

A DEEP DIVE INTO DELTALAKE PERFORMANCE



Microsoft®
Most Valuable
Professional



**ADVANCING
ANALYTICS**

Databricks
BEACONS★



@ADVANCINGANALYTICS



@ADVANALYTICSUK



/ADVANCING ANALYTICS



www.advancinganalytics.co.uk

THE EVOLUTION OF LAKEHOUSES



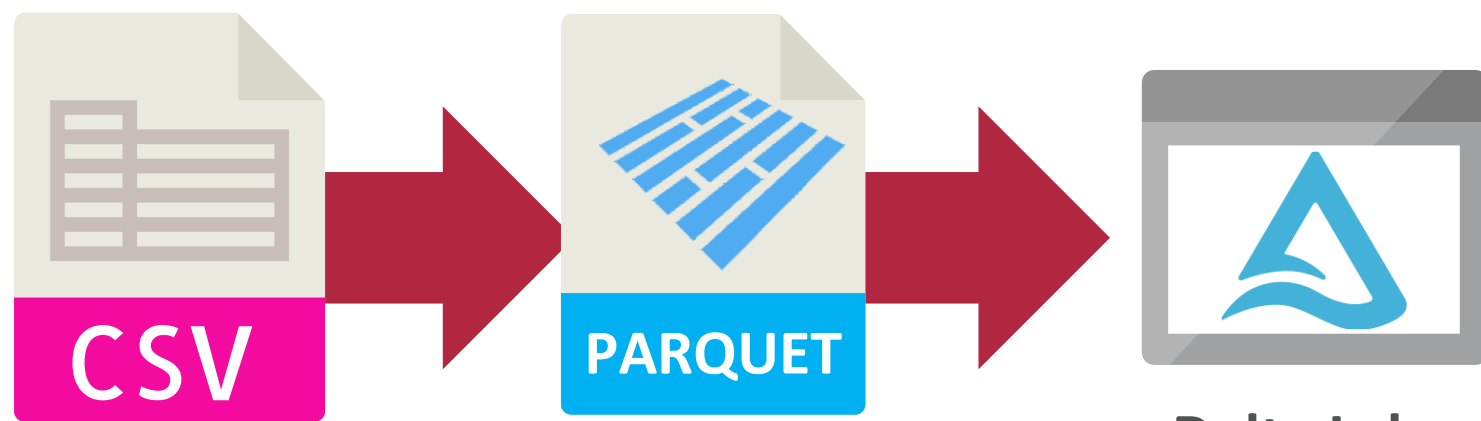
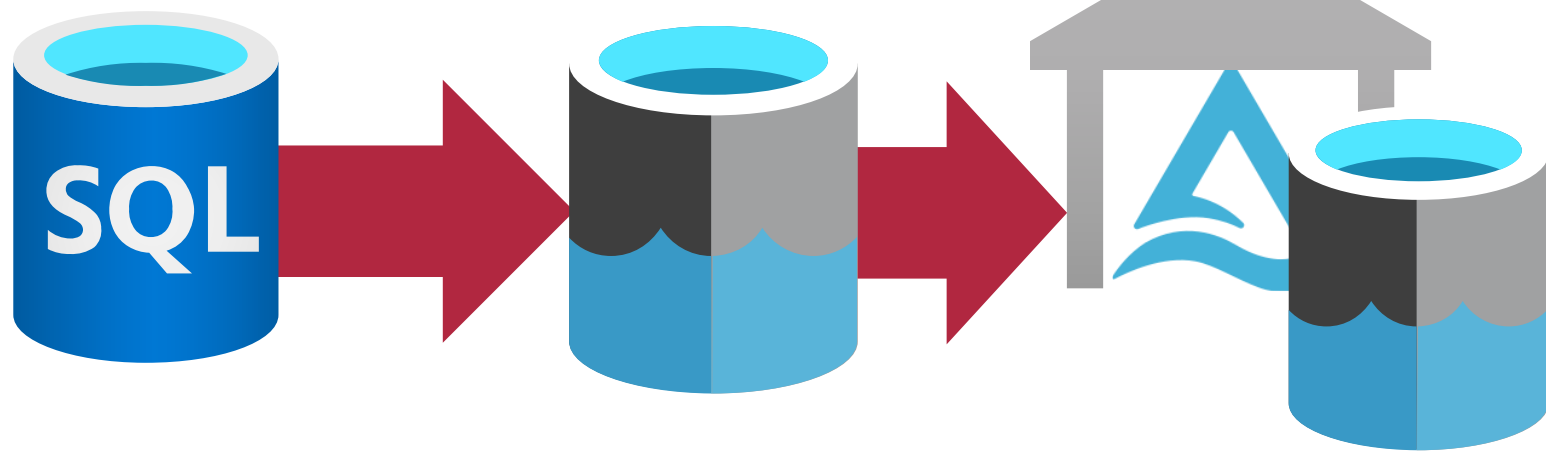
@ADVANCINGANALYTICS



@ADVANALYTICSUK

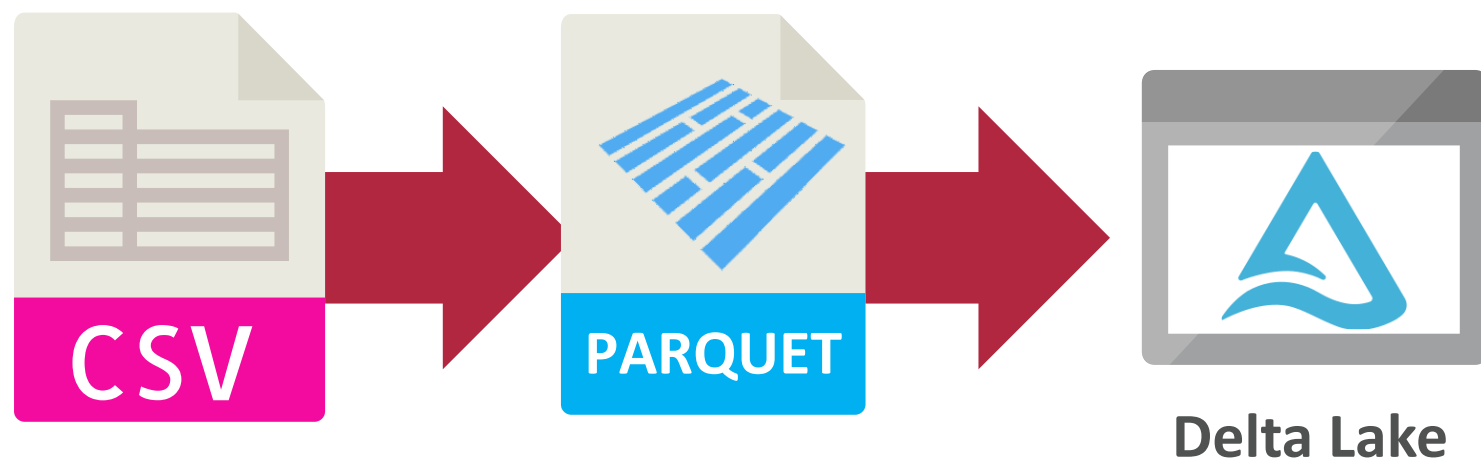
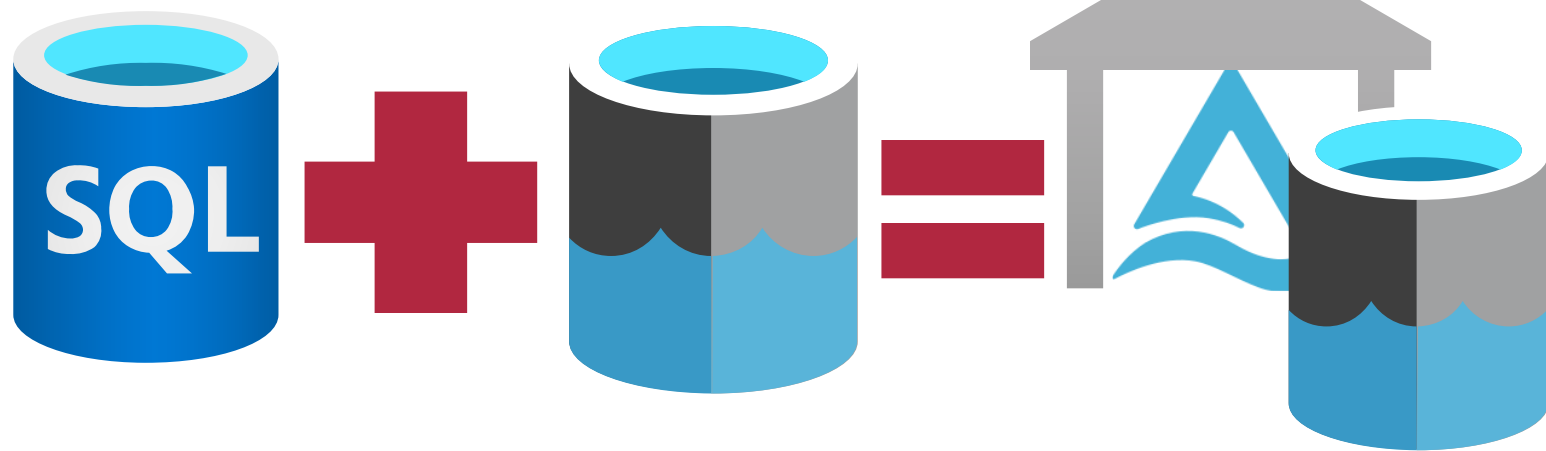


/ADVANCING ANALYTICS



Delta Lake







www.advancinganalytics.co.uk

SO WHAT'S WRONG WITH PARQUET?



