computers

Article

Lite^2: A Schemaless Zero-Copy Serialization Format

Tianyi Chen [†], Xiaotong Guan [†], Shi Shuai [†], Cuiting Huang [†] and Michal Aibin [*,†]

Khoury College of Computer Sciences, Northeastern University, 410 West Georgia Street, Vancouver, BC V6B 1Z3, Canada
* Correspondence: vancouver@northeastern.edu
† These authors contributed equally to this work.

Abstract: In the field of data transmission and storage, serialization formats play a crucial role by converting complex data structures into a byte stream that can be easily stored, transmitted, and reconstructed. Despite the myriad available serialization formats, ranging from JSON to Protobuf, each has limitations, particularly in balancing schema flexibility, performance, and data copying overhead. This paper introduces Lite^2, a novel data serialization format that addresses these challenges by combining schemaless flexibility with the efficiency of zero-copy operations for flat or key–value pair data types. Unlike traditional formats that often require a predefined schema and involve significant data copying during serialization and deserialization, Lite^2 offers a dynamic schemaless approach that eliminates unnecessary data copying, optimizing system performance and efficiency. Built upon a contiguously stored B-tree structure, Lite^2 enables efficient data lookup and modification without deserialization, thereby achieving zero-copy operations.

Keywords: data serialization; data formats; schemaless; zero-copy

Citation: Chen, T.; Guan, X.; Shuai, S.; Huang, C.; Aibin, M. Lite^2: A Schemaless Zero-Copy Serialization Format. *Computers* **2024**, *13*, 89. https://doi.org/10.3390/computers13040089

Academic Editor: Paolo Bellavista

Received: 26 January 2024
Revised: 18 March 2024
Accepted: 25 March 2024
Published: 28 March 2024

1. Introduction

If we need to send data from one place to another, we typically need to serialize the data into an array of bytes. That is achieved through data serialization formats. There are many ways to achieve this; the most common solution is JSON [1]. Initially designed for communication between Javascript clients and servers, JSON gained popularity through its human-readable characteristics and flexibility. It has gained support from mainstream programming languages such as C, C++, Python, Java, and more. Over time, its use cases have expanded beyond client–server communication to include pure back-end applications, system configuration files, and even database storage.

Despite being a popular choice, JSON is only guaranteed to be the best fit for some scenarios. For example, when a schema is already defined, there is no need to send the names of attributes in the message. In such cases, formats like Protobuf [2] and Apache Avro [3] can offer several advantages. Popić et al. [4] demonstrated that, in IoT scenarios, the total message size of JSON could be 5.9 times larger than Protobuf when transmitting the same data, indicating that Protobuf is more compact than JSON. Additionally, the schema brings type safety and more efficient serialization/deserialization. It is especially important when working with client–server [5] and high-availability applications [6].

However, even with a more efficient serialization/deserialization process, the cost of using such formats is still considered significant. According to a study by Zeller et al. [7], serialization and Remote Procedure Call (RPC) costs are responsible for 12% of all fleet cycles across all applications at Google. Palkar et al. [8] found that modern big data applications can spend 80–90% of CPU time parsing data. Therefore, researchers have been exploring more efficient serialization formats that provide "zero-copy" operations to completely reduce the serialization cost. Cap'n Proto [9] and FlatBuffers [10] are two popular solutions in this category.

The evolution of serialization formats did not benefit all applications; while schema-based formats like Protobuf offer significant advantages in terms of data size, there are cases where maintaining the flexibility of JSON is desired. Schema-based formats cannot serve as a simple drop-in replacement in such instances. In a practical situation, developers are currently seeking an alternative to JSON. Specifically, the server must retrieve a large JSON from the database for each request, modify several entries, and store these entries back into the database. This is a perfect fit for zero-copy serialization since 99% of the JSON entries will not be accessed for each request yet are still wasting CPU time. However, to the best of our knowledge, existing zero-copy solutions all require schemas to use.

This paper aims to propose a novel data serialization format. To address the above issue, we need to thoroughly understand existing protocols, e.g., various kinds of JSON encoders and protocols other than JSON, like Protobuf and CBOR. We analyze the characteristics of each protocol, including its strengths and weaknesses. For instance, although Protobuf is generally faster, its implementation library may not be optimized, leading to slower parsing times due to extra memory allocation or copying [11]. As a result, we identified the best fit and designed a novel data serialization format—Lite^2. In the landscape of data serialization formats, various solutions cater to different needs, including handling complex nested data structures. Lite^2 is a novel approach optimized for scenarios demanding high-performance operations with schemaless data, primarily focusing on flat, key–value pair structures. This distinction is crucial for understanding Lite^2's design goals and comparative benchmarks presented in this paper.

This paper is structured into eight sections. Section 2 outlines five existing approaches to data serialization. Section 3 details the inner workings of the Lite^2 data serialization format. Section 4 describes a thorough experiment on our novel format compared to other data formats. Section 5 presents the results and discusses the experiment. Finally, Section 6 summarizes the findings and contributions of this paper, followed by Section 7 on applications and Section 8, which concludes our work.

2. Related Works

Data transfer protocols and serialization techniques are popular topics in academia and industry [12–14]. Currently, there are a variety of data forms, each with its own applicable scenarios, advantages, and disadvantages. There is no best solution. This section compares the five most commonly used data forms: JSON, CBOR, Apache Avro, Protobuf, and zero-copy serializations.

2.1. JSON

JSON (JavaScript Object Notation) is designed as a general-purpose data interchange format. It was introduced as part of the JavaScript Language Standard but became widely adopted by mainstream programming languages, e.g., C/C++, Java, Python, etc. JSON's main feature is that it is human-readable while still being easy for machines to encode and decode. Its popularity makes it the default choice of data-interchange format.

As JSON becomes so popular, it is common for a web server to parse JSON. There is a chance that JSON parsing and generation become the bottleneck of a back-end system. Thus, any improvement in JSON processing could speed up the system simultaneously. There are several competitive JSON parsers, e.g., RapidJson [15], simdjson [16], and sajson [17]. The simdjson parser claimed to be the most performant, with substantial speedups in multiple tasks. They achieved this by extensively utilizing single instruction, multiple data (SIMD) instructions. In short, SIMD is a way to achieve data-level parallelism by applying the same CPU instruction to an array of data. It is a common way to accelerate data processing [18] and is why simdjson can achieve the same tasks with far fewer CPU cycles.

