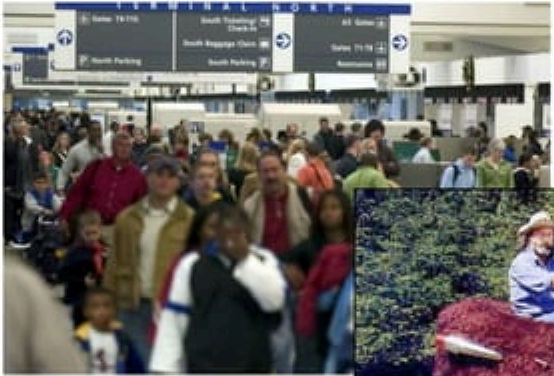
Spark + Parquet in Depth

Robbie Strickland
VP, Engines & Pipelines, Watson Data Platform
@rs_atl

Emily May Curtin
Software Engineer, IBM Spark Technology Center
@emilymaycurtin

SPARK SUMMIT EAST 2017

Atlanta…



SPARK SUMMIT EAST 2017

Atlanta!!!!

Outline

- Why Parquet
- Parquet by example
- How Parquet works
- How Spark squeezes out efficiency
- What's the catch
- Tuning tips for Spark + Parquet


STORAGE
H HD
HISTORY.COM

SPARK SUMMIT EAST 2017

Goals for Data Lake Storage

- Good Usability
  - Easy to backup
  - Minimal learning curve
  - Easy integration with existing tools
- **Resource efficient**
  - Disk space
  - Disk I/O Time
  - Network I/O

- **AFFORDABLE**
  - **CA$$$H MONEY**
  - **DEVELOPER HOURS → $$$**
  - **COMPUTE CYCLES → $$$**
- **<u>FAST QUERIES</u>**

Little Costs Matter at Actual Scale

"Very Large Dataset"

Weather-Scale Data

Disk and Network I/O *Hurt*

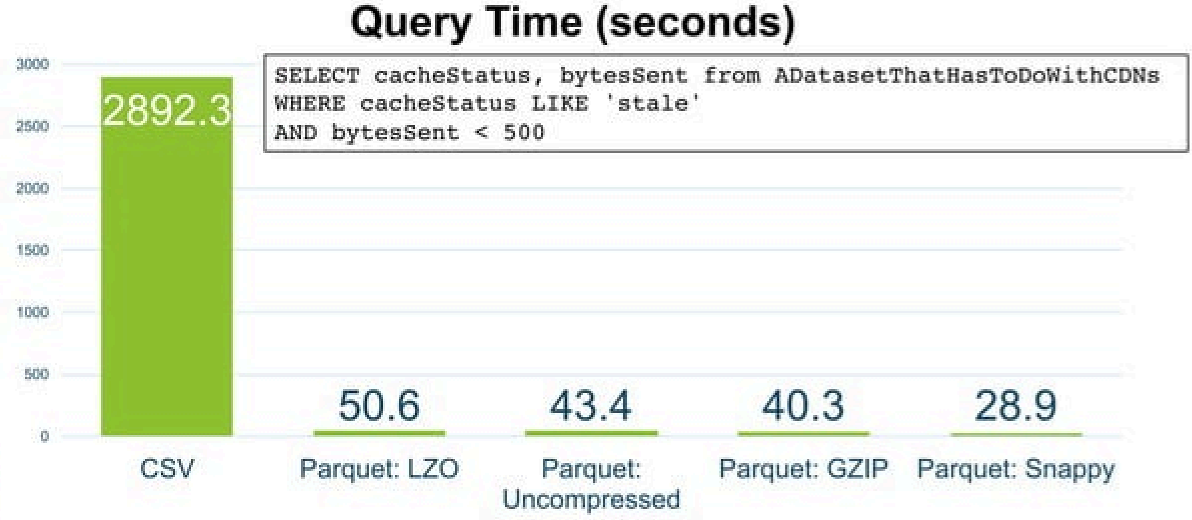
Action	Computer Time	"Human Scale" Time
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	**50–150 µs**	**2–6 days**
Rotational disk I/O	**1–10 ms**	**1–12 months**
Internet: SF to NYC	**40 ms**	**4 years**
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

Source: Systems Performance: *Enterprise and the Cloud* by Brendan Gregg via CodingHorror.com "The Infinite Space Between Words"

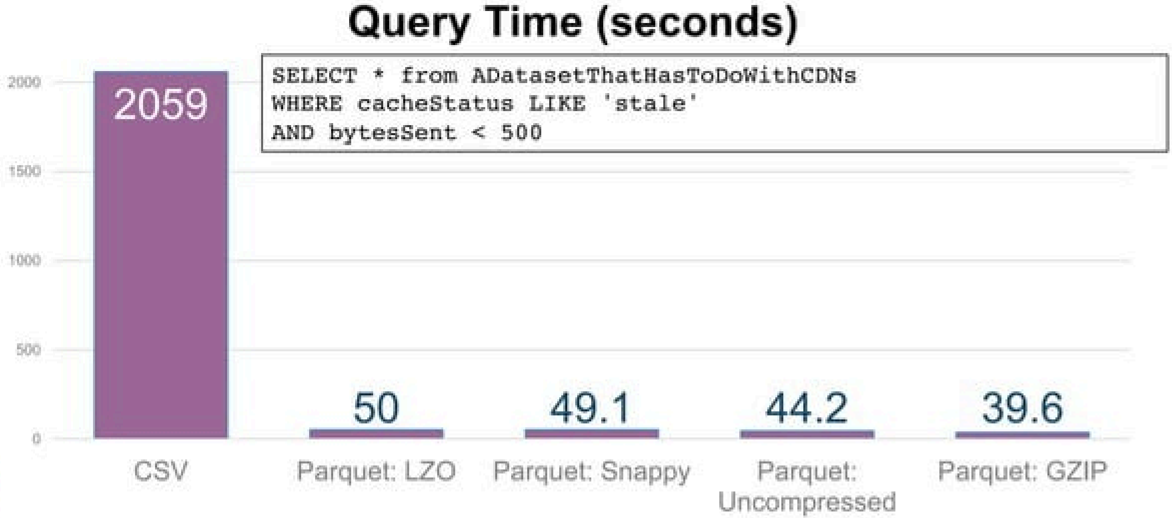
Options For Multi-PB Data Lake Storage

	Files	Compressed Files	Databases
Usability	Great!	Great!	OK to BAD (not as easy as a file!)
Administration	None!	None!	LOTS
Spark Integration	Great!	Great!	Varies
Resource Efficiency	BAD (Big storage, heavy I/O)	OK… (Less storage)	BAD (Requires storage AND CPU)
Scalability	Good-ish	Good-ish	BAD (For multi-petabyte!)
CO$$$$T	OK…	OK…	TERRIBLE
QUERY TIME	TERRIBLE	BAD	Good!