@ADVANCINGANALYTICS

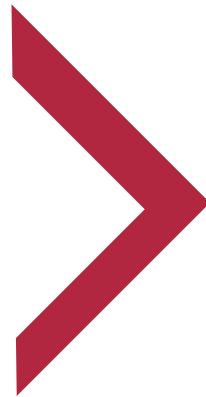


@ADVANALYTICSUK

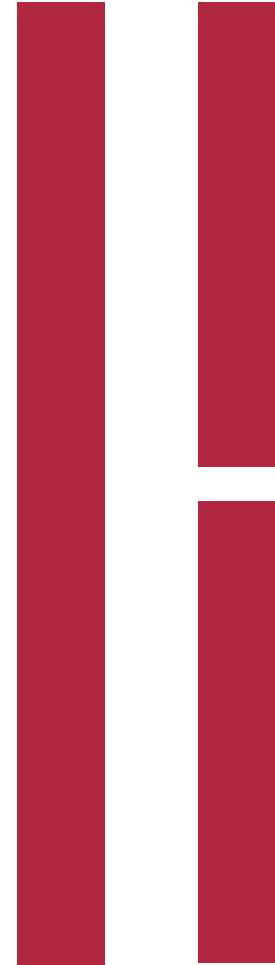


/ADVANCING ANALYTICS

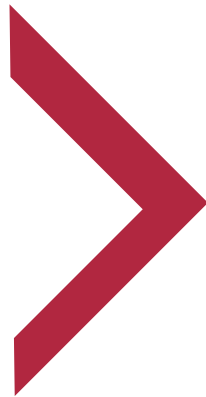
THE PROBLEMS OF PARQUET



Small files have a heavy performance impact

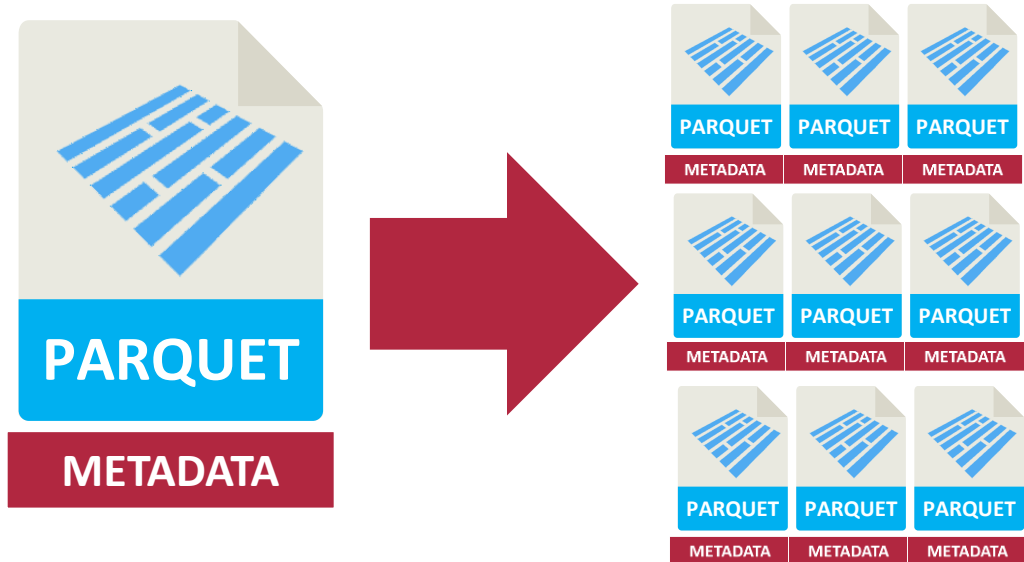


THE PROBLEMS OF PARQUET



Small files have a heavy performance impact

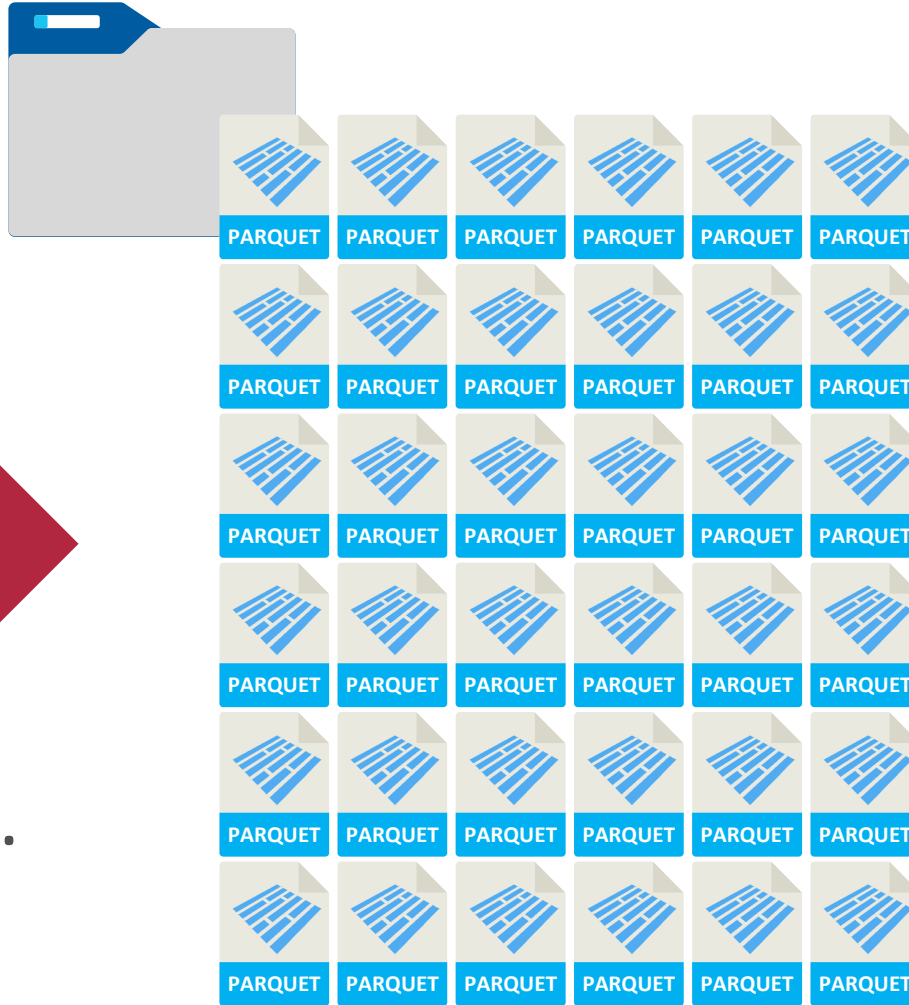
METADATA IN DATA FILES



Metadata scan = reading all files



NO INDEXES



**SELECT * FROM
MyTable WHERE...**





www.advancinganalytics.co.uk

WHAT IS DELTA?



@ADVANCINGANALYTICS



@ADVANALYTICSUK



/ADVANCING ANALYTICS

WHAT IS DELTA?

*Delta Lake is an **optimised,**
managed format for organising &
working with **Parquet** files*

"It's Parquet, but better"





www.advancinganalytics.co.uk

SOUNDS FANCY... HOW DOES IT WORK?