Simdjson had been ported to mainstream languages due to its high-performance guarantee. However, maintaining the performance of the ported version is a non-trivial task. The issue is particularly outstanding in Python due to its highly dynamic nature. The research community has made some efforts to address this issue. One of which is Cython [19], which is a Python language extension that allows explicit type declarations and is compiled directly to C. The combination of Cython and simdjson results in cysimdjson [20], which is becoming one of the fastest Python JSON libraries.

2.2. CBOR

CBOR (Concise Binary Object Representation) is a data format designed to achieve goals such as the possibility of a tiny amount of code, a relatively small number of messages, and scalability without version negotiation [21].

CBOR is a relatively new standard for representing data on constrained devices but several research groups have studied it. For data transmission within a security information and event management (SIEM) system [22], Rix et al. developed an approach of transforming XML to CBOR using JSON as an intermediate step to reduce network load. In addition, they applied GZIP compression. This combination compressed the example file from 358 bytes (XML) to 251 (XML/GZIP) and 63 bytes using CBOR and GZIP. It shows the significant benefits of CBOR/GZIP in their use cases. Ilgner et al. evaluated CBOR for Bluetooth and 3G communication [23]. They encountered challenges because CBOR consumes a lot of static memory for CBOR object variables. CBOR libraries do not support 64-bit integers and longer objects must be broken.

However, CBOR still has advantages while sensor data must be encoded for transmission [24]. In many usage scenarios, CBOR, due to its unique combination of features, becomes a viable alternative for JSON [25].

2.3. Apache Avro

Apache Avro is another powerful data serialization format. It was integrated with Hadoop by Doug Cutting, the father of Hadoop [26].

Apache Avro is one of the most widely used data serialization approaches as it has many advantages over other data serialization formats [3]. Firstly, Apache Avro is a language-neutral data serialization system. Thanks to its efficient binary format, it does not require code generation and can interoperate with various programming languages [27]. Secondly, it has better performance in data file size and reading/writing execution time when compared with other data formats such as XStream [28], Protostuff [29], Thrift, and so on [30]. Thirdly, given Apache Avro was created as a subproject of Hadoop, it is still a preferred tech for serialization and deserialization in the Hadoop ecosystem [28,29].

However, Apache Avro also has a significant drawback—heavy dependence on the schema. Schema designing and updating are of high importance in the application of Apache Avro, and the binary data implementation needs specified knowledge and experience [27].

2.4. Protobuf

Protocol Buffers is a data exchange format created by Google to send data over a network. It is a binary file format. Protobuf employs a language known as Protocol Definition Language (PDL) [2]. Data structures defined in this language can be compiled into code in various languages.

Protocol buffers can store data in a more efficient format than JSON or XML, allowing for faster read and write times and more minor storage requirements [31,32]. Protocol buffers can be used to serialize data before sending them over the network to deserialize them on the other end. It can help reduce the bandwidth needed to transfer data and make them more compact and efficient to process. Due to the format's efficiency in size and speed, distributed systems and mobile apps can benefit from its adoption.

However, it is more challenging to get started with than formats like JSON or XML because the structure is more complex [31,32]. Also, corresponding adaptation and transformation work is required. Compared with JSON and XML, the versatility still needs improvement [33].

2.5. Zero-Copy Serializations

With the recent trend of splitting a monolithic application into multiple microservices, a remote procedure call (RPC) has become essential to the whole architecture. To make an RPC call, the caller must serialize the parameter and send it to the target microservice. Some researchers proposed "zero-copy" serialization formats to further reduce the cost of the RPC [7]. Though still dependent on schemas, such formats have no cost of serialization or deserialization because they represent the object the same way in memory and the database. Typical solutions include Cap'n Proto [9] and Flatbuffers [10].

2.6. Gaps in Previous Research

Table 1 summarizes the properties of several compared protocols. The current approach seems to have provided solutions for the following cases:

- If users want maximum flexibility, they should try JSON.
- If they want a better balance of performance and flexibility, they can choose the better JSON parser or CBOR.
- Protobuf/Avro is better if they have the schema in advance.
- As for better performance with a given schema, they should try zero-copy serialization formats like Cap'n Proto.

As Table 1 illustrates, current serialization formats offer varying degrees of flexibility and performance, with none achieving an optimal balance of both in all scenarios. While we acknowledge the extensive support for nested records and arrays by established formats such as JSON and Protobuf, it is crucial to highlight the distinct niche Lite^2 aims to fill. Specifically designed for efficiency and high performance in simpler data models, Lite^2 excels in environments where data schemas are flat or primarily composed of key–value pairs. This focus allows for specialized optimizations that significantly benefit applications demanding rapid data access and minimal overhead, often constrained by traditional formats' schema rigidity.

Recognizing the diversity of data structures encountered in real-world applications, we propose Lite^2 not as a universal replacement but as a complementary option for specific use cases. A hybrid model employing Lite^2 for performance-critical, flat data segments alongside formats like JSON or Protobuf for more complex, nested structures could offer a balanced solution in scenarios where data complexity varies. By leveraging Lite^2 for its intended strengths, developers can achieve notable performance improvements without sacrificing the expressiveness and flexibility provided by self-describing formats. This approach underlines our contribution to the serialization landscape: introducing a format tailored for specific efficiency and performance needs, thereby enriching the toolkit available to developers for optimized data handling across a spectrum of application requirements.

Table 1. Protocol comparison table.

	JSON	**CBOR**	**Avro**	**Protobuf**	**Cap'n Proto**	**Lite^2**
Binary	No	Yes	Yes	Yes	Yes	Yes
Schema-IDL	No	No	Yes	Yes	Yes	No
Human-readable	Yes	No	No	No	No	No
Self-describing	Yes	Yes	No	No	No	Yes
Zero-copy operations	No	No	No	No	Yes	Yes

3. Problem Statement

Suppose we have a dataset represented as a dictionary, denoted as D. This dictionary consists of n key–value pairs. Specifically, it is defined as follows:

$$D^n = \{k_i : v_i\}, \tag{1}$$

where $i \in [1, n]$.

Here, k_i represents the ith key and v_i is its corresponding value in the dictionary. The index i ranges from 1 to n, covering all key–value pairs in the dictionary.

A data serialization format, referred to as F, is a transformation that converts D into an array of bytes. This transformation can be expressed mathematically as

$$B = F(D), \tag{2}$$

where $B \in [0, 255]^m$ and m is the length of B.

