NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE | NTU Academy for Professional and Continuing Education

(SCTP) Advanced Professional Certificate

Data Science and AI

2.3 Data Encoding and Data Flow

Module Overview

2.1 Introduction to Big Data and Data Engineering

2.2 Data Architecture

2.3 Data Encoding and Data Flow

2.4 Data Extraction and Web Scraping

2.5 Data Warehouse

2.6 Data Pipelines

2.7 Testing and Data Orchestration

2.8 Out of Core/Memory Processing

2.9 Big Data Ecosystem and Batch Processing

2.10 Event Streaming and Stream Processing

Learning Objective

By the end of this module, you will be able to

- **Select** appropriate **data encoding formats** and **data flow mechanisms** based on business scenarios
- **Encode and decode** data in various formats
- Exchange data via **RPC**

...and able to Join the conversation at the water cooler

Alice: Hey Bob, did you hear about the **REST API** that went to therapy? It had too many status issues and couldn't maintain a stable relationship.

Bob: Speaking of relationships, I just migrated our service from REST to gRPC and the performance boost is like going from a bicycle to a rocket ship.

Alice: Oh, you're one of *those* people now - next you'll be telling me about how Protocol Buffers changed your life.

Charlie: *joins conversation* You're both wrong - Event-driven architecture with Kafka is clearly superior, it's like gossiping - fire and forget!

Alice: Yeah, until you need to debug a production issue and realize your data is flowing through seventeen different microservices like a game of telephone.

Bob: *pulls out laptop* Let me show you this encoding benchmark I ran last night comparing Avro, Arrow and Protocol Buffers.

Agenda

Data Encoding Formats

- JSON
- XML
- Apache Thrift
- Protocol Buffers
- Apache Avro
- Apache Parquet
- Apache ORC
- Apache Arrow

Modes of Data Flow

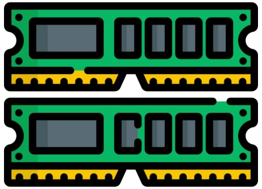
- Databases
- Service calls
  - Rest API
  - GraphQL
  - RPC
- Asynchronous message passing
  - Message queue
  - Publish-Subscribe (Pub-sub)

Data Encoding

Programs handle data in two primary formats:

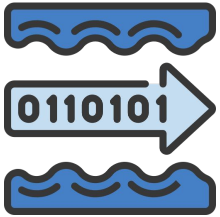
In-Memory Representation:

Data is structured (objects, arrays, etc.) for fast CPU access using pointers.

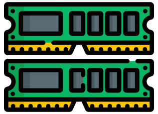


Serialized Representation:

For storage or transfer, data is converted into a byte sequence (JSON, binary) as pointers are not valid across processes.

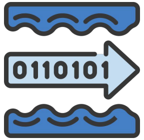


Data Encoding



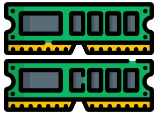
Encoding
Serialization
Marshalling

Converting in-memory data to a byte sequence.



Decoding/Parsing
Deserialization
Unmarshalling

Converting the byte sequence back to in-memory data.



Data needs to be transformed between in-memory structures and serialized formats for tasks like saving to files or sending over networks.

Data Encoding Formats

Data encoding is a common problem, leading to many different libraries and formats.

Language-specific formats have drawbacks:

- Tied to a specific programming language, **hindering** cross-language data reading.
- Decoding may require instantiating arbitrary classes, posing **security** risks.
- Versioning is often overlooked, creating forward and backward **compatibility issues**.
- Potential **inefficiency** in CPU time and encoded data size.

