READ MODE · ANSWER FIRST

Apache Parquet File Format

An interactive, visual explanation of Apache Parquet File Format, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

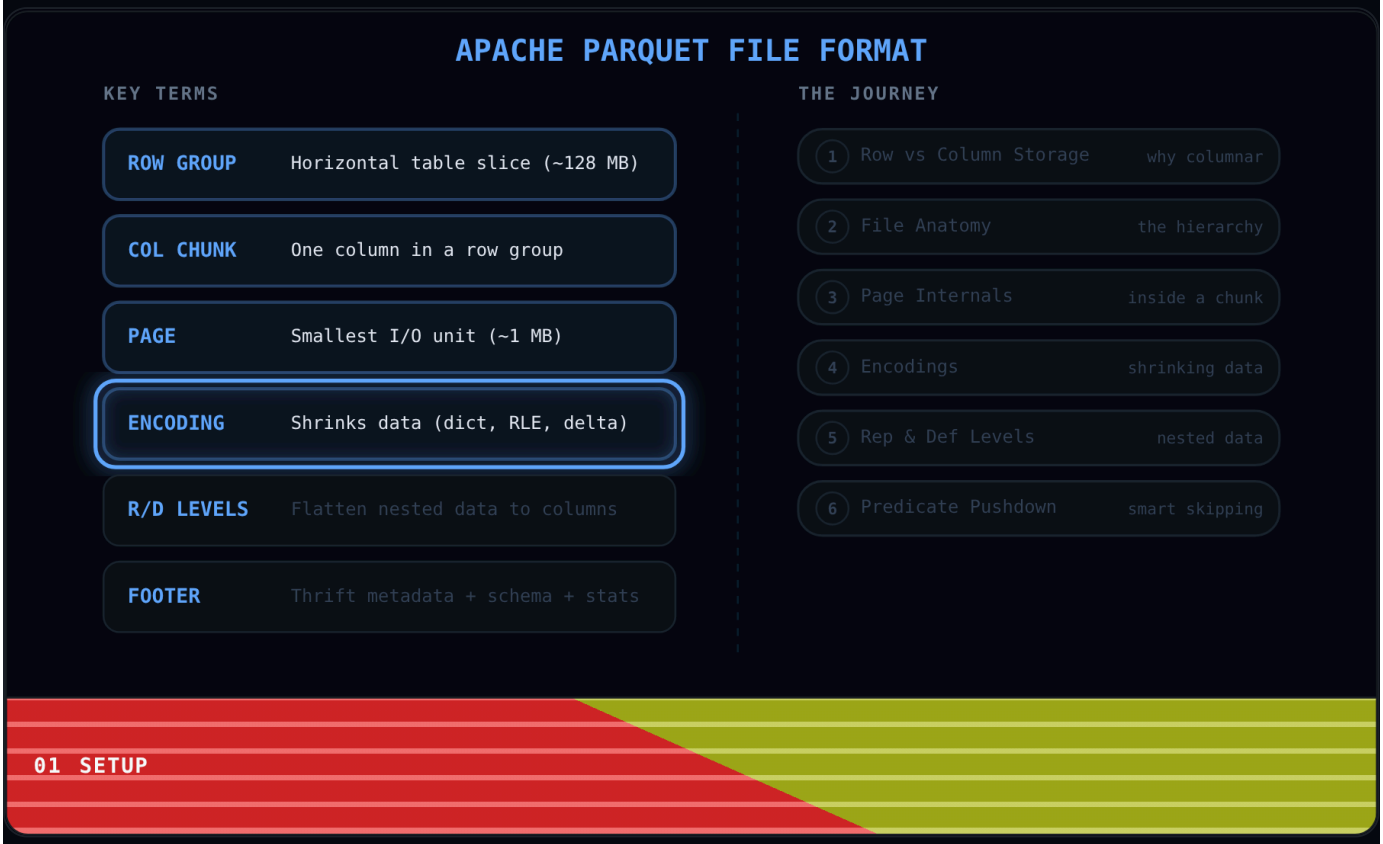
The whole story in 6 lines

Apache Parquet File Format becomes easier to reason about when every stage is connected as one system.