CSV vs. Parquet Column Selection Query



CSV vs. Parquet Table Scan Query



Parquet Format

"Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language."

- Binary Format
- API for JVM/Hadoop & C++
- Columnar
- Encoded
- Compressed
- Machine-Friendly

Parquet By Example

Introducing the Dataset

Very Important Dataset

Title	Released	Label	PeakChart.UK	Certification.BVMI	Certification.RIAA	(omitted for space...)
Led Zeppelin	01/12/1969	Atlantic	6		8x Platinum	...
Led Zeppelin II	10/22/1969	Atlantic	1	Platinum	Diamond	...
Led Zeppelin III	10/05/1970	Atlantic	1	Gold	6x Platinum	...
Led Zeppelin IV	11/08/1971	Atlantic	1	3x Gold	Diamond	...
Houses of the Holy	03/28/1973	Atlantic	1	Gold	Diamond	...
Physical Graffiti	02/24/1975	Swan Song	1	Gold	Diamond	...
Presence	03/31/1976	Swan Song	1		3x Platinum	...
In Through The Out Door	08/15/1979	Swan Song	1		6x Platinum	...
Coda	11/19/1982	Swan Song	4		Platinum	...

One Row, Two Different Ways

```
{
  "Title" : "Led Zeppelin IV",
  "Released" : "11/8/1971",
  "Label" : "Atlantic",
  "PeakChart.UK" : 1,
  "PeakChart.AUS" : 2,
  "PeakChart.US" : 2,
  "Certification.ARIA" : "9x Platinum",
  "Certification.BPI" : "6x Platinum",
  "Certification.BVMI" : "3x Gold",
  "Certification.CRIA" : "2x Diamond",
  "Certification.IFPI" : "2x Platinum",
  "Certification.NVPI" : "Platinum",
  "Certification.RIAA" : "Diamond",
  "Certification.SNEP" : "2x Platinum"
}
```

```
{
  "TITLE": "LED ZEPPELIN IV",
  "RELEASED": "11/8/1971",
  "LABEL": "ATLANTIC",
  "PEAKCHART": {
      "UK": 1,
      "AUS": 2,
      "US": 2 },
  "CERTIFICATION": {
      "ARIA": "9X PLATINUM",
      "BPI": "6X PLATINUM",
      "BVMI": "3X GOLD",
      "CRIA": "2X DIAMOND",
      "IFPI": "2X PLATINUM",
      "NVPI": "PLATINUM",
      "RIAA": "DIAMOND",
      "SNEP": "2X PLATINUM" }
}
```

The Flat Schema Data

```
Title   Released   Label  PeakChart.UK   PeakChart.AUS  PeakChart.US   PeakChart.Mars
Certification.ARIA Certification.BPI  Certification.BVMI Certification.CRIA Certification.IFPI
Certification.NVPI Certification.RIAA Certification.SNEP
Led Zeppelin   01/12/1969 Atlantic   6  9  10    2x Platinum    2x Platinum       Diamond    Gold
Gold   8x Platinum    Gold
Led Zeppelin II    10/22/1969 Atlantic   1  1  1     4x Platinum    4x Platinum    Platinum   9x
Platinum    Gold      Diamond    Platinum
Led Zeppelin III   10/5/1970  Atlantic   1  1  1        Platinum   Gold   3x Platinum    Gold   Gold
6x Platinum    Platinum
Led Zeppelin IV    11/8/1971  Atlantic   1  2  2     9x Platinum    6x Platinum    3x Gold    2x
Diamond 2x Platinum    Platinum   Diamond    2x Platinum
Houses of the Holy 03/28/1973 Atlantic   1  1  1        Platinum   Gold           Diamond    Gold
Physical Graffiti  02/24/1975 Swan Song  1  1  1     3x Platinum    2x Platinum    Gold
Diamond    Gold
Presence   03/31/1976 Swan Song  1  4  1        Platinum              3x Platinum
In Through The Out Door    08/15/1979 Swan Song  1  3  1     2x Platinum    Platinum              6x
Platinum
Coda   11/19/1982 Swan Song  4  9  6        Silver            Platinum
```

The Nested Schema Data


```
{"Title":"Led Zeppelin", "Released":"01/12/1969", "Label":"Atlantic", "PeakChart":{"UK":6, "AUS":9,
"US":10}, "Certification":{"ARIA":"2x Platinum", "BPI":"2x Platinum", "CRIA":"Diamond", "IFPI":"Gold",
"NVPI":"Gold", "RIAA":"8x Platinum", "SNEP":"Gold"}}
{"Title":"Led Zeppelin II", "Released":"10/22/1969", "Label":"Atlantic", "PeakChart":{"UK":1, "AUS":1,
"US":1}, "Certification":{"ARIA":"4x Platinum",  "BPI":"4x Platinum", "BVMI":"Platinum", "CRIA":"9x
Platinum", "IFPI":"Gold", "RIAA":"Diamond", "SNEP":"Platinum"}}
{"Title":"Led Zeppelin III", "Released":"10/5/1970", "Label":"Atlantic", "PeakChart":{"UK":1, "AUS":1,
"US":1}, "Certification":{"BPI":"Platinum", "BVMI":"Gold", "CRIA":"3x Platinum", "IFPI":"Gold",
"NVPI":"Gold", "RIAA":"6x Platinum", "SNEP":"Platinum"}}
{"Title":"Led Zeppelin IV", "Released":"11/8/1971", "Label":"Atlantic", "PeakChart":{"UK":1, "AUS":2,
"US":2}, "Certification":{"ARIA":"9x Platinum", "BPI":"6x Platinum", "BVMI":"3x Gold", "CRIA":"2x
Diamond", "IFPI":"2x Platinum", "NVPI":"Platinum", "RIAA":"Diamond", "SNEP":"2x Platinum"}}
{"Title":"Houses of the Holy", "Released":"03/28/1973", "Label":"Atlantic", "PeakChart":{"UK":1,
"AUS":1, "US":1}, "Certification":{"BPI":"Platinum", "BVMI":"Gold", "RIAA":"Diamond", "SNEP":"Gold"}}
{"Title":"Physical Graffiti", "Released":"02/24/1975", "Label":"Swan Song", "PeakChart":{"UK":1,
"AUS":1, "US":1}, "Certification":{"ARIA":"3x Platinum", "BPI":"2x Platinum", "BVMI":"Gold",
"RIAA":"Diamond", "SNEP":"Gold"}}
{"Title":"Presence", "Released":"03/31/1976", "Label":"Swan Song", "PeakChart":{"UK":1, "AUS":4,
"US":1}, "Certification":{"BPI":"Platinum", "RIAA":"3x Platinum"}}
{"Title":"In Through The Out Door", "Released":"08/15/1979", "Label":"Swan Song", "PeakChart":{"UK":1,
"AUS":3, "US":1}, "Certification":{"ARIA":"2x Platinum", "BPI":"Platinum", "RIAA":"6x Platinum"}}
{"Title":"Coda", "Released":"11/19/1982", "Label":"Swan Song", "PeakChart":{"UK":4, "AUS":9, "US":6},
"Certification":{"BPI":"Silver", "RIAA":"Platinum"}}
```

