Parquet in Practice & Detail

What is Parquet? How is it so efficient? Why should I actually use it?

blueyonder

About me

- Data Scientist at Blue Yonder (@BlueYonderTech)
- Committer to Apache {Arrow, Parquet}
- Work in Python, Cython, C++11 and SQL

@ xhochy

uwe@apache.org


blue yonder

Agenda

- Origin and Use Case
- Parquet under the bonnet
- Python & C++
- The Community and its neighbours

About Parquet

1. Columnar on-disk storage format
2. Started in fall 2012 by Cloudera & Twitter
3. July 2013: 1.0 release
4. top-level Apache project
5. Fall 2016: Python & C++ support
6. State of the art format in the Hadoop ecosystem
   - often used as the default I/O option

Why use Parquet?

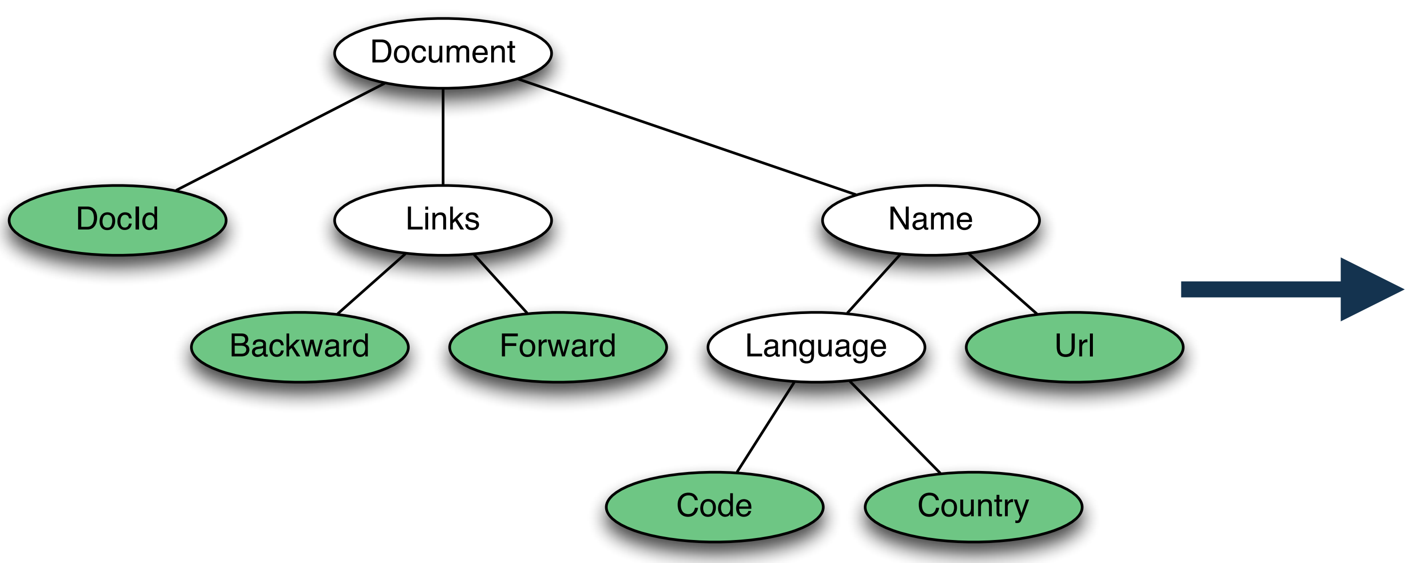
1. Columnar format
   —> vectorized operations
2. Efficient encodings and compressions
   —> small size without the need for a fat CPU
3. Query push-down
   —> bring computation to the I/O layer
4. Language independent format
   —> libs in Java / Scala / C++ / Python /…

Who uses Parquet?