@ADVANCINGANALYTICS

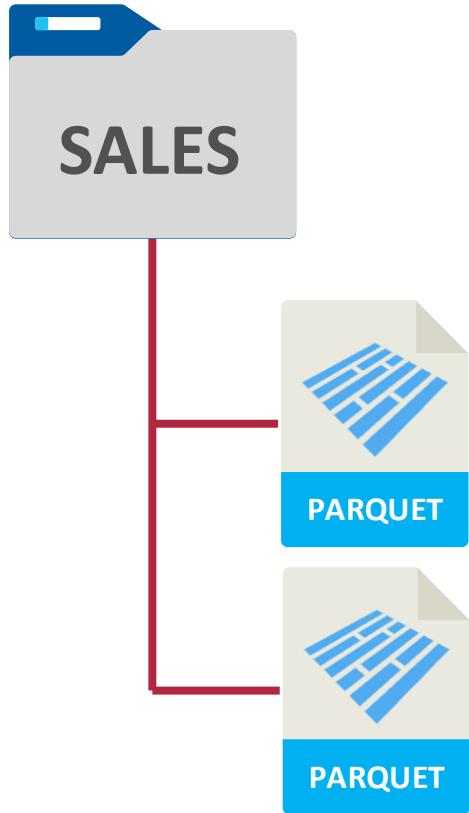


@ADVANALYTICSUK

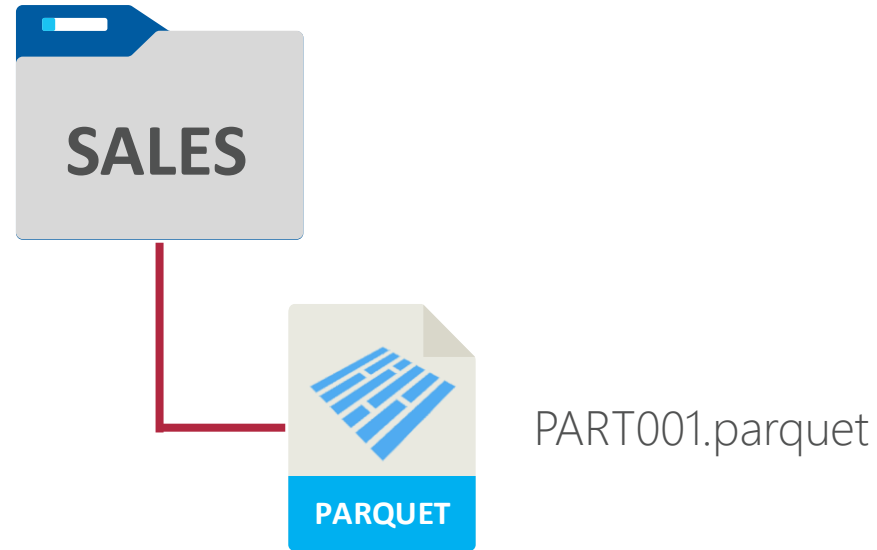


/ADVANCING ANALYTICS

BEFORE DELTA



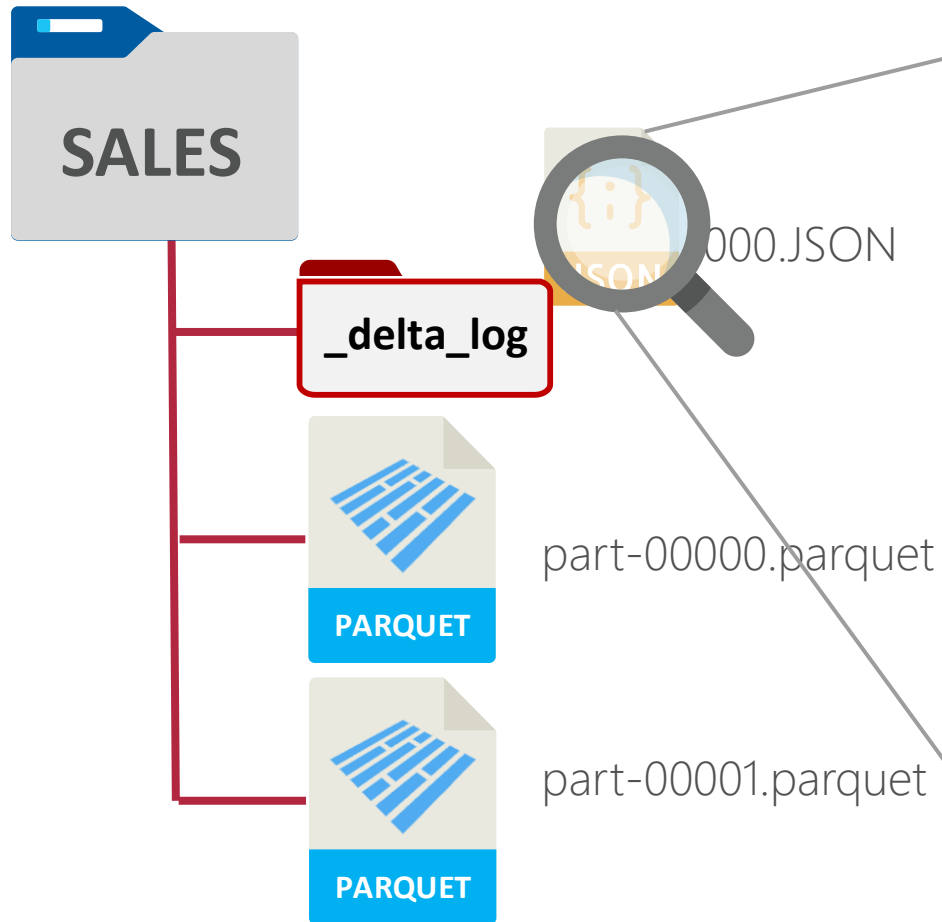
DELETE * **FROM** SALES **WHERE** Segment = 3



Only way to delete is to replace the existing files with a new file containing the non-deleted data



BUT WHAT ACTUALLY IS IT? - WITH DELTA:

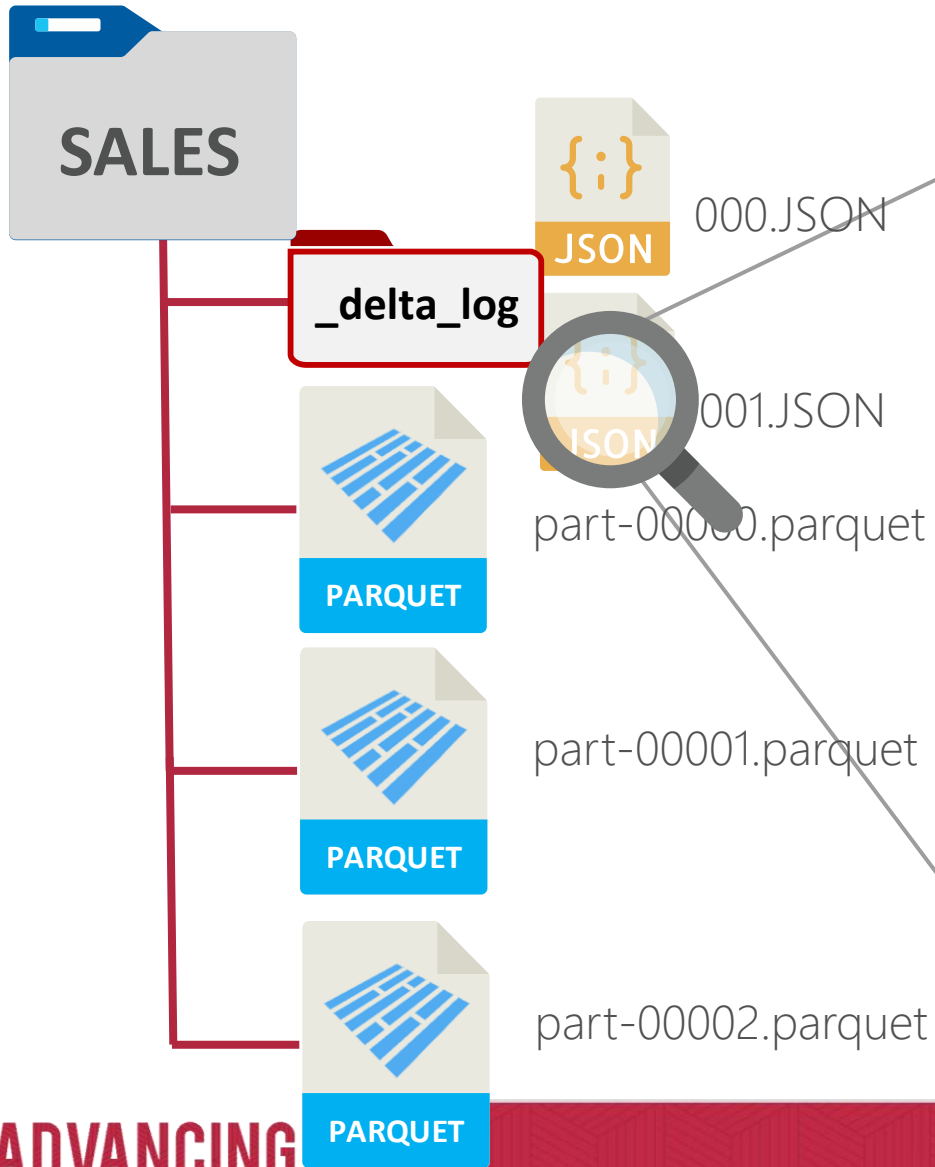


```
...
{
  "add": {
    "path": "part-00000.parquet",
    "partitionValues": {},
    "size": 255520,
    "modificationTime": 1572823237000,
    "dataChange": true,
    "stats": [...]},
  "add": {
    "path": "part-00001.parquet",
    "partitionValues": {},
    "size": 242520,
    "modificationTime": 1572823237000,
    "dataChange": true,
    "stats": [...]}
}
```

