READ MODE · ANSWER FIRST

Apache Arrow File Format

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

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

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

1. Side-by-side comparison of row-major struct layout vs column-major array layout in memory. Cache line hits on a columnar scan.
2. Structural hierarchy: ARROW1 magic → Schema message → RecordBatch messages → Footer → ARROW1 magic.
3. Bit-level validity bitmap, offsets buffer for variable-length types, and data buffer. INT64 vs UTF8 layout.
4. Flatbuffer metadata header describing buffer offsets and lengths, followed by the contiguous body with 64-byte padding.
5. Footer structure: schema copy + Block index. Read path: last 10 bytes → footer → seek to batch.
6. mmap the file, read footer, cast buffer pointers directly to typed arrays. No parse step.

STAGE 1 OF 8

Setup

Welcome. Apache Arrow is a language-independent columnar memory format. A Schema defines the structure of your data, listing every field by name and data type. Arrow schemas are strongly typed: an INT64 column is always 8 bytes per value, a UTF8 column stores variable-length strings. Let us learn the key terms you will see in every stage ahead.

A Record Batch is a chunk of rows stored in columnar layout. Each batch carries a slice of the table as a set of equal-length column arrays. Batches are the unit of data transfer. A file holds one or more of them.

A Buffer is a contiguous block of bytes. Every column in a record batch is made of buffers. An integer column has two buffers: a validity bitmap and a data buffer. A string column has three: validity, offsets, and character data. Buffers are the atoms of Arrow memory.

Arrow uses Flatbuffers for its metadata. Flatbuffers is a zero-copy serialization library. The schema, record batch metadata, and footer are all Flatbuffer messages. This means metadata can be read directly from a memory-mapped file without parsing or copying.

The Validity Bitmap is a bit array where each bit says whether the corresponding row is null or present. Bit 0 means null. Bit 1 means the value exists. One byte covers eight rows. This is how Arrow encodes nullability without sentinel values.

Memory Alignment means every buffer starts at a 64-byte boundary and is padded to a multiple of 64 bytes. This alignment lets SIMD instructions process column data at full speed without unaligned-access penalties. Now let us begin with the most fundamental idea: why columnar layout changes everything for analytical workloads.

STAGE 2 OF 8

Row vs Columnar Memory

02 ROW VS COLUMNAR MEMORY

Here is a small table with four columns: id (INT32), name (UTF8), price (FLOAT64), and qty (INT32). On the left, data is stored row by row. Each row is a struct with all four fields packed together. On the right, the same data is stored column by column. Each column is a contiguous array.

In row-major layout, a single row occupies a variable number of bytes because the name field has variable length. Accessing the price of row N requires skipping over every field that comes before it. There is no fixed stride.

In Arrow columnar layout, the price column is a flat FLOAT64 array. Every value is exactly 8 bytes. Accessing price[N] is a single pointer add: base + N * 8. No parsing. No field skipping. This fixed stride is what makes vectorized processing possible.

Watch what happens when we compute SUM(price). In the row store, the CPU loads cache lines that contain id, name, and qty bytes it will never use. Those wasted bytes evict useful data from cache. The scan touches every byte in the table to read one column.

In the columnar store, the CPU loads only the price array. Every byte in every cache line is a price value. L1 cache hit rate approaches 100%. For a table with 200 columns, this means reading 0.5% of the data instead of 100%. Try switching the Scan Target to SELECT * in the sidebar and notice that the advantage disappears when all columns are needed.

That is the core tradeoff. Columnar layout sacrifices random row access for massive throughput on analytical scans. Arrow locks this decision into its memory format so that every library in every language gets the same cache-friendly layout. Next, let us open an Arrow IPC file and see how it organizes these columnar arrays on disk.

STAGE 3 OF 8

IPC File Anatomy

03 IPC FILE ANATOMY

Now that we understand why Arrow uses columnar layout, let us look at how an Arrow IPC file organizes that data on disk. Every Arrow file starts and ends with the same 8-byte magic sequence: ARROW1 followed by two padding bytes. This lets any reader instantly verify the file format.

