# Large-Scale Data Management and Distributed Systems

# VI. Column-oriented Storage

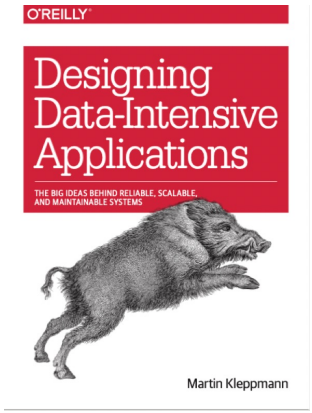
Vania Marangozova

Vania.Marangozova@imag.fr

2023-2024

# References

- Designing Data-Intensive Applications by Martin Kleppmann
  - Chapter 3: Column-oriented storage
  - Chapter 4: Formats for Encoding Data
- Presentation of Prof Michael Stonebraker @EPFL (2013) (https://slideshot.epfl.ch/play/suri_stonebraker): Everything You Learned in Your DBMS Class is Wrong



# In this lecture

- Transacton-based systems vs Data Warehouses
- Principles of column-oriented storage
- Representation of large data on disks

# Transaction-based Systems

- Early days: write to a database = business transaction
  - Sell at Walmart: Client xxx bought yyy for zzz on ttt in store sss
- Applications are interactive
  - Interaction with the end users
  - Online Transaction Processing Systems (OLTP)
- A transaction
  - Involves a sequence of reads/updates of a small number of records
  - **Requires high availability and low latency**

# Row-based Traditional OLTP

- Typically using the relational model
- Data is organized in tables which contain **rows**
- One row = une piece of information containing different facts/attributes

Table

| | Country | Product | Sales |
|---|---|---|---|
| Row 1 | India | Chocolate | 1000 |
| Row 2 | India | Ice-cream | 2000 |
| Row 3 | Germany | Chocolate | 4000 |
| Row 4 | US | Noodle | 500 |

# Traditional Row-Based Storage

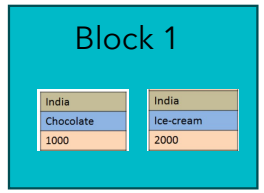
- Persistent storage is divised in blocks
- Each block contains several rows
- When a row needs to be accessed, the whole row (the whole block !) needs to be loaded into memory
- Speeding up with *indexes*
  - *indexes are application specific, depending on the type of queries*

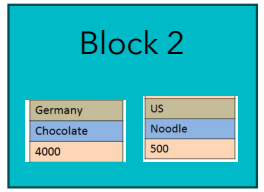
**Table**

| | Country | Product | Sales |
|---|---|---|---|
| Row 1 | India | Chocolate | 1000 |
| Row 2 | India | Ice-cream | 2000 |
| Row 3 | Germany | Chocolate | 4000 |
| Row 4 | US | Noodle | 500 |

**Row Store**

| | |
|---|---|
| Row 1 | India |
| | Chocolate |
| | 1000 |
| Row 2 | India |
| | Ice-cream |
| | 2000 |
| Row 3 | Germany |
| | Chocolate |
| | 4000 |
| Row 4 | US |
| | Noodle |
| | 500 |



# A Detour on RDBMS Indexes

- An index is an additional data structure
  - does not change the contents of the DB
  - impacts performance: overhead on a write operation: do the write **and** update the index
    - need to do the write very fast
    - nothing compares to a sequential append to a file as random access is suicidal
- The simplest index = hash index <key, value>
  - **in memory** structure
  - key -> file offset
  - good for **frequent** updates of values