Formant Name	Where	Description	Format Type
Pickle	Module 1	Python language-specific format for encoding in-memory objects into byte sequences	Binary
CSV	Module 1	Comma Separated Value	Textual
JSON	Unit 2.1	JavaScript Object Notation	Textual
XML	Self-studies	eXtensible Markup Language	Textual

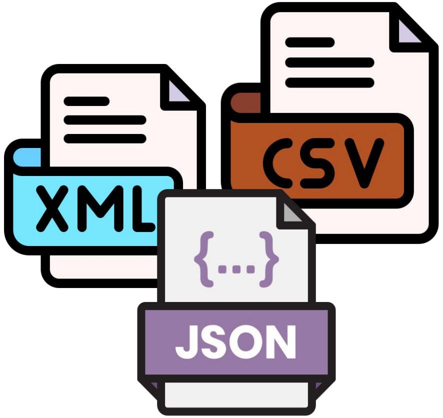
Binary > Textual Format

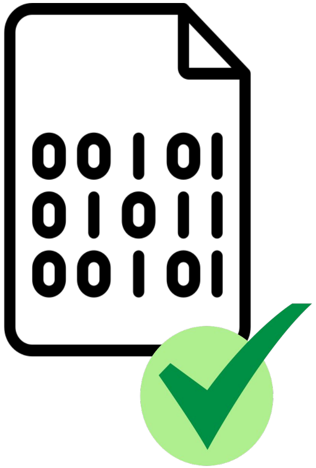
Data Type Clarity

Textual formats (XML, CSV) may not distinguish between numbers and digit strings.

JSON distinguishes but lacks precision for integers/floats, often requiring an external schema.

Binary formats can represent data types precisely.





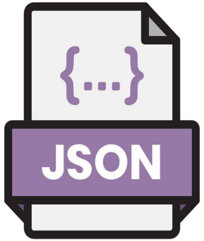
Binary > Textual Format

json

```json
{
  "user": "john_doe",
  "age": 28,
  "email": "john@example.com",
  "active": true
}
```

xml

```xml
<user>
  <name>john_doe</name>
  <age>28</age>
  <email>john@example.com</email>
  <active>true</active>
</user>
```

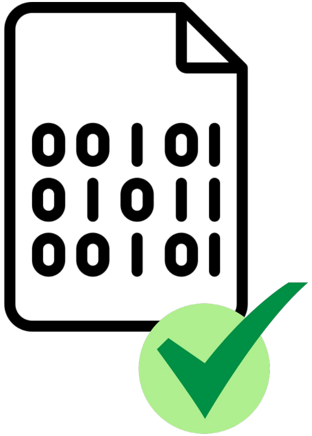


Binary Data Handling

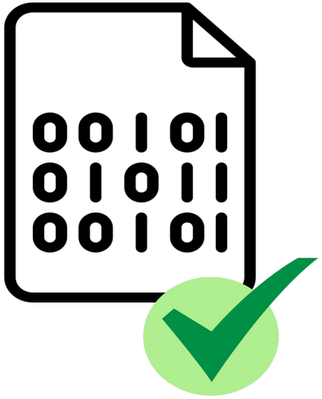
JSON and XML are designed for Unicode strings, not raw binary data.

Base64 encoding is a common but inefficient workaround, increasing data size by 33%.

Binary formats handle binary data directly.



Binary > Textual Format



Schema Support

Textual formats often lack robust schema support.

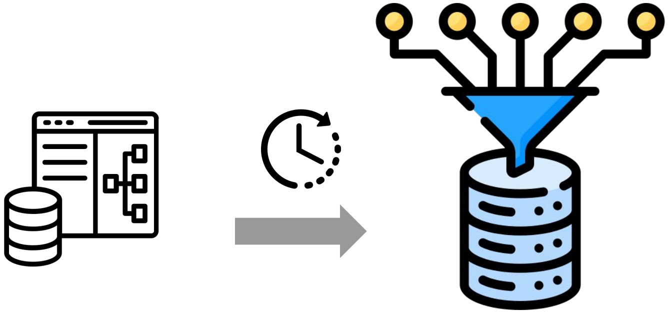
While XML has widely used schemas (XSD, DTD), JSON schema usage is inconsistent.

Without a schema, data interpretation can be difficult.

Binary formats often include or require schemas, ensuring data is interpreted correctly.

Schema Evolution

- Build systems that **adapt** to change easily.
- Application feature changes often require data format or schema changes.
- Relational databases assume one schema at a time, changed through migrations (ALTER statements)
- NoSQL databases that don't enforce a schema (*schemaless*) can contain a mixture of older and newer data formats.