In this formula, B is the resulting array of bytes. Each element of B is within the range of 0 to 255, reflecting the byte structure, and m is the total number of bytes in the array.

Moreover, the data serialization format provides an inverse operation to revert B into the original dictionary D, represented as follows:

$$D = F^{-1}(B) \tag{3}$$

Different data serialization formats (F) may require varying structures of D and produce different byte arrays B. To objectively compare these formats, we establish specific operations that D must support. For any query q, the dictionary should either return the corresponding value v, if q exists in D, or indicate its absence:

$$D[q] = \begin{cases} v_q, & if\ q\ exists\ in\ D \\ NULL, & otherwise \end{cases} \tag{4}$$

Additionally, for any key–value pair k, v, the dictionary should insert them if k is new or update the existing value of k. This insertion or update operation is denoted as $D[k] = v$.

The primary criterion for comparing these formats is their time efficiency in performing various operations. The time efficiency is defined as the duration to complete specific tasks, detailed as follows:

$$T_{de} = Elapsed(F^{-1}(B)) \tag{5}$$

$$T_{se} = Elapsed(F(D^n)) \tag{6}$$

$$T^p_{read} = T_{de} + \sum_{i=0}^{p} Elapsed(D^n[q_i]) \tag{7}$$

$$T^q_{write} = \sum_{i=0}^{q} Elapsed(D^n[k_i] = v_i) + T_{se} \tag{8}$$

$$T^p_{read+write} = T_{de} + \sum_{i=0}^{p} Elapsed(D^n[k_i] = v_i) + T_{se} \tag{9}$$

These metrics evaluate the time elapsed in three common scenarios: (1) deserializing the data and accessing a few entries, (2) deserializing the data, updating a few entries, and then serializing the data, and (3) creating an instance of data in the specified format from an empty state. The format with the lowest elapsed time for these operations is deemed the most efficient.

4. Method

Although we did not find an algorithm that exactly meets our needs, we know about a very similar application—SQL [34]. Take MySQL as an example; it mostly satisfies the requirement of being schemaless zero-copy. As it can store arbitrary keys and therefore be schemaless, it does not require loading everything into the memory before performing a lookup, thus being zero-copy.

However, MySQL stores the data on hard disk drives directly. It does not need to ensure the contiguity of the stored content, which is critical to a serialization format. A serialization format should always serialize the data into a continuous chunk of bytes, whereas an SQL database can split them everywhere on the disk.

Another reason we do not use MySQL directly is because it is not dedicated to data serialization and, therefore, includes some unnecessary design from our point of view. Drawing inspiration from MySQL's schemaless and zero-copy capabilities, the Lite^2 format incorporates these advantages while discarding the unnecessary overhead associated with a full-fledged database system. The goal is to create a lightweight, memory-efficient tool suitable for scenarios where data structures need to be serialized and deserialized rapidly and frequently without the need for a complete database management solution. Lite^2 is designed to be a dedicated serialization tool, optimized for speed and flexibility, and free from the constraints and complexities of database management systems. We named it "Lite^2" since it is lighter than SQLite [35].

4.1. Contiguously Stored B-Tree

The underlying data structure of SQL databases is B-tree [36]. A B-tree is a tree data structure that stores sorted data and performs operations such as searches, insertions, and deletions in logarithmic time. It is essentially a generalized binary search tree [37] that allows storing more than two children in a node. It also has a self-balancing mechanism by ensuring all paths from the root to the leaf have approximately the same length. These properties make it practical in database applications. While Lite^2 leverages the well-established principles of B-trees for organizing data, its novelty lies in adapting these principles to a zero-copy, schemaless serialization context. Unlike traditional B-trees used primarily for database indexing, Lite^2's B-tree-inspired structure is designed from the ground up to optimize for memory efficiency and direct operations on serialized data. This design facilitates lookup, insertion, and deletion without deserialization, reducing operation latencies and memory overhead.

Lite^2 format aims to store a B-tree in contiguous memory. The memory layout is designed as follows:

MAGIC	VER	DEPTH	ROOT	LEAK	TREE_SECTION
SQLite	1	1	3	3	VAR

4.1.1. Header Section

Each serialized dataset starts with a magic string "SQLite" as an identification, followed by one byte of version number to allow format versioning and evolution. Then, we use one byte to store the depth of the tree and three bytes to store the offset of the root node. The term offset refers to the zero-based index of a location in the serialized bytes, e.g., the offset of magic is 0 while the offset of version is 7. We also keep a "leak" field to track the length of the deleted entries in the tree section.

4.1.2. Nodes

The header section is followed by the tree section, which consists of two types of objects: nodes and key–value (KV) entries.

PARENT_OFFSET	PARENT_IDX	LEN	HASHES	KV_OFFSET	CHILDREN_OFFSET
3	1	1	$3*19$	$3*19$	$3*20$

Nodes form the main body of the B-tree. To self-balance, we need to store the offset of the parent node and the position of the current node as a child in the parent node within each node. That results in the parent offset and parent index field at the beginning of each node. We also need to track the number of children in each node, which is the purpose of the length field.

The node is then followed by 3 fixed-sized sections: hashes of keys, which are used for ordering, KV offsets, where the entries are stored, and child offsets, which are the locations of the child nodes.

The order of the tree is configurable and, in our case, we chose 19. The reasoning behind this is that the typical cache line of an x86-64 CPU contains 64 bytes. As hashes are represented by 3 bytes each, having an order of 19 will allow all the values required for comparison ($3 + 1 + 1 + 3 * 19 = 62$ bytes) to fit into a single cache line (64 bytes).

4.1.3. Entries

Entries are where the keys and values are stored. They are either deleted or referenced from precisely one node in the B-tree.

CTRL	LEN	KEY_LEN	VAL_LEN	KEY	VAL
1	3	3	3	VAR	VAR

If the most significant bit of CTRL is set to 1, we consider this entry deleted. It is then followed by its entire length (LEN), key length, value length, and the actual key and value. Storing the entire length allows us to allocate a few more bytes for an entry so that it does not need to relocate too frequently when the value grows in size.

4.1.4. Lookup and Insertion

The lookup is essentially a B-tree lookup (as shown in Algorithm 1).