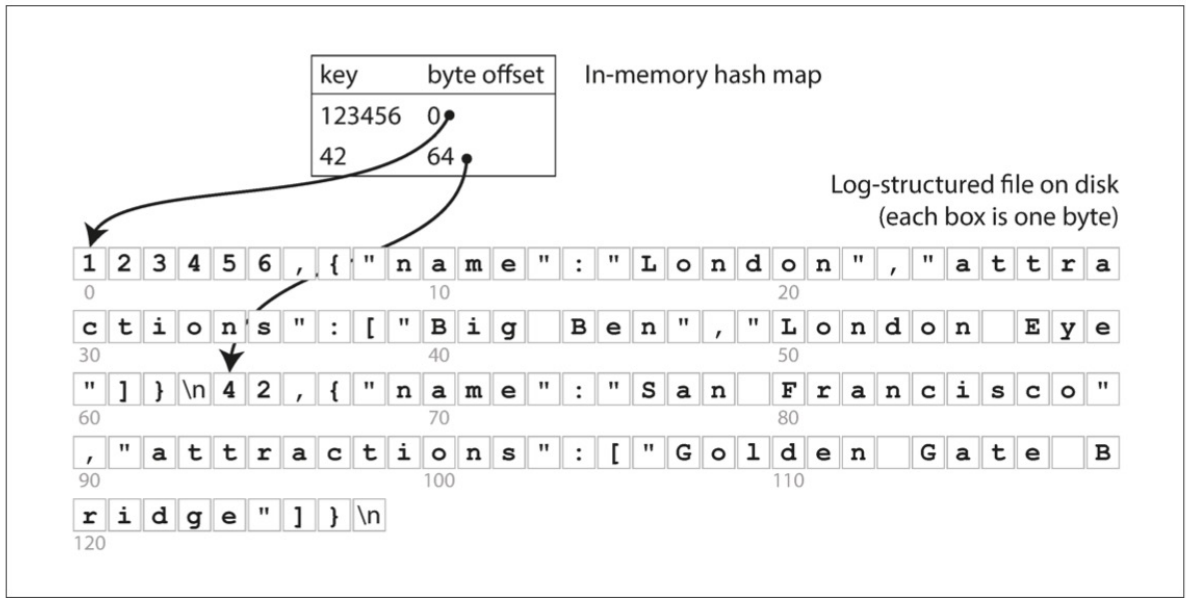
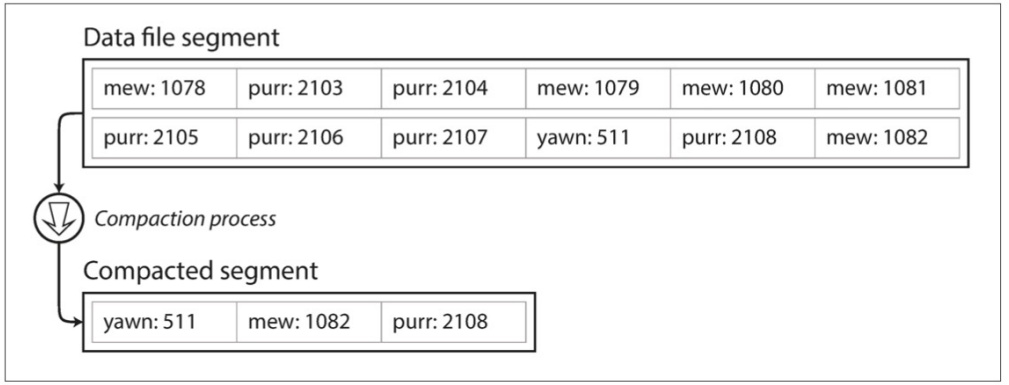