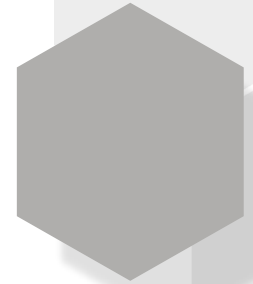


BUT WHAT ACTUALLY IS IT? - WITH DELTA:

DELETE * FROM SALES WHERE
Segment = 3



```
...
{
  "add": {
    "path": "part-00002.parquet",
    "partitionValues": {},
    "size": 255520,
    "modificationTime": 1572823237000,
    "dataChange": true,
    "stats": [...]}
  "remove": {
    "path": "part-00000.parquet",
    "modificationTime": 1572823237000,
    "dataChange": true}
  "remove": {
    "path": "part-00001.parquet",
    "modificationTime": 1572823237000,
    "dataChange": true}
}
```



SELECT * FROM SALES



000.JSON



001.JSON



part-00000.parquet



part-00001.parquet



part-00002.parquet





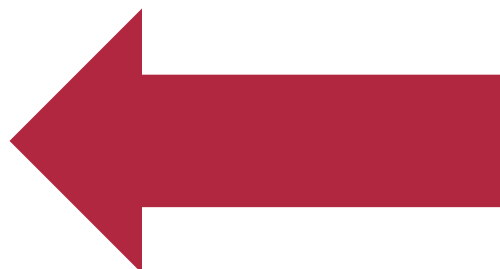
000.JSON



001.JSON



part-00000.parquet



part-00001.parquet



part-00002.parquet



SELECT * FROM SALES
TIMESTAMP AS OF
"2023-05-18T22:15:12.013Z"



www.advancinganalytics.co.uk

DELTA TABLE PERFORMANCE



@ADVANCINGANALYTICS



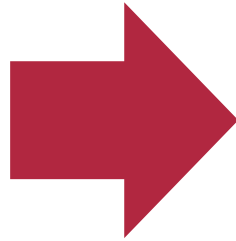
@ADVANALYTICSUK



/ADVANCING ANALYTICS

PERFORMANCE TUNING

Lake-Based Performance Tuning has two aspects:



1 – Read Less Data

2 – Optimize for Parallelism



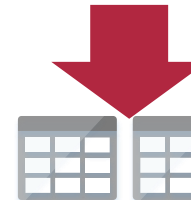
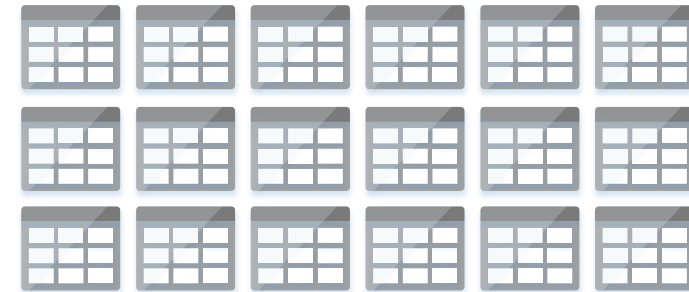
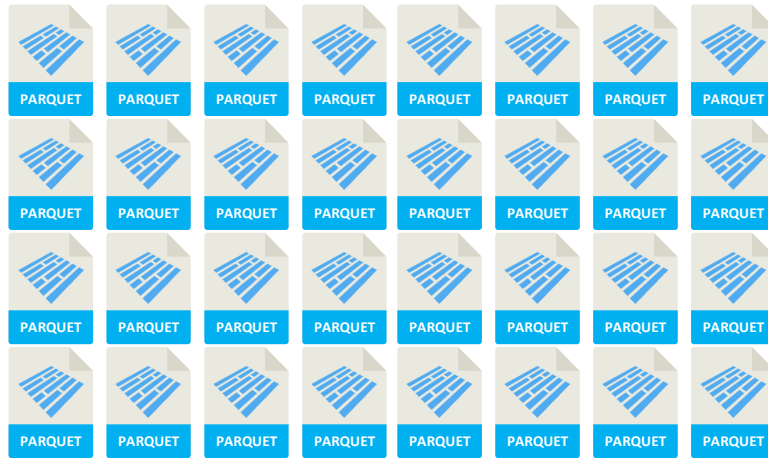
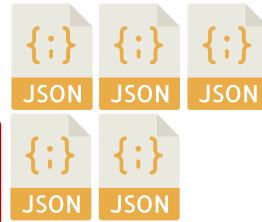


www.advancinganalytics.co.uk

STEP 1 - PARTITIONING

DELTA BY DEFAULT

```
SELECT * FROM Sales  
WHERE Year = 2023
```



In Memory
Filter

LAKE PARTITIONING SYNTAX

Spark automatically detects partitioning when folders are named:

`[COLUMNNAME]=[VALUE]`

For example, we might partition our Sales folder by Year, Month and Day:

`SALES / YEAR=2023 / MONTH=202301 / DAY=20230101`



PARTITIONING

We don't create Lake partitions manually – use Spark to write out partitioned Data!

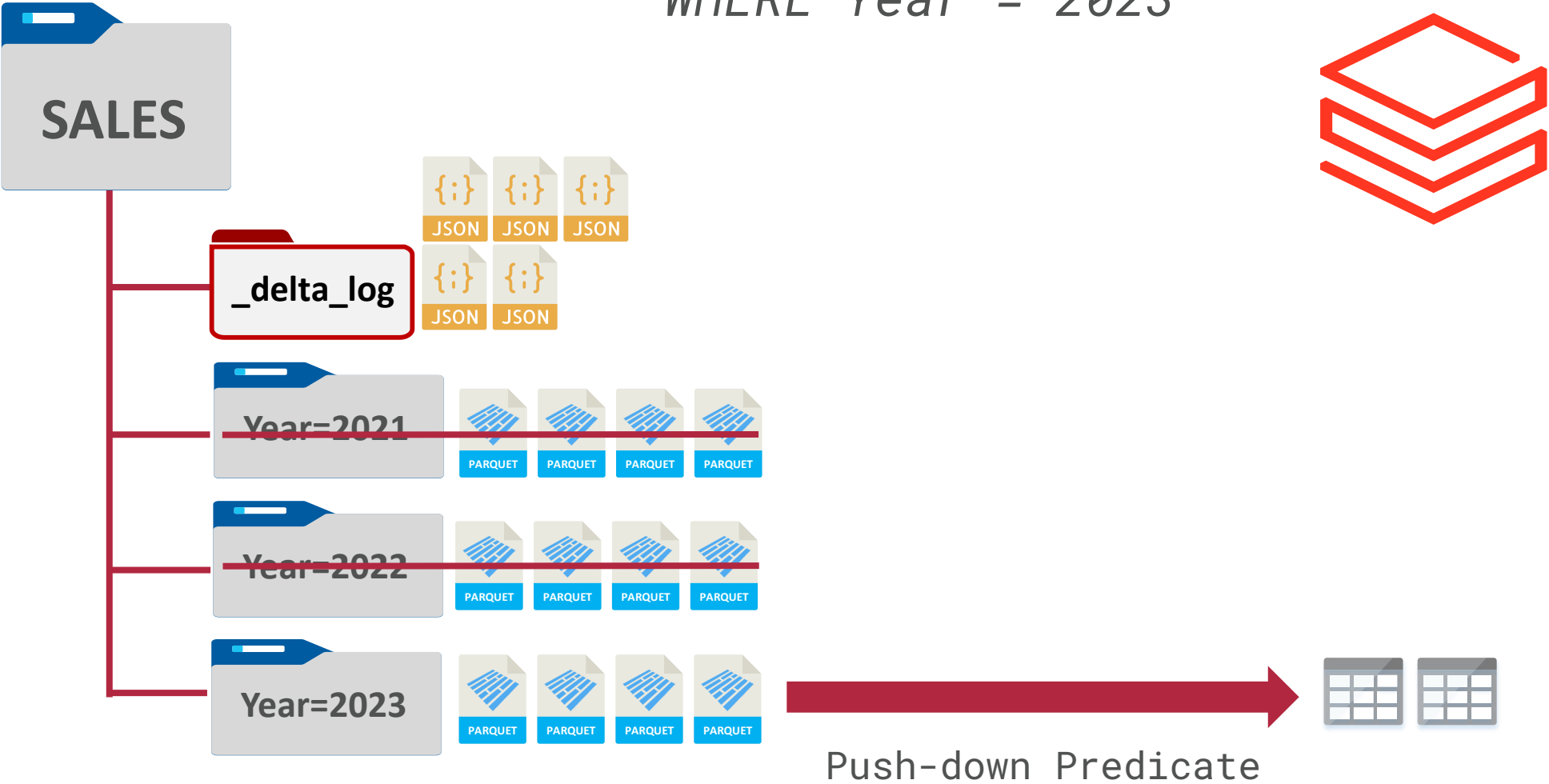
```
#Write DataFrame to Lake
(df
  .write
  .partitionBy("Year")
  .format("delta")
  .save("/mnt/lake/sales/"))
```

Spark will create sub-folders for each unique value of our partition columns



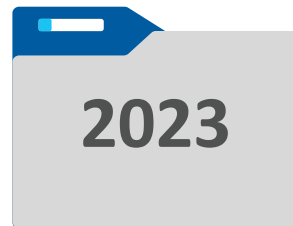
DELTA BY DEFAULT

```
SELECT * FROM Sales
WHERE Year = 2023
```





DELTA PERFORMANCE



Partitioning



STEP 2 – FILE SKIPPING





DATA SKIPPING

Delta Tables collect statistics within the transaction log, this includes a “min/max” value for the **first 32 columns** in the table.