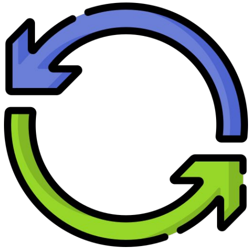


In the **Ingestion** stage, code changes are needed when data format or schema changes but cannot happen instantaneously.

For example, when a new field is added to a record, your code starts reading and writing that field.

Schema Evolution

Old and new versions of code and data, formats may potentially coexist. Compatibility is needed in both directions:



Backward Compatibility

Newer code can read data written by older code.

This is common when ingesting older data using newer code.

Achieved by handling known older formats or retaining old code.

Forward Compatibility

Older code can read data written by newer code.

This is common when ingesting newer data before code changes are deployed.

Can be tricky, as code must ignore new changes.

Code-along

Jupyter Notebook

Binary Formats

Binary Formats

We will now explore some popular binary encoding formats.

- Apache Thrift
- Protocol Buffers
- Apache Avro
- Apache Parquet
- Apache ORC
- Apache Arrow

Open-source frameworks, mostly being developed by the *Apache Software Foundation.*

The main advantages of binary formats are:

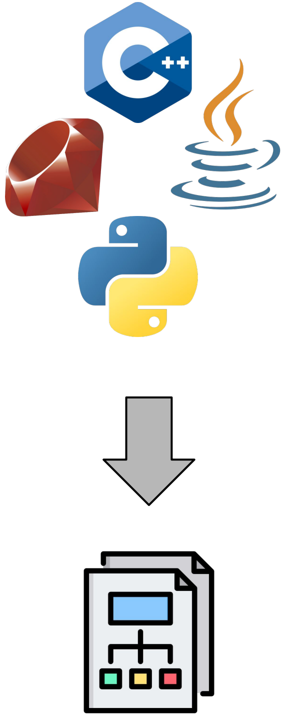
- **Efficient and compact**: The encoded data is smaller and faster to serialize/deserialize compared to textual formats like JSON or XML.
- **Schema evolution**: Allow forward and backward compatibility when extending or modifying the schema, suitable for evolving systems.

Apache Thrift

- Thrift is an **interface definition language** and **binary communication protocol** used for defining and creating services for programming languages, e.g. C++, Java, Python, Ruby.
- Define data types and service interfaces in a simple definition file.
- The compiler generates code from the input file to build RPC clients and servers that **communicate seamlessly between programming languages**.
- Originally developed at Facebook.

Protocol Buffers (Protobufs)

- **Cross-platform** data format used to serialize structured data with **speed prioritization**.
- Like Thrift, it support various programming languages, enabling **interoperability between services written in different languages**.
- Define a **schema** in a *.proto* file, specifying data structure and fields, with **backward & forward** schema compatibility.
- The file is compiled by the **Protobufs** compiler to generates source code in target languages.
- Generated code provides classes/structs for data manipulation and serialization/deserialization.
- Originally developed at Google.



schema.proto

Apache Avro

- **Language-agnostic** and **platform-neutral** system for efficient structured data serialization/deserialization.
- Aims for compact, fast serialization format for **data exchange** between systems/services.
- Uses a schema to specify data structure, with human-editable IDL and machine-readable JSON variants.
- Allows data serialization **with** or **without** a schema, enabling **dynamic typing** and self-describing data.
- **Native integration** with big data frameworks like **Hadoop, Spark, and Kafka**.

Apache Parquet

- **Column-oriented** data storage format designed for efficient data storage and processing.
- Optimized for analytics workloads, reading/processing specific columns.
- Unlike **row-based** *Thrift*, *Protobuf* and *Avro*, it stores column values together.
- Language-agnostic, usable with various languages and big data frameworks (Hadoop, Spark, etc.).
- Parquet file is an *Hadoop Distributed File System* (.hdfs*)* file that includes metadata that allows column splitting across files.
- Metadata contains data schema that references multiple files.