Algorithm 1: Lookup operation in Lite[2]

```
Result: Find the value for the given key
Input   : Key to find
Output: Value associated with the key or NOT FOUND
currentNode ← root;
while currentNode ≠ NULL do
    for key in currentNode.keys do
        if key is equal to the search key then
            return associated value;
        else if key is greater than the search key then
            currentNode ← child node at the current index;
            break;
    end
    if all keys in currentNode are less than the search key then
        currentNode ← rightmost child;
    end
end
return NOT FOUND;
```

We start from the root node whose offset is stored in the header and search down the tree until we reach the target or nothing. We chose a linear search to find the child to descend to instead of a binary search. Linear search outperforms binary search with small search ranges in our micro-benchmarks.

Insertion follows the same process as that defined for B-tree. The new nodes and entries are allocated at the end of the buffer, as seen in Algorithm 2.

Algorithm 2: Insertion operation in Lite^2

```
Result: Insert a new key–value pair
Input: Key and value to insert
1 Perform Lookup(key) to determine insert location;
2 if key already exists then
3     Update the value and return;
4 Insert key–value pair in the determined location;
5 if node exceeds maximum capacity then
6     Split the node;
7     Propagate split if necessary, up to the root;
8     if root is split then
9         Create a new root;
```

4.1.5. Deletion

Finally, let us discuss the deletion in Algorithm 3.

Algorithm 3: Deletion operation in Lite^2

```
Result: Mark a key–value pair as deleted
input: Key to delete
1 Perform Lookup(key) to locate the key–value pair;
2 if key is found then
3     Mark the entry as deleted in its control field;
4     Increment the "leak" counter in the header;
5 if the "leak" counter exceeds a threshold then
6     Rebuild the tree from non-deleted entries;
```

We perform a lazy deletion instead of a full one. The deleted entry will be marked in its CTRL field and its length will be added to the leak field in the header. If the leak field reaches some thresholds, we rebuild a new tree from the old one to eliminate all the deleted entries.

4.1.6. Zero-Copy Operations

One of the key innovations of Lite^2 is its ability to perform read and write operations directly on the serialized data form. This approach eliminates the need for a separate deserialization step, significantly accelerating data access times and reducing computational overhead. We can directly perform B-tree lookup on the bytes buffer and insert into or delete from the bytes buffer. Therefore, they are zero-copy operations.

4.2. Python Binding

PyO3 is a Rust library that allows you to write native Python modules or run Python code and modules from Rust. It provides a safe and fast way to interface between Rust and Python. PyO3 can be used with minimal configuration by using Maturin, a tool for building and publishing Rust-based Python packages. We followed the official guidelines and built our binding. We adhered to official guidelines while creating our binding and avoided several errors by using PyO3:

- We refrained from directly modifying Python bytes since they are semantically immutable and their contents should never change.
- We avoided keeping a raw reference to Python bytes in the Rust object, as it is unsafe and could cause a segmentation fault. The Python caller can deconstruct the bytes object at any time. Accessing these bytes from native code after deconstructing them will result in a segmentation fault.

- We avoided returning a reference of the data on the native heap to Python for similar reasons. If the Python caller drops the Rust object, the reference is invalidated. Any access to that reference results in a use-after-free error, causing segmentation faults.

Designs that contain the above-mentioned errors cannot be compiled with PyO3, which saved us a significant amount of time debugging and reworking our implementation.

4.3. Benchmarking

Our benchmarking design incorporates essential features such as extensibility and configurability. The former enables the easy addition of data serialization formats, while the latter allows testing with varying data and numbers of rounds.

Despite Python's lack of performance efficiency, it presents competitive capabilities for benchmarking and testing. With its dynamic type system, we are able to create a benchmark system that meets our requirements. For testing and benchmarking, we have chosen Pytest, which supports small, readable tests and complex functional testing for applications and libraries. The use of plain assert statements and adherence to conventions for Python test discovery further enhance its utility. Furthermore, its vast collection of plugins and extensions strengthens its applicability. We have implemented our benchmarks using the pytest-benchmark extension to provide precise and statistically significant outcomes.

4.4. Test Strategies

Our testing strategy involved multiple approaches, including unit tests, random tests, and fuzz tests. To avoid redundancy, we kept our Rust tests minimal and focused on testing in Python. We also maintained a list of concerns and addressed them with corresponding test cases:

- Can the format handle oversized keys and values properly? Our format has a theoretical size limit and it is essential to handle oversized data correctly.
- Does the library violate Python type guarantees? As we interact only with "bytes", which have immutable semantics, it is crucial to avoid any mutations.
- Does the library operate in a memory-safe way? It is vital to avoid use-after-free situations.
- Does the format function correctly during insertion, query, and deletion operations?
- Does the library behave well with different key distributions? For a data structure built upon hash functions, it is crucial to handle hash collisions correctly.
- Does the library require an appropriate amount of space? Even if the library passes all the above tests, it is crucial to test its space requirements as space is one of our concerns.

5. Simulation Scenarios

For the process of our benchmark algorithm, we will consider it as a scenario where a company needs to process a large amount of data from multiple sources. The data sources include JSON files, binary files, and others. The developers must identify the most efficient format between the 11 frameworks shown in Table 2.

Table 2. Data serialization formats.

Lite^2	MessagePack
JSON	Orjson
Cap'n Proto	UJSON
CBOR	Rapidjson
Proto Buffers	Simdjson
Cysimdjson	

We can break down data processing into three cases: parsing and accessing the data, inserting and dumping the data, and parsing, updating, and dumping the data. Each case corresponds to the efficiency in read-heavy applications, write-heavy applications, and read–write balance applications, as shown in Table 3.

Table 3. Performance metrics.

Read efficiency	Parse the data and access a number of entries
Write efficiency	Insert a number of entries and serialize
Read–write efficiency	Parse the data, update some entries, and serialize

To conduct this simulation, we first establish the parameters for the number of entries (N_e) and average entry (key and value) length ($\overline{L_e}$). Subsequently, we generate sample data conforming to each data serialization format. This sample data is written to disk. We will calculate the elapsed time based on Equations (7)–(9) for each setting (N_e, $\overline{L_e}$) by performing them multiple times. This evaluation will help us understand how each format performs under various workload conditions.

