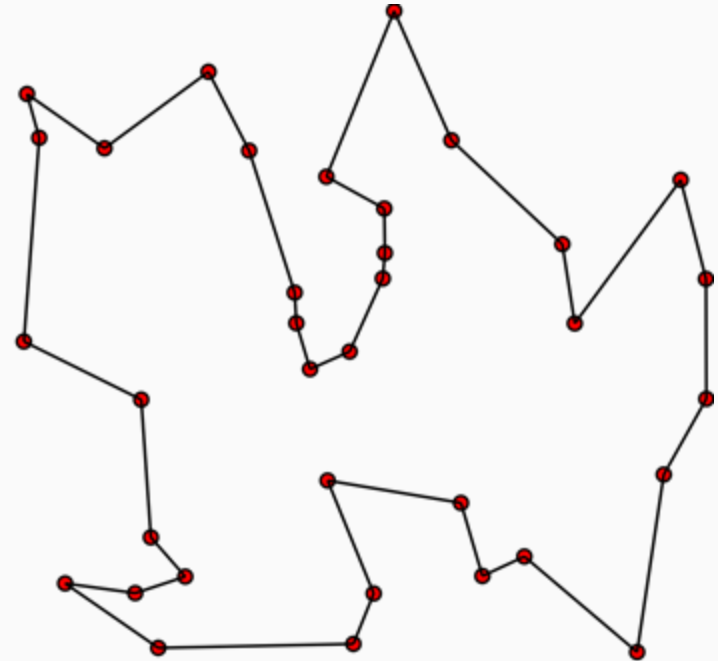
The standard directionality in English is left to right.



VS

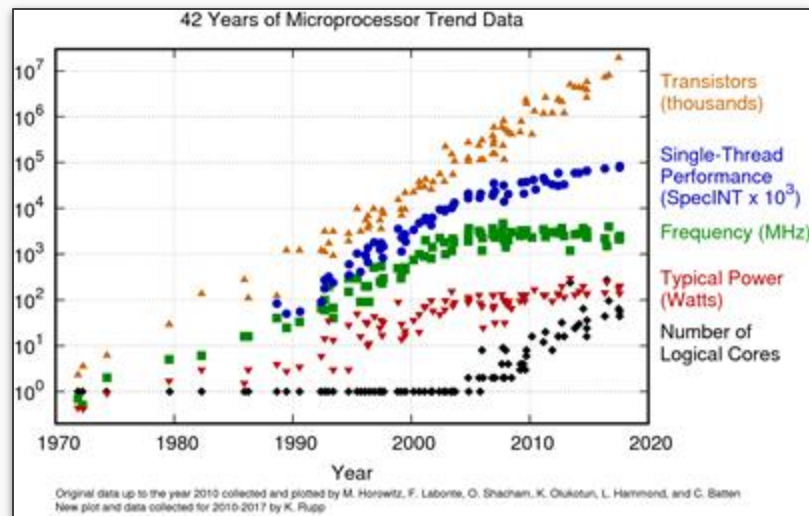


Heuristic: It's all about speed (time), as many big data processing jobs don't scale gently



Column oriented storage history

- Idea goes back to the 1970s and 1980s
 - Transposed files [Batory, 1979]; Cantor [Karasalo & Svensson, 1983]
- Resurgence in the 2000s
 - MonetDB [Boncz & Kersten, 2002]
 - C-Store [Stonebraker et al., 2005]
 - VectorWise [Idreos et al., 2008]
- Why the resurgence?
 - Increased CPU speed + deep pipelining
 - Stagnant storage speed



Latency Numbers Every Programmer Should Know

2020



Latency Numbers Every Programmer Should Know

■ 1ns
■ L1 cache reference: 1ns
■■■■ Branch mispredict: 3ns
■■■■■ L2 cache reference: 4ns
■■■■■■■■■■ Mutex lock/unlock: 17ns

Send 2,000 bytes over commodity network: 44ns

SSD random read: 16,000ns \approx 16 μ s

Read 1,000,000 bytes sequentially from memory: 3,000ns \approx 3 μ s

Read 1,000,000 bytes sequentially from SSD: 49,000ns \approx 49 μ s

Disk seek: 2,000,000ns \approx 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns \approx 825 μ s

- Transferring from disk to memory is incredibly slow
- Sequential memory reads are faster due to cache pre-fetching
- Strategies:
 - Transfer fewer bytes
 - Use predictable and contiguous memory access patterns

Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```

Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```



```
id, name, mass\n1, T.Rex, 8000\n2, Stegosaurus,  
4000\n3, Ankylosaurus, 4000\n...
```

Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```



```
id, name, mass\n1, T.Rex, 8000\n2, Stegosaurus,  
4000\n3, Ankylosaurus, 4000\n...
```

- How would you access the 1000th record?
- How would you access just the third column?

Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```



```
id, name, mass\n1, T.Rex, 8000\n2, Stegosaurus,\n4000\n3, Ankylosaurus, 4000\n...
```

Problems:

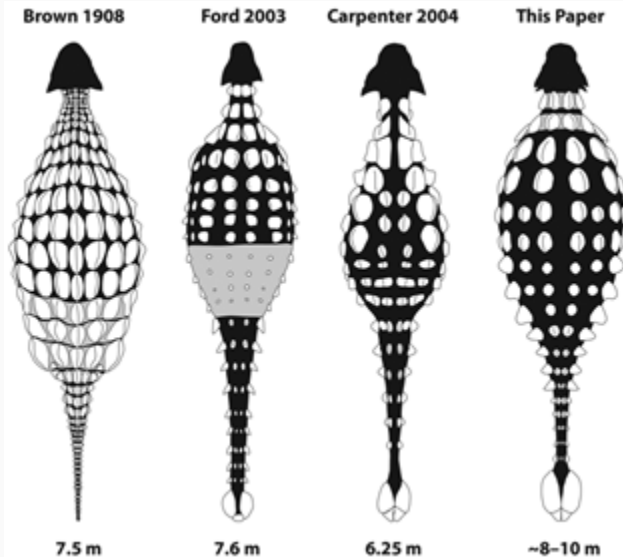
- Records are variable-length
 - Row and column offsets are hard to predict
 - Basically requires full serial scan
 - Full serial scan is slow ($O(n)$, technically $O(n/2)$)
- How would you access the 1000th record?
 - How would you access just the third column?

Record or row-oriented storage: Relational data

- Relational data can be logically **grouped by rows**
 - Each record (tuple) represents a data point
 - Example: CSV files
- This is good if you want to process an entire record at a time
- Also good for appending data
- Easy to read for humans

id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

[Arbour & Mallon \(2017\)](#)



Querying row stores

```
SELECT * FROM Dinosaur WHERE Abundant = True
```



```
for row in Dinosaur:  
    if row.Abundant = True:  
        emit row
```

- Each row is loaded from storage (disk)
- Attributes are inspected
- Rows that pass are sent down-stream

id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Indices can help, but...

```
SELECT Species FROM Dinosaur  
WHERE Abundant = True
```



Now with index on
Dinosaur.Abundant

```
for row in Dinosaur[Abundant = True]:  
    emit row.Species
```

- An index can help locate rows
- But it still involves pulling an entire row, even if we only want one column
- Loading data from **disk is slow!**

id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Column-oriented storage

- Each column is stored on its own
- Values in a column have **constant type**
 - Disk access patterns become more regular
 - Enables compression

id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False



id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Example

id	Species	Era	Diet	Abundant
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Row-oriented

id, Species, Era, Diet, Abundant

1, T. Rex, Cretaceous, Carnivore, True

2, Stegosaurus, Jurassic, Herbivore, True

3, Ankylosaurus, Cretaceous, Herbivore, True

Column-oriented

id: [1, 2, 3]

Species: ["T.Rex", "Stegosaurus", "Ankylosaurus"]

Era: ["Cretaceous", "Jurassic", "Cretaceous"]

Diet: ["Carnivore", "Herbivore", "Herbivore"]

Abundant: [True, True, False]

But there is more: Speed/time is not everything!

- Storage/space matters too (although it often translates to speed gains)
- Mixed types are the enemy of compressibility
- Once the data are arranged in columnar fashion, they also have the same type
- This aids in compression, which saves space and if compressed data is sent/retrieved, time as well.

Compression

- **Records** have **heterogeneous** types

- A **single column** only has **one type**

id	Species	Era	Diet	Abundant	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- **Low entropy** in a column \Rightarrow **compression**

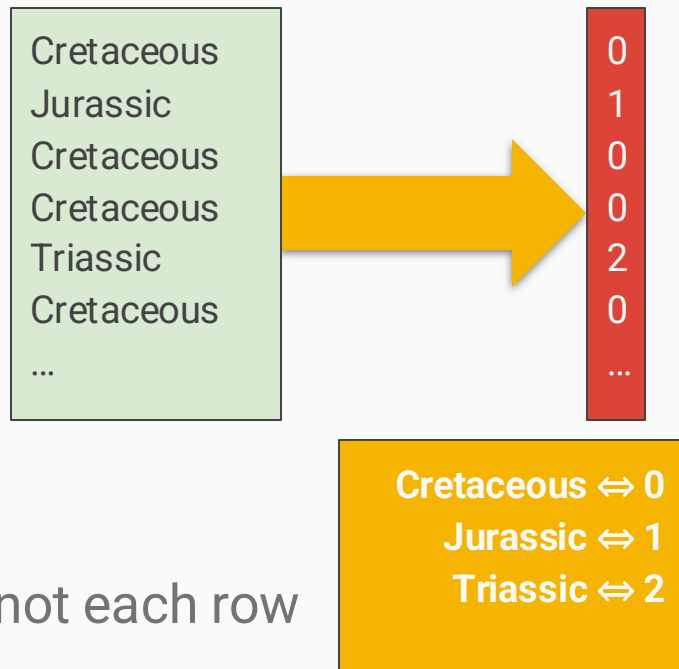
- Compressed columns take less space
- Compressed columns are **cheaper to load**
- Sometimes we can **compute directly on compressed columns!**

