Machine Learning Systems Design

Data Lifecycle

Lecture 5: Data Engineering Fundamentals

CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io

Agenda

1. What is data engineering?
2. Data sources
3. Data models
4. Data storage
5. Data format
6. Data flow

4. Data storage
and retrieval

Data storage engines

There are two families of storage engines:

- Log-structured
- Page-oriented (B-trees)

Log-structured storage engine

The world’s simplest database, implemented as two Bash functions:

```bash
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Log-structured storage engine

and it works!

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Log-structured storage engine

But, old versions of the values are not overwritten

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Time and space complexity

	`db_set`	`db_get`
time complexity	?	?
space complexity	?	?

Time and space complexity

	`db_set`	`db_get`
time complexity	$O(1)$	$O(N)$
space complexity	$O(k)$	$O(1)$

Time and space complexity

	`db_set`	`db_get`
time complexity	$O(1)$	$O(N)$
space complexity	$O(k)$	$O(1)$

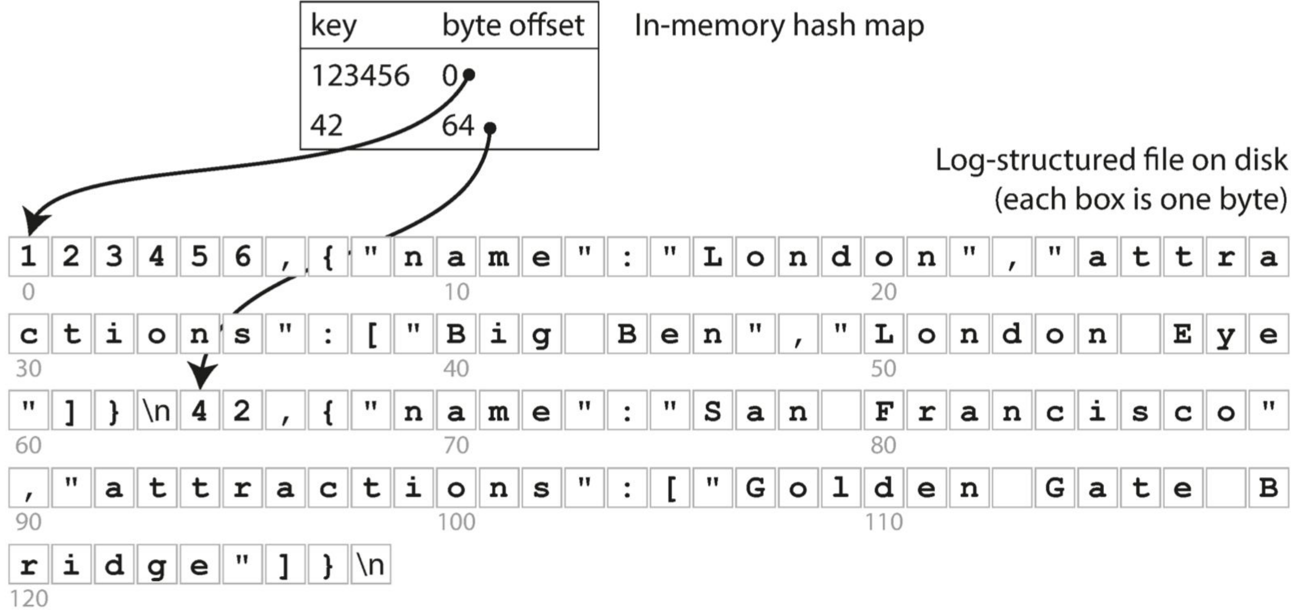
Time and space complexity