- Query Engines
  - Hive
  - Impala
  - Drill
  - Presto
  - …
- Frameworks
  - Spark
  - MapReduce
  - …
  - **Pandas**

Nested data

- More than a flat table!
- Structure borrowed from Dremel paper
- *https://blog.twitter.com/2013/dremel-made-simple-with-parquet*



Columns:
docid
links.backward
links.forward
name.language.code
name.language.country
name.url

Why columnar?

2D Table

1	a	$	1.1
2	b	€	1.1
3	c	$	0.9

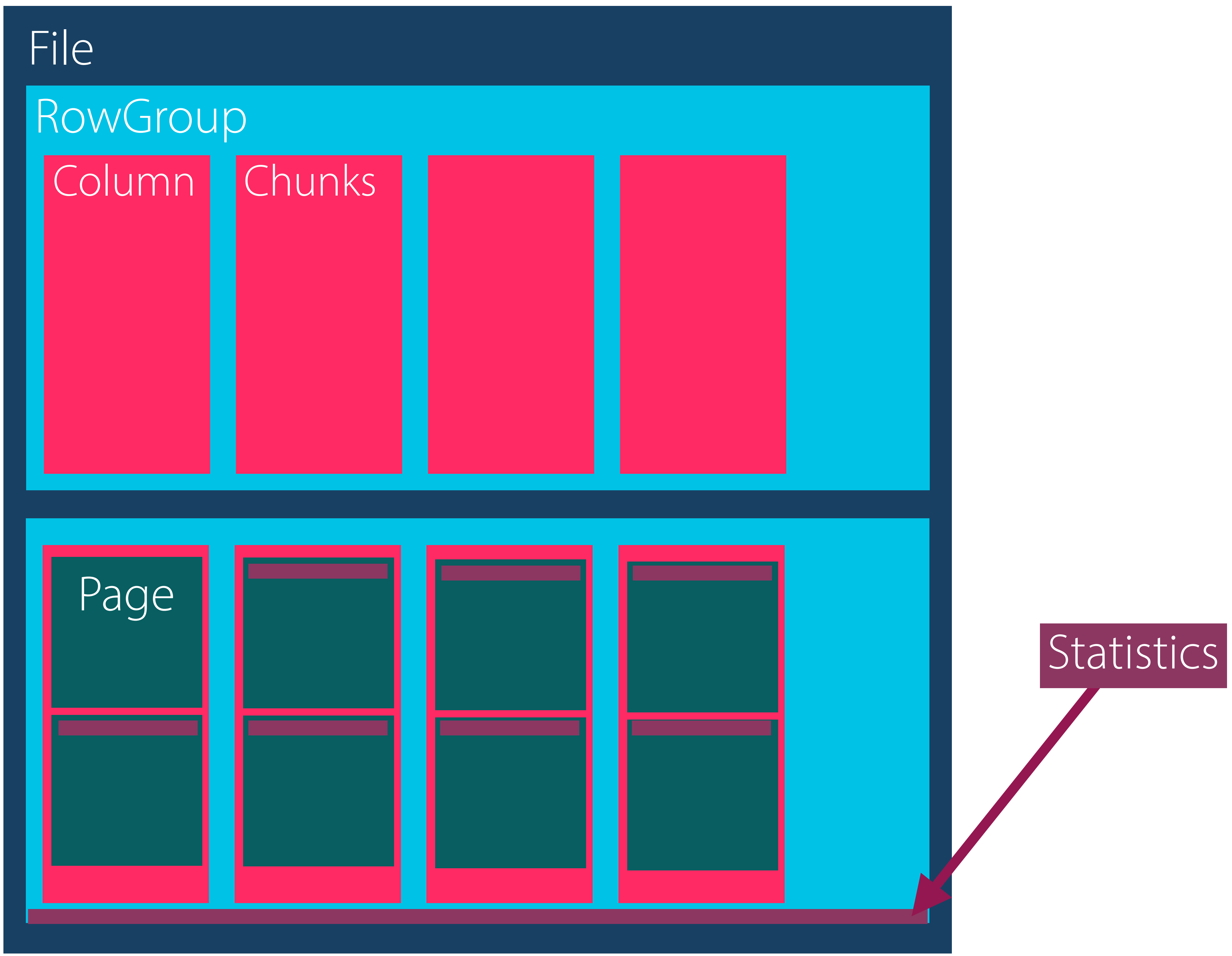
row layout

1	a	$	1.1	2	b	€	1.1	3	c	$	0.9

columnar layout

1	2	3	a	b	c	$	€	$	1.1	1.1	0.9

File Structure



Encodings

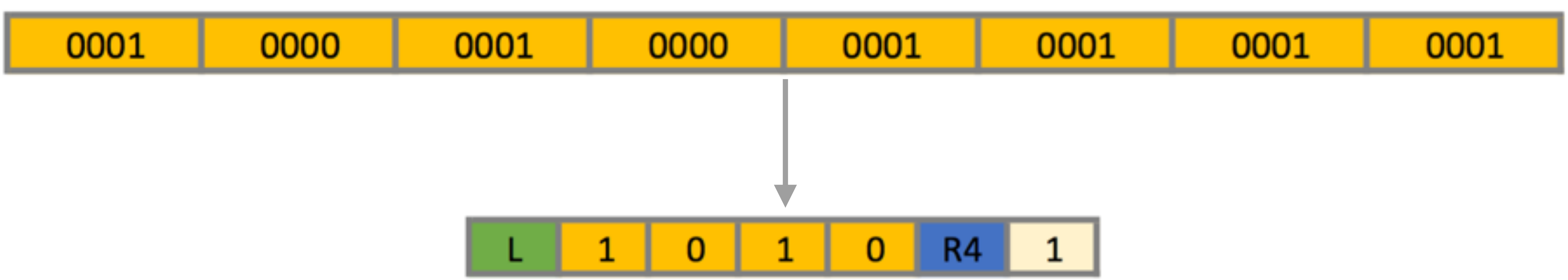
- Know the data
- Exploit the knowledge