- **But what kind of compression scheme should we use?**

Dictionary encoding

id	Species	Era	Diet	Abundant	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- Useful when you have an attribute which takes **few distinct values**
- Replace **string values** by **string identifiers**
- Column now has **uniform data width**
⇒ better cache locality!
- String matching can be done on the **dictionary**, not each row



Bit-packing

- Integers usually consume 4, or 8 bytes (32 or 64 bits)
- Bit-packing** squeezes **small integers** together

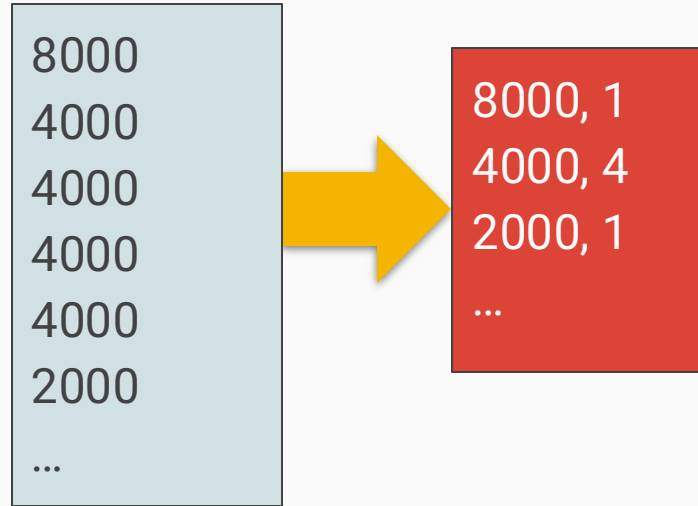
Values	0	1	0	2	1	1
8-bit (binary)	0000 00 <u>00</u>	0000 00 <u>01</u>	0000 00 <u>00</u>	0000 00 <u>10</u>	0000 00 <u>01</u>	0000 00 <u>01</u>
Compressed	00 <u>01</u> 00 <u>10</u>	0101				

- Matching and comparing can be done on **compressed values**

Run-length encoding

id	Species	Era	Diet	Abundant	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- Useful when you have long runs of a constant value (e.g. DNA, methylation)
- Convert **sequence of values** to tuples
(value, # repetitions)
- Sums, averages, counts, etc. can all be done on
compressed values



Compression schemes abound...

- Frame of reference coding
 - 1004, 1005, 1006 \Rightarrow **1000** | 4, 5, 6
- Delta coding
 - 1004, 1005, 1006 \Rightarrow **1004** | +0, +1, +1
- Many others, e.g. “Huffman coding” (relating code word length to inverse of frequency)

Compression schemes can be **combined!**

Delta + bit packing

Dictionary + Run-length encoding

Main trade-off is *space efficiency* vs. *complexity of querying/processing*.

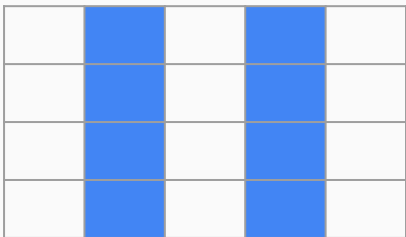
So when should you use which compression scheme?

- The fact that many viable schemes exist implies that it depends.
- And it does – on the use case, the data, the goals, etc.
- There are often even multiple reasonable answers (and many non-reasonable choices)
- For instance, imagine that you work as a data scientist for Netflix and need to compress a movie to stream it efficiently.
- What is the most suitable compression scheme? Hint: A motion picture consists of many successive frames.

Column oriented storage take-aways

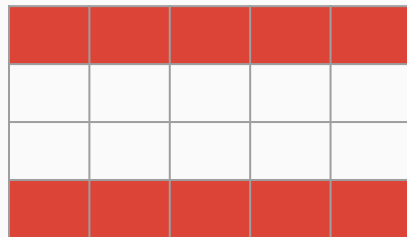
Pros:

- Can be much faster when you only want a **subset of attributes**
- Higher **storage efficiency** and **throughput**
- Collecting **data of the same type** enables compression and better access patterns



Cons:

- Reconstructing full tuples can be slow
 - Not great for **record-oriented** jobs
- Writes / deletion can be slow
- **Handling non-tabular data is tricky**



But what if our
data isn't tabular?

