APACHE Spark™

LOAD BIG DATA EFFICIENTLY

PART 9: FASTER DATA LOADS IN A NUTSHELL

DATA WITH NIKK THE GREEK

The File Format Matters

File format matters

Unstructured

text, e.g. txt

videos, e.g. mp4

sound, e.g. mp3

pictures, e.g. png

Unorganised and unformatted information and data

Semi-structured

{json}

CSV

xml

Contains tags or markers that create some structure or semantic but not fully enforced

Structured

Parquet

AVRO

Enforce schema and data type rules, tabular format

File format matters

- CSV and JSON are both human readable, row level formats allowing fast writes and easy to open and process
- Compression is possible but not as fast as Avro and Parquet. It also limits the splitability
- CSV is further easy coruptable
- Avro is very efficient to save big amounts of data fast
- Parquet is very efficient in loading data including predicate pushdown but also writing data
- Compressable
- Avro and Parquet are not human readable and support is more limited especially for Avro

Size, write and loading times

10 Million row dataset with 8 files has been used

	Size	Monthly Costs
JSON	100 000 GB	2.000 €
AVRO	6 000 GB	120 €

Format	Size	Write time	Load time
JSON	1208 MB	3 s	6 s
CSV with infer schema	593 MB	3 s	12 s
PARQUET	81.5 MB	2 s	0,6 s
AVRO	69.2 MB	1 s	0,9 s

JSON vs. AVRO

- 94 % size
- 66 % write time
- 85 % load time

CSV vs. PARQUET

- 86 % size
- 33 % write time
- 95 % load time

Smaller size and faster reads and writes save you compute and storage costs

Detailed loading times

Format	Load time	of which meta data	of which actual load
JSON	6 s	3 s	3 s
CSV with infer schema	12 s	25 ms + 6 s	6 s
PARQUET	0.6 s	30 ms	0.6 s
AVRO	0.9 s	0 s	0.9 s

Small file Problem

Small file problem

10 Million row dataset with
Large: 8 files and Small: 100.000 files has been used

Format	Meta Time (S)	Loading Time (S)	Total (S)	Meta Time (L)	Loading Time (L)	Total (L)
JSON	21 s + 6,8 min	18 s	**7,7 min**	3 s	3 s	**6 s**
CSV	19 s + 86 ms + 6,9 min	30 s	**7,7 min**	25 ms + 6s	6 s	**12 s**
PARQUET	0,3 s	5,8 min	**5,8 min**	30 ms	0,6 s	**0,6 s**
AVRO	-	5,1 min	**5,1min**	-	0,9 s	**0,9 s**

Filter and write

col1	col2
1	A
2	B
3	A

col1	col2
4	B
5	B
6	B

col1	col2
7	A
8	A
9	B

col1	col2
10	B
11	A
12	A

Driver

- Assigns data to partitions
- Delegates the work to executors, meaning partitions to tasks
- Create execution plans
- Saves meta data about the files

Worker 1

Core1 Core2

Worker 2

Core1 Core2

Reference: https://youtu.be/kCydZHkqXc0

Problem Cause of Small files

- File meta data stored in driver memory
- Efforts querying data including open file, closing file and checking the storage files and directories
- Scheduling overhead for delegating the partitions to tasks
- Reduced parallalism due to more created partitions
- Increased CPU costs due to serialisation and deserialisation

Reference: Small File, Large Impact — Addressing the Small File Issue in Spark | by Siraj | Medium

Solving the Problem

- Increase the file size. Either by checking the source options or having an intermediate process saving files in bigger files. I like Delta and you the bin packing (optimize) option
- Use file formats like Parquet or Avro with less meta data to be stored
- Reduce meta data by defining the schema
- Reduce number files per partition and thus increase number of partitions e.g. by using maxPartitionBytes or openCostInBytes

The force of the Schema

The force of the schema

```
ddl_schema = "id bigint, date date, timestamp timestamp, idstring string, idfirst
string, idlast string"

spark_schema = t.StructType(
    [
        t.StructField('id', t.LongType(), True),
        t.StructField('date', t.DateType(), True),
        t.StructField('timestamp', t.TimestampType(), True),
        t.StructField('idstring', t.StringType(), True),
        t.StructField('idfirst', t.StringType(), True),
        t.StructField('idlast', t.StringType(), True)
    ]
)

sdf_parquet = spark.read.format("parquet").schema(ddl_schema).load(path_parquet)
```