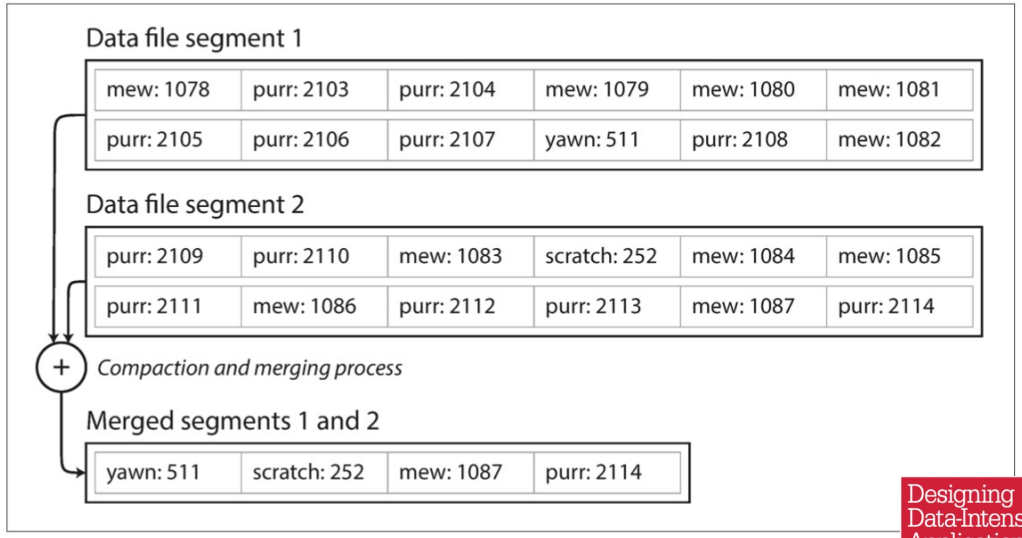

*Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.*

Designing Data-Intensive Applications

# Optimize the storage ?

- **Compaction**
  - break the log into segments of a certain size
  - close a segment file when it reaches a certain size
  - make subsequent writes to a new segment file
  - compact segments
  - as segments become smaller, merge several segments together
  - segments are never modified




Designing Data-Intensive Applications

# SSTables Indexes

- Hash-based indexes' limitations
  - must fit into memory: problem with a great number of keys
  - range queries are not efficient: need to query key by key
- SSTables = Sorted String Table
  - <key, value> sequence is **sorted by key**
  - merging segments is simple and efficient: mergesort
  - ne need to keep an index of **all the keys** in memory: sparce index is enough
  - segments may by compressed

# Construction of SSTables

- Use (balanced) tree data structure for representation in memory
  - this in memory index is called **memtable**

1. write => update memtable
2. size memtable > threshold => write it to dosk as SSTable
3. read => search in memtable, then in most recent segment, etc
4. periodically do compaction

- To face crashes, maintain an additional append log of recent writes
  - a separate log on disk to which every write is immediately appended (as in the hash)

    => Log-Structured Merge-Tree (LSM-Tree)

# Back to Data Management Systems

- Busienesses have not only their OLTP systems, but also maintain Data Warehouses
- Data Warehouse = data system used for data analytics
  - How much have the PPP store sold this month ?
  - Which product is the most popular ?
  - ...
- OLAP = Online Analytic Processing Systems (OLAP)
- Access pattern
  - Scan over a large number of records
  - Compute statistics (make dashboards, ... business intelligence... data analyst)

# Why not execute anayltics queries on the OLTP directly ?

- Performance
  - Disturb the OLTP requests
  - Not adapted to serve OLAP requests
- Complexity
  - An entreprise might have multiple OLTP systems