Apache ORC

- Optimized Row Columnar (ORC) is a **column-oriented** file format for storing large-scale, structured data in a highly efficient and optimized manner.
- Like *Parquet*, ORC is optimized for big data processing and analytics workloads.
- Specifically designed for use with *Apache Hive*, but it can also be used with other big data frameworks like Spark and Impala.
- ORC file footer contains the type schema information, the number of rows, and the statistics about each of the columns.

Apache Arrow

- A cross-language platform for **in-memory** analytics.
- Defines **language-independent columnar memory** format for flat and hierarchical data, optimized for efficient analytics on modern hardware.
- Supports zero-copy reads for fast data access without serialization overhead.
- Used by many popular projects to efficiently ship columnar data or as basis for analytic engines.
- Libraries available for C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust.

https://www.menti.com/alutu5836gw1

Menti-moment…

Instructions

Go to

www.menti.com

Enter the code

1952 2786

Or use QR code

Business Decision Framework for Data Encoding

Business Need	Best Format Type	Top Options
High-Performance Processing	Binary	Protocol Buffers, Apache Arrow
Analytics & Reporting	Columnar Binary	Parquet, ORC
Cross-System Integration	Schema-Based Binary	Avro, Thrift
Human Readability & Debugging	Textual	JSON, XML
Schema Evolution & Compatibility	Self-Describing	Avro (both ways), Protobuf (backward)
Security & Untrusted Data	Schema-Enforced	Protocol Buffers, Avro

Modes of Data Flow

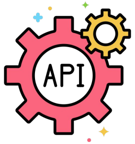
Data needs encoding to move between processes without shared memory

Databases

Writers encode

Readers decode

Backward and forward compatibility

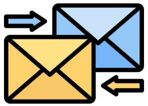


Service Calls

REST API

RPC

GraphQL



Messages

Message broker for async. communication.

Message queues (FIFO)

Publish-subscribe (broadcast)

https://www.youtube.com/watch?v=hltLrjabkiY

Service Calls: REST, RPC and GraphQL

REST

REpresentational **S**tate **T**ransfer

Uses standard HTTP methods and formats like JSON or XML.

Focus on resources and stateless communication

Ideal for web-based applications

RPC

Remote **P**rocedure **C**all

Emphasizes executing remote functions

Relies on specific binary encoding using IDL *(Interface Definition Language)*

Good for performance

GraphQL

Graph Query **L**anguage

Allows clients to request precise data

Reduces over/under-fetching

Single end point

Good for complex data and client driven data retrieval

What is gRPC?

A modern, high-performance implementation of RPC developed by Google. Designed for efficient communication in distributed systems and uses Protocol Buffers (Protobufs) for data serialization.

Cross-language support: gRPC enables seamless communication between applications written in different programming languages.

High efficiency: Using Protobufs (a compact binary format), it achieves better speed and smaller message sizes compared to JSON or XML used in traditional RPC.

Streaming capabilities: gRPC supports bi-directional streaming, allowing both the client and server to send data continuously.

HTTP/2 protocol: Leveraging HTTP/2 for transport ensures features like multiplexing, better flow control, and reduced latency.

RPC and Data Encoding

RPC often relies on specific data encoding formats to ensure that data is transmitted and interpreted correctly between the client and server.

Protobufs

Avro

Apache Thrift™

Message Passing

- Asynchronous communication method bridging RPC and databases
- Sender does not wait for reply
- Messages (requests) are delivered with low latency, similar to RPC
- Messages are sent via a message broker, which temporarily stores them, unlike direct RPC

Two main types:

- **Message queues**: Direct communication, FIFO *(First-In, First-Out)* processing
- **Publish-Subscribe**: Indirect communication, message broadcasting

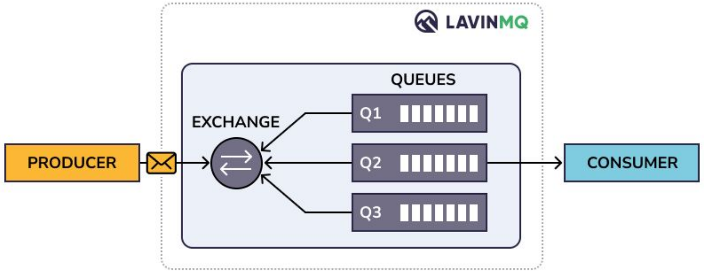
Message Queues

Direct Communication: Messages are sent directly from a sender (producer) to a queue and then retrieved by a receiver (consumer).