1. Side-by-side comparison of row-oriented vs column-oriented storage showing I/O difference for a projection query.
2. Structural hierarchy: magic bytes → row groups → column chunks → pages → footer.
3. Detail view of a column chunk: page header, dictionary page, data pages with repetition/definition level sections.
4. Interactive comparison of PLAIN, DICTIONARY, RLE and DELTA encodings on the same data.
5. Dremel encoding: the R and D levels that allow nested and repeated fields to be stored as flat column arrays.
6. Show how min/max stats at row group and page level enable skipping, plus column projection.

STAGE 1 OF 8

Setup



Welcome. Apache Parquet is the columnar file format that most of the modern data stack reads and writes. Before we open one up, let us learn six terms you will see in every stage ahead.

A Row Group is a horizontal slice of your table. Parquet chops a large dataset into independent row groups so that a reader can process one chunk at a time without loading the entire file into memory.

Inside each row group, data for a single column is stored together as a Column Chunk. This is what makes Parquet columnar. If you only need three columns out of fifty, you read three column chunks and skip the rest.

Each column chunk is further divided into Pages. A page is the smallest unit of I/O and compression. Parquet compresses and encodes data one page at a time.

Encoding is the trick that makes Parquet files small. Instead of storing raw values, Parquet applies algorithms like dictionary encoding, run-length encoding and delta encoding to squeeze columns down to a fraction of their original size.

For nested data, Definition and Repetition Levels track which fields are present and where repeated groups restart. This is how Parquet flattens a deeply nested record into flat columns without losing the structure.

The File Footer is a Thrift-serialized metadata block at the end of every Parquet file. It holds the schema, row group locations, column statistics and everything a reader needs to plan its I/O. Now let us begin with the most fundamental idea: why store data by columns at all.

Row vs Column Storage

Here is a small table with four columns: id, name, age and city. On the left, the data is stored row by row, the way a traditional database like MySQL or Postgres would write it to disk. On the right, the same data is stored column by column, the Parquet way.

In row storage, each row lives together on disk. The id, name, age and city for a single record are adjacent bytes. This is great when you need the entire row, like fetching a user profile by primary key.

In column storage, all the ids are packed together, then all the names, then all the ages, then all the cities. Each column forms its own contiguous run on disk.

Now watch what happens when we run SELECT age FROM table. In the row store, the reader must scan every row and skip over the id, name and city fields just to get the age values. All that skipped data is wasted I/O.

In the column store, the reader jumps straight to the age column chunk and reads only those bytes. No wasted I/O. For a table with fifty columns, this means reading 2% of the file instead of 100%. Try switching the Query control to SELECT * and notice how the advantage disappears when you need all columns.

That is the core insight. Columnar storage trades row-fetch speed for massive I/O savings on analytical queries that touch a few columns across millions of rows. Every feature of Parquet builds on this foundation. Next, let us look at how a Parquet file organizes these columns internally.

STAGE 3 OF 8

File Anatomy

Now that we understand why columnar storage matters, let us open a Parquet file and see how it is organized. Every Parquet file starts and ends with a 4-byte magic number: PAR1. This lets any reader confirm it is looking at a valid Parquet file before parsing further.

Between the magic bytes, the file is divided into one or more Row Groups. Each row group holds a horizontal slice of the table. A row group typically contains 128 MB of uncompressed data by default, though this is configurable. Try changing the Row Groups control in the sidebar to see how the file structure changes.

Inside each row group, data is organized by column. Each column gets its own Column Chunk. If the table has four columns and two row groups, there are eight column chunks total. Column chunks within a row group are written sequentially.

Each column chunk is further divided into Pages. Pages are the smallest unit of compression and encoding. A typical page holds around 1 MB of data. There are three page types: data pages that hold the actual values, dictionary pages that hold lookup tables and index pages for offset tracking.

At the very end of the file sits the File Footer. This is the metadata brain of the entire file. It contains the full schema definition, the byte offset and size of every row group and column chunk, column statistics like min/max values and null counts, and the encoding and compression codec used for each column.

A reader starts by reading the footer from the end of the file. From the footer, it knows exactly which byte ranges to fetch for the columns and row groups it needs. This design means a reader never has to scan the whole file. It plans its I/O from the footer and fetches only what the query requires.

That is the full anatomy: magic bytes, row groups holding column chunks of pages, and a footer that maps it all. Next, let us zoom into a single column chunk and see what lives inside its pages.