Results Absolute

Format	Experiment	Meta Time (S)	Loading Time (S)	Total (S)	Meta Time (L)	Loading Time (L)	Total (L)
JSON	w/o schema	21 s + 6,8 min	18 s	**7,7 min**	3 s	3 s	**6 s**
JSON	w schema	-	18 s	**18 s**	-	3 s	**3 s**
CSV	w/o schema	19 s + 86 ms + 6,9 min	30 s	**7,7 min**	25 ms + 6s	6 s	**12 s**
CSV	w schema	-	32 s	**32 s**	-	6 s	**6 s**
PARQUET	w/o schema	0,3 s	5,8 min	**5,8 min**	30 ms	0,6 s	**0,6 s**
PARQUET	w schema	-	33 s	**33 s**	-	0,4 s	**0,4 s**
AVRO	w/o schema	-	5,1 min	**5,1min**	-	0,9 s	**0,9 s**
AVRO	w schema	-	16 s	**16 s**	-	1 s	**1 s**

Results % to schema

Format	w/o vs with schema small	w/o vs with schema large	Small vs large w/o schema	Small vs large with schema
JSON	- 96 %	- 50 %	- 99 %	- 83 %
CSV	- 93 %	- 50 %	- 97 %	- 81 %
PARQUET	- 91 %	- 33 %	- 99,99 %	- 99 %
AVRO	- 95 %	- 0 %	- 99,99 %	- 94 %

Smaller files with Open Cost Per Bytes

Open Cost Per Bytes to deal with small files

Open Cost Per Bytes

- Represents the cost of creating a new partition
- based on the config "spark.sql.files.openCostInBytes"
- defaults to 4 MB
- Technically it adds the cost, e.g. 4 MB, to each file which is called padding
- Official description: The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over-estimate, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.

Reference: https://youtu.be/vkOxEHEKYhA

Results of open Cost In Bytes

OpenCost MB	JSON	AVRO
1 MB	5,8 min	5,5 min
2 MB	21 s	40 s
4 MB	21 s	41 s
6 MB	21 s	42 s
8 MB	22 s	2,1 min
10 MB	1,8 min	3,7 min
w/o schema 4 MB	7,7 min	5,1 min