# Content of Data Warehouses

- Extract from the OLTP
- Transform the data
- Load into the OLAP

- Contains a read-only copy of the data of all the OLTP systems in the company

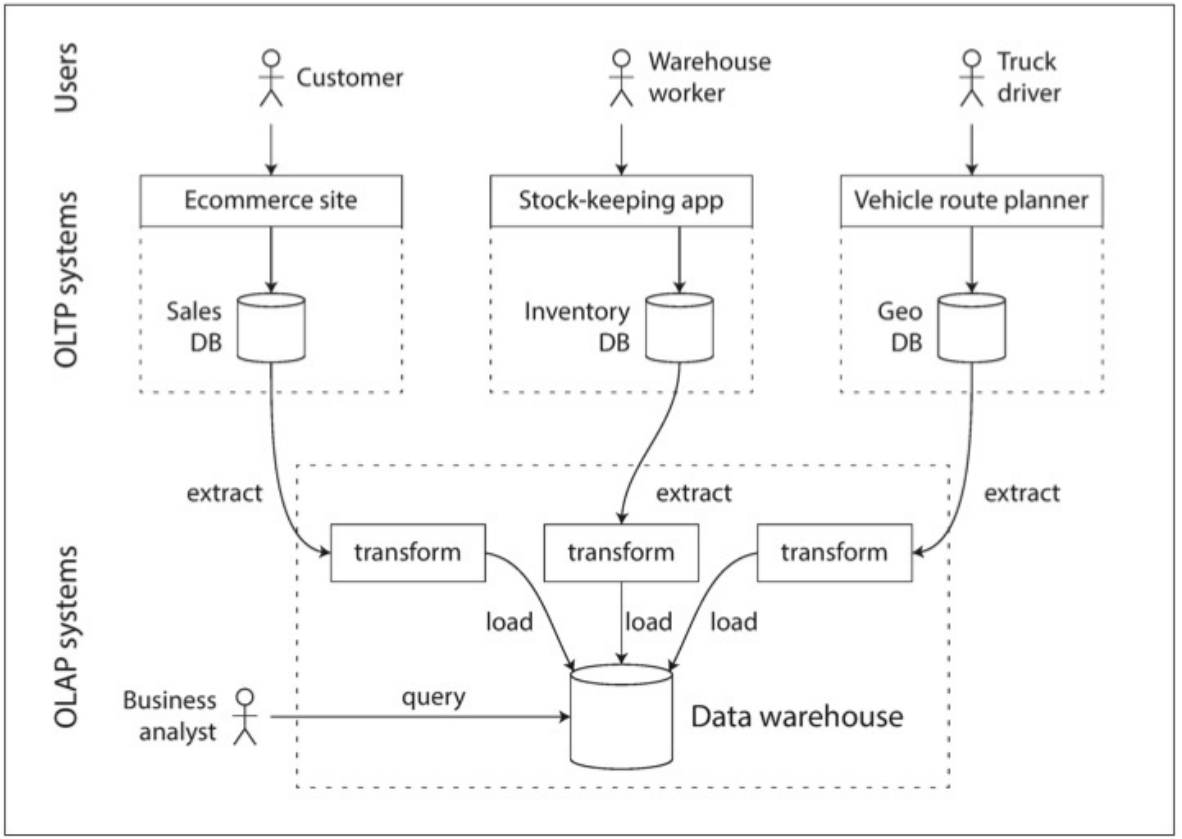



*Figure 3-8. Simplified outline of ETL into a data warehouse.*

# Differences between OLTP & OLAP

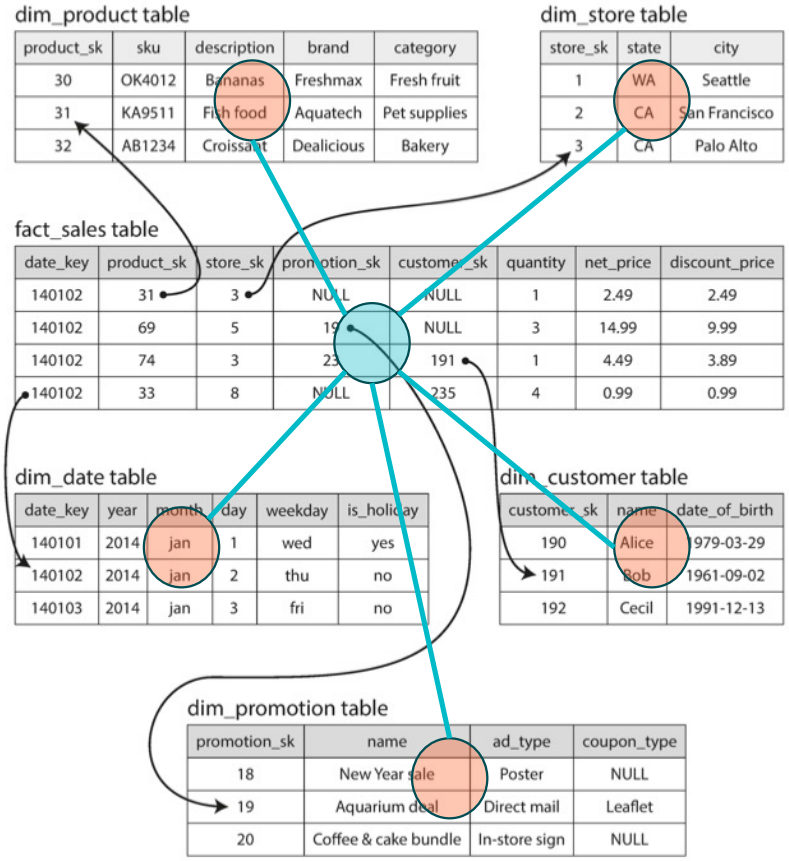
*Table 3-1. Comparing characteristics of transaction processing versus analytic systems*

| Property | Transaction processing systems (OLTP) | Analytic systems (OLAP) |
|---|---|---|
| Main read pattern | Small number of records per query, fetched by key | Aggregate over large number of records |
| Main write pattern | Random-access, low-latency writes from user input | Bulk import (ETL) or event stream |
| Primarily used by | End user/customer, via web application | Internal analyst, for decision support |
| What data represents | Latest state of data (current point in time) | History of events that happened over time |
| Dataset size | Gigabytes to terabytes | Terabytes to petabytes |