Parquet By Example

Writing Parquet Using Spark

Writing To Parquet: Flat Schema

```
val flatDF = spark
  .read.option("delimiter", "\t")
  .option("header", "true").csv(flatInput)
  .rdd
  .map(r => transformRow(r))
  .toDF

flatDF.write
  .option("compression", "snappy")
  .parquet(flatOutput)
```

SPARK SUMMIT EAST 2017

Writing To Parquet: Flat Schema

```
/*Oh crap, the Ints are gonna get pulled in as Strings unless we transform*/

case class LedZeppelinFlat(
                            Title: Option[String],
                            Released: Option[String],
                            Label: Option[String],
                            UK: Option[Int],
                            AUS: Option[Int],
                            US: Option[Int],
                            ARIA: Option[String],
                            BPI: Option[String],
                            BVMI: Option[String],
                            CRIA: Option[String],
                            IFPI: Option[String],
                            NVPI: Option[String],
                            RIAA: Option[String],
                            SNEP: Option[String]
                          )
```

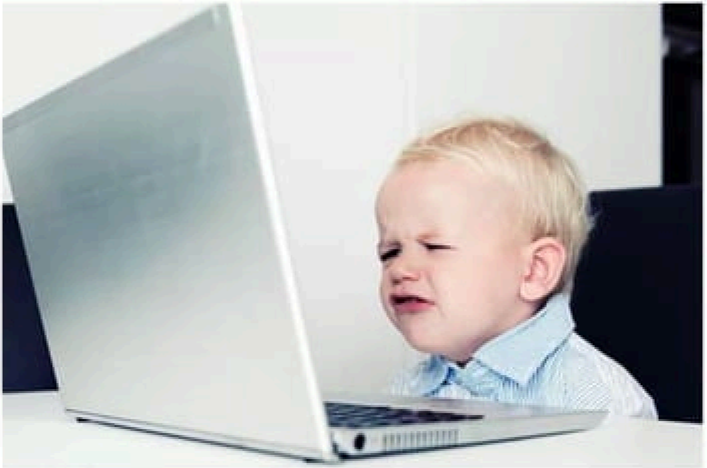
Writing To Parquet: Flat Schema

```
def transformRow(r: Row): LedZeppelinFlat = {
  def getStr(r: Row, i: Int) = if(!r.isNullAt(i)) Some(r.getString(i)) else None
  def getInt(r: Row, i: Int) = if(!r.isNullAt(i)) Some(r.getInt(i)) else None

  LedZeppelinFlat(
    getStr(r, 0),
    getStr(r, 1),
    getStr(r, 2),
    getInt(r, 3),
    getInt(r, 4),
    getInt(r, 5),
    getStr(r, 7),
    getStr(r, 8),
    getStr(r, 9),
    getStr(r, 10),
    getStr(r, 11),
    getStr(r, 12),
    getStr(r, 13),
    getStr(r, 14)
  )
}
```

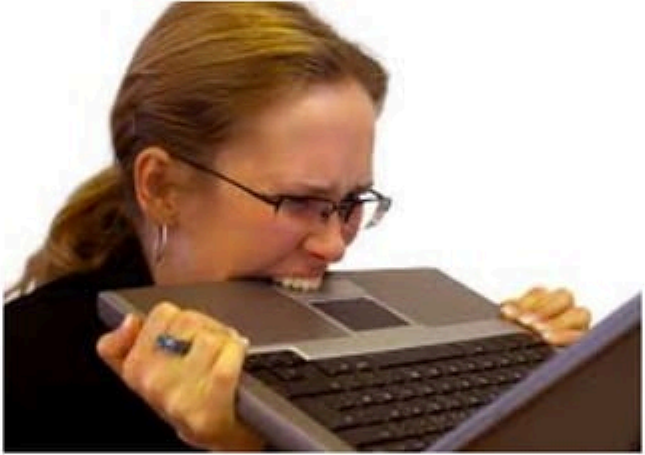
Writing To Parquet: Flat Schema

```
val outDF = spark
  .read.option("delimiter", "\t")
  .option("header", "true").csv(flatInput)
  .rdd
  .map(r => transformRow(r))
  .toDF

outDF.write
  .option("compression", "snappy")
  .parquet(flatOutput)
```

Writing To Parquet: Flat Schema



Writing To Parquet: Flat Schema… In Java



Writing To Parquet: Flat Schema… With MapReduce

Writing To Parquet: Nested Schema

```
val nestedDF = spark.read.json(nestedInput)

nestedDF.write
  .option("compression", "snappy")
  .parquet(nestedOutput)
```

Writing To Parquet: Nested Schema

Parquet By Example

Let's See An Example!

Parquet Schema Two Different Ways

FLAT SCHEMA

```
TITLE:                  OPTIONAL BINARY O:UTF8 R:0 D:1
RELEASED:               OPTIONAL BINARY O:UTF8 R:0 D:1
LABEL:                  OPTIONAL BINARY O:UTF8 R:0 D:1
PEAKCHART.UK:           REQUIRED INT32 R:0 D:0
PEAKCHART.AUS:          REQUIRED INT32 R:0 D:0
PEAKCHART.US:           REQUIRED INT32 R:0 D:0
CERTIFICATION.ARIA:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.BPI:      OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.BVMI:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.CRIA:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.IFPI:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.NVPI:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.RIAA:     OPTIONAL BINARY O:UTF8 R:0 D:1
CERTIFICATION.SNEP:     OPTIONAL BINARY O:UTF8 R:0 D:1
```

Nested Schema

```
Title:          OPTIONAL BINARY O:UTF8 R:0 D:1
Released:       OPTIONAL BINARY O:UTF8 R:0 D:1
Label:          OPTIONAL BINARY O:UTF8 R:0 D:1
PeakChart:      OPTIONAL F:3
.AUS:           OPTIONAL INT64 R:0 D:2
.UK:            OPTIONAL INT64 R:0 D:2
.US:            OPTIONAL INT64 R:0 D:2
Certification:  OPTIONAL F:8
.ARIA:          OPTIONAL BINARY O:UTF8 R:0 D:2
.BPI:           OPTIONAL BINARY O:UTF8 R:0 D:2
.BVMI:          OPTIONAL BINARY O:UTF8 R:0 D:2
.CRIA:          OPTIONAL BINARY O:UTF8 R:0 D:2
.IFPI:          OPTIONAL BINARY O:UTF8 R:0 D:2
.NVPI:          OPTIONAL BINARY O:UTF8 R:0 D:2
.RIAA:          OPTIONAL BINARY O:UTF8 R:0 D:2
.SNEP:          OPTIONAL BINARY O:UTF8 R:0 D:2
```