	`db_set`	`db_get`	
time complexity	$O(1)$	$O(N)$	index
space complexity	$O(k)$	$O(1)$	

Time and space complexity

	`db_set`	`db_get`	
time complexity	$O(1)$	$O(N)$	Index (hashmap) → $O(1)$
space complexity	$O(k)$	$O(1)$	

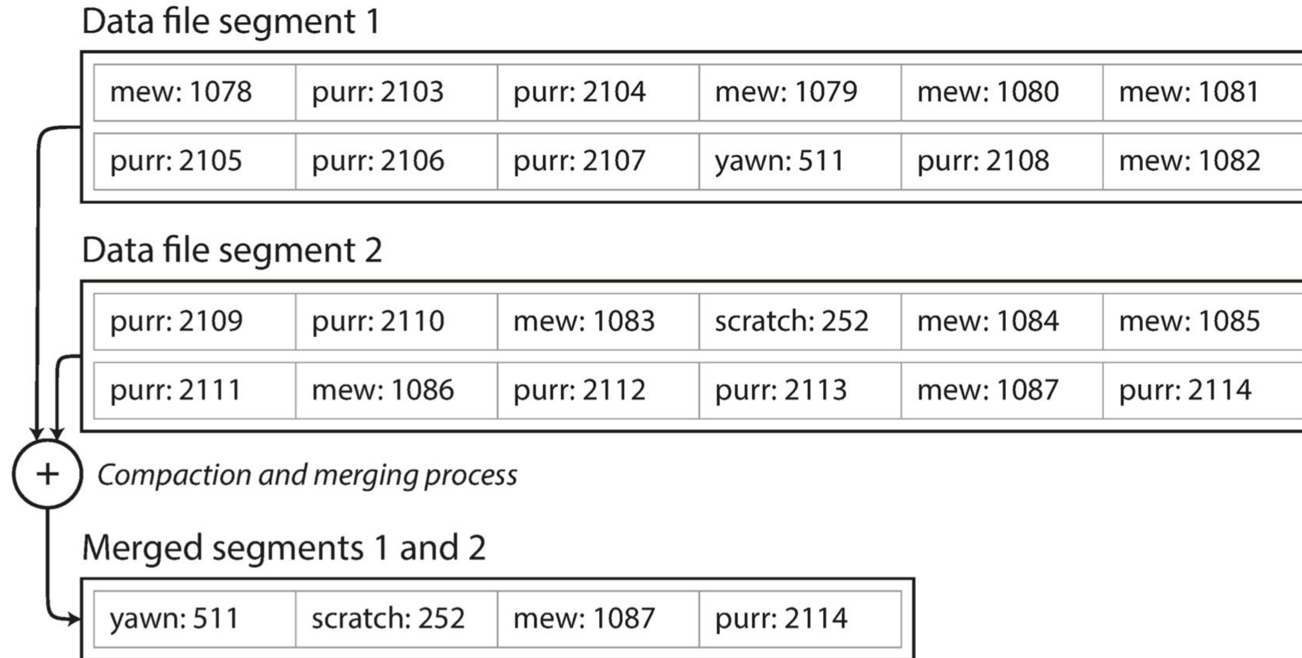
Hashmap index



Time and space complexity

	`db_set`	`db_get`	
time complexity	$O(1)$	$O(N)$	Index (hashmap) $\rightarrow O(1)$
space complexity	$O(k)$	$O(1)$	
	Compaction $\rightarrow O(1)$		

Segmenting, compaction and merge



Practical implementation

Lots of detail goes into making this simple idea work in practice…

- File format
- Deleting records
- Crash recovery
- Partially written records
- Concurrency control

SSTable and LSM-Tree

We can make a simple change to the format of our segment files: we require that the sequence of key-value pairs is sorted by key. We call it Sorted String Table (SSTable). Advantages are:

- Simple and efficient merge (mergesort)
- No need to keep an index of all the keys in memory
- We can have block compression

SSTable and LSM-Tree

Sparse index in memory

key	byte offset
...	...
hammock	100491
handbag	102134
handsome	104667
hangout	106812