Reference: https://youtu.be/C-ykod0Yyf4

Predicate and Aggregate Pushdown

Predicate Pushdown

Name	Female	Age
Steven	False	33
Peter	False	47
Ana	True	21
Laura	True	37

Goal: Get all data WHERE Age > 35

```
sdf.filter(f.col("Age") > 35)
```

Predicate Pushdown

Source

All data loaded

Name	Female	Age
Steven	False	33
Peter	False	47
Ana	True	21
Laura	True	37

Filter

Name	Female	Age
Peter	False	47
Laura	True	37

Predicate Pushdown

Name	Female	Age
Peter	False	47
Laura	True	37

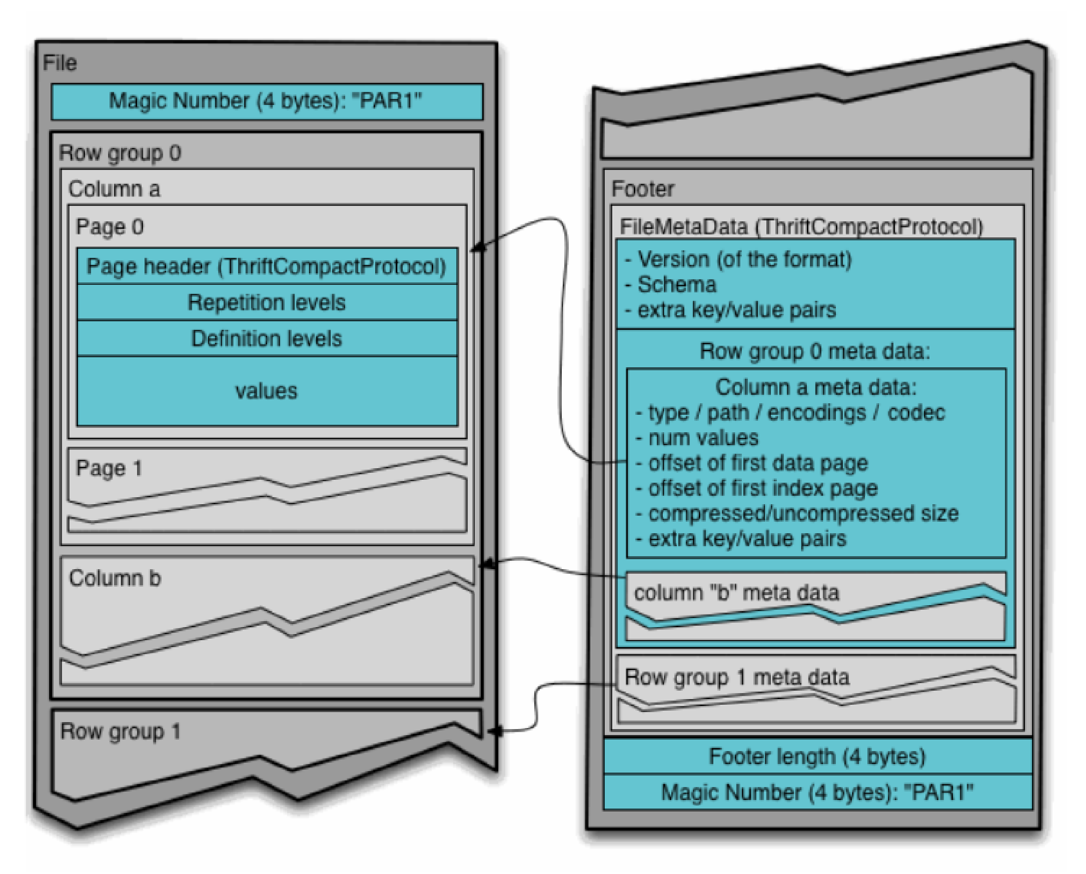
Further transformations

Predicate Pushdown

- **Predicate Pushdown is an optimization technique filtering data at the source and often relies on statistics**
- **Benefits:**
  - Less I/O meaning less data to load
  - Less memory usage
  - Faster queries
- **Parquet supports Predicate Pushdown using statistics saved in meta data footer**
- **Since Spark 3.1.0 also possible on Avro, CSV, JSON**

Parquet

- Row Groups are a logical division on row level of a parquet defaulting to 128 MB
- Column part relates to column chunk of row groups
- Pages are invisible units where the encoding and compression happens
- Footer containing file metadata which can be used for predicate pushdown:
  - File level: num rows/ columns, schema
  - Row group: num rows/ columns
  - Column level: min, max, null count, distinct value counts, page indexes etc.



Parquet format: Documentation | Parquet (apache.org)
Parquet Page Indexes: Speeding Up SELECT Queries with Parquet Page Indexes - Cloudera Blog

Load time and output rows for column/ row filters

Format	Load all data	Column filter	Row filter
JSON	16 s (10,000,000 rows)	5 s (10,000,000 rows)	4 s (300 rows)
CSV	13 s (10,000,000 rows))	4 s (10,000,000 rows)	4 s (308 rows)
PARQUET	2 s (10,000,000 rows)	0.6 s (10,000,000 rows)	0.1 s (20,000 rows)
AVRO	3 s (10,000,000 rows)	2 s (10,000,000 rows)	1 s (300 rows)

Aggregate Pushdown

- Filter and select push downs work on all data sources
- Aggregate Pushdowns work not on JSON, CSV, AVRO
- Activate aggregate pushdown for Parquet as follows:
  - `spark.conf.set("spark.sql.sources.useV1SourceList", "")`
  - `spark.conf.set("spark.sql.parquet.aggregatePushdown", "true")`
- Aggregate Pushdown has the following limitations:
  - No nested columns and string columns supported for min/max
  - Filter and aggregates are only for partitioned columns supported
- Aggregate Pushdown speeds up the performance significantly of counts, min and max
- V2 Source API seems unclear if more efficient than V1 but SQL interface seems different and Batch Scan is always on.

Better Partitions when loading data

Spark Architecture

col1	col2
1	A
2	B
3	A

col1	col2
4	B
5	B
6	B

col1	col2
7	A
8	A
9	B

col1	col2
10	B
11	A
12	A

Driver

- Assigns files to partitions
- Delegates the partitions as tasks to the worker
- Each core executes one task at the same time