Schema Breakdown

COLUMN NAME	Title
OPTIONAL / REQUIRED / REPEATED	OPTIONAL
DATA TYPE	BINARY
ENCODING INFO FOR BINARY	0:UTF8
REPETITION VALUE	R:0
DEFINITION VALUE	D:0

FLAT SCHEMA

```
TITLE:          OPTIONAL BINARY O:UTF8 R:0 D:1
RELEASED:       OPTIONAL BINARY O:UTF8 R:0 D:1
LABEL:          OPTIONAL BINARY O:UTF8 R:0 D:1
PEAKCHART.UK:   REQUIRED INT32 R:0 D:0

. . .
```

Repetition and Definition Levels

- Document: R = 0, D = 0
  - DocId: *required*
  - Links: *optional* D = 1
    - Backward: *repeated* R = 1, D = 2
    - Forward: *repeated* R = 1, D = 2
  - Name: *repeated* R = 1, D = 1
    - Language: *repeated* R = 2, D = 2
      - Code: *required*
      - Country: *optional* D = 3
    - Url: *optional* D = 2

required: same Repetition and Definition level as parent
optional: same Repetition level as parent, increment Definition level
repeated: increment both Repetition and Definition levels

R = 0	R = 1	R = 2
Document.DocId		
Document.Links	Backward	
Document.Links	Forward	
Document	Name	Language.Code
Document	Name	Language.Country
Document	Name.Url	

D = 0	D = 1	D = 2	D = 3
Document.DocId			
Document	Links	Backward	
Document	Links	Forward	
Document	Name	Language.Code	
Document	Name	Language	Country
Document	Name	Url	

Source: https://github.com/apache/parquet-mr

One Parquet Row, Two Ways

```
TITLE = LED ZEPPELIN IV
RELEASED = 11/8/1971
LABEL = ATLANTIC
PEAKCHART.UK = 1
PEAKCHART.AUS = 2
PEAKCHART.US = 2
CERTIFICATION.ARIA = 9X PLATINUM
CERTIFICATION.BPI = 6X PLATINUM
CERTIFICATION.BVMI = 3X GOLD
CERTIFICATION.CRIA = 2X DIAMOND
CERTIFICATION.IFPI = 2X PLATINUM
CERTIFICATION.NVPI = PLATINUM
CERTIFICATION.RIAA = DIAMOND
CERTIFICATION.SNEP = 2X PLATINUM
```

```
Title = Led Zeppelin IV
Released = 11/8/1971
Label = Atlantic
PeakChart:
.AUS = 2
.UK = 1
.US = 2
Certification:
.ARIA = 9x Platinum
.BPI = 6x Platinum
.BVMI = 3x Gold
.CRIA = 2x Diamond
.IFPI = 2x Platinum
.NVPI = Platinum
.RIAA = Diamond
.SNEP = 2x Platinum
```

SPARK SUMMIT EAST 2017

Parquet By Example

Reading and Querying Using Spark

Slightly Different Queries

```
// Many ways, this is just one!

val flatParquet = "s3a://…../LedZeppelin-FlatSchema.parquet/"
val flatdf = spark.read.parquet(flatParquet)
flatdf.createOrReplaceTempView("LedZeppelinFlat")

val nestedParquet = "s3a://…../LedZeppelin-NestedSchema.parquet/"
val nesteddf = spark.read.parquet(nestedParquet)
nesteddf.createOrReplaceTempView("LedZeppelinNested")

val flatQuery= "select Title, US from LedZeppelinFlat where US = 1"
val nestedQuery = "select Title, PeakChart.US from LedZeppelinNested
where PeakChart.US = 1"

spark.sql(flatQuery)
spark.sql(nestedQuery)
```

Same Result

FINISHED

```
%sql

select Title, US from LedZeppelinFlat
where US = 1
```

Title	US
Led Zeppelin II	1
Led Zeppelin III	1
Houses of the Holy	1
Physical Graffiti	1
Presence	1
In Through The Out Door	1

FINISHED

```
%sql

select Title, PeakChart.US from LedZeppelinNested
where PeakChart.US = 1
```

Title	US
Led Zeppelin II	1
Led Zeppelin III	1
Houses of the Holy	1
Physical Graffiti	1
Presence	1
In Through The Out Door	1

How Parquet Works

SPARK SUMMIT EAST 2017

Parquet Structure In the Filesystem

- Groups of rows, partitioned by column values, compressed however you like. (GZIP, LZO, Snappy, etc)
- In general LZO wins size benchmarks, Snappy good balance between size and CPU intensity.

```
led-zeppelin-albums.parquet/
•   _SUCCESS
•   _common_metadata
•   _metadata
•   Year=1969/
    – Part-r-00000-6d4d42e2-c13f-4bdf-917d-2152b24a0f24.snappy.parquet
    – Part-r-00001-6d4d42e2-c13f-4bdf-917d-2152b24a0f24.snappy.parquet
    – …
•   Year=1970/
    – Part-r-00000-35cb7ef4-6de6-4efa-9bc6-5286de520af7.snappy.parquet
    – ...
```

Data In Columns On Disk

Title	Date	Chart

Row-Oriented data on disk

Led Zeppelin IV	11/08/1971	1	Houses of the Holy	03/28/1973	1	Physical Graffiti	02/24/1975	1

Column-Oriented data on disk

Led Zeppelin IV	Houses of the Holy	Physical Graffiti	11/08/1971	03/28/1973	02/24/1975	1	1	1

Encoding: Incremental Encoding

Led_Zeppelin	0 Led_Zeppelin
Led_Zeppelin_II	12 _II
Led_Zeppelin_III	15 I
Led_Zeppelin_IV	14 V
58 bytes*	24 bytes*

ENCODING

58% Reduction

*not counting delimiters

SPARK SUMMIT EAST 2017

Encoding: Dictionary Encoding

Atlantic
Atlantic
Atlantic
Atlantic
Atlantic
Atlantic
Swan Song
Swan Song
Swan Song
Swan Song