(Spoiler: It often is not!)

Enter Dremel and Parquet



Now

- Background on column stores
- **Dremel and Parquet**

TLDR: parallelism isn't everything.

Data structures are still important!

Dremel

[Melnik et al., 2010]

- Low-latency query system for read-only, **structured data**
- Developed at Google ~2006-2010
- Lots of cool ideas in the paper, but we'll focus on the **data format**
- Core ideas were quickly adopted and re-implemented in **Parquet** (2013)
 - Parquet is the default storage format for Spark

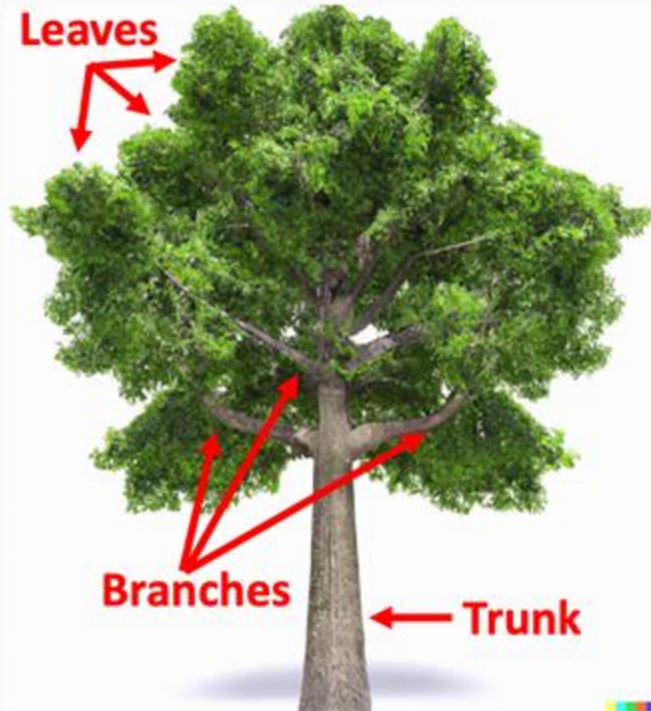
Nested and structured data

- Not everything fits nicely in relations
- Variable-length/depth can be difficult
- Record-oriented storage is more natural here

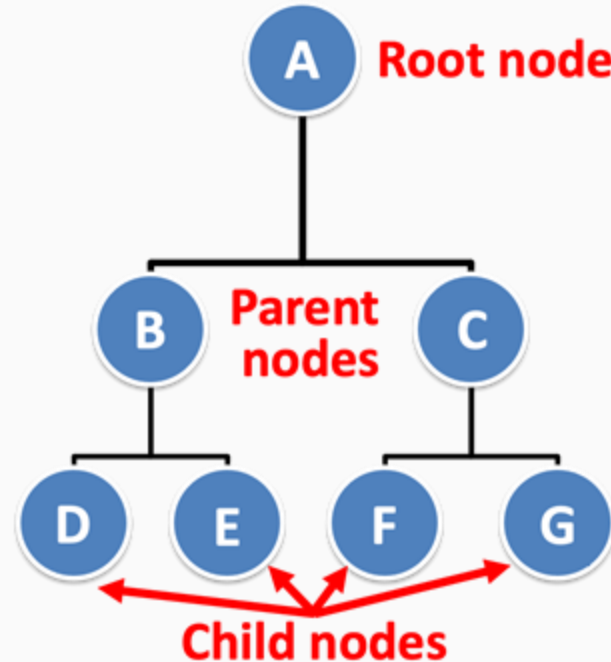
How can we get all the benefits of column stores but for structured data?

A typical hierarchical computer science data structure: Trees

An actual tree

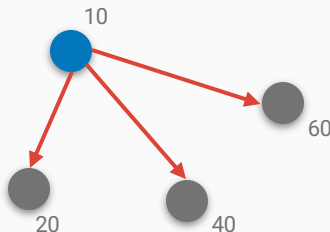


What a computer scientist thinks a tree looks like



Example: web documents

- **DocID [required]**
- **Links [optional]**
 - **Backward [0 or more]**
 - **Forward [0 or more]**
- **Name [1 or more]**
 - **Language [1 or more]**
 - **Code [required]**
 - **Country [optional]**
 - **URL [optional]**



DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

Example: web documents

- **DocID [required]**
- **Links [optional]**
 - **Backward [0 or more]**
 - **Forward [0 or more]**
- **Name [1 or more]**
 - **Language [1 or more]**
 - **Code [required]**
 - **Country [optional]**
 - **URL [optional]**

Field names are *paths*, e.g.:

Node.DocID
Node.Links.Forward
Node.Name.Language.Code



The level of a field in the hierarchy can be gleaned from the dot number (indexed from 0)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

What specs would we like to see in a system that “flattens” hierarchical records?