STAGE 4 OF 8

Page Internals

We just saw that column chunks are made of pages. Now let us zoom into a single column chunk for the "city" column and see what each page actually contains. Think of the column chunk as a self-contained file within the file.

Every page begins with a Page Header. The header is a small Thrift-encoded structure that tells the reader the page type, the uncompressed and compressed sizes, the encoding used and how many values the page holds. The reader parses this header first to know how to decode the rest.

If the column uses dictionary encoding, the first page in the chunk is a Dictionary Page. It holds a sorted array of unique values. For the city column, this might be just four entries: Austin, Boston, Chicago, Denver. Every subsequent data page references these entries by integer index rather than storing the full strings.

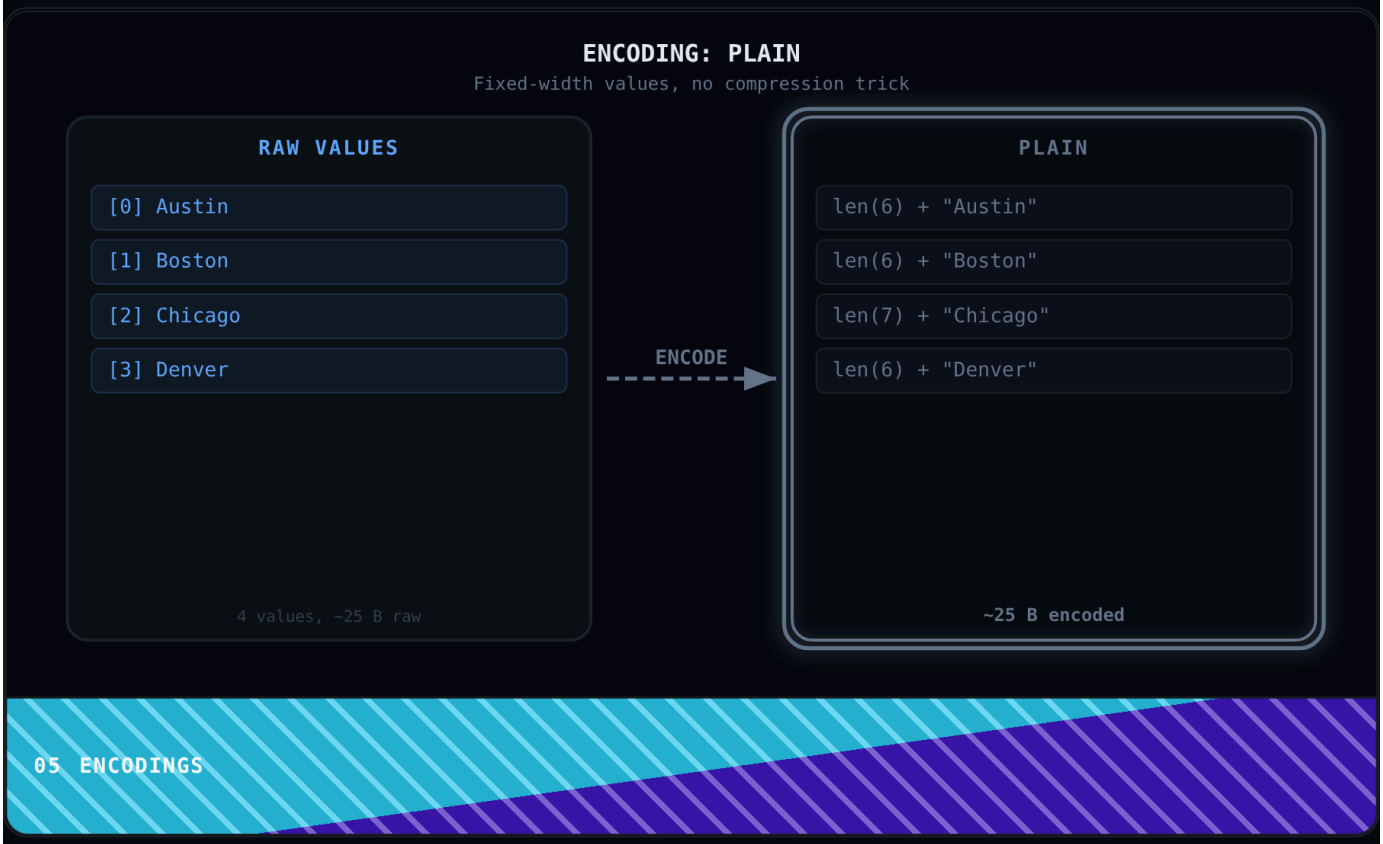
After the dictionary page come the Data Pages. Each data page has three sections. First: the repetition levels, which track where repeated fields restart. Second: the definition levels, which track whether a value is null or present. Third: the actual encoded values. For non-nested, non-nullable columns, the first two sections are empty.

