**Part 1: - Foundations of Data Systems**

**Chapter 1: - Reliable, Scalable, and Maintainable Applications**

**Thinking About Data Systems**

**Reliability**

* **Hardware Faults**
* **Software Errors**
* **Human Errors**
* **How Important is Reliability**

**Scalability**

* **Describing Load**
* **Describing performance**
* **Approaches for coping with Load**

**Maintainability**

* **Operability: Make life easy for operations**
* **Simplicity: Making Complexity**
* **Evolvability: Making Change Easy**

**Chapter 2: - Data Models and Query Languages**

**Relational Model Versus Document Model 2**

* **The Birth of NoSQL**
* **The Object-Relational Mismatch**
* **Many-to-One and Many-to-Many Relationships**
* **Are Document Databases Repeating History**
* **Relational Versus Document Databases Today**

**Query Languages for Data**

* **Declarative Queries on the Web**
* **MapReduce Querying**

**Graph-Like Data Models**

* **Property Graphs**
* **The Cypher Query Language**
* **Graph Queries in SQL**
* **Triple-Stores and SPARQL**
* **The Foundation: Datalog**

**Chapter 3: - Storage and Retrieval**

**Data Structures That Power Your Database**

* **Data Structures That Power Your Database**
* **Hash Indexes**
* **SSTables and LSM-Trees**
* **B-Trees**
* **Comparing B-Trees and LSM-Trees**
* **Other Indexing Structures**

**Transaction Processing or Analytics**

* **Data Warehousing**
* **Stars and Snowflakes: Schemas for Analytics**

**Column-Oriented Storage**

* **Column Compression**
* **Sort Order in Column storage**
* **Writing to Column-Oriented Storage**
* **Aggregation: Data Cubes and Materialized Views**

**Chapter 4: - Encoding and Evolution**

One of the most important points included in this chapter is how to keep both backword and forward compatibility while adding new features.

**Backward compatibility**: Newer Code can read data written by older code

**Forward compatibility**: older Code can read data written by newer code.

This chapter discusses the different formats of encoding like (JSON, XML, Protocol Buffers, Thrift and Avro) and how they handle schema changes and support systems where old and new data, new and old code need to coexist

**Formats for encoding data**

**Programs usually work with data (at least) two different representations**

1. in memory data kept in objects, structs, lists and other data structures
2. or in file or even send it over the network, so you have to encode the data as a sequence of bytes.

* **Language Specific Formats**

1. Many languages come with built in support for encoding in-memory objects into byte sequence for example java has java.io.Serializable
2. Language Specific formats have deep problems like that encoding is often tied to a particular programming language, so we need to standardize the encoding formats.

* **JSON, XML, and Binary Variants**

1. JSON and XML are the obvious contenders standardized encodings that can be written and read by many programming languages
2. They are widely known, widely supported, and almost as widely disliked. XML is often criticized for being too verbose and unnecessarily complicated
3. JSON, XML, and CSV are textual formats, they also have some subtle problems
4. ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string
5. JSON distinguishes strings and numbers, but it doesn’t distinguish integers and floating-point numbers, and it doesn’t specify a precision
6. This is a problem when dealing with large numbers; for example, integers greater than 2^53 an example is Twitter, which uses a 64-bit number to identify each tweet. The JSON returned by Twitter’s API includes tweet IDs twice, once as a JSON number and once as a decimal string

Despite these flaws, JSON, XML, and CSV are good enough for many purposes. It’s likely that they will remain popular, especially as data interchange formats (i.e., for sending data from one organization to another). In these situations, as long as people agree on what the format is, it often doesn’t matter how pretty or efficient the format is. The difficulty of getting different organizations to agree on anything outweighs most other concerns.

1. Binary encoding you could choose a format that is more compact or faster to parse, JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example)
2. Example 4-1. Example record which we will encode in several binary formats in this chapter { "userName": "Martin", "favoriteNumber": 1337, "interests": ["daydreaming", "hacking"] }
3. Applying MessagePack - a binary encoding for JSON- will turn this string “example 4-1” from 81 bytes long into 66 bytes next sections will show different encoding formats to minimize this string.

* **Thrift and Protocol Buffers**

1. Apache Thrift and Protocol Buffers (protobuf) are binary encoding libraries that are based on the same principle. Protocol Buffers was originally developed at Google, Thrift was originally developed at Facebook, and both were made open source in 2007–08
2. Both Thrift and Protocol Buffers require a schema for any data that is encoded. To encode the data in Example 4-1 in Thrift, you would describe the schema in the Thrift interface definition language (IDL)
3. Thrift and Protocol Buffers each come with a code generation tool that takes a schema definition like the ones shown here, and produces classes that implement the schema in various programming languages
4. Encoding Example 4-1 in that format takes **59** bytes.
5. The big difference compared to Figure 4-1 is that there are no field names (userName, favoriteNumber, interests). Instead, the encoded data contains field tags, which are numbers (1, 2, and 3)
6. Field tags and schema evolution
7. We said previously that schemas inevitably need to change over time. We call this schema evolution. How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility
8. You can add new fields to the schema, provided that you give each field a new tag number. If old code (which doesn’t know about the new tag numbers you added) tries to read data written by new code, including a new field with a tag number it doesn’t recognize, it can simply ignore that field (forward compatibility)
9. if you add a new field, you cannot make it required. If you were to add a field and make it required, that check would fail if new code read data written by old code, to maintain backward compatibility, every field you add after the initial deployment of the schema must be optional or have a default value.
10. Datatypes and schema evolution , you can change datetype —but there is a risk that values will lose precision or get truncated.
11. Protocol Buffers is that it does not have a list or array datatype, but instead has a repeated marker for fields

* **Avro**

1. Apache Avro is another binary encoding format that is interestingly different from Protocol Buffers and Thrift. It was started in 2009 as a subproject of Hadoop, Avro also uses a schema to specify the structure of the data being encoded
2. The 4-1 example now takes **32** bytes long.
3. Avro uses writer’s schema and the reader’s schema to handle schema evolution.