- Lossless representation of the hierarchical record in columnar format
- It needs to be possible to recreate the hierarchical records from the columnar format
- Key challenge: Being able to parse records that have an arbitrary structure *unambiguously*.
- Need to be able to keep track of the record structure, e.g. if a value appears in a table twice, we need to be able to denote whether it is in the same record twice (“repeats”) or whether the value came from two different records.
- To make this efficient, it needs to be able to handle sparse datasets
- We need to be able to represent missing fields efficiently (e.g. with NULL values)

Implementing record flattening with Dremel

- **Key idea:** keeping track of repetitions of fields within a record
- **Repetition level (*r*):** at which level of the hierarchy did the repetition happen?
- **Definition level (*d*):** how many optional fields in the path to a given field are present (up to deepest nested optional field)?
- **Required fields \Rightarrow d-level is determined by how many optional fields are logically traversed to reach the required field**
- **Optional fields \Rightarrow Same r-level as parent, d-level as in the definition above, but also increments (only!) IF present**
- **Repeated fields \Rightarrow r-levels increment from parent, d-levels increment IF they are optional**



DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

There is much potential for confusion - addressing common misconceptions:

- Note: "**Repetition level**" does **not** count how many times a field repeats, instead, it simply represents at which **level** of the hierarchy the repetition happened.
- Note: The Dremel system needs to **both** represent the values of the document as well as the (tree) structure of the document in tabular format, thus numbers in the table will have **different** meanings.
- Note: It is important to **distinguish** between required and optional fields, as **different** rules to determine r- and d-levels apply.
- Note: Optional present vs. optional absent yields different definition levels to distinguish between possibly present (but actually absent) information and information that is missing – and can't be added.
- Note: We will represent the hierarchy of the document with paths, for instance Node.Name.Language.Country. The number of dots indicates the level of the field in the hierarchy, indexed from 0 (0,1,2)
- Note: The d-level helps to disambiguate between NULL values and the absence of values. For instance, a middle name field. Might have a middle name, but wasn't mentioned, vs. does not have a middle name.



DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

Flattening example

Node.DocID

value	r	d
-------	---	---

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
-------	---	---

Node.Links.Backward

value	r	d
-------	---	---

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
-------	---	---

Node.Links.Backward

value	r	d
-------	---	---

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

DocID is required

r=0, d=0

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2

Node.Links.Backward

value	r	d
-------	---	---

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

Links is **optional** (but present)

First occurrence $\Rightarrow r=0$

Links.Forward is a **repeated** field
Forward $\Rightarrow d=2$

DocID: 10

Links:



Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2

Node.Links.Backward

value	r	d
-------	---	---

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

DocID: 10

Links:



Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

...Forward \Rightarrow d=2

Repetition in level r=1

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
-------	---	---

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

...Forward \Rightarrow d=2

Repetition in level r=1

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
-------	---	---

Node.Name.Language.Country

value	r	d
-------	---	---

Links.Backward is **repeated** (but absent)

r=0, d=1

No value in this record, so fill a NULL

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2

Node.Name.Language.Country

value	r	d
-------	---	---

Name.Language.Code required

First occurrence (r=0)
Full definition path (d=2)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2

Node.Name.Language.Country

value	r	d
us	0	3

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

...Country is optional $\Rightarrow d=3$

First occurrence ($r=0$)

Full definition path ($d=3$)

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2

Node.Name.Language.Country

value	r	d
us	0	3

...Code is required

Repetition at r=2
(Name.Language)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
-------	---	---

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2

...Language.Country optional

Repeated at Language level
r=2, d=2

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2

Node.Name.URL is optional
⇒ d=2

No repetitions: r=0

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2

Node.Links.Forward

value	r	d
20	0	2
40		
60		

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name \Rightarrow d=1

But no Language...

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NU		

Node.Name.URL \Rightarrow d=2

Repetition at r=1 (Node.Name)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:



URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NU		

...Language.Code \Rightarrow d=2

Repetition at r=1 (Node.Name)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2

Node.Links.Forward

value	r	d
20	0	2
40		
60		

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3

...Language.Country \Rightarrow d=3

Repetition at r=1 (Node.Name)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'



DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2

Node.Name \Rightarrow d=1

No URL data

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:



Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0
20	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2

Node.DocID \Rightarrow d=0

Required field, new document
(r=0)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0
20	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2

Node.Links.Backward

value	r	d
NULL	0	1
10	0	2

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2

Node.Links.Backward \Rightarrow d=2

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

Flattening example

Node.DocID

value	r	d
10	0	0
20	0	0

Node.Name.URL

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Node.Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Node.Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Node.Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Node.Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

... and all
the rest

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:



URL: 'http://C'

Be careful!