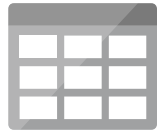
Using a technique called Data Skipping, Spark will selectively ignore files if it knows they do not contain data it needs.

File	Min Values	Max Values
PART001.parquet	{Year:2023, CustomerId:3, Product:42}	{Year:2023, CustomerId:22, Product:198}
PART002.parquet	{Year:2023, CustomerId:45, Product:1}	{Year:2023, CustomerId:87, Product:57}
PART003.parquet	{Year:2023, CustomerId:27, Product:123}	{Year:2023, CustomerId:124, Product:264}

```
SELECT * FROM Sales WHERE CustomerId = 28
```



SKIPPING STATISTICS



First 32 Columns

Col1

Col2

Col3

Col4

Col5

Col6



Col33

Col34

If the column you want to skip on isn't in the first 32 columns in the table, you need to:

1. Rearrange your table to prioritise the first 32 columns. or
2. Change the config to store more statistics

```
#Set number of indexed columns before writing  
spark.conf.set("spark.databricks.delta.dataSkippingNumIndexedCols", 64)
```

Capturing more statistics, especially on large strings & binary columns, will slow down your write action!



DELTA PERFORMANCE





STEP 3 – BETTER COMPRESSION





Parquet

Metadata – Schema, Structure, Dictionaries

A	1	Fred
A	1	Bob
A	1	Fred
B	1	Bob
C	1	Fred
C	1	Bob
C	1	Fred
B	1	Bob

CSV File



Col1	Col2	Col3
A*3	1*8	1
B		2
C*3		1
B		2
		1
		2
		1
		2

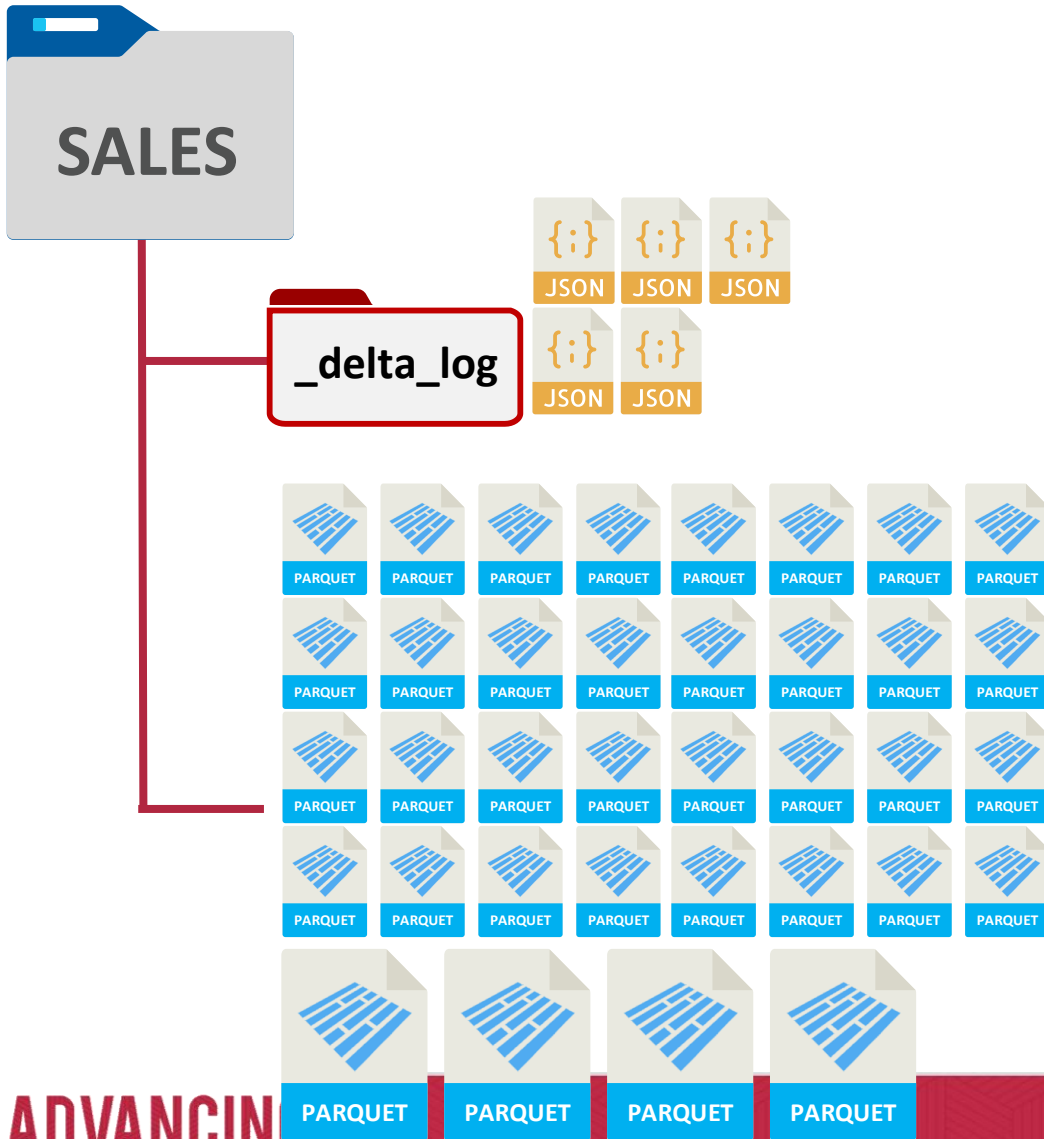
Parquet File

Dictionary

1 = Fred
2 = Bob

This compression technique is called **Run Length Encoding** and is one of the reasons why columnstores are so good for storing analytical data!

SMALL/INCREMENTAL UPDATES



Small Files don't compress well – we therefore see poor performing read speeds over many small files

The **Optimize** command compacts small files into larger, better compressed files
This is treated like all other updates, files are NOT deleted

OPTIMIZE

The Optimize command can be thought of as a maintenance process similar to defragmenting an index

```
--Optimize an entire table in SQL  
OPTIMIZE [database].[table]
```

It should be run regularly – how regularly depends on how often your data is changing, and the performance impact on end users!



Z-ORDERING

Whilst it sounds complicated, we can summarize Z-ordering as:

"Sort the data on specific columns before writing to files, to optimize data skipping"

```
--Optimize an entire table  
OPTIMIZE [database].[table] ZORDER BY [ColumnName]
```



Z-ORDERING EXPLAINED

"select count() from Employees
where Name = 'Brad'"*

A	1	Bob
A	2	Fred



A	1	Andy
A	2	Tom



A	1	Brad
A	2	Tim



A	1	Dan
A	2	Jan



In this example, we have
not ordered our small files,
so data skipping only hits
occasional lucky successes



Z-ORDERING EXPLAINED

A	1	Bob
A	2	Fred

A	1	Andy
A	2	Tom

A	1	Brad
A	2	Tim

A	2	Dan
A	1	Jan



ZOrder
by
Name

A	1	Andy
A	1	Bob
A	1	Brad
A	2	Dan

A	2	Fred
A	1	Jan
A	2	Tim
A	2	Tom



"select count() from
Employees where
Name = 'Brad'"*



HILBERT SPACE-FILLING Z-CURVE

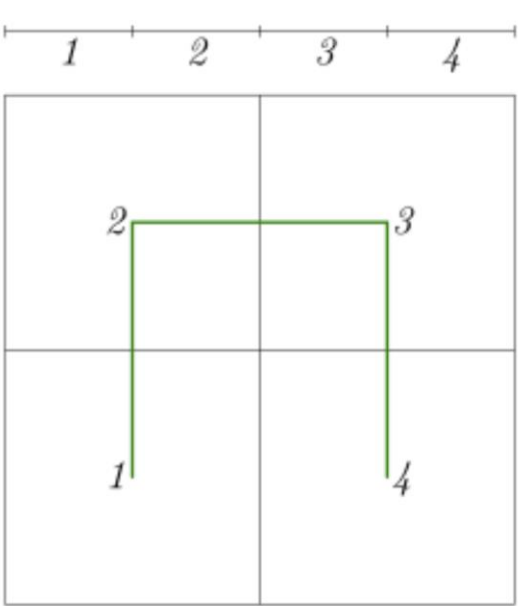


Fig. 1.