Worker 1

Core1 Core2

Worker 2

Core1 Core2

Reference Architecture Basics: https://youtu.be/kCydZHkqXc0

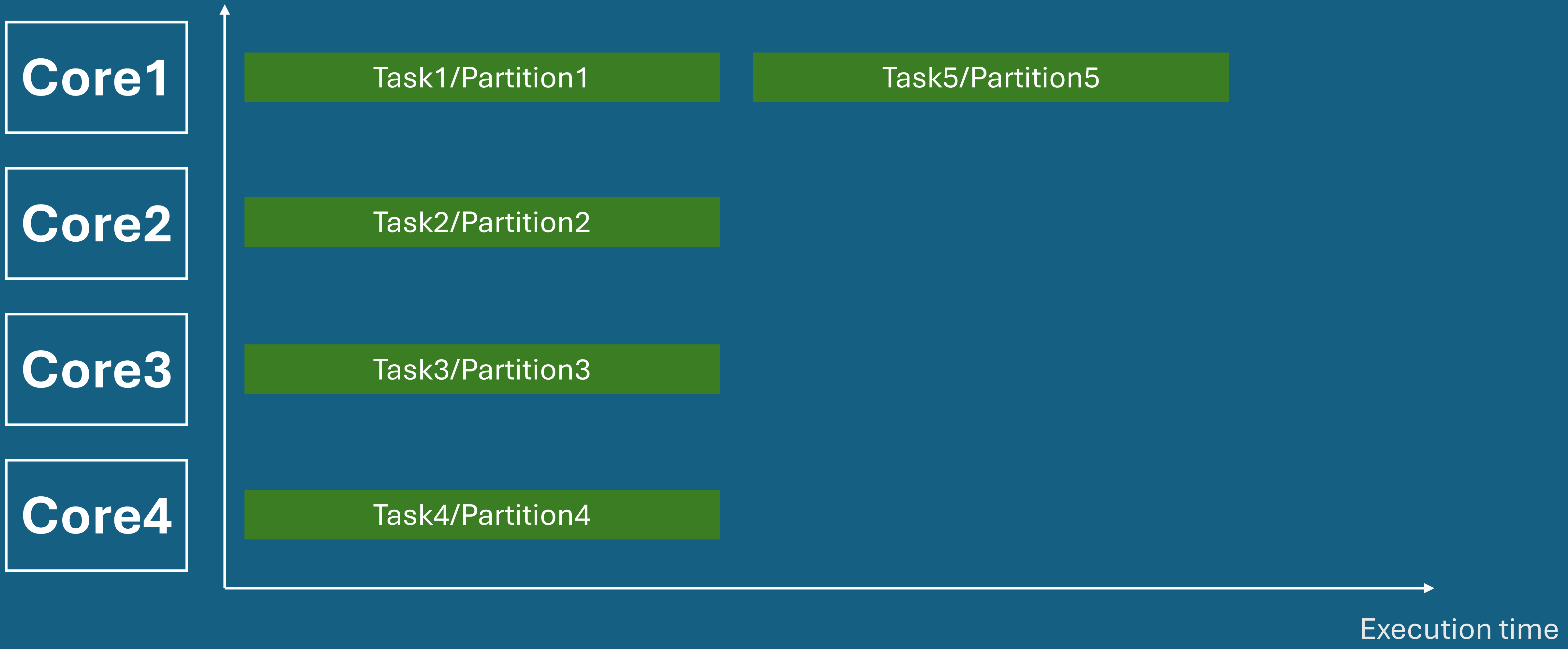
What determines the number of partitions?

- Num of cores:
  - Spark tries to create at least the number of partitions equal to your number of cores
  - Can be changed with conf `spark.sql.files.minPartitionNum`
- Num of parquet files and its row groups: Parquet is only splitable on Row group level for partitioning
- Max Partition size:
  - Default 128 MB as the default row group size
  - Can be changed with conf `spark.sql.files.maxPartitionBytes`
- Max Cost per Bytes:
  - Represents the cost of creating a new partition, defaulting to 4 MB
  - Can be changed with conf `spark.sql.files.openCostInBytes`