Designing Data-Intensive Applications

# Data Warehouse Data Model

- Star or Snowflake model
- A central table = fact table
  - usually contains data with **x100 attributes**
  - Example : purchase, date, client, ...
- Dimension tables
  - giving more information on some "dimensions" of the facts
  - Example : store table



# Typical Data Warehouse Query

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date    ON fact_sales.date_key    = dim_date.date_key
  JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

- Concerns **only a few attributes** from the rows of the fact table

- What about performance ?
  - In **row-oriented storage** need to read and bring **all the row** i.e. x100 more data than needed

# Back to Row-oriented Storage

All the values from one row of a table are stored next to each other

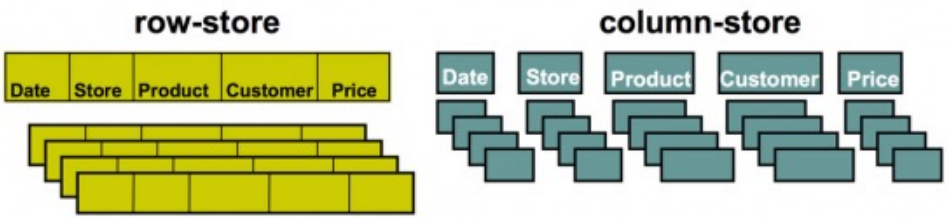
- Inefficient data compression: data of different types are next to each other
- Inefficient read operations
  - interested in a few entries in a row but need to read the whole row
  - may want to filter elements based on a condition on one entry but need to read all rows

# Column-oriented Storage Idea

Store all the values

from one row together ✗

from each column together ✓




fact_sales table

| date_key | product_sk | store_sk | promotion_sk | customer_sk | quantity | net_price | discount_price |
|---|---|---|---|---|---|---|---|
| 140102 | 69 | 4 | NULL | NULL | 1 | 13.99 | 13.99 |
| 140102 | 69 | 5 | 19 | NULL | 3 | 14.99 | 9.99 |
| 140102 | 69 | 5 | NULL | 191 | 1 | 14.99 | 14.99 |
| 140102 | 74 | 3 | 23 | 202 | 5 | 0.99 | 0.89 |
| 140103 | 31 | 2 | NULL | NULL | 1 | 2.49 | 2.49 |
| 140103 | 31 | 3 | NULL | NULL | 3 | 14.99 | 9.99 |
| 140103 | 31 | 3 | 21 | 123 | 1 | 49.99 | 39.99 |
| 140103 | 31 | 8 | NULL | 233 | 1 | 0.99 | 0.99 |

Columnar storage layout:

| | |
|---|---|
| date_key file contents: | 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103 |
| product_sk file contents: | 69, 69, 69, 74, 31, 31, 31, 31 |
| store_sk file contents: | 4, 5, 5, 3, 2, 3, 3, 8 |
| promotion_sk file contents: | NULL, 19, NULL, 23, NULL, NULL, 21, NULL |
| customer_sk file contents: | NULL, NULL, 191, 202, NULL, NULL, 123, 233 |
| quantity file contents: | 1, 3, 1, 5, 1, 3, 1, 1 |
| net_price file contents: | 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99 |
| discount_price file contents: | 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99 |

# Column-oriented Storage

- Each column in a different file
  - The number of distinct values in a column is often small (not the case for rows)
  - A column contains data of the same type => **eficient compression**
- To optimize access : **sort** values in a column
  - all columns sorted the same way to maintain index relation ☺
  - there may be different sorting possibilities that suit different queries: materialized views

| Country | Status |
|---|---|
| US | Processing |
| US | Shipped |
| US | Shipped |
| US | Shipped |
| JP | Canceled |
| JP | Processing |
| JP | Processing |
| JP | Shipped |
| UK | Canceled |
| UK | Canceled |
| UK | Processing |
| UK | Processing |
| KE | Canceled |
| KE | Shipped |
| KE | Shipped |
| KE | Shipped |