We will run performance tests in this simulation for the following combinations:

- N_e is 100 and $\overline{L_e}$ is 40
- N_e is 1000 and $\overline{L_e}$ is 160
- N_e is 100,000 and $\overline{L_e}$ is 640

In developing our benchmarking methodology, several vital parameters were carefully selected to evaluate Lite^2's performance comprehensively. These parameters were chosen to simulate a range of real-world scenarios that Lite^2 will likely encounter, ensuring that our assessment captures its strengths and potential limitations under various conditions:

1. **Data Size (N_e) and Entry Size ($\overline{L_e}$)**: The sizes of datasets and individual entries were chosen based on common use cases in data serialization where schemaless data formats are prevalent [38]. These sizes reflect the varied nature of data these systems handle, from configuration files to sensor data streams.
2. **Read–Write Operation Mix**: The mix of read and write operations in our benchmarks was determined by analyzing typical workload patterns in target application domains. This mix aims to provide a balanced view of Lite^2's performance across different data access patterns.
3. **Concurrency Levels**: Given the increasing importance of concurrency in modern applications [5], particularly those based in cloud and distributed computing environments [6,39], we included varying levels of concurrency in our benchmarks. The chosen levels represent the load conditions under which data serialization formats must operate efficiently.

Where applicable, we benchmarked Lite^2 against other formats using parameters that align with those used in their respective evaluations. This approach ensures a fair comparison, highlighting Lite^2's performance relative to established options in similar conditions. The simulation scenarios are summarized in Table 4.

Table 4. Parameters for each simulation scenario.

Dataset	Entries	Average Key–Value Length
Simulation I	100	40
Simulation II	1000	160
Simulation III	100,000	640

We designed the N_e to span three orders of magnitude. Our experiments indicate that the three sets of data presented are sufficiently representative, so there is no need to present additional data. This systematic approach facilitates an empirical comparison of the performance of the three serialization formats under various conditions, ultimately assisting developers and architects in identifying the most efficient format for their specific needs.

6. Results

Lite^2 is a highly competitive format for datasets of various sizes. When the number of operations is small (as shown in the results in Figures 1–3), Lite^2 always shows unparalleled performance, proving our zero-copy mechanism's effectiveness. The advantage decreases as the number of operations grows and is surpassed by Protobuf in the 100-entry configuration. This is due to the different search algorithms for entry lookup. For Lite^2, this is achieved through a B-tree search with an $O(logN)$ theoretical time complexity, where N represents the number of elements in the tree. Protobuf, on the other hand, can access the elements by offset like an array, which has a time complexity of $O(1)$. When performed frequently, the faster searching mechanism of Protobuf offsets the costly serialization process. Such a trade-off is likely to be observed in smaller datasets. In larger datasets, the requirement for achieving the offset becomes more unrealistic, making Lite^2 a better choice.

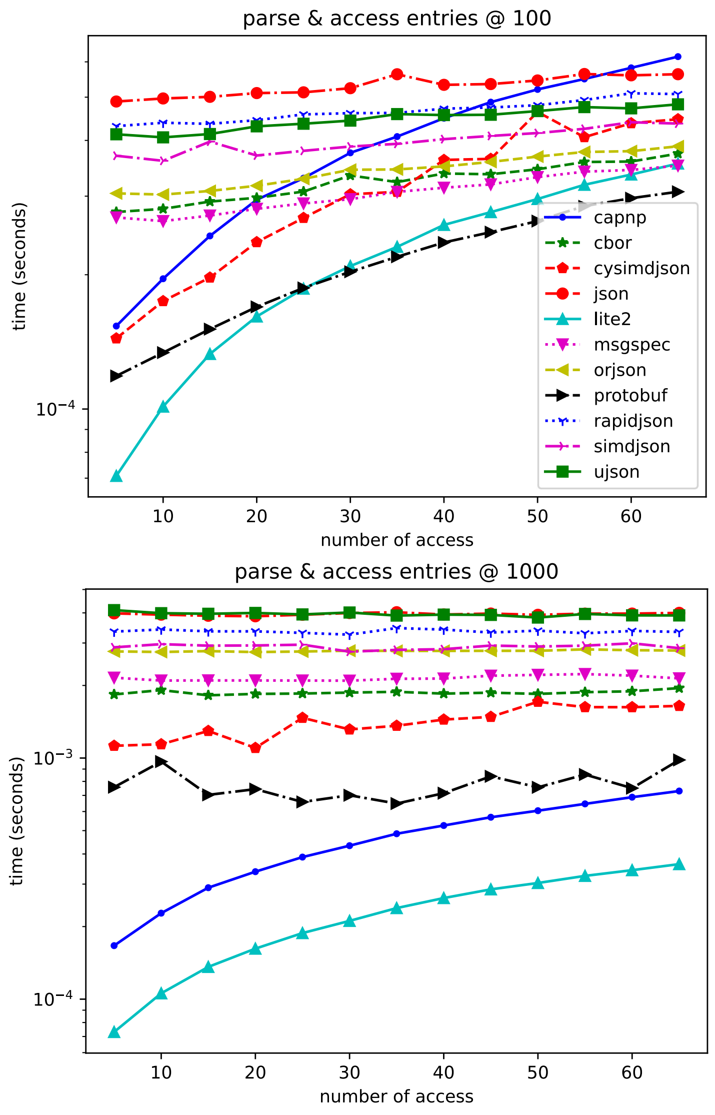


Figure 1. *Cont.*

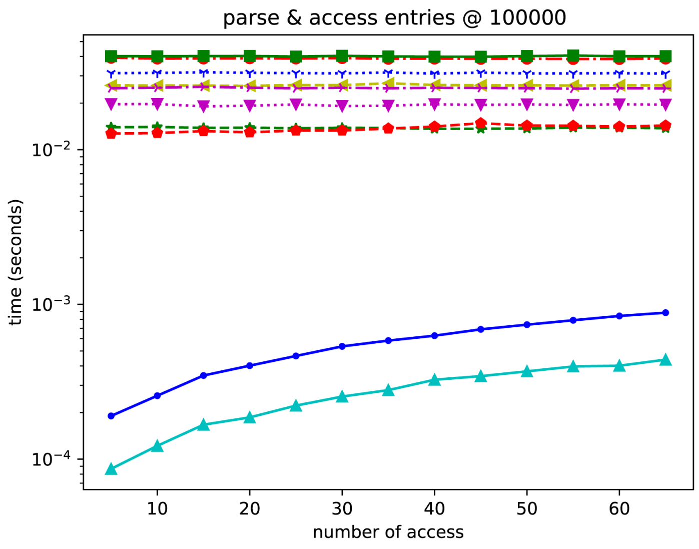
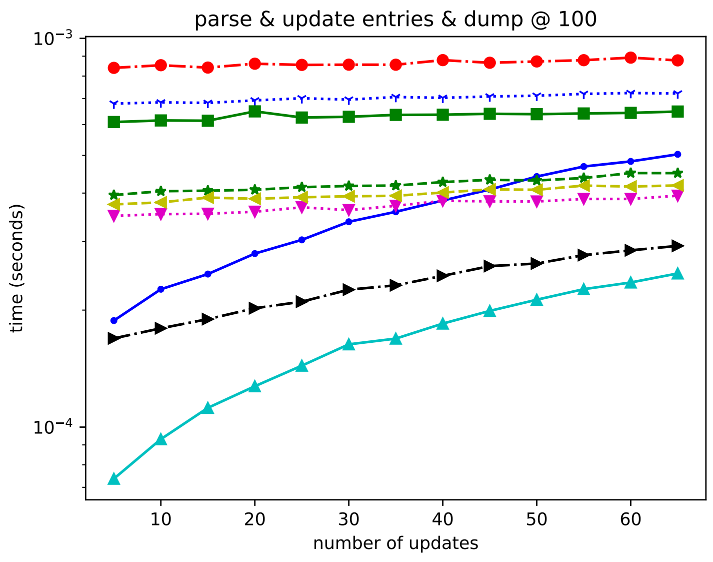