- Cheaper than universal compression
- Example dataset:
  - NYC TLC Trip Record data for January 2016
  - *1629 MiB as CSV*
  - *columns: bool(1), datetime(2), float(12), int(4)*
  - Source: http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

Encodings — PLAIN

- Simply write the binary representation to disk
- Simple to read & write
- Performance limited by I/O throughput
- —> 1499 MiB

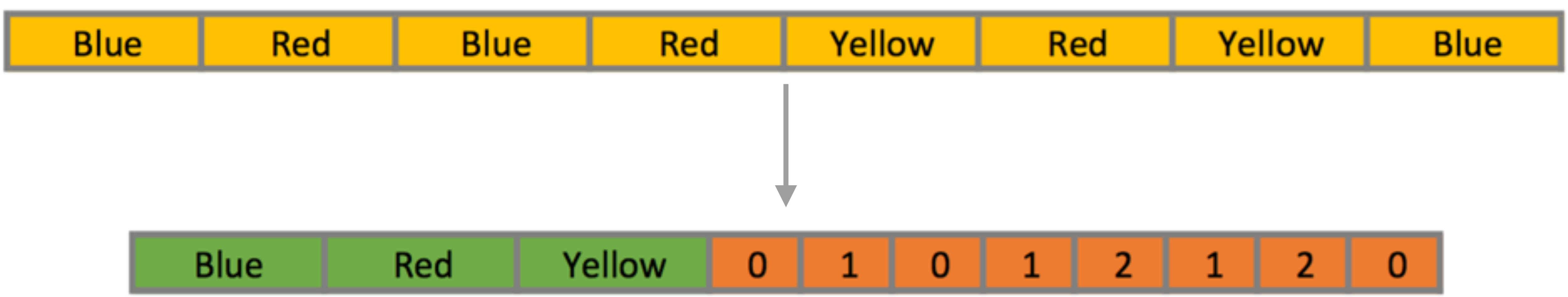
Encodings — RLE & Bit Packing

- bit-packing: only use the necessary bit
- **R**un**L**ength**E**ncoding: *378 times „12"*
- *h*ybrid: dynamically choose the best
- Used for Definition & Repetition levels