After encoding, the entire page body is compressed using the codec specified in the column metadata. Switch the Compression control in the sidebar to see how different codecs affect the page. Snappy prioritizes speed. ZSTD offers better compression ratios at the cost of more CPU. The page header always stores both sizes so the reader knows how many bytes to read and how many to expect after decompression.

That is the full page anatomy: a header describing the page, an optional dictionary, then data pages carrying levels and values. The reader processes pages sequentially within a column chunk. Now that we know what lives inside the pages, let us look at the encoding algorithms that make the values so compact.

STAGE 5 OF 8

Encodings



We just saw that pages hold encoded values. Now let us see the encoding algorithms themselves. Parquet picks the best encoding for each column based on the data type and cardinality. Switch the Encoding control in the sidebar to compare how each algorithm transforms the same raw data.

PLAIN encoding stores values as-is. Integers are stored as fixed-width 32 or 64 bit values. Strings are stored as a length prefix followed by UTF-8 bytes. There is no compression trick here. This is the baseline that every other encoding improves upon.

DICTIONARY encoding replaces repeated values with small integer indices. The encoder builds a sorted dictionary of unique values, then each data page stores indices instead of full values. For a city column with only 20 unique cities across a million rows, this turns variable-length strings into tiny integers. The dictionary itself is stored once in the dictionary page.

RLE/Bit-Packing encodes runs of repeated values and packs small integers into fewer bits. If a boolean column has 1000 consecutive true values, RLE stores that as the pair (1000, true) instead of 1000 individual bits. Bit-packing takes integers that only need 3 bits and packs eight of them into three bytes instead of eight.

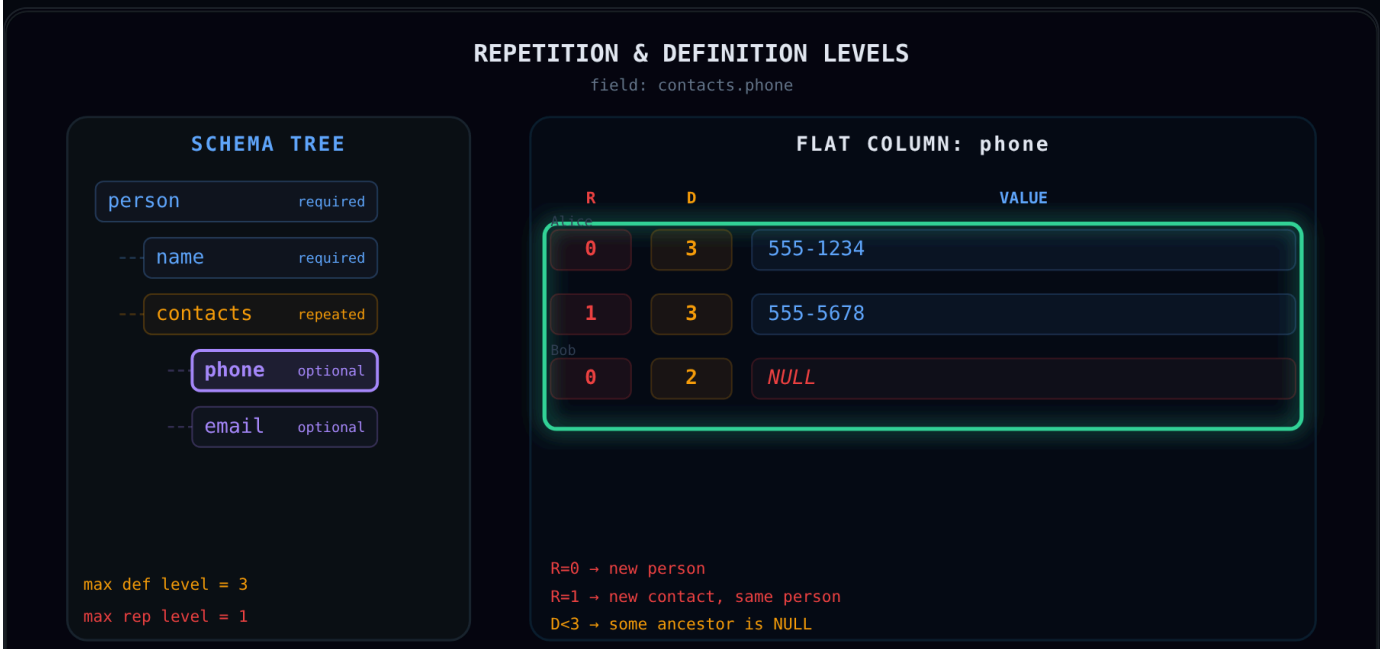
DELTA encoding stores the difference between consecutive values instead of the values themselves. For a sorted timestamp column where entries are one second apart, the deltas are all 1. Those tiny deltas compress extremely well. Parquet uses DELTA_BINARY_PACKED for integers and DELTA_LENGTH_BYTE_ARRAY for variable-length strings.