FIFO Processing: Messages are processed in a "First-In, First-Out" order. The oldest message in the queue is processed first.

Single Consumer: Each message is typically processed by only one consumer. Messages are not duplicated or lost.

Decoupling: Producers and consumers are decoupled. They don't need to know each other's identities or locations. This makes the system more flexible and scalable.



Use Cases:

- Task distribution
- Event handling
- Workload management

Publish-Subscribe (Pub-Sub)

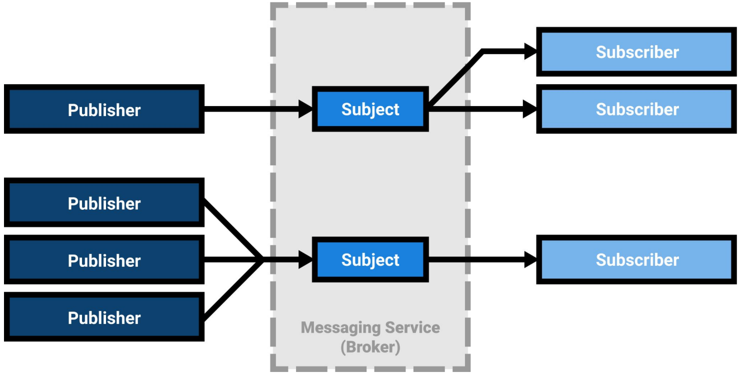
Indirect Communication: Publishers send messages to specific topics, and subscribers receive messages from the topics they are interested in.

Message Broadcasting: Each message sent to a topic is typically broadcast to multiple subscribers who are listening to that topic.

Decoupling: Publishers and subscribers are decoupled. They don't need to know each other's identities or locations. This allows them to operate independently.

Use Cases:

- Real-time event broadcasting
- Log aggregation
- Notification systems



Source: Messaging Pattern: Publish-Subscribe - A. Rothuis https://www.menti.com/alutu5836gw1

Menti-moment…

Instructions

Go to

www.menti.com

Enter the code

1952 2786

Or use QR code

Enterprise Decision Framework for Data Flow Modes

Business Need	Flow Mechanism	Top Options
Web Applications & Public APIs	REST	Standard HTTP-based APIs
High-Performance Microservices	RPC	gRPC (Google), Thrift RPC
Complex Data Retrieval	Query Language	GraphQL
Workload Distribution	Message Queue	RabbitMQ
Event Broadcasting	Publish-Subscribe	Kafka, Cloud Pub/Sub
Data Persistence & History	Database	Relational/NoSQL depending on structure
Decoupled Systems	Asynchronous Messaging	Kafka, RabbitMQ

https://www.youtube.com/watch?v=XG-EVX6PEFo

Revisiting the water cooler conversation...

Alice: Hey Bob, did you hear about the **REST API** that went to therapy? It had too many status issues and couldn't maintain a stable relationship.

Bob: Speaking of relationships, I just migrated our service from REST to gRPC and the performance boost is like going from a bicycle to a rocket ship.

Alice: Oh, you're one of *those* people now - next you'll be telling me about how Protocol Buffers changed your life.

Charlie: *joins conversation* You're both wrong - Event-driven architecture with Kafka is clearly superior, it's like gossiping - fire and forget!

Alice: Yeah, until you need to debug a production issue and realize your data is flowing through seventeen different microservices like a game of telephone.

Bob: *pulls out laptop* Let me show you this encoding benchmark I ran last night comparing Avro, Arrow and Protocol Buffers.

End of Lesson - Exit Ticket

Survey Link

https://www.menti.com