Immediately after the opening magic comes the Schema message. This is a Flatbuffer-encoded description of every field: name, data type, nullability, and any nested children. The schema is written once and tells any reader the exact structure of every record batch that follows.

After the schema come one or more Record Batch messages. Each message has two parts: a Flatbuffer metadata header describing buffer offsets, lengths, and null counts, followed by the raw buffer body. The body is the actual column data. Try changing the Record Batches control in the sidebar to see how the file grows.

Optionally, the file can include Dictionary Batch messages before or between record batches. Dictionary batches define lookup tables for dictionary-encoded columns. The actual record batch data then stores integer indices instead of full values. This is how Arrow handles categorical strings efficiently.

At the very end, just before the closing magic bytes, sits the Footer. The footer is another Flatbuffer message. It contains a copy of the schema plus a Block index: an array of (offset, metadataLength, bodyLength) triples, one per record batch. This index is what makes random access possible.

A reader opens an Arrow file by reading the last 10 bytes to find the footer length, then reads the footer, then uses the Block index to seek directly to any record batch it needs. No sequential scan. No parsing of intermediate data. The file format is designed for random access from the end. Next, let us zoom into the buffer layout of a single column.

STAGE 4 OF 8

Buffer Layout

INT64 COLUMN LAYOUT

VALIDITY BITMAP 1 bit per row

1	0	1	1	1	1

DATA BUFFER 8 bytes per value

42	??	7	100	0	13

We just saw that record batches contain raw buffer bodies. Now let us zoom in and see what those buffers actually look like for a single column. The buffer layout depends on the data type. Switch the Data Type control to compare INT64 and UTF8 layouts.

Every nullable column starts with a Validity Bitmap. This is a bit array packed 8 rows per byte. Bit position 0 is the least significant bit. A 1 bit means the value is present. A 0 bit means null. For 1000 rows, the validity bitmap is only 125 bytes plus padding to the next 64-byte boundary.

For a fixed-width type like INT64, the data buffer is a flat array of 8-byte values. Row N lives at offset N * 8. There is no length prefix, no delimiter, no framing of any kind. The CPU can load values directly into SIMD registers. A null row still occupies 8 bytes in the data buffer. The validity bitmap is the only place that records nullness.