Figure 1. Parse and access entries.



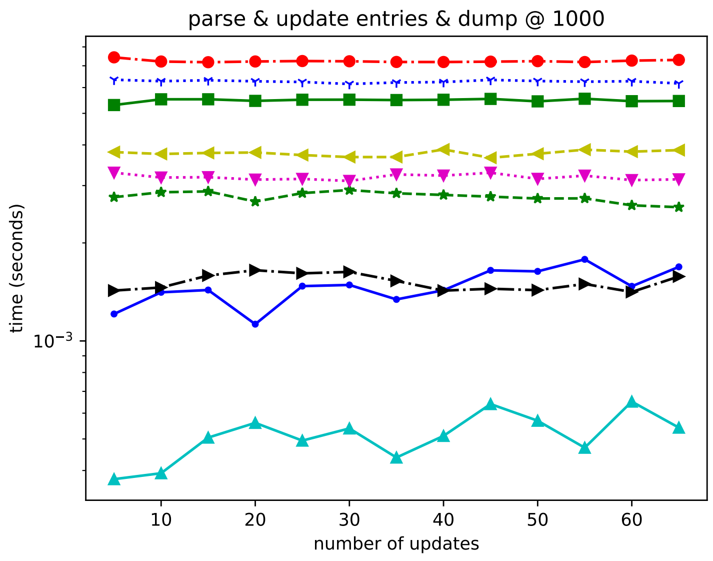


Figure 2. *Cont.*

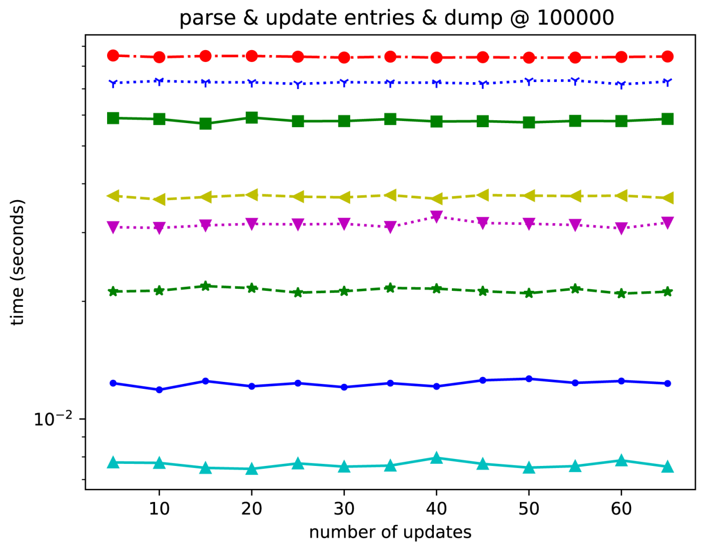


Figure 2. Parse and update entries and dump.

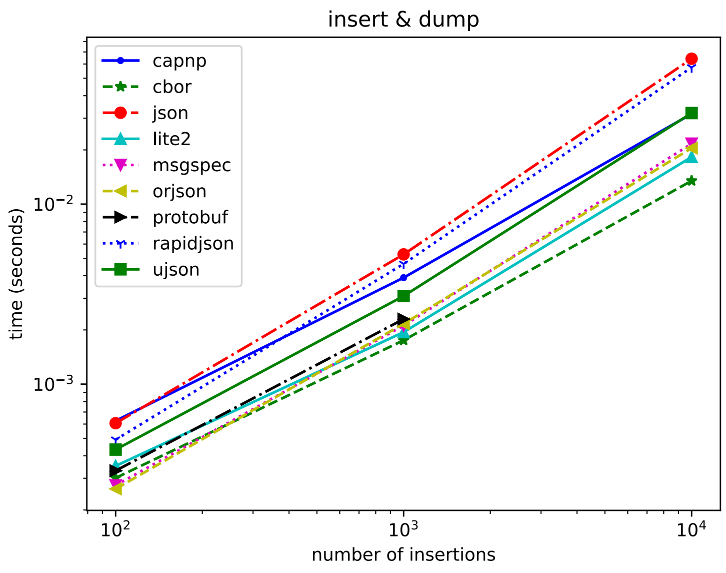


Figure 3. Insert and dump.

Lite^2 is comparable to other formats regarding pure insertion performance, even though it is not the fastest. Furthermore, it has a propensity to become more time-effective as the number of entries rises. Lite^2 is a highly recommended data format for read and update scenarios involving sizable schemaless datasets. It is the best-performing data structure for these jobs thanks to its schematic format, direct memory access to serialized data, and great performance.

7. Applications

Due to the nature of the analysis of simulation outcomes, Lite^2 has a particular impact on obtaining and updating data in multiple fields.

7.1. Web Server

Web servers commonly need to deal with data. For example, an e-commerce website must store the merchandise with its properties, such as various discounts, prices, availability, etc. The properties could grow into a large key–value object and, due to their interdependent nature, splitting them might not be a choice. The data are in rapid iteration, meaning applying schema will greatly increase the development complexity. The server may need to fetch the data, access or modify a few entries of the data, and write them back to the database. In such a scenario, the serialization and deserialization could spend significantly more CPU cycles than the insertion/searching. A schemaless zero-copy serialization format like Lite^2 would be a perfect fit.