Encodings — Dictionary

- **PLAIN_DICTIONARY / RLE_DICTIONARY**
- every value is assigned a code
- *Dictionary:* store a map of *code* —> *value*
- *Data:* store only codes, use ***RLE*** on that
- —> 329 MiB (22%)



Compression

1. Shrink data size independent of its content
2. More CPU intensive than encoding
3. encoding+compression performs better than compression alone with less CPU cost
4. LZO, Snappy, GZIP, Brotli
   —> If in doubt: use Snappy
5. GZIP: 174 MiB (11%)
   Snappy: 216 MiB (14 %)

```
Row group 0:  count: 10906858  16,73 B records  start: 4  total
--------------------------------------------------------------------
                       type      encodings count     avg size
VendorID               INT64     G RBR     10906858  0,09 B
tpep_pickup_datetime   INT64     G RBR_    10906858  0,86 B
tpep_dropoff_datetime  INT64     G RBR_    10906858  2,78 B
passenger_count        INT64     G RBR     10906858  0,23 B
trip_distance          DOUBLE    G RBR     10906858  1,34 B
pickup_longitude       DOUBLE    G RBR     10906858  1,87 B
pickup_latitude        DOUBLE    G RBR     10906858  1,96 B
RatecodeID             INT64     G RBR     10906858  0,04 B
store_and_fwd_flag     BOOLEAN   G RB_     10906858  0,01 B
dropoff_longitude      DOUBLE    G RBR     10906858  1,90 B
dropoff_latitude       DOUBLE    G RBR     10906858  2,11 B
payment_type           INT64     G RBR     10906858  0,16 B
fare_amount            DOUBLE    G RBR     10906858  0,98 B
extra                  DOUBLE    G RBR     10906858  0,04 B
mta_tax                DOUBLE    G RBR     10906858  0,01 B
tip_amount             DOUBLE    G RBR     10906858  0,93 B
tolls_amount           DOUBLE    G RBR     10906858  0,09 B
improvement_surcharge  DOUBLE    G RBR     10906858  0,00 B
total_amount           DOUBLE    G RBR     10906858  1,35 B
```

https://github.com/apache/parquet-mr/pull/384

Query pushdown

1. Only load used data
   1. skip columns that are not needed
   2. skip (chunks of) rows that not relevant
2. saves I/O load as the data is not transferred
3. saves CPU as the data is not decoded

1	a	$	1.1
2	b	€	1.1
3	c	$	0.9

1	a	$	1.1
2	b	€	1.1
3	c	$	0.9

1	a	$	1.1
2	b	€	1.1
3	c	$	0.9

Competitors (Python)

- **HDF5**
  - binary (with schema)
  - fast, just not with strings
  - not a first-class citizen in the Hadoop ecosystem
- **msgpack**
  - fast but unstable
- **CSV**
  - The universal standard.
  - row-based
  - schema-less

C++

1. General purpose read & write of Parquet
   - data structure independent
   - pluggable interfaces (allocator, I/O, …)
2. Routines to read into specific data structures
   - Apache Arrow
   - …

Use Parquet in Python

```
import pyarrow
import pyarrow.parquet

A = pyarrow

def save_as_compressed_parquet():
    table = A.from_pandas_dataframe(df, timestamps_to_ms=True)
    A.parquet.write_table(table, 'table.parquet', compression='SNAPPY')
```

https://pyarrow.readthedocs.io/en/latest/install.html#building-from-source

Get involved!

1. Mailinglist: dev@parquet.apache.org
2. Website: https://parquet.apache.org/
3. Or directly start contributing by grabbing an issue on https://issues.apache.org/jira/browse/PARQUET
4. Slack: https://parquet-slack-invite.herokuapp.com/

Questions?!

We’re hiring!

blueyonder