84 bytes*

ENCODING

0 → Atlantic
1 → Swan Song

~98% Reduction

0
0
0
0
0
0
1
1
1
1

1.25 bytes + dictionary size

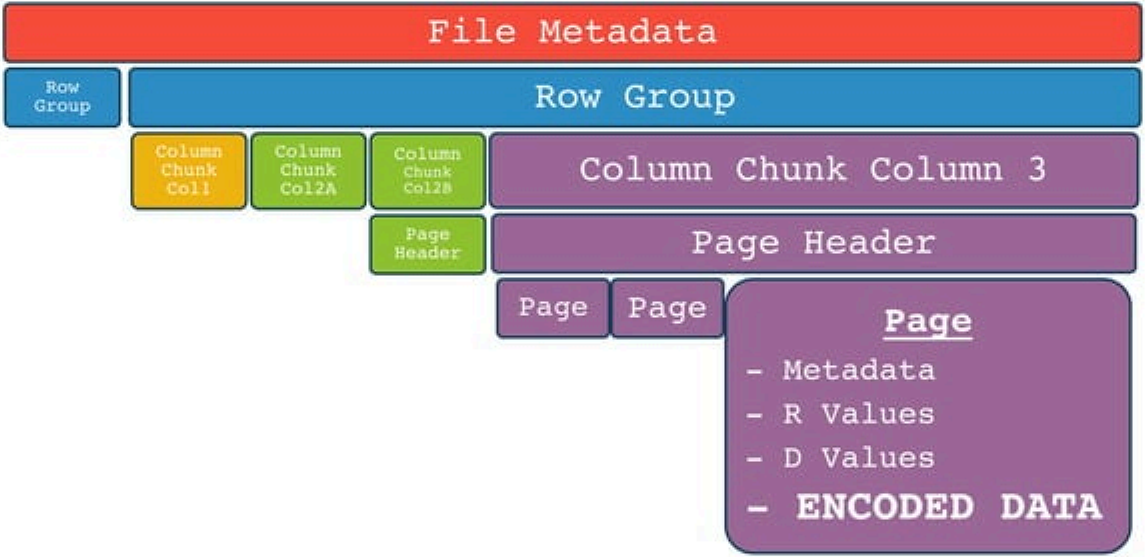
More Encoding Schemes

- Plain (bit-packed, little endian, etc)
- Dictionary Encoding
- Run Length Encoding/Bit Packing Hybrid
- Delta Encoding
- Delta-Length Byte Array
- Delta Strings (incremental Encoding)

SPARK SUMMIT EAST 2017

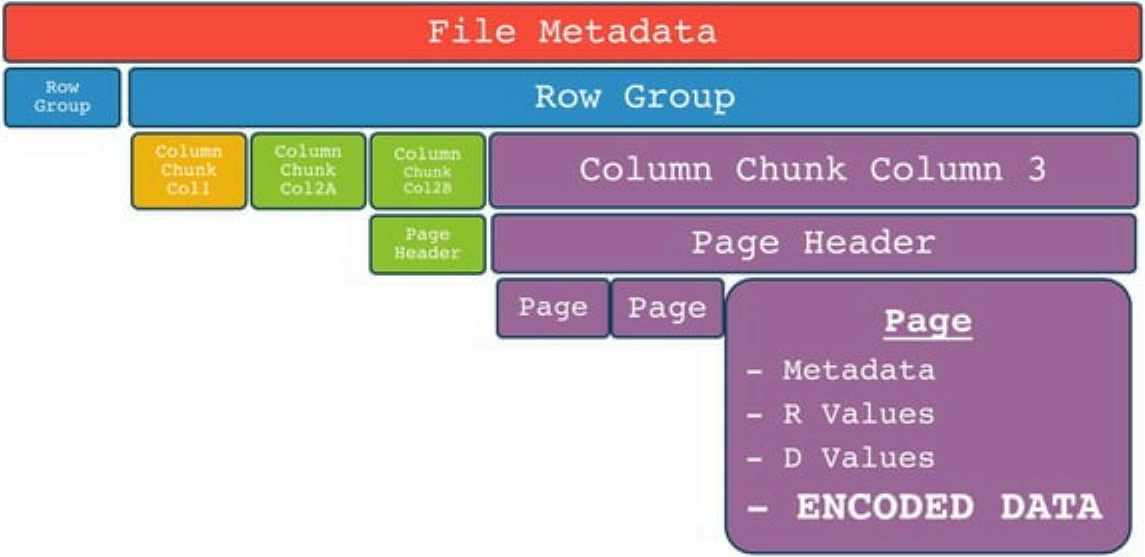
See https://github.com/apache/parquet-format/blob/master/Encodings.md for more detail