* **The Merits of Schemas**

binary encodings based on schemas are also a viable option. They have a number of nice properties

1. they can omit field names from the encoded data this keeps the string size much smaller.
2. For users of statically typed programming languages, the ability to generate code from the schema is useful, since it enables type checking at compile time. In summary, schema evolution allows the same kind of flexibility as schemaless schema-on-read JSON databases provide.

* **Modes Of DataFlow**

**DataFlow Through Database**

1. In a database, the process that writes to the database encodes the data, and the pro‐ cess that reads from the database decodes it
2. Different values written at different times: A database generally allows any value to be updated at any time. This means that within a single database you may have some values that were written five milli‐ seconds ago, and some values that were written five years ago and thus Most rela‐ tional databases allow simple schema changes, such as adding a new column with a null default value, without rewriting existing data.v When an old row is read, the database fills in nulls for any columns that are missing from the encoded data on disk
3. Archival storage : Perhaps you take a snapshot of your database from time to time, say for backup pur‐ poses or for loading into a data warehouse (see “Data Warehousing” on page 91). In this case, the data dump will typically be encoded using the latest schema, even if the original encoding in the source database contained a mixture of schema versions from different eras. Since you’re copying the data anyway, you might as well encode the copy of the data consistently.

**DataFlow Through Service: REST and RPC**

1. The web works this way: clients (web browsers) make requests to web servers, mak‐ ing GET requests to download HTML, CSS, JavaScript, images, etc., and making POST requests to submit data to the server. The API consists of a standardized set of proto‐ cols and data formats (HTTP, URLs, SSL/TLS, HTML, etc.). Because web browsers, web servers, and website authors mostly agree on these standards, you can use any web browser to access any website (at least in theory!)
2. Web services When HTTP is used as the underlying protocol for talking to the service, it is called a web service. This is perhaps a slight misnomer, because web services are not only used on the web, but in several different contexts. For example client application running on a user’s device or One service making requests to another service owned by the same organization
3. There are two popular approaches to web services: REST and SOAP. They are almost diametrically opposed in terms of philosophy, and often the subject of heated debate among their respective proponents
4. REST is not a protocol, but rather a **design philosophy** that builds upon the principles of HTTP It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication.
5. SOAP is an XML-based protocol for making network API requests.vii Although it is most commonly used over HTTP, it aims to be independent from HTTP and avoids using most HTTP features. Instead, it comes with a sprawling and complex multitude of related standards (the web service framework, known as WS-\*) that add various features , The API of a SOAP web service is described using an XML-based language called the Web Services Description Language, or WSDL and moreover WSDL is not designed to be human-readable, and as SOAP messages are often too complex to construct manually.
6. **The problems with remote procedure calls (RPCs**) : Web services are merely the latest incarnation of a long line of technologies for making API requests over a network, many of which received a lot of hype but have serious problems. **Enterprise JavaBeans (EJB)** and **Java’s Remote Method Invocation (RMI)** are limited to Java. The **Distributed Component Object Model** (**DCOM**) is limited to Microsoft platforms.
7. The RPC model tries to make a request to a remote net‐ work service look the same as calling a function or method in your programming lan‐ guage, within the same process.
8. The client and the service may be implemented in different programming lan‐ guages, so the RPC framework must translate datatypes from one language into another. This can end up ugly, since not all languages have the same types

**Current directions for RPC :-**

1. Various RPC frameworks have been built on top of all the encodings mentioned in this chapter: for example, Thrift and Avro come with RPC support included, gRPC is an RPC implementation using Protocol Buffers, Finagle also uses Thrift, and Rest.li uses JSON over HTTP
2. This new generation of RPC frameworks is more explicit about the fact that a remote request is different from a local function call.

**Message Passing Dataflow**

1. asynchronous message-passing systems, which are somewhere between RPC and databases. They are similar to RPC in that a client’s request (usually called a message) is delivered to another process with low latency. They are similar to databases in that the message is not sent via a direct net‐ work connection, but goes via an intermediary called a message broker (also called a message queue or message-oriented middleware), which stores the message temporarily.
2. Using a message broker has several advantages compared to direct RPC
3. It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability
4. It avoids the sender needing to know the IP address and port number of the recipient (which is particularly useful in a cloud deployment where virtual machines often come and go)
5. It allows one message to be sent to several recipients
6. Message Brokers: in general, message brokers are used as follows: 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. There can be many producers and many consumers on the same topic

**Distributed actor frameworks**

* + - The actor model is a programming model for concurrency in a single process. Rather than dealing directly with threads (and the associated problems of race conditions, locking, and deadlock), logic is encapsulated in actors and each actor typically represents one client or entity, it may have some local state (which is not shared with any other actor), and it communicates with other actors by sending and receiving asynchro‐ nous messages
    - A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework.
    - Three popular distributed actor frameworks : [Akka, Orleans, Erlang]

**Part 2: - Distributed Data**

**Chapter 5: Replication**