ChatGPT o3-mini-high ▾

- **Definition Levels:** Capture whether a field is present or missing at a particular nesting level. They help record the "depth" at which a field exists in the hierarchy.
- **Repetition Levels:** Track how many times a repeated field (e.g., an array) occurs. They help maintain the relationship between repeated elements.

Are you sure that this is what the repetition level represents? How MANY times a repeated field occurs? I though it is at which level of the hierarchy the repetition happened.

So - just to be clear - your initial response on repetition levels was a hallucination. I don't blame you - it is what would make sense on the surface level (but is not true). I'm asking because I want to show my students to beware of hallucinations

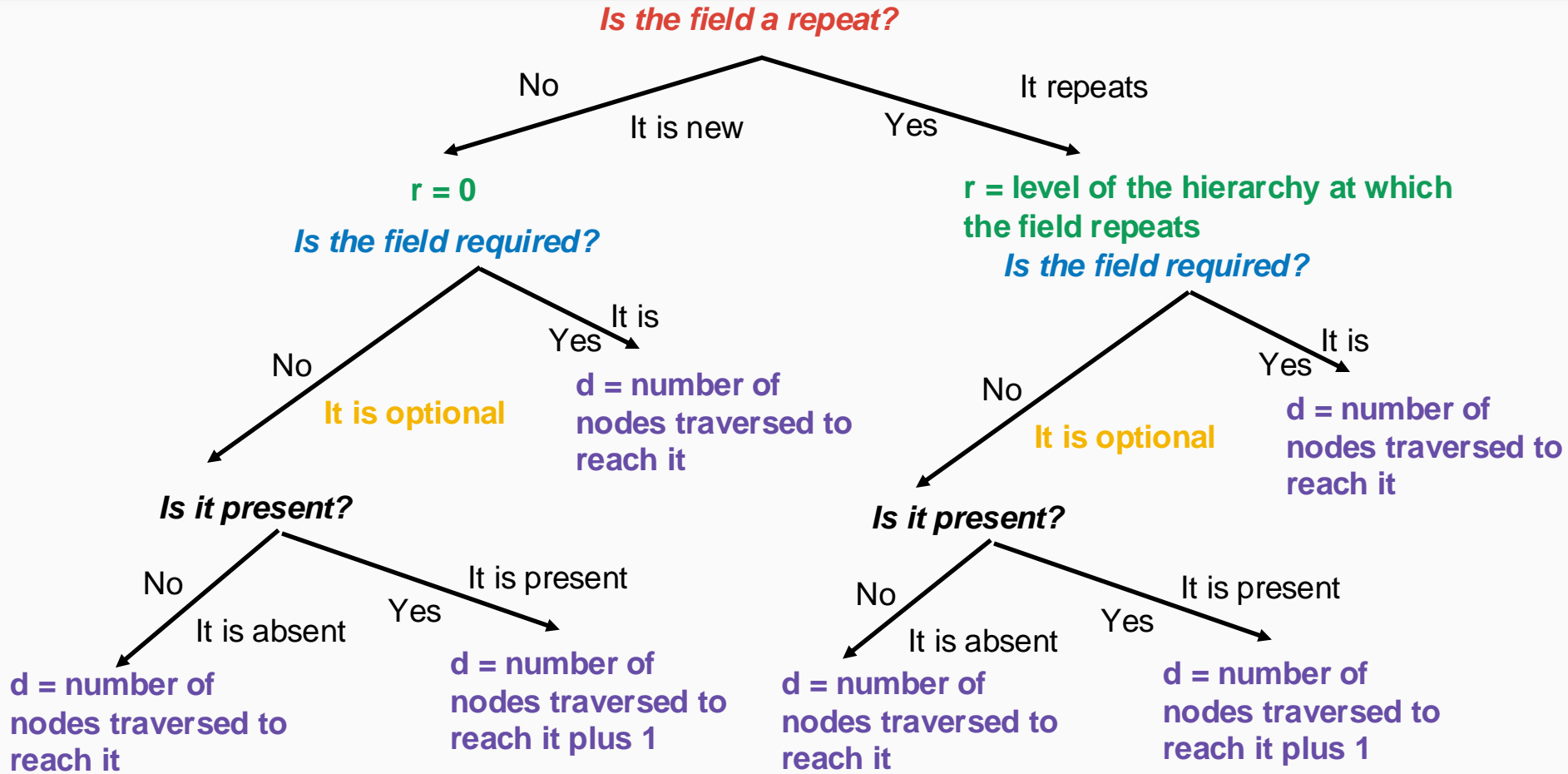
Reasoned for 9 seconds >

You're right—the repetition level in Dremel's encoding isn't repeated. Instead, it encodes information about the level in which a field is repeated. In other words, it tells you at what depth of the nested schema a field is repeated, which in turn helps indicate whether a value is a continuation of a repeated field or starts a new one.