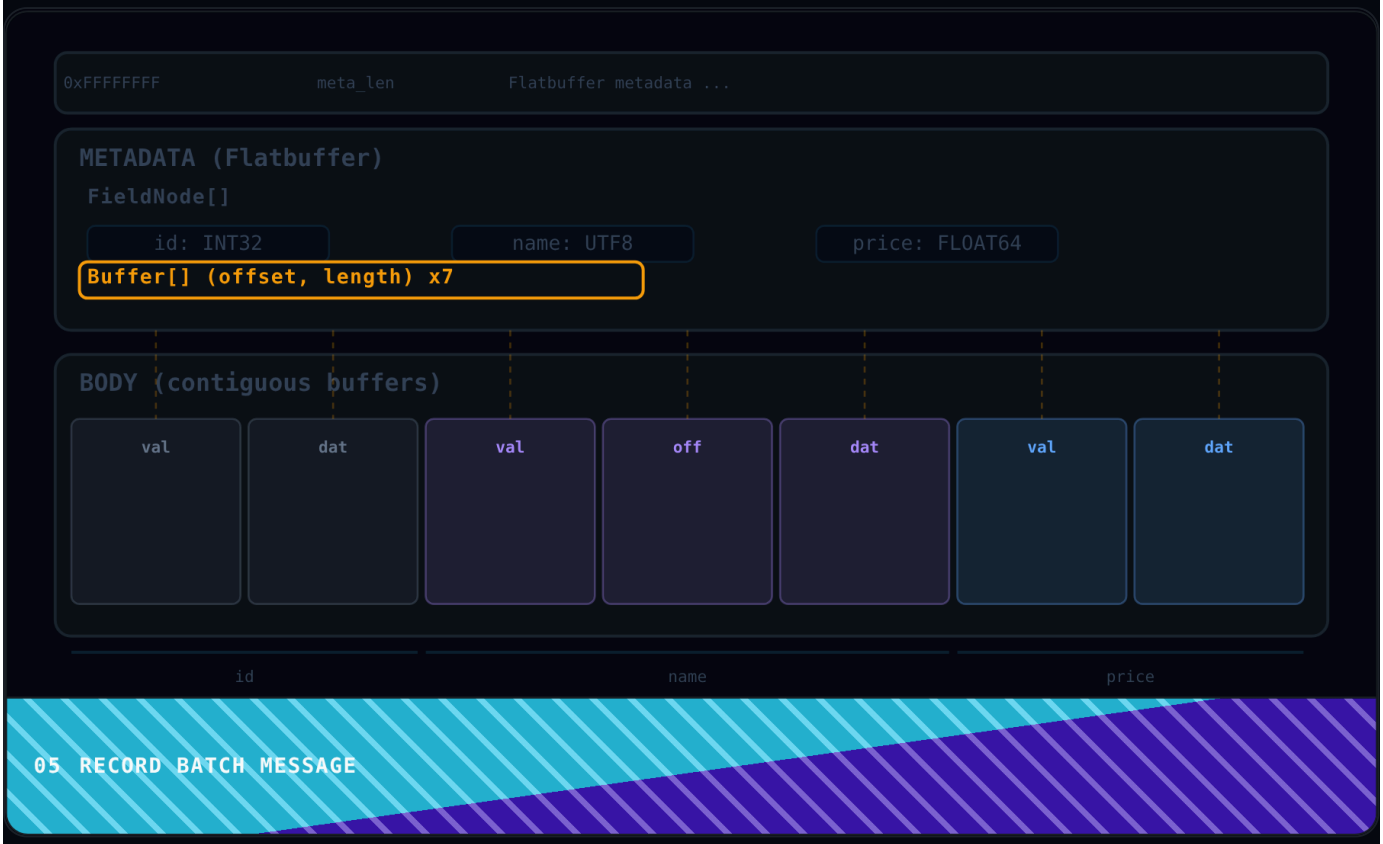
For a variable-length type like UTF8, the layout adds an Offsets buffer between validity and data. The offsets buffer is an array of INT32 values. offset[i] is the byte position where string i starts in the data buffer. offset[i+1] minus offset[i] is the byte length of string i. So N strings require N+1 offset entries.

The UTF8 data buffer stores all string bytes packed end to end with no delimiters. Reading string i means: check validity bit i, read offset[i] and offset[i+1], then slice data[offset[i]..offset[i+1]]. This avoids the overhead of per-string length prefixes that formats like Protobuf or Thrift use.

Every buffer is padded to a multiple of 64 bytes. If the validity bitmap for 5 rows is 1 byte, it gets 63 bytes of zero-padding. This alignment guarantee means a consumer can always hand the buffer directly to a SIMD intrinsic or a GPU kernel without copying. That is the buffer contract. Next, let us see how multiple columns pack together into a record batch message.

STAGE 5 OF 8

Record Batch Message



We now know what a single column looks like in memory. But a record batch contains multiple columns. How does Arrow pack them into one contiguous message that can be written or sent as a single unit? The answer is a two-part message: metadata header plus body.

The metadata header is a Flatbuffer message. It contains the number of rows in the batch, and for each column, a FieldNode with the column length and null count. After the field nodes comes a list of Buffer descriptors. Each descriptor is a (offset, length) pair pointing into the body. The header is a map of the body.

The body is where all the actual bytes live. Column buffers are written one after another in schema order. For each column, the validity bitmap comes first (if nullable), then type-specific buffers (offsets for variable-length, data for all). Every buffer start is aligned to a 64-byte boundary by inserting zero-padding between buffers. Try changing the Columns control to see how the body grows.