# Optimizations for Read Operations

- Limiting the amount of data read
  - Context: A request that selects a subset of columns
  - Solution with columnar storage: we can read only the files corresponding to these columns
- Filtering based on a condition
  - Context: We are interested in items corresponding to a condition

    SELECT * FROM Customers WHERE Country='Mexico'
  - Solution with columnar storage
    - Check the condition by reading only the corresponding column
    - Store a summary (min, max, etc) of the column at the beginning of each partition to fully skip reading when possible

# Write Operations

- Updates in place would be tricky and slow
  - Columns are compressed
  - We need to keep the ordering of rows
- Use a LSM-tree approach
  - Accumulate data in memory
  - Write in bulk to the storage

# File Formats on Disks

- One representation in memory to suit processing
- Another representation on disks (file formats) to store and access data efficiently
- To pass from the first to the second and vice versa:
    - encoding/decoding
    - marshalling/unmarshalling
    - serialization/deserialization

File formats in Big Data

- CSV
- JSON
- Avro (JSON-based for Hadoop)
- Parquet (efficient columnar data representation for Hadoop)
- ORC
- ...

Several formats are supported by:

- Distributed file systems (eg, HDFS)
- NoSQL databases (eg, MongoDB)
- Data processing engines (eg, Spark)

# Text Files

- Examples of such formats
  - CSV
  - JSON
  - XML
- Advantages
  - Readable by humans
- Drawbacks
  - High storage footprint
  - Very low read performance

# Binary Encoding Formats

- Examples
  - Avro (Hadoop)
  - Thrift (Facebook)
  - Protocol Buffers (Google)
- Idea
  - Describe the data using a schema
  - Pack all fields describing an item (a row) in a binary format
- Advantages
  - Can lead to huge space reduction

# Protocol Buffers (Google)

*JSON*

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

**81 bytes**

*Protocol Buffers*

Schema IDL

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

see result next slide

# Protocol Buffers Example (cont.)

Byte sequence (59 bytes):

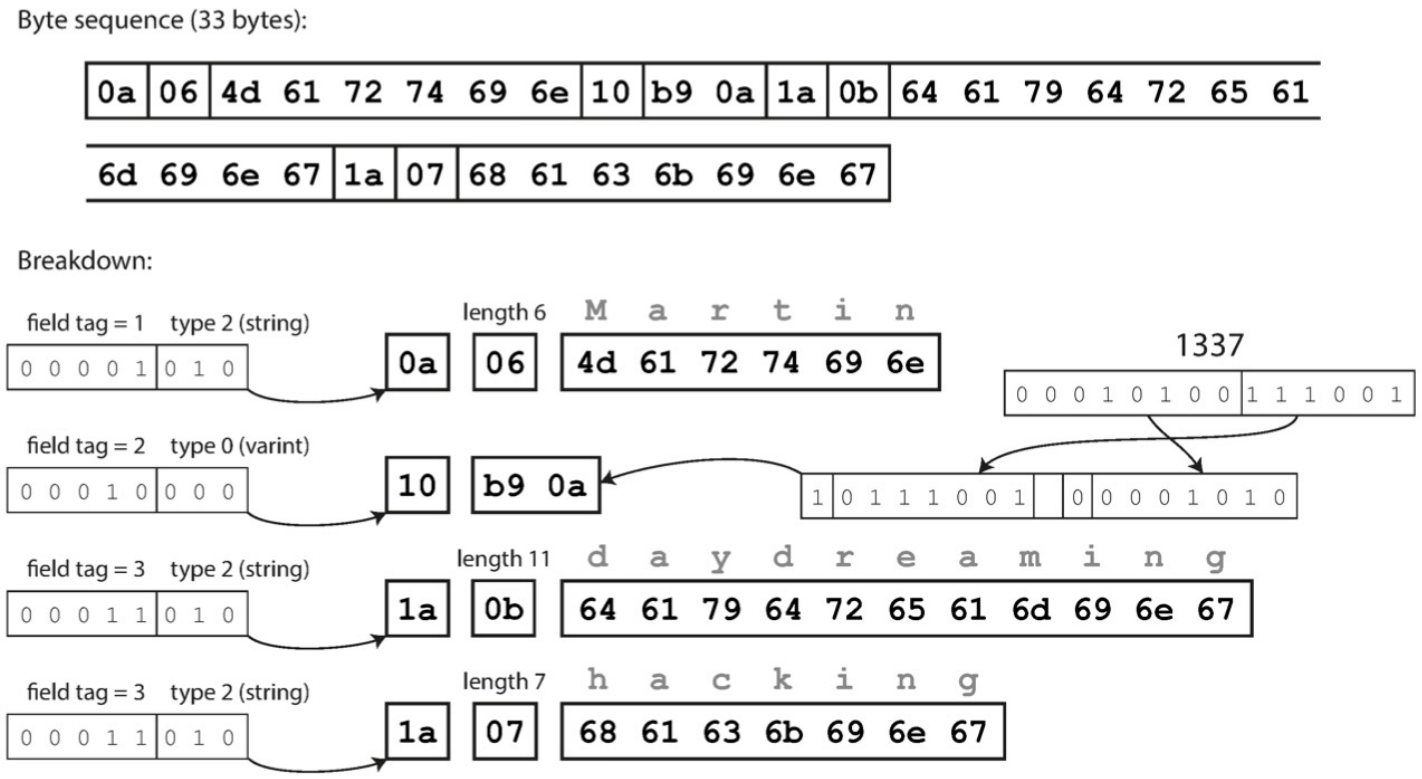
```
0b 00 01 00 00 00 06 4d 61 72 74 69 6e 0a 00 02 00 00 00 00
00 00 05 39 0f 00 03 0b 00 00 00 02 00 00 00 0b 64 61 79 64
72 65 61 6d 69 6e 67 00 00 00 07 68 61 63 6b 69 6e 67 00
```