...	...

Sorted segment file (SSTable) on disk

.... hand: 91541

handbag: 8786 | handcuffs: 2729 | handful: 44662

handicap: 70836 | handiwork: 45521 | handkerchief: 20952

handlebars: 3869 | handoff: 5741 | handprinted: 33632

compressible block

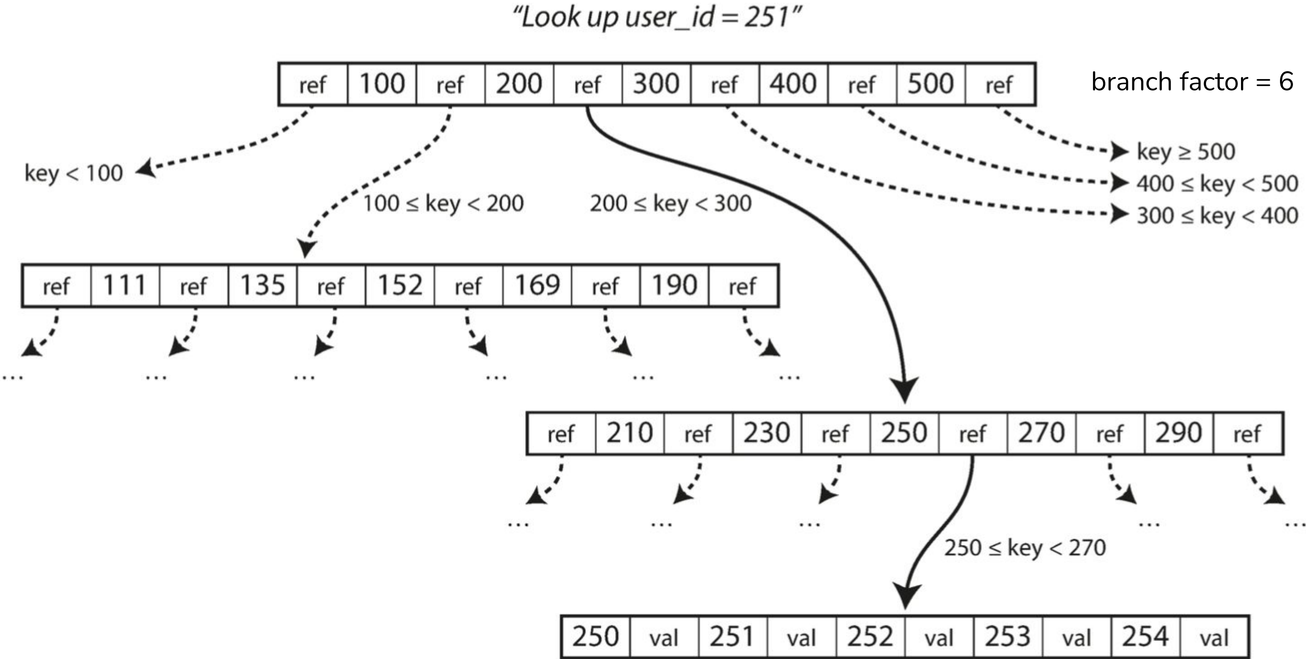
handsome: 86478 | handwaving: 44005 | handwriting: 22846

....

Practical implementation

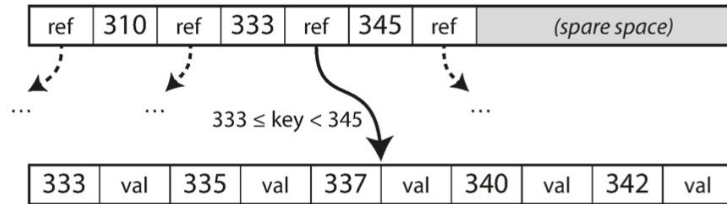
- Constructing and maintaining SSTables
- Making an LSM-tree out of SSTables
- Performance optimizations

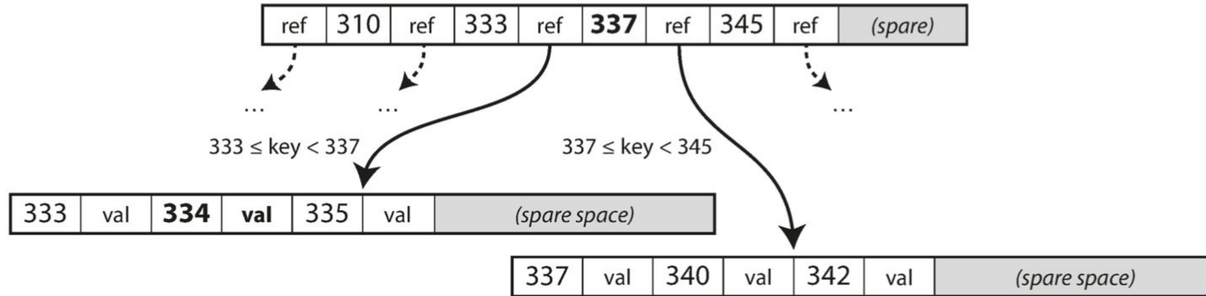
Page-oriented storage engine (B-Trees)



Looking up a key using a B-tree index

Page-oriented storage engine (B-Trees)





Growing a B-tree by splitting a page

Practical implementation

- Making B-trees reliable
- B-tree optimizations

Other Indexing Structures

- Storing values within the index
- Multi-column indexes
- Full-text search and fuzzy indexes
- Keeping everything in memory

Data Storage Engines & Processing

Databases optimized for

Transactional processing

Analytical processing

OnLine Transaction Processing (OLTP)

- Transactions: tweeting, ordering a Lyft, uploading a new model, etc.
- Operations:
    - Insert when generated
    - Occasional update/delete

OnLine Transaction Processing

- Transactions: tweeting, ordering a Lyft, uploading a new model, etc.
- Operations:
  - Inserted when generated
  - Occasional update/delete
- Requirements
  - Low latency
  - High availability

OnLine Transaction Processing

- Transactions: tweeting, ordering a Lyft, uploading a new model, etc.
- Operations:
  - Inserted when generated
  - Occasional update/delete
- Requirements
  - Low latency
  - High availability
  - ACID not necessary
    - **Atomicity**: all the steps in a transaction fail or succeed as a group
      - If payment fails, don't assign a driver
    - **Isolation**: concurrent transactions happen as if sequential
      - Don't assign the same driver to two different requests that happen at the same time

See ACID:
Atomicity,
Consistency,
Isolation,
Durability

OnLine Transaction Processing

- Transactions: tweeting, ordering a Lyft, uploading a new model, etc.
- Operations:
  - Inserted when generated
  - Occasional update/delete
- Requirements
  - Low latency
  - High availability
- Typically row-major

Row