7.2. Financial and Securities Markets

Efficiency in reading and updating data is crucial in the financial and securities markets. As information on the financial market is always changing, traders and investors must act rapidly to get access to real-time data. Here is a specific illustration and explanation: High-Frequency Trading (HFT) is a trading strategy based on algorithms and high-speed computer programs to exploit small price differences in the market to realize profits. This procedure necessitates the real-time acquisition and processing of a sizable volume of schemaless data, such as market conditions, news, social media, etc. Time-saving measures are therefore essential for reading and updating data.

Real-time access to data from numerous sources, formats, and structures is necessary for high-frequency trading systems, including real-time stock prices, market quotations, news, and social media. These data must be transferred at high speed and low latency to enable trading algorithms to make judgments quickly. High-frequency trading systems must also constantly update internal data to track market movements and modify trading tactics in real time. The system must be adaptable and effective to quickly update data because these data may not have a structure. In conclusion, Lite^2 is quite helpful for managing such events. It can read and update data quickly.

7.3. Internet of Things (IoT) and Real-Time Monitoring Systems

Time efficiency is crucial for accessing and updating data in the Internet of Things (IoT) and real-time monitoring systems [40]. IoT devices and sensors produce a lot of real-time data, necessitating frequent updates to keep the system functional. The intelligent transportation system uses real-time monitoring and Internet of Things technology to analyze and dispatch road or air traffic situations. To address the issue of congestion quickly and boost the effectiveness of road transportation, traffic data must be acquired and updated in real-time during this process.

7.4. Virtual Reality Entertainment Applications

In a VR entertainment application, the sensor data generated by different devices can have varying schema structures, and the data generated by these sensors can be highly dynamic and unpredictable.

Let us assume a virtual reality platform is designed to allow users to connect and collaborate with others in real-time. It provides a range of tools and features for users to build their virtual worlds and experiences. When users connect to it, many data need to be handled, including information about the users' avatars, their movements and actions, and their interactions with objects and other users. Additionally, chat messages or other forms of communication between users may need to be transmitted and processed.

This is a good application scenario for Lite^2. First, a schemaless data framework can allow for more efficient and flexible processing of the varying users' data. Second, a zero-copy data format would reduce the overhead associated with copying data, improving performance and reducing memory usage. This is especially important in VR applications, where high frame rates and low latency are critical for a smooth, immersive experience.

7.5. Real-Time Criminal Record Matching in CCTV Surveillance System

A CCTV surveillance system is deployed in a public security setting, let us say a city, to monitor people's movements. The system consists of multiple CCTV cameras capturing video streams and generating data in real-time. The data includes video frames, metadata, and other relevant information.

When the CCTV cameras detect suspicious activities or individuals, the system performs real-time criminal record matching against a mass database of criminal records. Upon a successful match, the system generates an alarm in real-time, notifying the security department to take action, like dispatching security officers to the location for further investigation or intervention.

This is a good application scenario for Lite^2. The format's schemaless nature is particularly beneficial as the data captured by the CCTV cameras may not adhere to a fixed schema due to frequent changes in criminal records and varying data formats from different sources. Zero-copy enables quick and efficient data retrieval and matching, crucial in public security scenarios where speed is paramount.

8. Conclusions and Future Research

We have developed a novel schemaless serialization format known as Lite^2. This format has been thoroughly benchmarked to assess its performance in various scenarios. In situations where the number of operations required after deserialization is low, Lite^2 demonstrates a significant performance advantage over JSON, with up to a 100-fold increase in efficiency. This performance improvement is attributable to Lite^2's unique design, which allows it to excel in specific use cases where traditional formats like JSON may struggle to keep up. By leveraging the strengths of Lite^2, developers can optimize their applications for maximum performance, particularly in situations where quick data retrieval and processing are significant.

When comparing Lite^2 and SQL databases, both use a B-tree data structure. Despite this commonality, drawing a direct performance comparison between the two would not yield valid conclusions. This is because Lite^2 and SQL databases have different purposes and design goals. Lite^2 is a schemaless serialization format aimed at efficient data retrieval and storage, optimized for high-performance reading and writing of key–value pairs, reflecting a specific subset of database operations that benefit from zero-copy and schemaless characteristics. On the other hand, SQL databases are designed for structured data storage, retrieval, and manipulation with a predefined schema.

The underlying reasons for the limitations in comparing Lite^2 and SQL databases could be their different architectures, optimizations, and implementations. To better understand the performance and applicability of Lite^2 in the context of database systems, further exploration and research are needed. This could involve investigating more suitable comparison metrics or methodologies, and examining the specific use cases where Lite^2 might offer advantages over traditional database systems, which is not the focus of this framework nor the paper.

Limitations and Future Work

Lite^2, in its current implementation, is designed to handle map-type data structures, which consist of key–value pairs. This limitation means that the format is unsuitable for representing list or array-like structures. This could be a drawback in some use cases where the data are inherently sequential or where list-based structures are more appropriate.

The key–value structure utilized by Lite^2 currently only supports string data types for both keys and values. This restriction can affect the format's flexibility when dealing with more complex or diverse data types, such as integers, floats, or nested objects. Extending support to a wider variety of data types could significantly enhance Lite^2's adaptability and usefulness across different applications.

To address these limitations and increase the applicability of Lite^2, future work should focus on expanding its capabilities to support various data structures, accommodating diverse data types, and enhancing its performance in data creation scenarios. This would produce a more versatile and efficient serialization format for various applications. Proposed developments include introducing transaction support, enabling atomicity and consistency across operations, and implementing range queries to facilitate more complex data retrieval patterns. Additionally, exploring mechanisms for optimizing Lite^2 for workloads that involve joins and aggregations will be crucial for broadening its applicability to a wider array of database applications. These advancements will leverage Lite^2's core strengths while addressing the multifaceted nature of database workloads.

The performance of Lite^2 when creating data from scratch is relatively ordinary, which means that it may not stand out compared to other serialization formats in scenarios where a substantial amount of new data needs to be generated and processed. This average performance can be attributed to the format's specific design choices and underlying data structures, which may not be optimized for the rapid generation of new data. To enhance Lite^2's performance in data creation scenarios, future work should investigate potential improvements in its underlying data structures, algorithms, or implementation techniques. Optimizing the format for creating data from scratch would make it more competitive and applicable across a broader range of scenarios, increasing its overall utility and effectiveness in various applications.