Reference Influence on Partitions: https://youtu.be/vkOxEHEKYhA

Perfect distributions of Partitions

Bad example

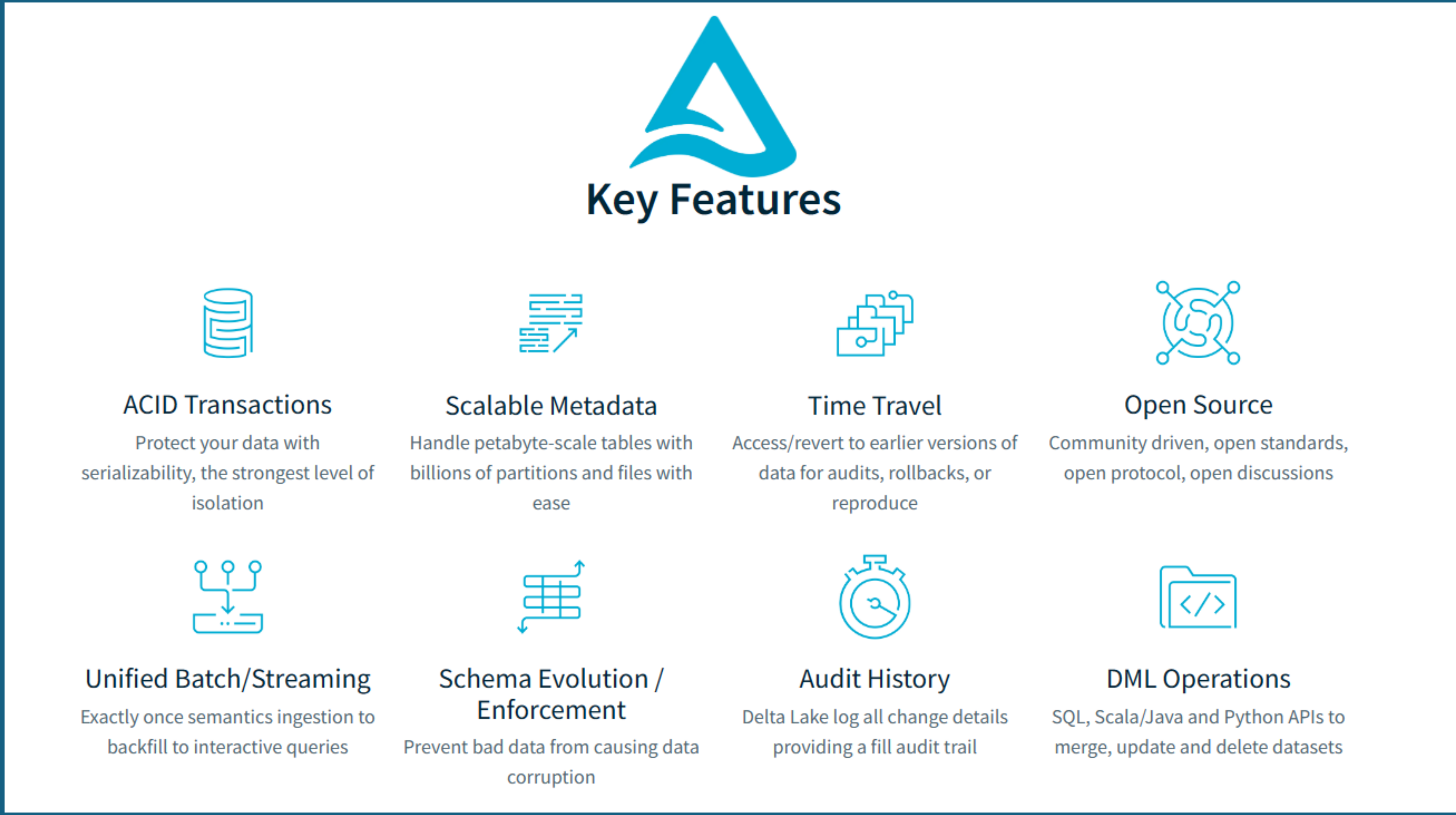

Core1
Core2
Core3
Core4
Task1/Partition1
Task5/Partition5
Task2/Partition2
Task3/Partition3
Task4/Partition4
Execution time

Basic rules of good partitions

- Good parallelisation:
  - Factor 2-4 of your number of cores (exceptions for smaller files)
  - Uniform datasets generate also uniform partitions
- Partition size:
  - To big partitions can lead to out of memory issues
  - Max partition size is at 128 MB, 100 MB to 1 GB is recommended
  - It depends of course on your machine and your other operations
- Distribution overhead:
  - A high number of partitions can create a distribution overhead
  - Execution time should make 90 % of the whole execution time
  - Exception: Small file problem where the distribution overhead is ok

Tip to check no. partitions: `sdf.rdd.getNumPartitions()`

Use Delta Lake



[Home | Delta Lake]

Summary

- Use Big data formats like Parquet and Avro
- Define the schema whenever possible
- Avoid small files if possible. If not reduce the overhead using the schema and conf `spark.sql.files.openCostInBytes`
- Apply column filter and row filter as close to the source as possible to use predicate pushdown
- Optionally the same for aggregate pushdown
- Use conf `spark.sql.files.maxPartitionBytes` to create a well distributed partitions