Slicing and Dicing Within A Compressed File

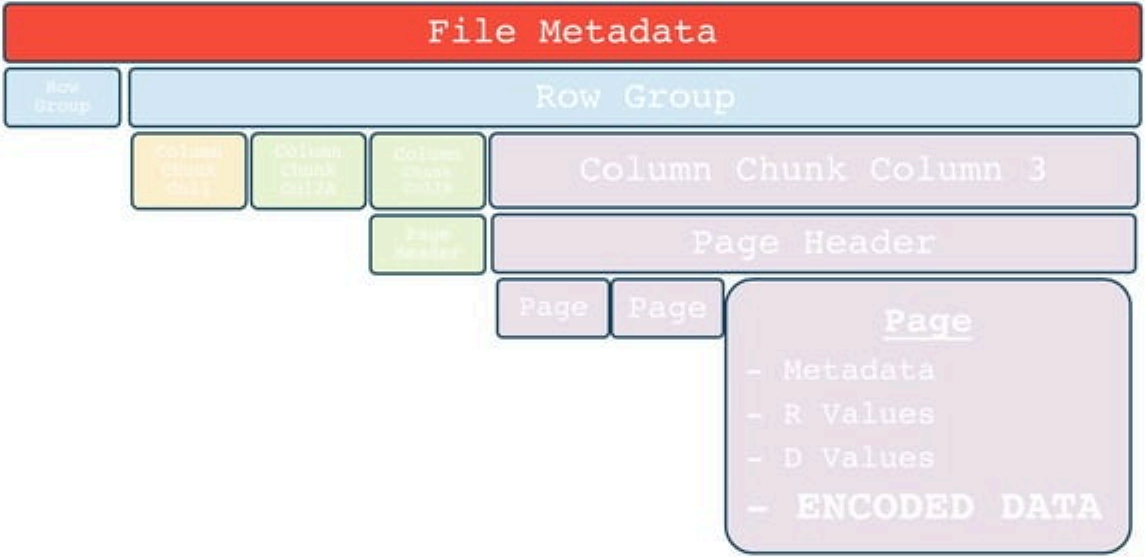


Slicing and Dicing Within A Compressed File

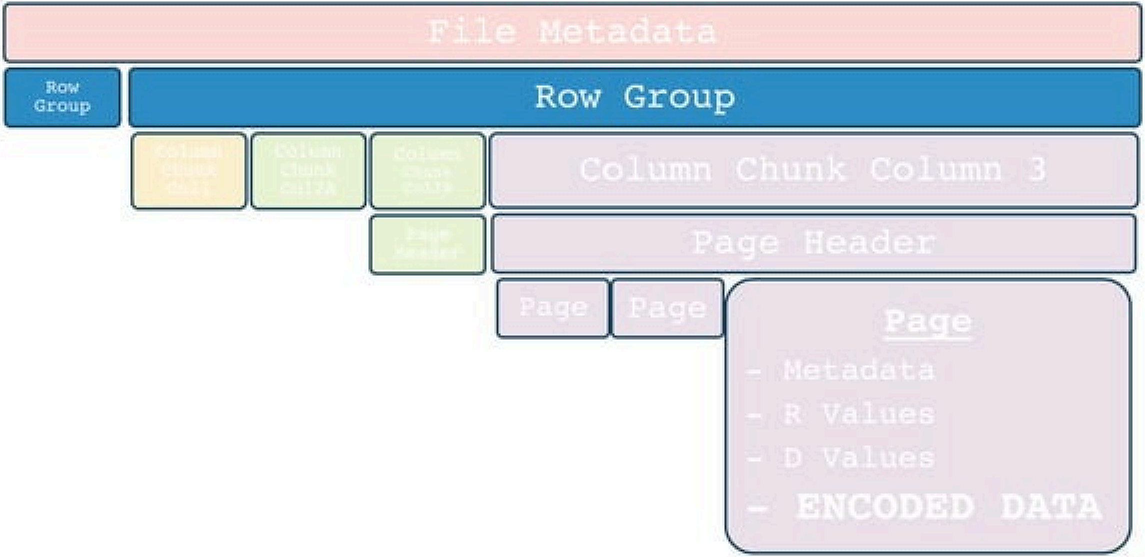
Slicing and Dicing Within A Compressed File



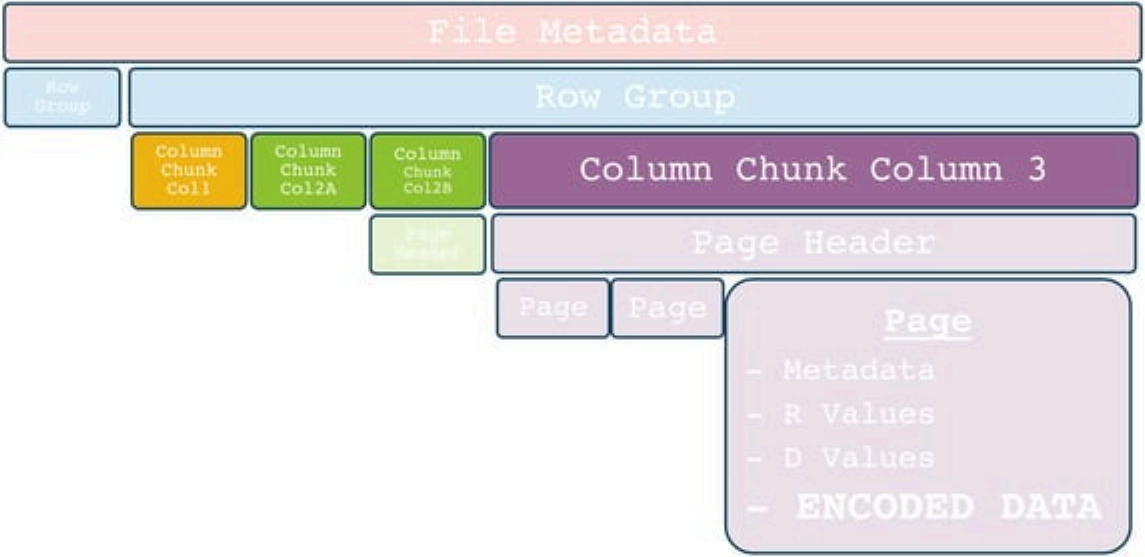
Slicing and Dicing Within A Compressed File



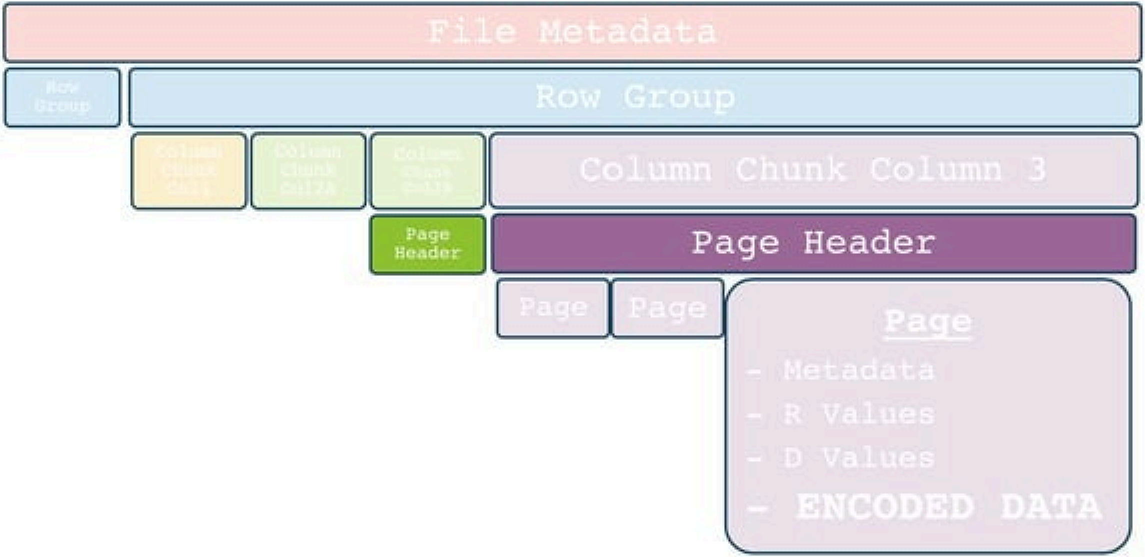
Slicing and Dicing Within A Compressed File



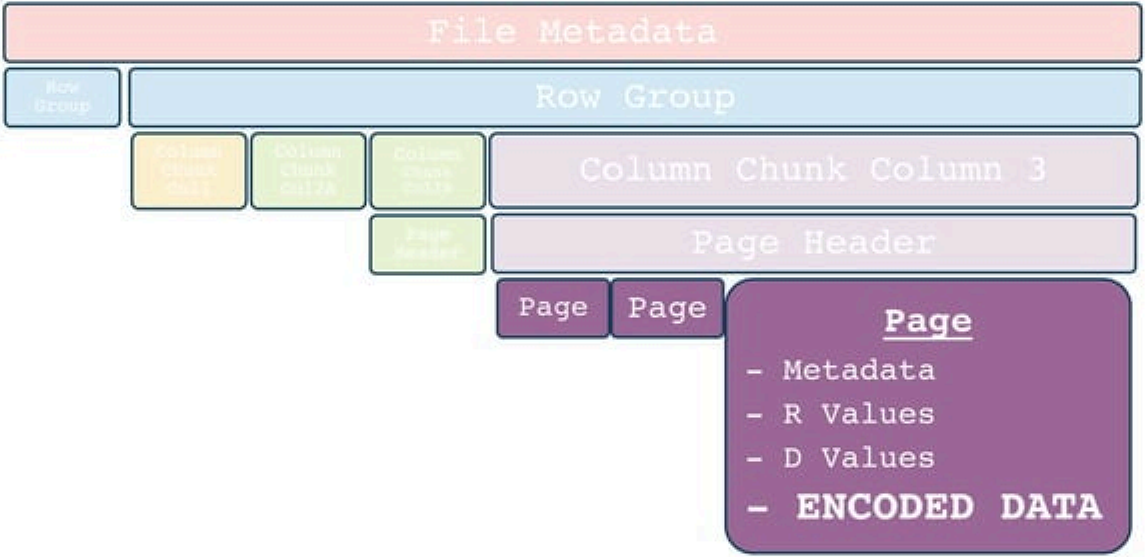
Slicing and Dicing Within A Compressed File