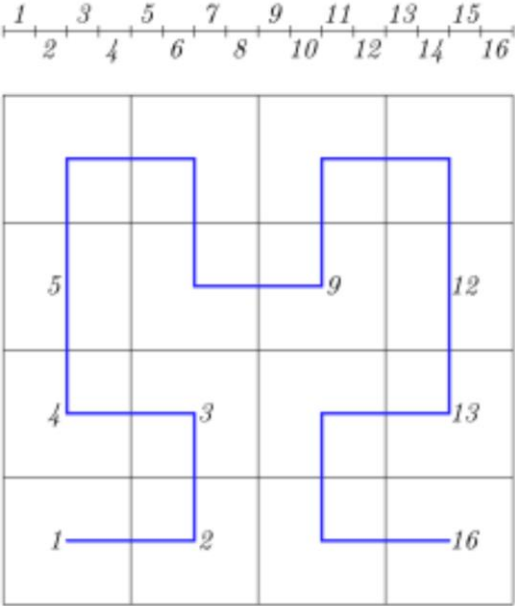


Fig. 2.

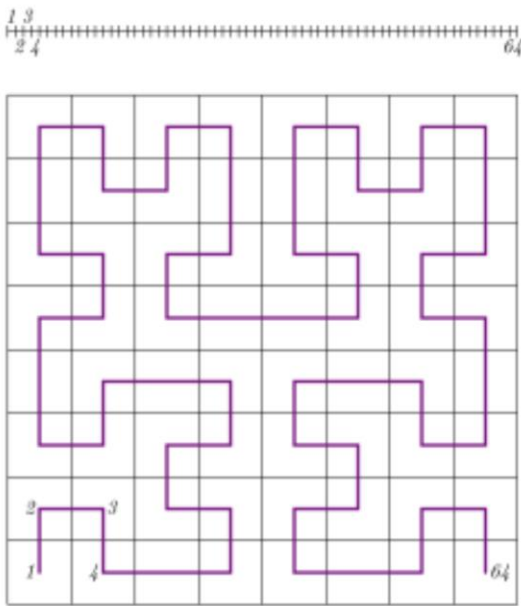
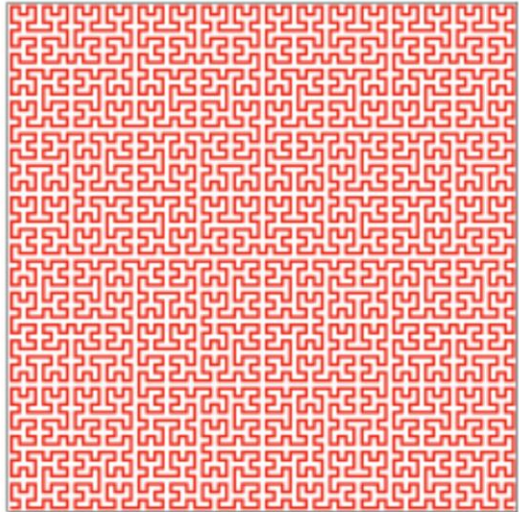
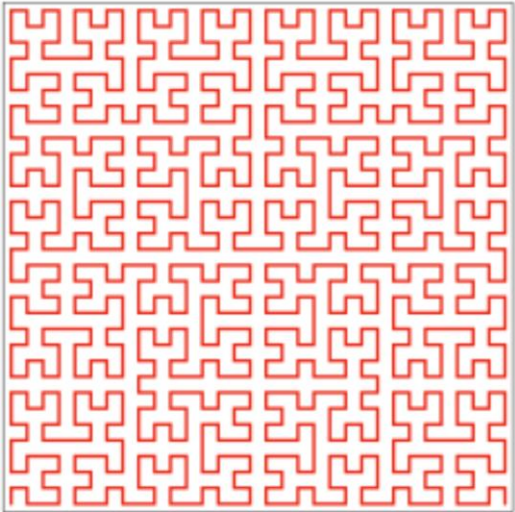
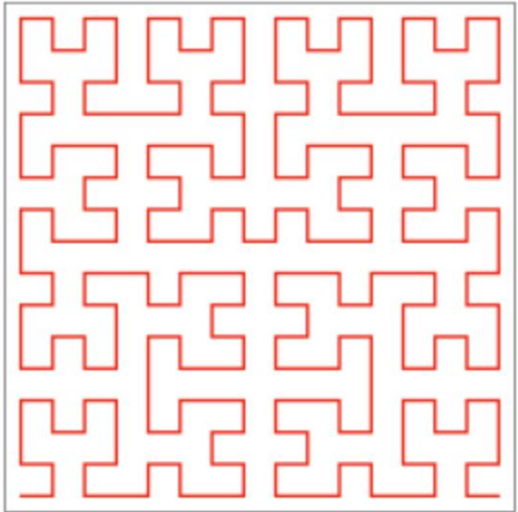


Fig. 3.



Z-ORDERED DATA EXAMPLE

The nature of space-filling curves means you might be surprised by how the data is distributed



File	Min Values	Max Values
PART001.parquet	{Year:2023, CustomerId:0, Product:0}	{Year:2023, CustomerId:50, Product:50}
PART002.parquet	{Year:2023, CustomerId:0, Product:51}	{Year:2023, CustomerId:50, Product:100}
PART003.parquet	{Year:2023, CustomerId:51, Product:0}	{Year:2023, CustomerId:100, Product:50}
PART003.parquet	{Year:2023, CustomerId:51, Product:51}	{Year:2023, CustomerId:100, Product:100}



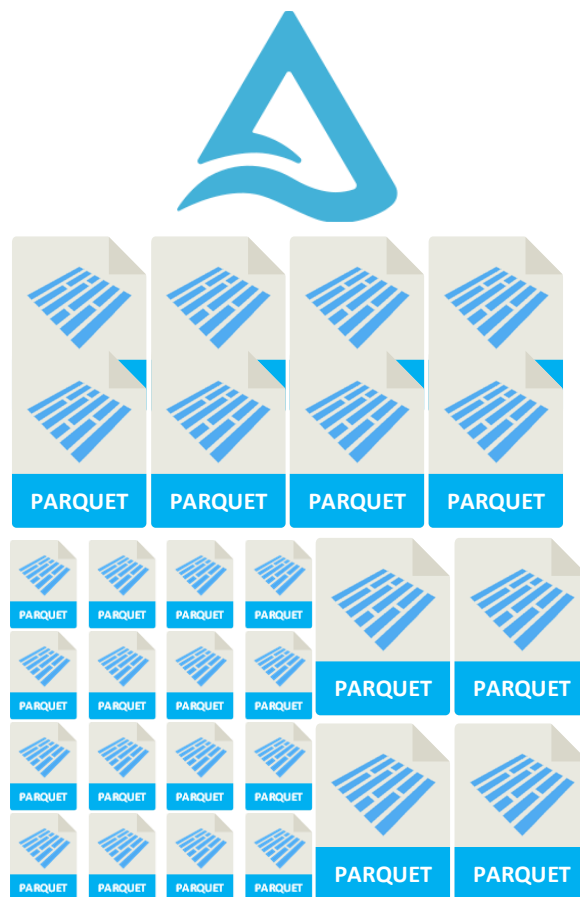
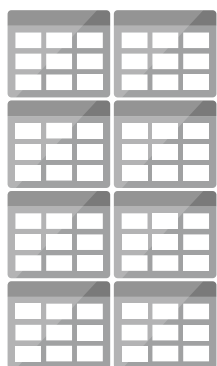
AUTO-OPTIMIZE

We have a choice - you can optimize deliberately and specifically at certain times, or we can do a little bit of optimization every time we write using ***Auto-Optimize***

```
--Enable AutoOptimize  
set spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true;  
set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true;
```

Optimized Writes - Change the query plan to attempt to write out files of at least 128Mb (this is less than the 1Gb default that Optimize uses!)