In practice, Parquet often layers these encodings. A string column might use DICTIONARY encoding, and the dictionary indices themselves get RLE/bit-packed. The column metadata in the footer records exactly which encoding was used so the reader knows how to decode.

Encoding is what makes Parquet files remarkably small. A well-encoded column can be 10 to 100 times smaller than the raw data. Next, let us tackle the most subtle part of the format: how Parquet flattens

nested data without losing the structure.

Repetition & Definition Levels



06 REPETITION & DEFINITION LEVELS

Parquet does not just store flat tables. It handles arbitrarily nested and repeated data, like a JSON record with arrays of objects. The trick comes from Google's Dremel paper: two small integers per value called the Repetition Level and the Definition Level.

Consider a schema: person with a required name and a repeated contacts group, where each contact has an optional phone and an optional email. The schema tree has three levels of nesting: the root person, the repeated contacts list and the leaf fields.

The Definition Level tells the reader how many levels of the path are actually defined for a given value. For the phone field, the max definition level is 3 (person, contacts, phone all defined). A definition level of 2 means contacts exists but phone is null. A definition level of 1 means the contacts list itself is empty. The reader uses this to reconstruct nulls at every level of nesting.

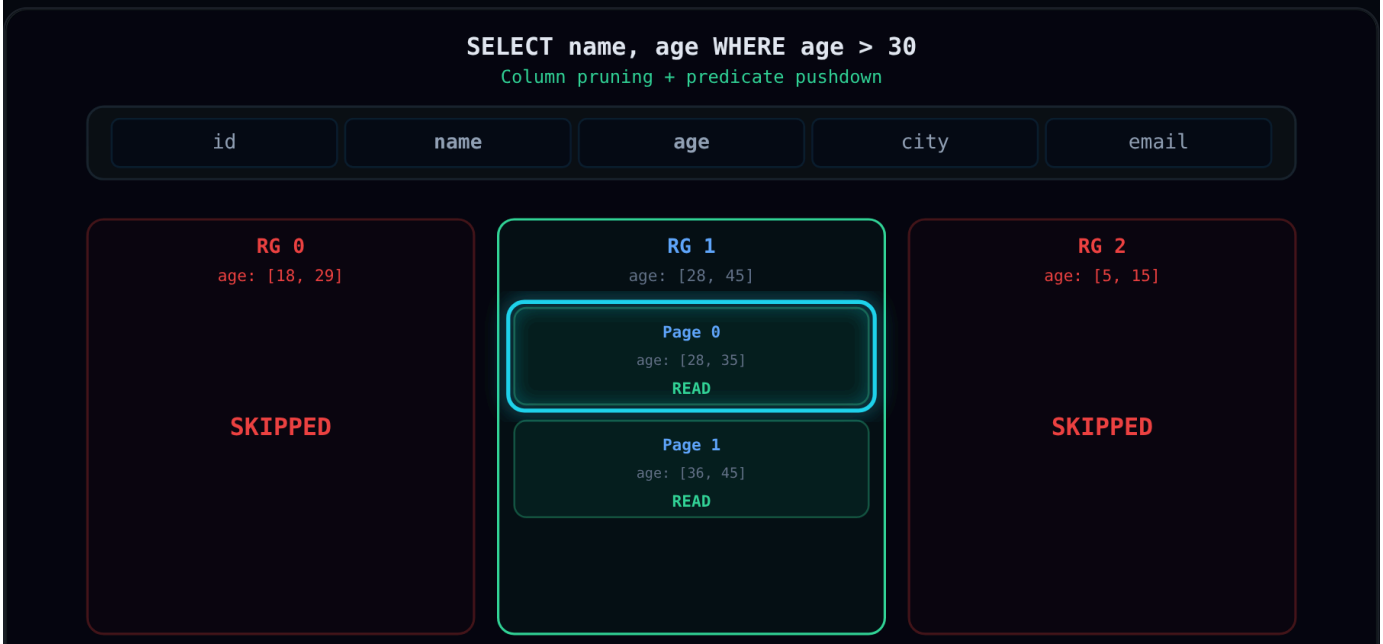
The Repetition Level tells the reader at which level of nesting a new value starts a new repeated element. A repetition level of 0 means a brand new top-level record. A repetition level of 1 means a new entry in the contacts list for the same person. Only repeated fields need repetition levels.