```
INSERT INTO RideTable(RideID, Username, DriverID, City, Month, Price)
VALUES ('10', 'memelord', '3932839', 'Stanford', 'July', '20.4');
```

OnLine Analytical Processing (OLAP)

- How to get aggregated information from a large amount of data?
  - e.g. what's the average ride price last month for riders at Stanford?
- Operations:
  - Mostly SELECT

OnLine Analytical Processing

- Analytical queries: aggregated information from a large amount of data?
  - e.g. what’s the average ride price last month for riders at Stanford?
- Operations:
  - Mostly SELECT
- Requirements:
  - Can handle complex queries on large volumes of data
  - Okay response time (seconds, minutes, even hours)

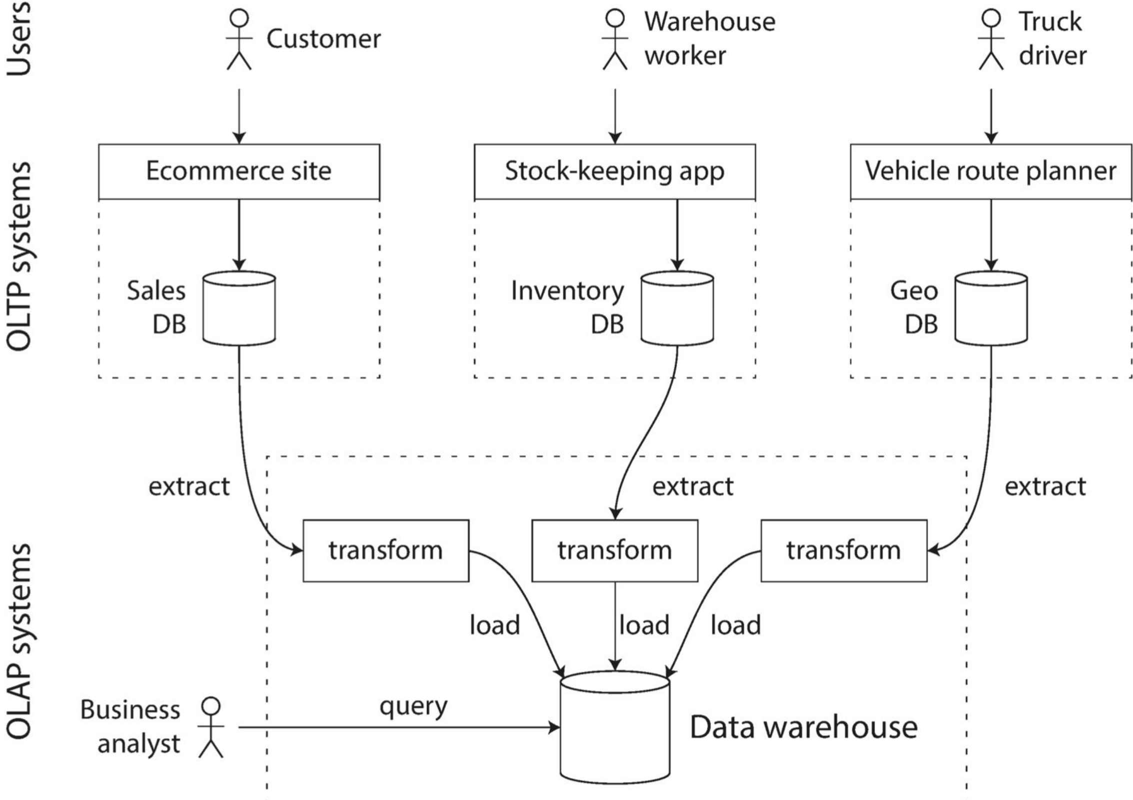
OnLine Analytical Processing

- Analytical queries: aggregated information from a large amount of data?
  - e.g. what's the average ride price last month for riders at Stanford?
- Operations:
  - Mostly SELECT
- Requirements:
  - Can handle complex queries on large volumes of data
  - Okay response time (seconds, minutes, even hours)
- Typically column-major

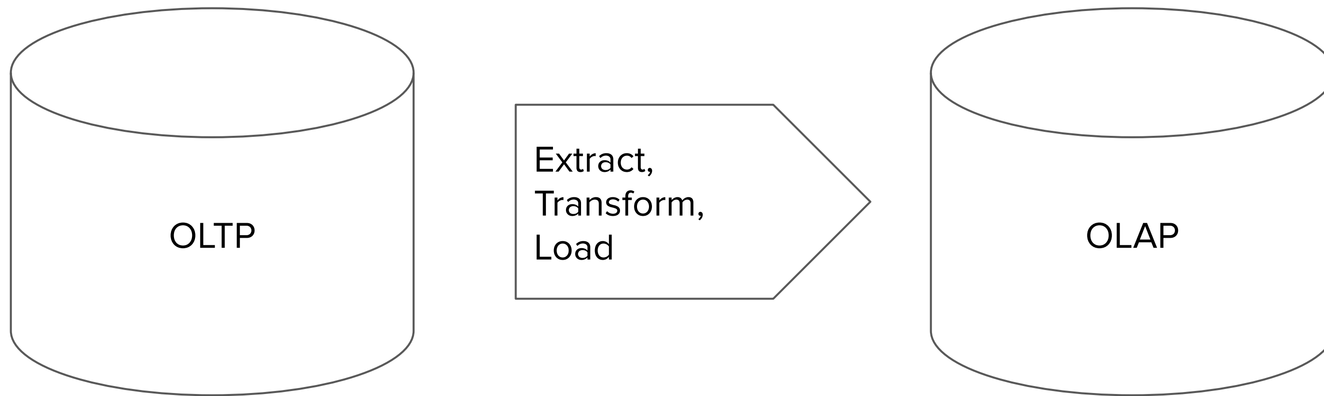
Column