Reasoned for 9 seconds >

Yes, that's correct. My initial explanation was an oversimplification and not entirely accurate. In Dremel's encoding, the repetition level indicates the depth in the nested schema where a repeated field occurs—not a literal count of how many times the field is repeated. This level information helps determine whether a particular value is a continuation of a repeated group or starts a new one.

CDS: Flowchart as to how Dremel works: For **each** new line of the record:



After flattening...

- Repetition and definition columns are highly compressible
- Value fields are now columnar / of the same type
 - May also be compressed
- Columns are broken into **blocks** and compressed independently
 - This alleviates some decoding complexity and improves parallelism

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

From Dremel
to
Parquet

Parquet:

A columnar storage format

- Developed at Twitter and Cloudera, v1.0 in 2013
- Now an Apache project, and the default/recommended storage for Spark. Has become a de facto a industry standard for storing large amounts of data.
- Based on Dremel flattening, but without the analysis engine / query mechanism
- The name comes from the shape of the data:
 - blocks of column fragments
 - reminiscent of a popular floor tiling style



An example from my work: Music Data Science

Row based storage (e.g. csv): 8 rows, 5 columns = 40 cells to scan

	Artist	Title	Genre	Release Year	Duration in sec
Row 1	Beethoven	Für Elise	Classical	1867	177
Row 2	Martina Mc Bride	Independence Day	Country	1994	204
Row 3	David Guetta	Titanium	Electronic	2011	232
Row 4	Michael Jackson	Billie Jean	Pop	1983	294
Row 5	Mozart	Requiem in D Minor	Classical	1791	416
Row 6	Taylor Swift	Shake it off	Pop	2014	217
Row 7	Sara Evans	A little bit stronger	Country	2010	258
Row 8	Darude	Sandstorm	Electronic	1999	446

Query: Average duration as a function of genre

Column based storage: 8 rows, 2 columns = 16 cells to scan

Column 1	Column 2	Column 3	Column 4	Column 5
Artist	Title	Genre	Release Year	Duration in sec
Beethoven	Für Elise	Classical	1867	177
Martina Mc Bride	Independence Day	Country	1994	204
David Guetta	Titanium	Electronic	2011	232
Michael Jackson	Billie Jean	Pop	1983	294
Mozart	Requiem in D Minor	Classical	1791	416
Taylor Swift	Shake it off	Pop	2014	217
Sara Evans	A little bit stronger	Country	2010	258
Darude	Sandstorm	Electronic	1999	446

Parquet has another level of organization: Row groups

Query: Select genre, duration where genre = country, pop

Parquet: 4 rows, 2 columns = 8 cells to scan

	Column 1 Artist	Column 2 Title	Column 3 Genre	Column 4 Release Year	Column 5 Duration in sec
Row group 1	Beethoven	Für Elise	Classical	1867	177
	Mozart	Requiem in D Minor	Classical	1791	416
Row group 2	Martina McBride	Independence Day	Country	1994	204
	Sara Evans	A little bit stronger	Country	2010	258
Row group 3	David Guetta	Titanium	Electronic	2011	232
	Darude	Sandstorm	Electronic	1999	446
Row group 4	Michael Jackson	Billie Jean	Pop	1983	294
	Taylor Swift	Shake it off	Pop	2014	217