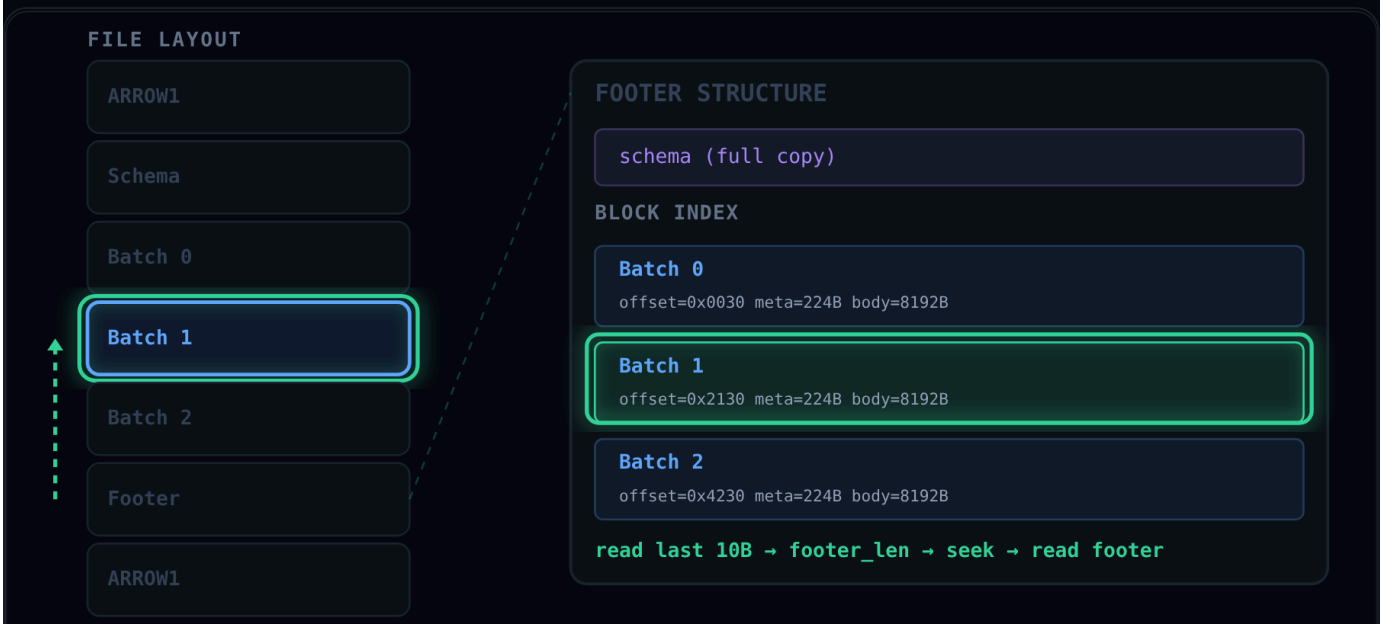
The buffer descriptors in the metadata use absolute byte offsets from the start of the body. This means a reader can seek to any individual buffer without parsing the buffers that came before it. Reading column 47 out of 200 is a single seek plus a read of exactly the bytes described by that buffer descriptor.

The full IPC message wraps everything in a framing envelope: a 4-byte continuation marker (0xFFFFFFFF), a 4-byte little-endian metadata length, the Flatbuffer metadata, padding to 64-byte alignment, then the body. This envelope is what gets written to the file or sent over a stream socket.

This design means writing a record batch is a sequential write of metadata plus body. Reading it is: parse the small metadata header, then use the buffer descriptors to access column data directly without any further parsing. The body bytes are the final in-memory representation. Next, let us look at how the footer enables random access to any batch in the file.

STAGE 6 OF 8

Footer & Random Access



06 FOOTER & RANDOM ACCESS

We now understand how each record batch is structured. But a file can hold hundreds of batches. How does a reader find batch N without scanning every byte before it? That is the job of the footer.

The footer is a Flatbuffer message written at the very end of the file, just before the closing ARROW1 magic. It contains a full copy of the schema (so the reader never needs to seek back to the beginning) and two Block arrays: one for record batches and one for dictionary batches.

Each Block is a triple: (offset, metadataLength, bodyLength). The offset is the absolute byte position of the message in the file. metadataLength is how many bytes the Flatbuffer metadata occupies. bodyLength is how many bytes the buffer body occupies. These three numbers are everything a reader needs to seek and read one batch.