```
SELECT AVG(Price)
FROM RideTable
WHERE City = 'Stanford' AND Month = 'July';
```

Data warehousing



ETL (Extract, Transform, Load)



Transform: the meaty part

- cleaning, validating, transposing, deriving values, joining from multiple sources, deduplicating, splitting, aggregating, etc.

ETL -> ELT

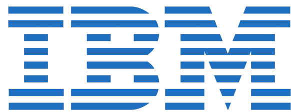
Structured -> unstructured -> structured

want more flexibility | tools & infra standardized

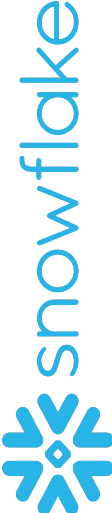
ETL -> ELT -> ETL

Decoupling storage & processing

- OLTP & OLAP: how data is stored is also how it's processed
  - Same data being stored in multiple databases
  - Each uses a different processing engine for different query types
- New paradigm: storage is decoupled from processing
  - Data can be stored in the same place
  - A processing layer on top that can be optimized for different query types



Decoupling storage & processing



Cloud Services Layer

Authentication, Optimizer, Metadata Manager, Security etc.

Compute Layer
or
Query Processing Layer

Virtual Warehouse

Virtual Warehouse

Storage Layer

Database

Database

Database

https://hevodata.com/blog/snowflake-architecture-cloud-data-warehouse/

5. Data format

How to store your data?

Programs usually work with data

- in memory, or
- over the network

thus, we need some kind of translation between the two representations.

- storing data: **encoding**/serialization/marshalling
- unloading data: **decoding**/deserialization/unmarshalling/parsing

How to store your data?

Data formats are
agreed upon standards
to serialize your data so that
it can be transmitted & reconstructed later

Data formats: questions to consider

- How to store multimodal data?
  - `{‘image’: [[200,155,0], [255,255,255], ...], ‘label’: ‘car’, ‘id’: 1}`
- Access patterns
  - How frequently the data will be accessed?
- The hardware the data will be run on
  - Complex ML models on TPU/GPU/CPU

Formats for Encoding Data

- Language-specific: *pickle*
- Language independent: *JSON*, *XML*, *and binary variants*
- Thrift and Protocol Buffers
- Avro

Data formats

	Format	Binary/Text	Human-readable	Example use cases
	JSON	Text	Yes	Everywhere
	CSV	Text	Yes	Everywhere
	Parquet	Binary	No	Hadoop, Amazon Redshift
	Avro	Binary primary	No	Hadoop
	Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Language specific	Pickle	Binary	No	Python, PyTorch serialization

Data formats

Language independent

Format	Binary/Text	Human-readable	Example use cases
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Binary	No	Python, PyTorch serialization

The difficulty of getting different organizations to agree on *anything* outweighs most other concerns.

MessagePack: a binary encodings for JSON

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

81 bytes to encode by the textual JSON encoding (with whitespace removed)

MessagePack: a binary encodings for JSON

Byte sequence (66 bytes):

83 a8 75 73 65 72 4e 61 6d 65 a6 4d 61 72 74 69 6e ae 66 61