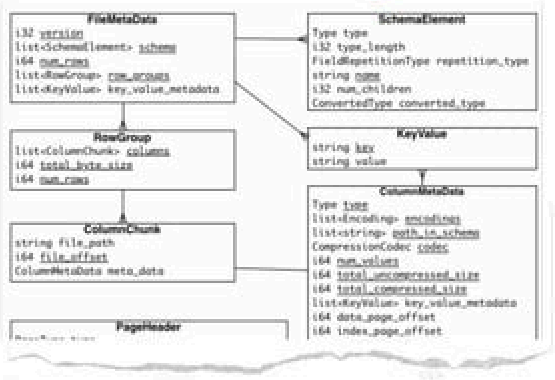
Slicing and Dicing Within A Compressed File



Slicing and Dicing Within A Compressed File



Format Spec



See the format spec for more detail:

https://github.com/apache/parquet-format

Getting Efficiency With Spark

Partitioning

```
dataFrame
    .write
    .partitionBy("Whatever", "Columns", "You", "Want")
    .parquet(outputFile)


// For a common example
dataFrame
    .write
    .partitionBy("Year", "Month", "Day", "Hour")
    .parquet(outputFile)
```

Spark Filter Pushdown

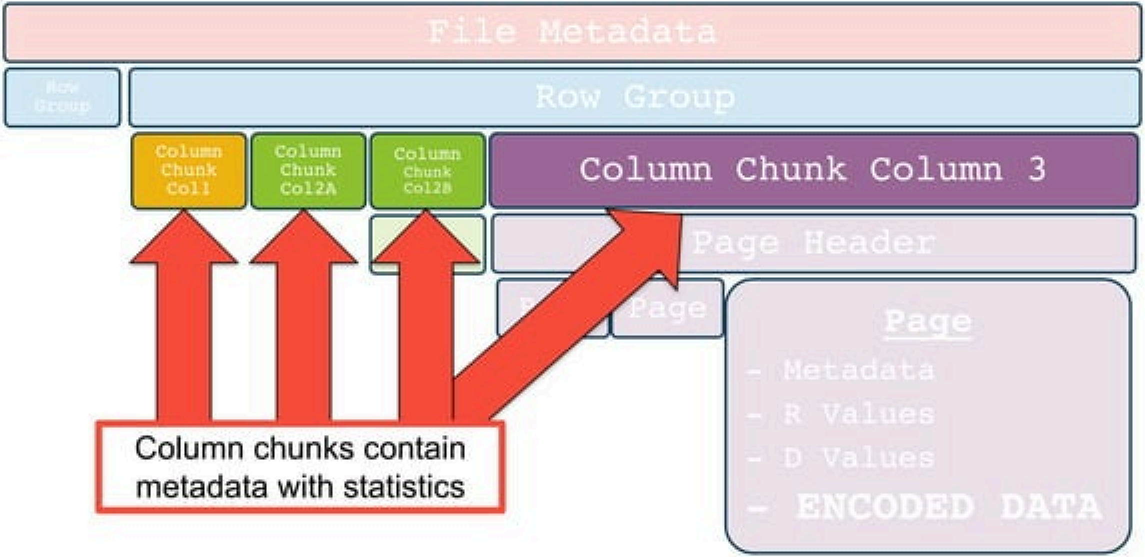
`spark.sql.parquet.filterPushdown` → **`true`** by default since 1.5.0

For **`Where`** Clauses, **`Having`** clauses, etc. in SparkSQL, The Data Loading layer will test the condition before pulling a column chunk into spark memory.

```
select cs_bill_customer_sk customer_sk, cs_item_sk item_sk
from catalog_sales,date_dim
where cs_sold_date_sk = d_date_sk
and d_month_seq between 1200 and 1200 + 11
```


Example From: https://developer.ibm.com/hadoop/2016/01/14/5-reasons-to-choose-parquet-for-spark-sql/

Slicing and Dicing Within A Compressed File



Physical Plan for Reading CSV

```
[ Scan
CsvRelation(hdfs://rhel10.cisco.com/user/spark/hadoopds1000g/date_dim/*,false,|,",null,PERMISS
IVE,COMMONS,false,false,StructType(StructField(d_date_sk,IntegerType,false),
StructField(d_date_id,StringType,false), StructField(d_date,StringType,true),
StructField(d_month_seq,LongType,true), StructField(d_week_seq,LongType,true),
StructField(d_quarter_seq,LongType,true), StructField(d_year,LongType,true),
StructField(d_dow,LongType,true), StructField(d_moy,LongType,true),
StructField(d_dom,LongType,true), StructField(d_qoy,LongType,true),
StructField(d_fy_year,LongType,true), StructField(d_fy_quarter_seq,LongType,true),
StructField(d_fy_week_seq,LongType,true), StructField(d_day_name,StringType,true),
StructField(d_quarter_name,StringType,true), StructField(d_holiday,StringType,true),
StructField(d_weekend,StringType,true), StructField(d_following_holiday,StringType,true),
StructField(d_first_dom,LongType,true), StructField(d_last_dom,LongType,true),
StructField(d_same_day_ly,LongType,true), StructField(d_same_day_lq,LongType,true),
StructField(d_current_day,StringType,true), StructField(d_current_week,StringType,true),
StructField(d_current_month,StringType,true), StructField(d_current_quarter,StringType,true),
StructField(d_current_year,StringType,true)))[d_date_sk#141,d_date_id#142,d_date#143,d_month_s
eq#144L,d_week_seq#145L,d_quarter_seq#146L,d_year#147L,d_dow#148L,d_moy#149L,d_dom#150L,d_qoy#
151L,d_fy_year#152L,d_fy_quarter_seq#153L,d_fy_week_seq#154L,d_day_name#155,d_quarter_name#156
,d_holiday#157,d_weekend#158,d_following_holiday#159,d_first_dom#160L,d_last_dom#161L,d_same_d
ay_ly#162L,d_same_day_lq#163L,d_current_day#164,d_current_week#165,d_current_month#166,d_curre
nt_quarter#167,d_current_year#168]]
```