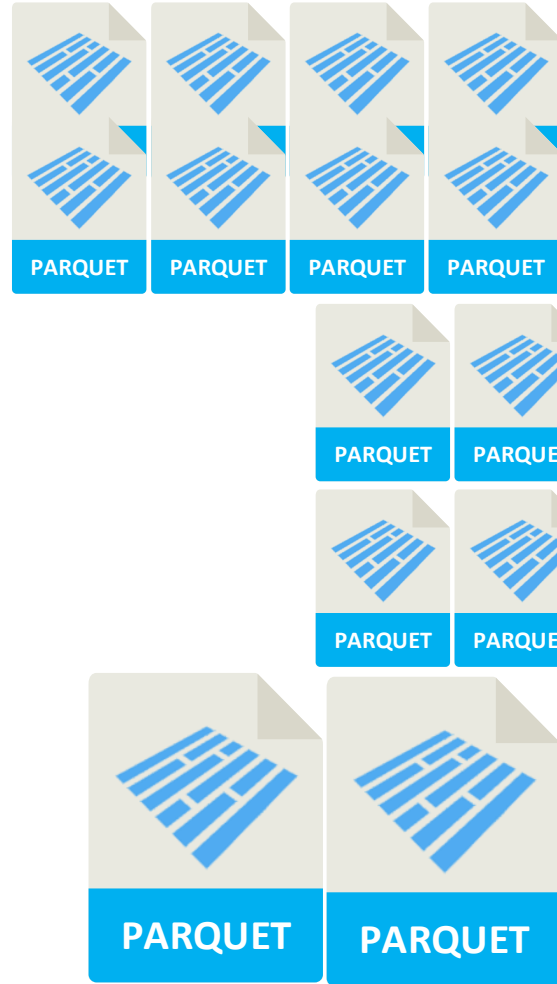
Auto Compaction - Run a lightweight optimize job after the write has finished, looking for further file compaction opportunities, again with 128mb



Optimised Writes

Auto Compaction



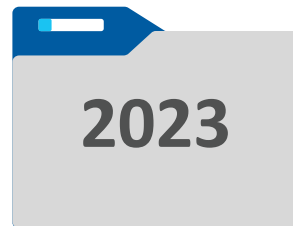


Optimize & Z-Order





DELTA PERFORMANCE



Partitioning



File
Skipping

+	0	1	4	5	16	17	20	21
0	0	1	4	5	16	17	20	21
2	2	3	6	7	18	19	22	23
8	8	9	12	13	24	25	28	29
10	10	11	14	15	26	27	30	31
32	32	33	36	37	48	49	52	53
34	34	35	38	39	50	51	54	55
40	40	41	44	45	56	57	60	61
42	42	43	46	47	58	59	62	63

+	0	1	4	5	16	17	20	21
0	0	1	4	5	16	17	20	21
2	2	3	6	7	18	19	22	23
8	8	9	12	13	24	25	28	29
10	10	11	14	15	26	27	30	31
32	32	33	36	37	48	49	52	53
34	34	35	38	39	50	51	54	55
40	40	41	44	45	56	57	60	61
42	42	43	46	47	58	59	62	63

+	0	1	4	5	16	17	20	21
0	0	1	4	5	16	17	20	21
2	2	3	6	7	18	19	22	23
8	8	9	12	13	24	25	28	29
10	10	11	14	15	26	27	30	31
32	32	33	36	37	48	49	52	53
34	34	35	38	39	50	51	54	55
40	40	41	44	45	56	57	60	61
42	42	43	46	47	58	59	62	63

Z-Ordering



STEP 4 – BLOOMS & V-ORDERS



BLOOM FILTER INDEXES

Bloom filter indexes are an additional metadata structure that indicates whether each file of the delta table ***might contain*** a given value

```
--Enable Bloom Filter Indexes
SET spark.databricks.io.skipping.bloomFilter.enabled = true;

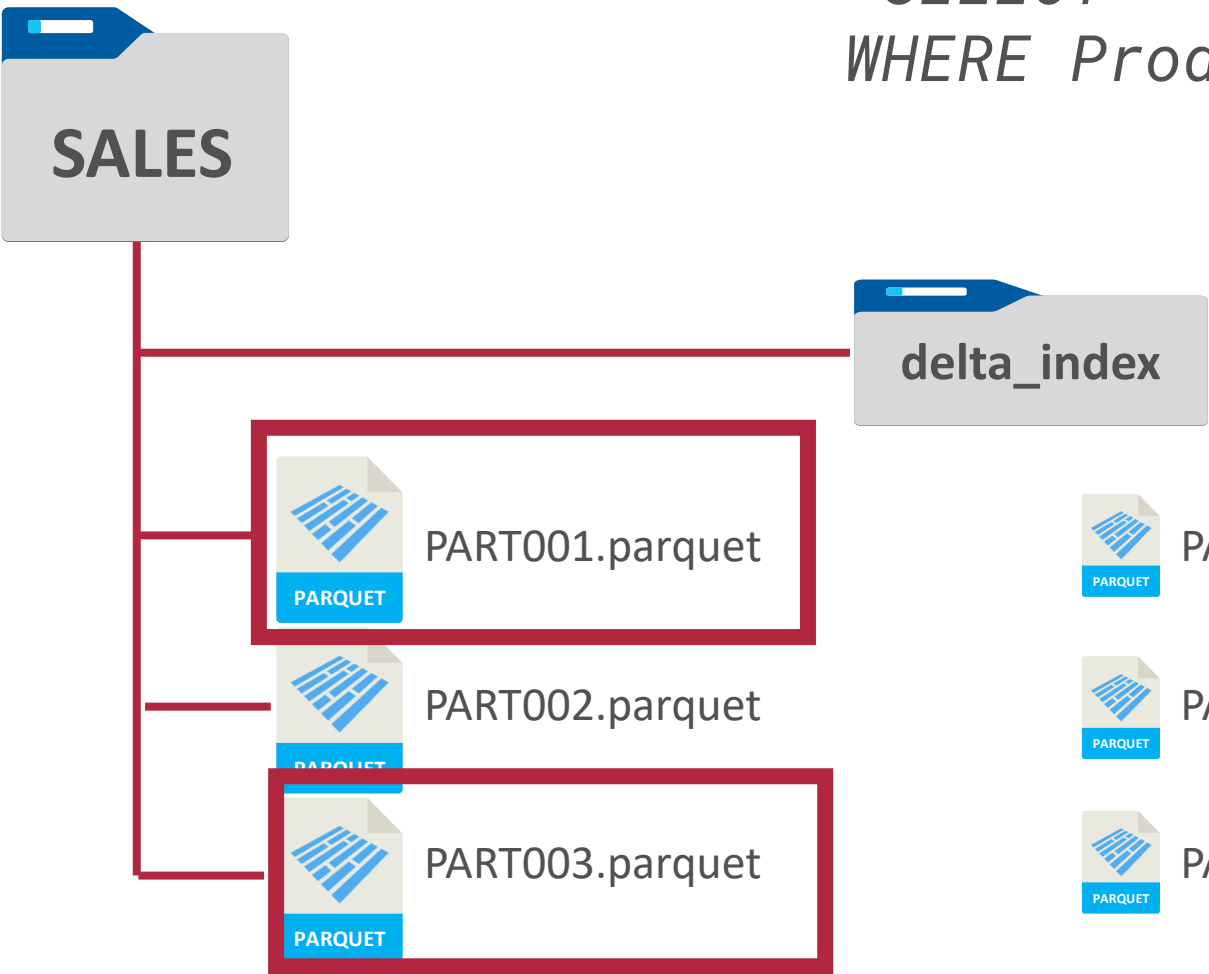
CREATE BLOOMFILTER INDEX
ON TABLE Sales
FOR COLUMNS(ProductID OPTIONS (fpp=0.1, numItems=10000))
```







fpp – the False Probability Percentage, the lower the fpp, the more confident the index is
numItems – max number of items the index should cover



BLOOM FILTER INDEX

```
SELECT * FROM Sales
WHERE ProductID = 173
```



-  PART001.parquet.index.parquet 
-  PART002.parquet.index.parquet 
-  PART003.parquet.index.parquet 





V-ORDERING IN FABRIC

A	B	C
SF*200	AG	1433
SE*134	MZ	232
LO*22	MA	11223
SF*123	DL	4622
SE*112	SW	2573
LO*3	TW	7382

Parquet File

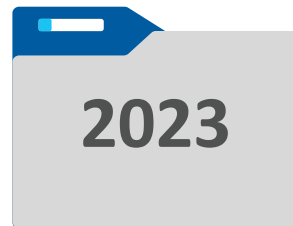
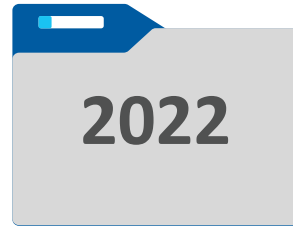


A	B	C
SF*323	AG	1433
SE*246	MZ	232
LO*25	MA	11223
	DL	4622
	SW	2573
	TW	7382

V-Ordered Parquet File



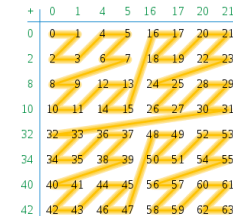
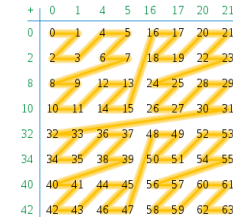
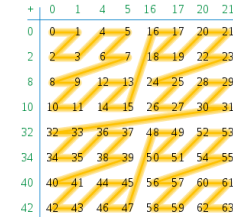
DELTA PERFORMANCE