The read path starts from the end. The reader reads the last 10 bytes of the file: a 4-byte little-endian footer length, then the 6-byte ARROW1 + padding magic. It subtracts the footer length to find the footer start, reads the footer, and now has random access to every batch. Use the Read Batch control to watch how the reader jumps to different batches.

This end-first design means a writer can stream record batches sequentially, appending each one, and only write the footer at the end when it knows all the offsets. A reader can open a multi-gigabyte file and access batch 500 out of 1000 with exactly three reads: the tail magic, the footer, and the target batch. No scanning.

The footer is what turns a flat byte sequence into a random-access columnar database file. Without it, Arrow IPC would only support sequential streaming. With it, any batch and any column within that batch can be accessed in constant time. Now let us see the ultimate payoff: the zero-copy read path.

Zero-Copy Read Path

07 ZERO-COPY READ PATH

Everything we have learned so far exists for one reason: to make reading columnar data as cheap as casting a pointer. This stage shows the zero-copy read path and why it is fundamentally different from traditional deserialization.

In a traditional read, the application calls read() to copy file bytes into a userspace buffer, then parses the buffer to extract values. Every value gets touched twice: once by the kernel copying it from page cache to userspace, once by the parser decoding it. For a 1 GB file, that is 2 GB of memory traffic.

Arrow uses mmap instead. The operating system maps the file directly into the process address space. No bytes are copied. The page cache IS the buffer. When the application accesses a memory address in the mapped region, the MMU translates it to the file page. The data appears in memory on demand.

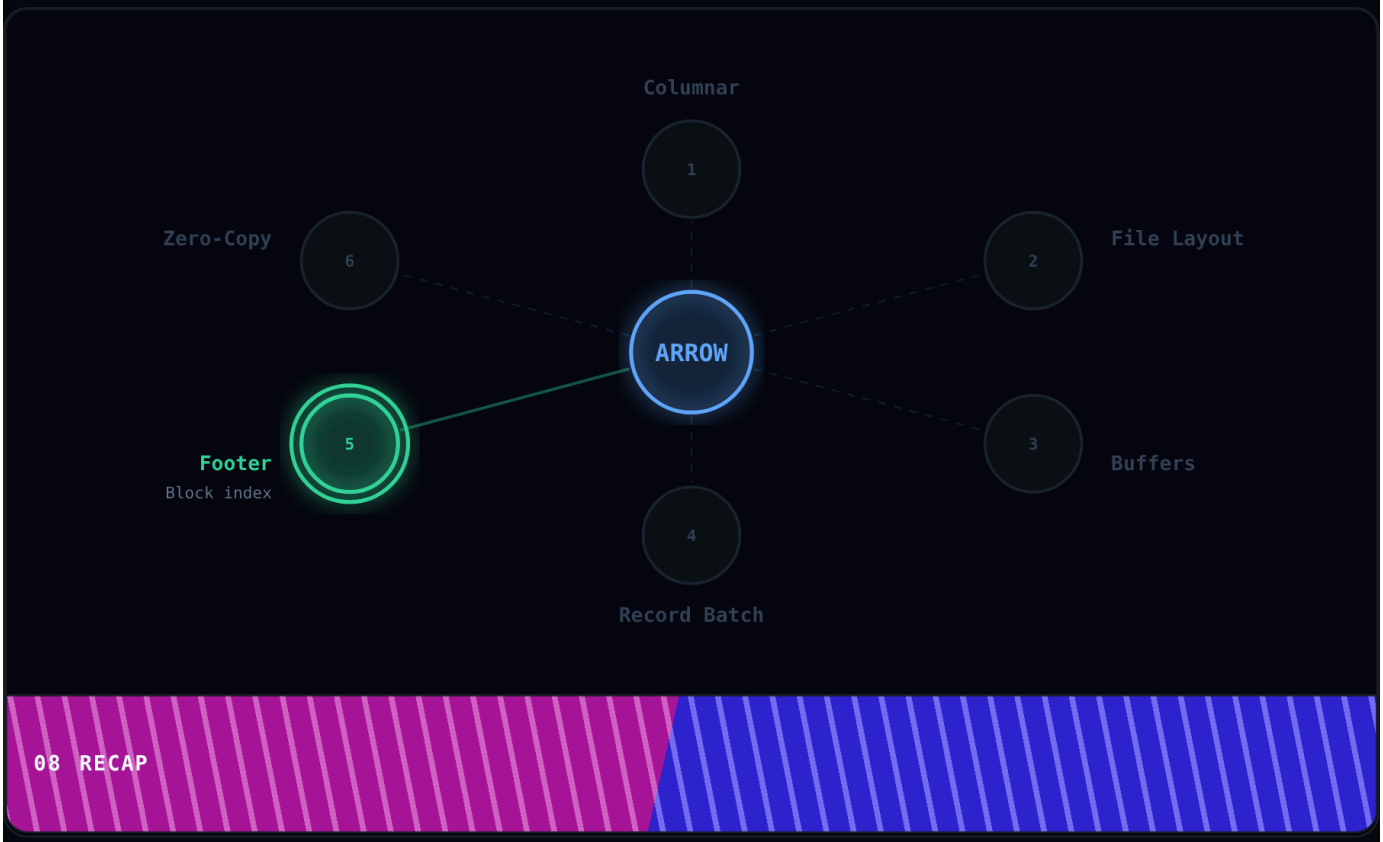
Because Arrow buffers are already in their final in-memory representation (fixed-width, aligned, padded), the reader does not need a parse step at all. It reads the footer to get buffer offsets, then computes a pointer: base_address + batch_offset + buffer_offset. That pointer IS an Arrow array. Try switching the Read Method to read+copy to see the extra copies a traditional format requires.