76 6f 72 69 74 65 4e 75 6d 62 65 72 cd 05 39 a9 69 6e 74 65
72 65 73 74 73 92 ab 64 61 79 64 72 65 61 6d 69 6e 67 a7 68
61 63 6b 69 6e 67

Breakdown:

object (3 entries) 83

string (length 8) a8 — u s e r N a m e: 75 73 65 72 4e 61 6d 65

string (length 6) a6 — M a r t i n: 4d 61 72 74 69 6e

string (length 14) ae — f a v o r i t e N u m b e r: 66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72

uint16 cd — 1337: 05 39

string (length 9) a9 — i n t e r e s t s: 69 6e 74 65 72 65 73 74 73

array (2 entries) 92

string (length 11) ab — d a y d r e a m i n g: 64 61 79 64 72 65 61 6d 69 6e 67

string (length 7) a7 — h a c k i n g: 68 61 63 6b 69 6e 67

66 bytes long binary encoding with MessagePack

Data formats

Apache Thrift (Facebook) and Protocol Buffers (Google) are binary encodings that are based on the same principle

Format	Binary/Text	Human-readable	Example use cases
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Binary	No	Python, PyTorch serialization

Protocol Buffers schema

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

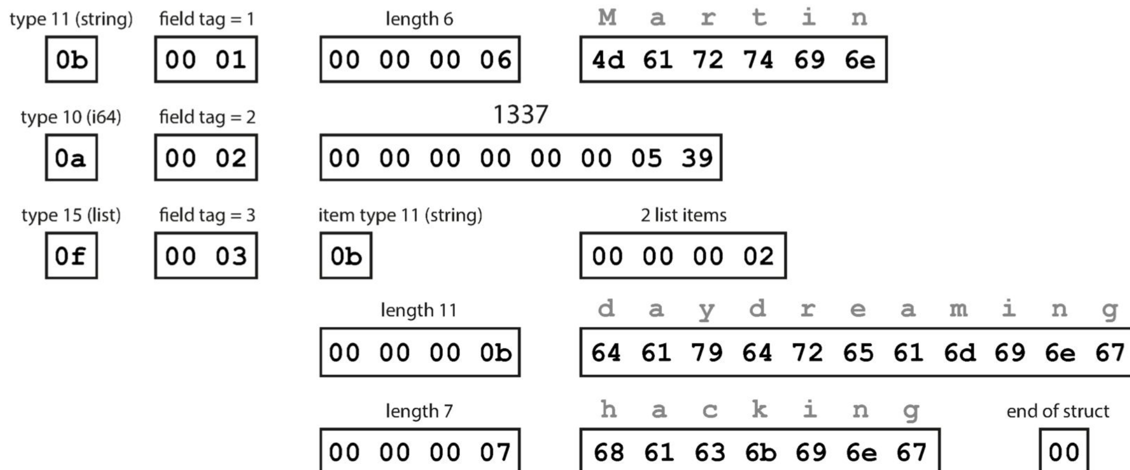
Protocol Buffers binary encoding

Thrift BinaryProtocol

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

Breakdown:



59 bytes long binary encoding with BinaryProtocol

and **34** bytes with CompactProtocol

Protocol Buffers schema evolution

How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?

Protocol Buffers schema evolution

How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?

Forward compatibility:

- You can change name of a field in the schema but cannot change a field's tag
- You can add new fields (with new tags) to the schema

Protocol Buffers schema evolution

How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?

Backward compatibility:

- Every field added after the initial deployment of the schema must be optional or have a default value.
- You can only remove optional fields

Protocol Buffers schema evolution

What about data types change?

Data formats

As a result of Thrift not being a good fit for Hadoop's use cases

Format	Binary/Text	Human-readable	Example use cases
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Binary	No	Python, PyTorch serialization

Data formats

	Format	Binary/Text	Human-readable	Example use cases
	JSON	Text	Yes	Everywhere
Row-major	CSV	Text	Yes	Everywhere
Column-major	Parquet	Binary	No	Hadoop, Amazon Redshift
	Avro	Binary primary	No	Hadoop
	Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
	Pickle	Binary	No	Python, PyTorch serialization

Row-major vs. column-major

Column-major:

- stored and retrieved column-by-column
- good for accessing features

	Column 1	Column 2	Column 3
Sample 1	...	...	...
Sample 2	...	...	...
Sample 3	...	...	...

Row-major:

- stored and retrieved row-by-row
- good for accessing samples

Row-major vs. column-major: DataFrame vs. ndarray

Pandas DataFrame: column-major

- accessing a row much slower than accessing a column and NumPy