Example From: https://developer.ibm.com/hadoop/2016/01/14/5-reasons-to-choose-parquet-for-spark-sql/

Physical Plan For Reading Parquet

```
+- Scan ParquetRelation[d_date_sk#141,d_month_seq#144L] InputPaths:
hdfs://rhel10.cisco.com/user/spark/hadoopds1tbparquet/date_dim/_SUCCESS,
hdfs://rhel10.cisco.com/user/spark/hadoopds1tbparquet/date_dim/_common_metadata
, hdfs://rhel10.cisco.com/user/spark/hadoopds1tbparquet/date_dim/_metadata,
hdfs://rhel10.cisco.com/user/spark/hadoopds1tbparquet/date_dim/part-r-00000-
4d205b7e-b21d-4e8b-81ac-d2a1f3dd3246.gz.parquet,
hdfs://rhel10.cisco.com/user/spark/hadoopds1tbparquet/date_dim/part-r-00001-
4d205b7e-b21d-4e8b-81ac-d2a1f3dd3246.gz.parquet, PushedFilters:
[GreaterThanOrEqual(d_month_seq,1200),
LessThanOrEqual(d_month_seq,1211)]]
```

Example From: https://developer.ibm.com/hadoop/2016/01/14/5-reasons-to-choose-parquet-for-spark-sql/

Get JUST the Data You Need

- Get just the partitions you need
- Get just the columns you need
- Get just the chunks of the columns that fit your filter conditions

What’s the Catch?

Limitations, Write Speed, Immutability

Limitations

- Pushdown Filtering doesn’t exactly work with object stores: AWS S3, etc. No random access
- Pushdown Filtering does not work on nested columns - SPARK-17636
- Binary vs. String saga – SPARK-17213

Write Speed → Who Cares!!
(In Most Cases)

Write Once

Read Forever

Which case will you optimize for?

Dealing With Immutability

- Write using partitioning
  - Reimagine your data as a timeseries
- Combine with a database (i.e. Cassandra)
- Append additional row groups

Parquet in a Streaming Context

Ongoing project In the Watson Data Platform

- Collect until watermark condition is met (time, size, number of rows, etc.)
- Groom collection
- Write groomed rows to parquet
- Append to existing as additional compressed files

Tuning and Tips for Spark + Parquet

Tuning In Spark (depending on your version)

- Use s3a if you're in AWS land
- `df.read.option("mergeSchema", "false").parquet("s3a://whatever")`
- Coalescing will change the number of compressed files produced
- Make sure your Parquet block size == your HDFS block size
- `sparkContext.hadoopConfiguration.set( "spark.sql.parquet.output.committer.class", "org.apache.spark.sql.parquet.DirectParquetOutputCommitter")`

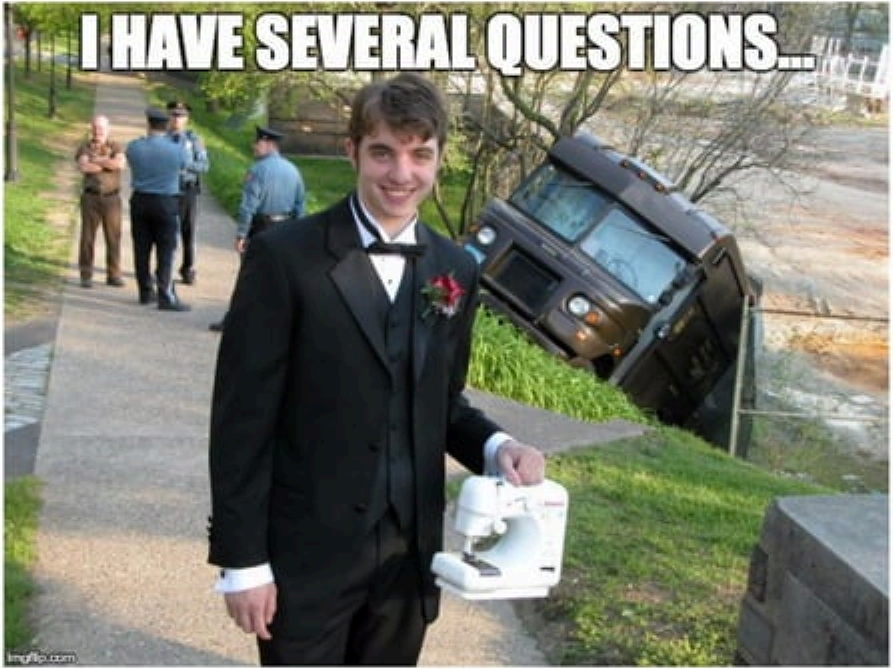
Let’s Summarize!

In Summary

	Parquet
Usability	Good!
Administration	None!
Spark Integration	FANTASTIC!!
Resource Efficiency	WONDERFUL!! (Storage, I/O, Data cardinality)
Scalability	FANTASTIC!!
CO$$$$T	¢ ¢ ¢
QUERY TIME	GOOD!!

In Summary

Parquet is a binary data storage format that,
in combination with Spark, enables fast queries by
getting you **just the data you need**,
getting it efficiently, and
keeping much of the **work out of Spark**.



SPARK SUMMIT EAST 2017

The Extra Slides

More About Those Benchmarks

File Format	Query Time (sec)	Size (GB)
CSV	2892.3	437.46
Parquet: LZO	50.6	55.6
Parquet: Uncompressed	43.4	138.54
Parquet: GZIP	40.3	36.78
Parquet: Snappy	28.9	54.83

```
SELECT cacheStatus, bytesSent from ADatasetThatHasToDoWithCDNs
WHERE cacheStatus LIKE 'stale'
AND bytesSent < 500
```

More About Those Benchmarks

- Wimpy cluster
  - 1 master
  - 3 workers
  - EC2 c4.4xlarge nodes
- All data in HDFS

Parquet vs. ORC

- ORC is columnar and indexed
- ORC does not handle nesting
- Table Scan benchmarks: comparable, ORC sometimes faster
- Selected Columns Benchmarks: Parquet wins
- **Benchmarks outdated**
  - Old versions of Spark
  - Old versions of ORC and Parquet spec

Parquet vs. Avro

- Avro is **row-major**
- Avro can be fast for table scans, but loses heavily on column-selection queries

Parquet vs. Kudu

- Parquet is immutable on disk
- Kudu is mutable on disk
- Trade-offs for both: http://www.slideshare.net/HadoopSummit/the-columnar-era-leveraging-parquet-arrow-and-kudu-for-highperformance-analytics

Robbie Strickland
VP, Engines & Pipelines, Watson Data Platform

Emily May Curtin
Software Engineer, IBM Spark Technology Center East