Partitioning



File
Skipping

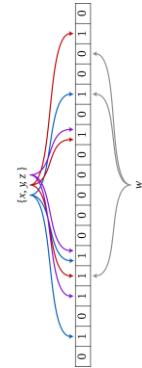


Z-Ordering



V-Ordering

	A	B	C
SF*323	AG	1433	
SE*246	MZ	232	
LO*25	MA	11223	
	DL	4622	
	SW	2573	
	TW	7382	



Bloom Index



www.advancinganalytics.co.uk

DATABRICKS PHOTON



@ADVANCINGANALYTICS



@ADVANALYTICSUK



/ADVANCING ANALYTICS

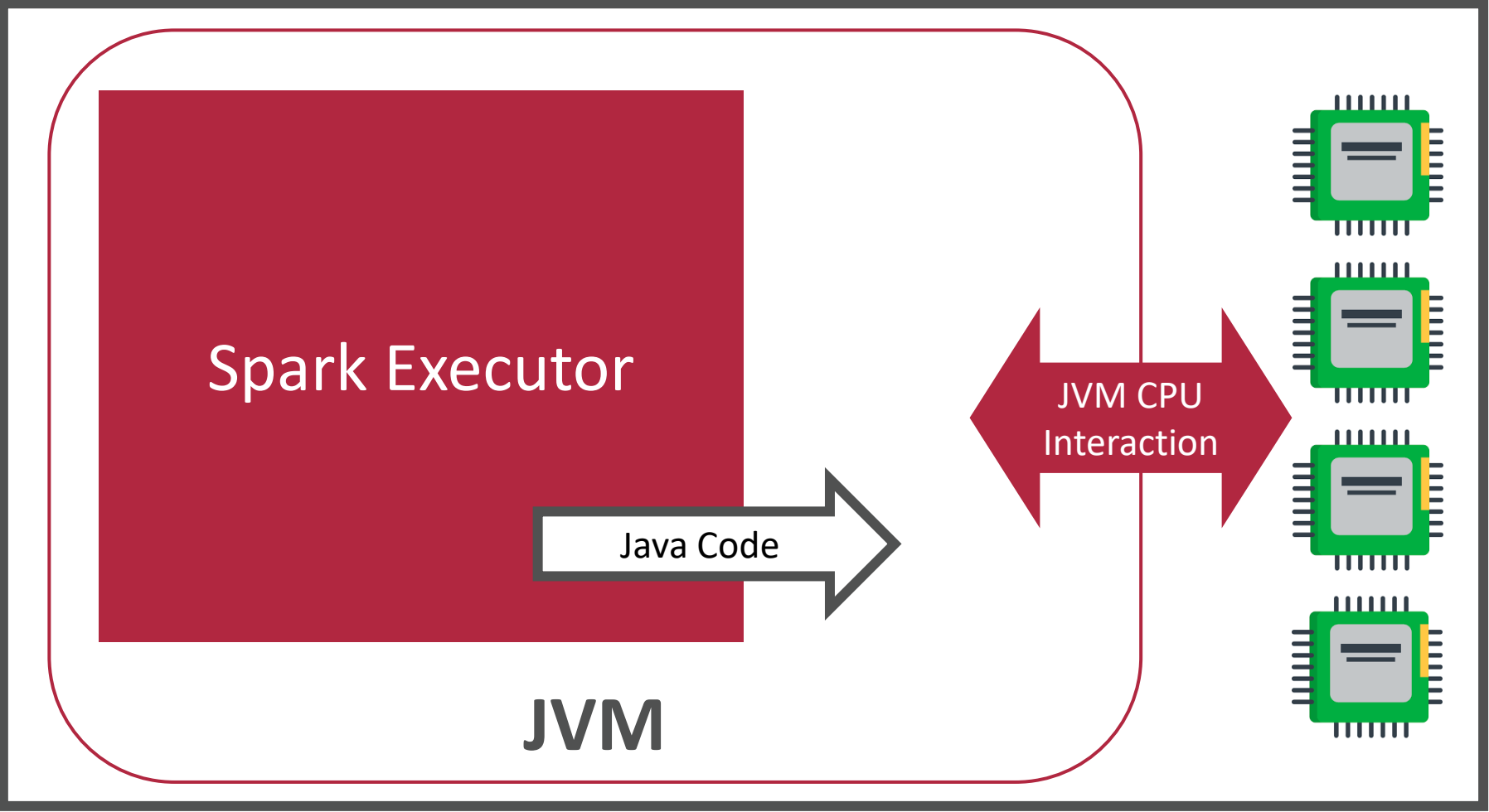
DATABRICKS PHOTON

Native, vectorised execution engine,
sitting on top of Databricks Spark.

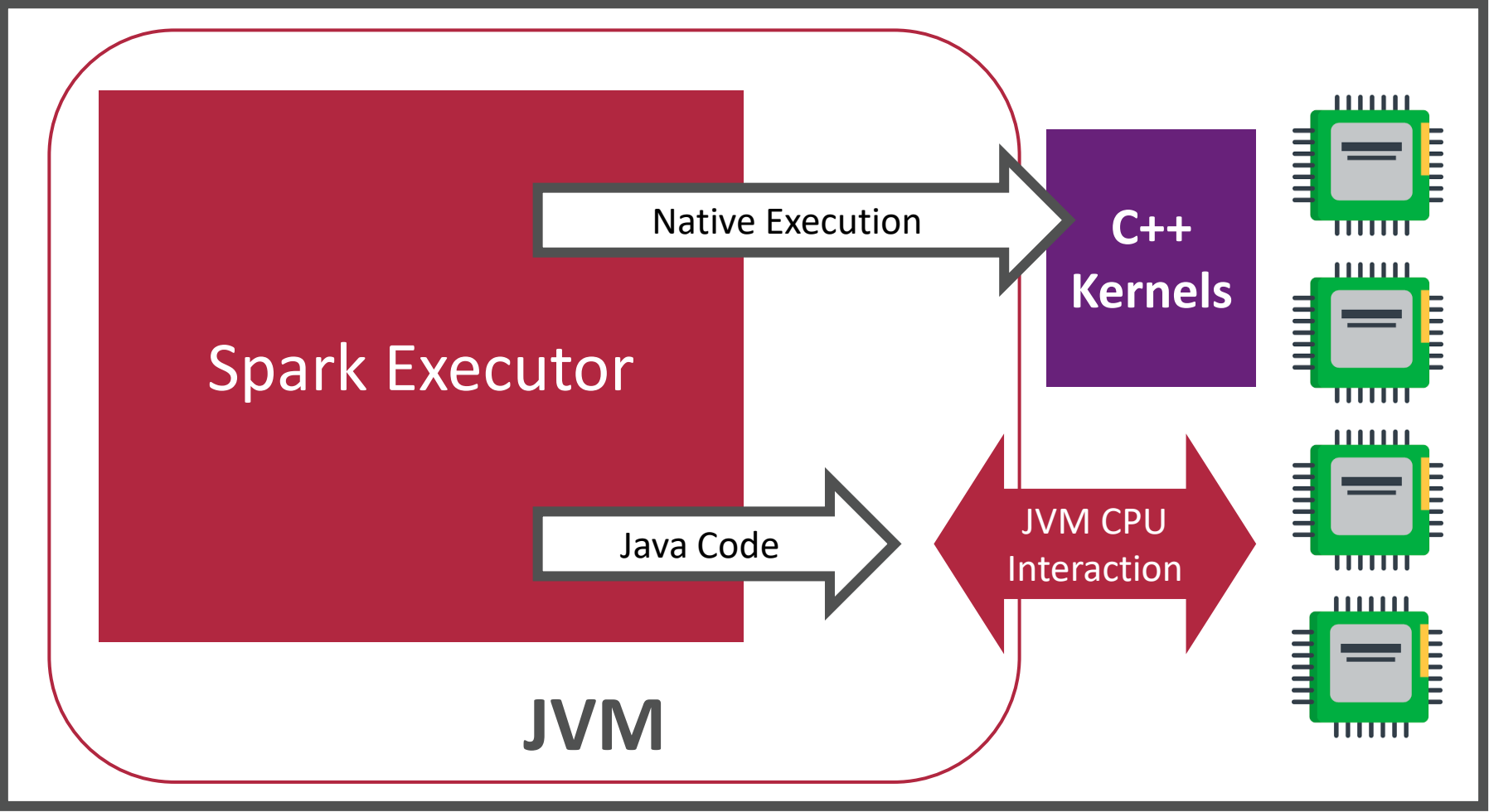
Uses C++ rather than Java, getting
much faster execution over the
traditional spark engine



CLASSIC SPARK



PHOTON EXECUTION



DATABRICKS PHOTON

Photon has to be specifically enabled on the cluster. Clusters with photon enabled with charge more DBUs than traditional spark clusters!



Databricks runtime version ?

Runtime: 13.1 (Scala 2.12, Spark 3.4.0)



Use Photon Acceleration ?

THANKS FOR LISTENING



Twitter: @MrSiWhiteley

youtube.com/c/AdvancingAnalytics

AdvancingAnalytics.co.uk



@ADVANCINGANALYTICS



@ADVANALYTICSUK



/ADVANCING ANALYTICS