**59 bytes**

Breakdown:

| | | | |
|---|---|---|---|
| type 11 (string) | field tag = 1 | length 6 | M a r t i n |
| 0b | 00 01 | 00 00 00 06 | 4d 61 72 74 69 6e |
| type 10 (i64) | field tag = 2 | 1337 | |
| 0a | 00 02 | 00 00 00 00 00 00 05 39 | |
| type 15 (list) | field tag = 3 | item type 11 (string) | 2 list items |
| 0f | 00 03 | 0b | 00 00 00 02 |
| | | length 11 | d a y d r e a m i n g |
| | | 00 00 00 0b | 64 61 79 64 72 65 61 6d 69 6e 67 |
| | | length 7 | h a c k i n g — end of struct |
| | | 00 00 00 07 | 68 61 63 6b 69 6e 67 — 00 |

# Compact Protocol Buffers Example (cont.)




**33 bytes**

# Parquet (column-oriented)

- A file consists of one or more row groups
  - A set of rows
- A row group contains exactly one column chunk per column
  - A column chunk is contiguous in the file
- Metadata are stored at the end of the file
  - Position of each column chunk
  - Statistics about each chunk
    - Min/Max statistics for numbers
    - Dictionary filtering for other columns (as long as less than 40k different values)

# Example

**https://blog.usejournal.com/sorting-and-parquet-3a382893cde5**

- Description of the data
  - Customer table with one column being the country

    SELECT * FROM Customers WHERE Country='Mexico'
  - Some numbers:
    - 10M rows
    - 10k rows per row group
    - 1% of the customers are from Mexico
- Amount of data read to answer the query
  - With a row-based format: All data
  - With an unsorted parquet file:
    - Probability of a row group with no customer from Mexico: $0.99^{10000} = 2.25 \times 10^{-32}$
    - So small that practically all row groups need to be read
- With a sorted parquet file: 1% of the row groups (10)

# Another example

**https://www.linkedin.com/pulse/ we-taking-only-half-advantage-columnar-file-format-eric-sun/**

- Description of the data
  - A website log dataset
  - Information in one entry:

    `timestamp, user_id, cookie, page_id, http_header, ...` •
  - Queries filter against page_id
- Some results
  - Avro:
    - Compressed data footprint: 1.4 TB
    - Amount of data read on query: 1.4 TB
  - ORC unsorted/sorted:
    - Compressed data footprint: 0.9 TB / 0.5 TB
    - Amount of data read on query: 300 GB / 200 MB