Let us walk through a concrete example. Person Alice has two contacts: phone 555-1234 and phone 555-5678. Person Bob has one contact with no phone. The phone column stores three values: (R=0,D=3,val=555-1234), (R=1,D=3,val=555-5678), (R=0,D=2,val=NULL). Switch the Field control to email to see the same logic for a different leaf.

Because R and D levels are small integers (bounded by the schema depth), they compress extremely well with RLE. A column of millions of phone numbers might need only a few hundred bytes for its level data. The levels are stored at the start of each data page, before the actual values.

Repetition and definition levels are what separate Parquet from a simple CSV-in-columns format. They let Parquet faithfully store Protocol Buffers, Avro records, JSON documents and any nested schema without flattening the structure away. Now let us see how all this structure enables smart query execution.

Predicate Pushdown



07 PREDICATE PUSHDOWN

Everything we have learned so far is how Parquet writes data. Now let us see the payoff at read time. A query engine does not read the whole file. It uses two techniques: column pruning and predicate pushdown. Both exploit the metadata we saw in the footer.

Column pruning is the simpler win. If the query is SELECT name, age FROM table, the reader consults the footer, finds the byte offsets of just the name and age column chunks, and skips everything else. With fifty columns, this alone eliminates 96% of I/O.

Predicate pushdown goes further. The footer stores min/max statistics for each column in each row group. For a WHERE age > 30 clause, the reader checks each row group. If a row group has max_age = 25, then no row in that group can match. The entire row group is skipped without reading a single data byte.