This is why Arrow calls itself a "zero-copy" format. A Parquet reader must decompress, decode, and materialize column data into buffers. An Arrow reader just maps the file and does pointer arithmetic. The tradeoff is file size: Arrow files are larger because they store raw values. But for inter-process communication, shared memory, and memory-mapped analytics, the zero-copy path is unbeatable.

That is the full payoff. Columnar layout for cache efficiency. Aligned buffers for SIMD. Flatbuffer metadata for zero-copy headers. A footer for random access. And mmap to skip deserialization entirely. Every design decision in the format serves this one goal: make the file bytes and the in-memory representation identical. Now let us step back and see the whole picture together.

STAGE 8 OF 8

Recap



08 RECAP

We have walked through the entire Apache Arrow IPC file format, from the fundamental choice of columnar layout to the zero-copy read path that it enables. Let us connect all six concepts into one picture.

We started with columnar memory layout: the decision to store each column as a contiguous array instead of packing rows together. This single choice is what makes SIMD processing, cache-line efficiency, and fixed-stride access possible.

We then opened the file and saw its anatomy: ARROW1 magic bytes, a schema message, record batch messages, optional dictionary batches, a footer, and closing magic. The format is self-describing and designed for random access from the end.

We zoomed into the buffer layout of a single column. Validity bitmaps encode nulls as single bits. Fixed-width types use flat data arrays. Variable-length types add an offsets buffer. Every buffer is padded to 64-byte alignment.

We saw how a record batch message packs multiple columns: a Flatbuffer metadata header with buffer descriptors, followed by a contiguous body where each buffer starts on a 64-byte boundary. The metadata is a map of the body.

We examined the footer and its Block index. Three numbers per batch: offset, metadata length, body length. The reader starts from the last 10 bytes, reads the footer, and can seek to any batch in constant time.

Finally, we saw the zero-copy read path. mmap the file, read the footer, compute a pointer to any buffer, and cast it directly to a typed array. No parsing. No copying. The file bytes are the in-memory representation.

Together, these six ideas make Apache Arrow the lingua franca of columnar data. Columnar layout for throughput. Typed buffers for safety. Alignment for SIMD. Flatbuffers for zero-copy metadata. A footer for random access. And mmap for zero-copy data. That is the complete Arrow file format.