```
# Get the column `date`, 1000 loops
%timeit -n1000 df["Date"]

# Get the first row, 1000 loops
%timeit -n1000 df.iloc[0]
```

```
1.78 µs ± 167 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
145 µs ± 9.41 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

NumPy ndarray: row-major by default

- can specify to be column-based

```
df_np = df.to_numpy()
%timeit -n1000 df_np[0]
%timeit -n1000 df_np[:,0]
```

```
147 ns ± 1.54 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
204 ns ± 0.678 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```


https://github.com/chiphuyen/just-pandas-things

Text vs. binary formats

	Text files	**Binary files**
Examples	CSV, JSON	Parquet
Pros	Human readable	Compact, schema as doc, forward/backward compatible
Store the number *1000000*?	7 characters -> 7 bytes	If stored as int32, only 4 bytes

You can unload the result of an Amazon Redshift query to your Amazon S3 data lake in Apache Parquet, an efficient open columnar storage format for analytics. Parquet format is up to 2x faster to unload and consumes up to 6x less storage in Amazon S3, compared with text formats. This enables you to save data transformation and enrichment you have done in

6. Data flow

How data flows?

The most common ways how data flows between processes:

- Via **databases**
- Via **service calls**
- Via **asynchronous message passing**

Data flow through databases

In a database, the process that writes to the database encodes the data, and the process that reads from the database decodes it.

We should also have backward and forward compatibility.

Data outlives code: Different values written at different times

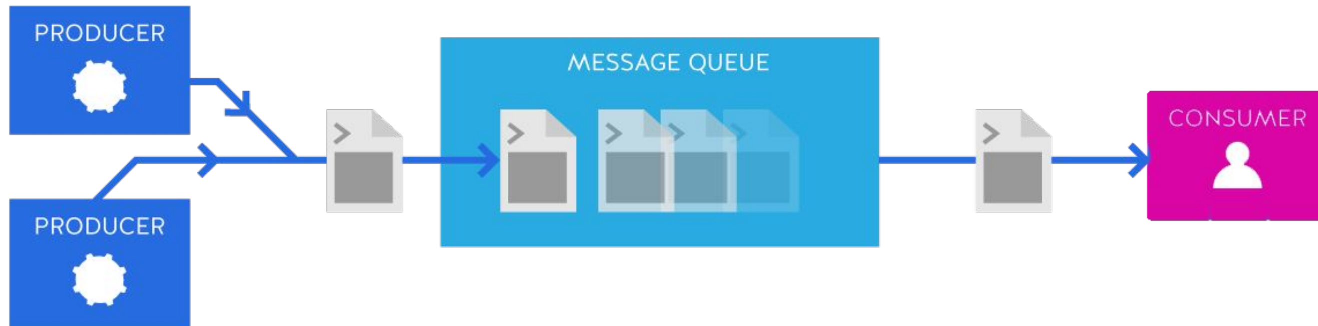
Data flow through services: REST and RPC

The **web** works this way:

clients (web browsers) make requests to web servers.

Message-passing data flow

Messages are encoded by the sender
and decoded by the recipient



One process sends a message to a named *queue* or *topic*, and the broker ensures that the message is delivered to one or more *consumers* of or *subscribers* to that queue or topic.

Machine Learning Systems Design

Data Lifecycle

Next Lecture: Data Preparation

CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io