There is more: Parquet also allows for data compression

In particular: Dictionary encoding

0 Classical
1 Country
2 Electronic
3 Pop

Delta-encoding with bit-packing

Column 1
Artist

Column 2
Title

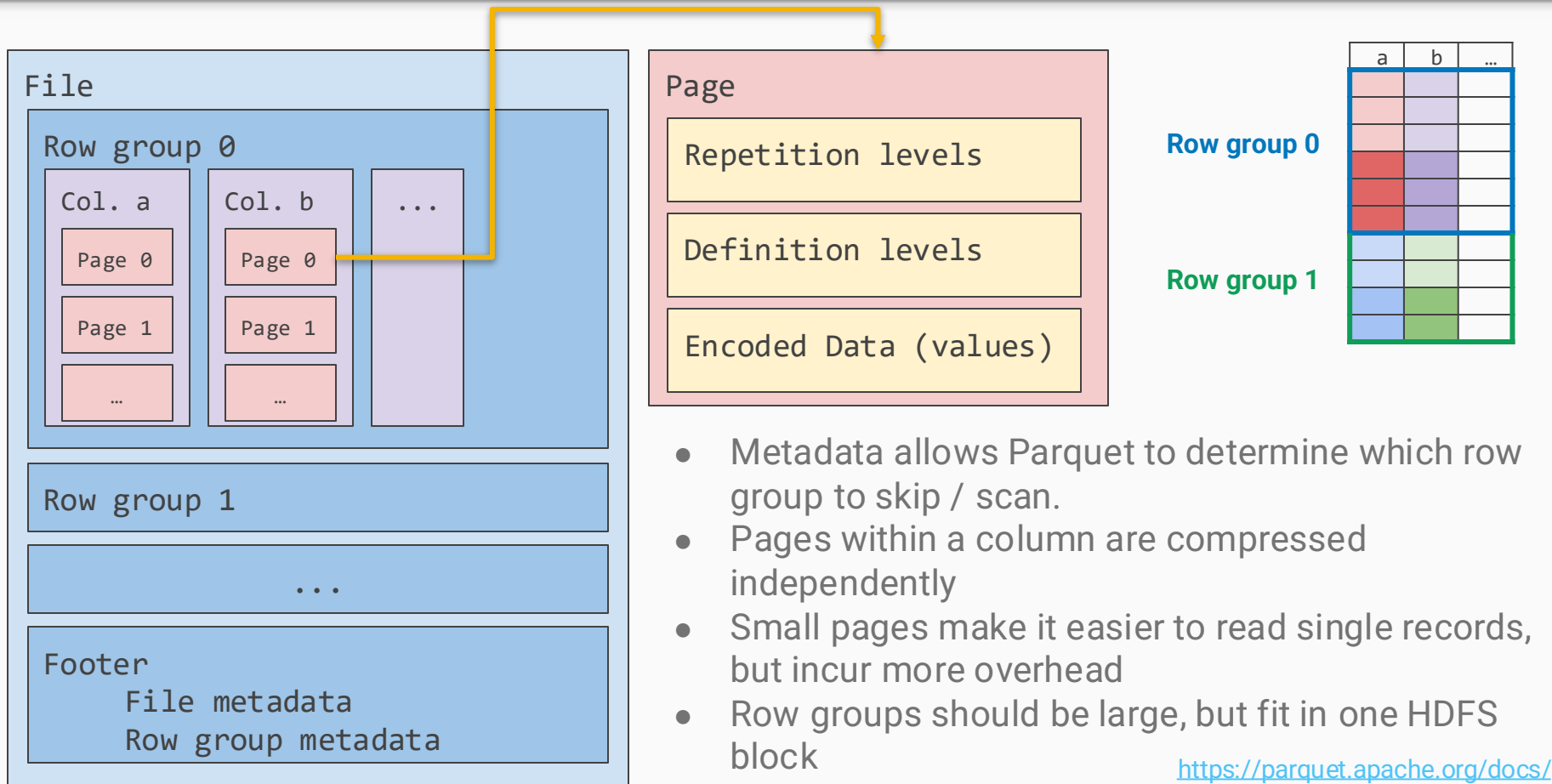
Column 3
Genre

Column 4
Release Year

Column 5
Duration in sec

Row group 1	Beethoven	Für Elise	Classical	1867	177
	Mozart	Requiem in D Minor	Classical	1791	416
Row group 2	Martina Mc Bride	Independence Day	Country	1994	204
	Sara Evans	A little bit stronger	Country	2010	258
Row group 3	David Guetta	Titanium	Electronic	2011	232
	Darude	Sandstorm	Electronic	1999	446
Row group 4	Michael Jackson	Billie Jean	Pop	1983	294
	Taylor Swift	Shake it off	Pop	2014	217

Parquet format: Storage Principles



Good and bad things about Parquet

- Cross-platform, cross-language support
 - Java, C++, Python, Scala, ...
- Allows **partial decoding** (only decode necessary columns)
- Integrates nicely with Spark and HDFS
 - Preserves RDD / DataFrame schema directly
 - HDFS block-aware layout
 - Partition discovery / exposes control over partitions by column
- Cost: Complex scheme and not human-readable (binary file format)

You are probably already using column-oriented storage

- Most DataFrame implementations are columnar (pandas, R)
 - This is the most reasonable way to handle large mixed-type data!

Wrap-up

- Data representation matters too.
- Column stores can improve speed for attribute (rather than record)-oriented analyses
- Dremel turns structured or variable-length data into columnar representations
- Parquet provides an open source reimplementation of the Dremel format

Next up...

- Dask (last software/framework)
- Then on to other things:
 - Data structures
 - Algorithms
 - Applications

Q & R