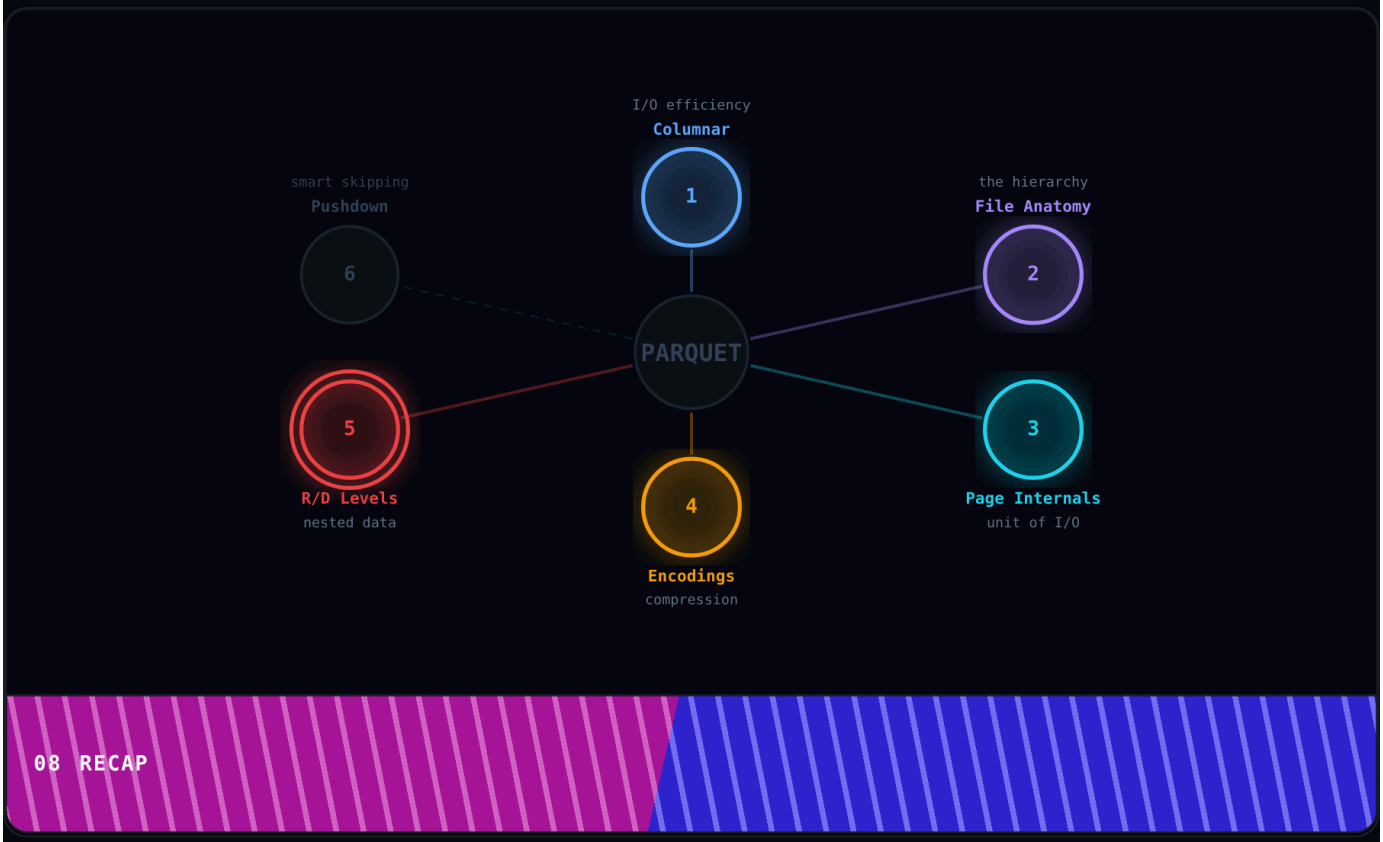
Page-level statistics make this even more precise. Within a row group that passes the row group filter, each data page also stores its own min and max values. The reader can skip individual pages where the predicate cannot be satisfied. This is sometimes called page index filtering.

Try switching the Predicate control to age < 10. Notice how different row groups and pages get skipped depending on the data distribution. The combination of column pruning and multi-level predicate pushdown means a typical analytical query might read less than 1% of the bytes in a large Parquet file.

This is why Parquet is the default format for data lakes. The columnar layout, encoding and statistics all work together to turn a full table scan into a series of surgical byte-range reads. Now let us step back and see the whole picture together.

STAGE 8 OF 8

Recap



08 RECAP

We have walked through the entire Parquet file format, from the high-level idea of columnar storage down to the byte-level encoding of nested fields. Let us connect all six concepts into one picture.

We started with columnar storage: the decision to group values by column instead of by row. This single idea is the foundation. Everything else in Parquet exists to make columnar storage fast, compact and expressive.

We then opened the file and saw its anatomy: magic bytes, row groups, column chunks, pages and the footer. The hierarchy gives readers a map. The footer tells them exactly where every piece of data lives.

Inside each column chunk, we zoomed into the page internals: headers, dictionary pages, data pages with their repetition and definition level sections. Pages are the unit of I/O and compression.

We examined the encoding algorithms that make columns compact. Dictionary encoding, RLE, bit-packing and delta encoding each exploit a different data pattern. They often layer on top of each other for maximum compression.

We tackled the Dremel-inspired repetition and definition levels that let Parquet flatten arbitrarily nested data into flat columns without losing structure. Two small integers per value replace the need for nested containers.

Finally, we saw predicate pushdown and column pruning: the read-time payoff. Statistics at the row group and page level let the reader skip data that cannot match the query, and column pruning means it only reads the columns it needs.

Together, these six ideas make Parquet the backbone of the modern data lake. Columnar layout for I/O efficiency. Hierarchical structure for random access. Encodings for compression. Dremel levels for nested data. And statistics for smart skipping. That is the complete Parquet file format.