Author Contributions: Conceptualization, T.C., S.S. and C.H.; Data curation, T.C. and C.H.; Formal analysis, T.C. and C.H.; Investigation, T.C.; Methodology, T.C., X.G., S.S. and C.H.; Resources, M.A.; Software, T.C., X.G., S.S. and C.H.; Supervision, M.A.; Validation, M.A.; Writing—original draft, T.C., X.G., S.S. and C.H.; Writing—review and editing, M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Dataset available on request from the authors.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Crockford, D.; Morningstar, C. *Standard ECMA-404 The JSON Data Interchange Syntax*; ECMA International: Geneva, Switzerland, 2017. [CrossRef]
2. Google. *Protocol Buffers: Google's Data Interchange Format*; Technical report; Google: Mountain View, CA, USA, 2008.
3. Proos, D.P.; Carlsson, N. Performance comparison of messaging protocols and serialization formats for digital twins in IoV. In Proceedings of the 2020 IFIP Networking Conference (Networking), Virtual, 22–25 June 2020; pp. 10–18.
4. Popić, S.; Pezer, D.; Mrazovac, B.; Teslić, N. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In Proceedings of the 2016 International Conference on Smart Systems and Technologies (SST), Osijek, Croatia, 12–14 October 2016; pp. 261–265. [CrossRef]
5. Tagdiwala, V.; Bharoliya, A.; Patel, P.; Shah, D.; Aibin, M. Robust Client and Server State Synchronisation Framework For React Applications: React-state-sync. In Proceedings of the 2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Regina, SK, Canada, 24–27 September 2023; pp. 475–481. [CrossRef]
6. Kit, N.K.K.; Aibin, M. Study on High Availability and Fault Tolerance. In Proceedings of the 2023 International Conference on Computing, Networking and Communications (ICNC), Honolulu, HI, USA, 20–22 February 2023; pp. 77–82. [CrossRef]
7. Wolnikowski, A.; Ibanez, S.; Stone, J.; Kim, C.; Manohar, R.; Soulé, R. Zerializer: Towards Zero-Copy Serialization. In *HotOS '21: Proceedings of the Workshop on Hot Topics in Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 206–212. [CrossRef]
8. Palkar, S.; Abuzaid, F.; Bailis, P.; Zaharia, M. Filter before You Parse: Faster Analytics on Raw Data with Sparser. *Proc. VLDB Endow.* **2018**, *11*, 1576–1589. [CrossRef]
9. capnproto. Cap'n Proto Serialization/RPC System - Core Tools and C++ Library. Online Resource. Available online: https://capnproto.org (accessed on 20 March 2024).
10. Google. FlatBuffers: Memory Efficient Serialization Library. Online Resource. Available online: https://google.github.io/flatbuffers/ (accessed on 20 March 2024).
11. Sumaray, A.; Makki, S.K. A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform. In *ICUIMC '12: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*; Association for Computing Machinery: New York, NY, USA, 2012. [CrossRef]
12. Cao, S.; Di Girolamo, S.; Hoefler, T. Accelerating Data Serialization/Deserialization Protocols with In-Network Compute. In Proceedings of the 2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI), Dallas, TX, USA, 13–18 November 2022; pp. 22–30. [CrossRef]
13. Luis, Á.; Casares, P.; Cuadrado-Gallego, J.J.; Patricio, M.A. PSON: A Serialization Format for IoT Sensor Networks. *Sensors* **2021**, *21*, 4559. [CrossRef] [PubMed]

14. Viotti, J.C.; Kinderkhedia, M. A Survey of JSON-compatible Binary Serialization Specifications. *arXiv* **2022**. arXiv:2201.02089.
15. Tencent. Tencent/Rapidjson: A Fast JSON Parser/Generator for C++ with both SAX/Dom Style API. Online Resource. Available online: https://github.com/Tencent/rapidjson (accessed on 20 March 2024).
16. Langdale, G.; Lemire, D. Parsing gigabytes of JSON per second. *VLDB J.* **2019**, *28*, 941–960. [CrossRef]
17. chadaustin. chadaustin/Sajson: Lightweight, Extremely High-Performance JSON Parser for C++11. Online Resource. Available online: https://github.com/chadaustin/sajson (accessed on 20 March 2024).
18. Hernández, A.F. Yet Another Survey on SIMD Instructions. 2013. Available online: https://www.semanticscholar.org/paper/Yet-Another-Survey-on-SIMD-Instructions-Hern/3a12e293f19c8a998ccf3a3741e21085681ec343 (accessed on 20 March 2024).
19. Behnel, S.; Bradshaw, R.; Citro, C.; Dalcin, L.; Seljebotn, D.S.; Smith, K. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* **2011**, *13*, 31–39. [CrossRef]
20. TeskaLabs. TeskaLabs/Cysimdjson: Very fast Python JSON Parsing Library. Online Resource. Available online: https://github.com/TeskaLabs/cysimdjson (accessed on 20 March 2024).
21. Bormann, C.; Hoffman, P. *Concise Binary Object Representation (CBOR)*; RFC 7049, Internet Engineering Task Force. 2013. Available online: https://tools.ietf.org/html/rfc7049 (accessed on 20 March 2024).
22. Rix, T.; Detken, K.O.; Jahnke, M. Transformation between XML and CBOR for network load reduction. In Proceedings of the 2016 3rd International Symposium on Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS), Offenburg, Germany, 26–27 September 2016; pp. 106–111.
23. Ilgner, H.; Pienaar, S. Implementing a compact data format for Bluetooth and 3G communication to monitor remote pipelines. In Proceedings of the 2016 International Conference on Advances in Computing and Communication Engineering (ICACCE), Durban, South Africa, 28–29 November 2016; pp. 45–50.
24. Driscoll, B.; Zhao, Z. Automation of NERSC Application Usage Report. In Proceedings of the 2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools), Atlanta, CA, USA, 18 November 2020; pp. 10–18.
25. Kalvoda, P. Implementace a Evaluace Protokolu CBOR. Bachelor's Thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2015. Available online: http://hdl.handle.net/20.500.11956/61865 (accessed on 20 March 2024).
26. Vohra, D. Apache avro. In *Practical Hadoop Ecosystem*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 303–323.
27. Mooney, P.; Minghini, M. Geospatial Data Exchange Using Binary Data Serialization Approaches. *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.* **2022**, *48*, 4. [CrossRef]
28. Maeda, K. Comparative survey of object serialization techniques and the programming supports. *Int. J. Comput. Inf. Eng.* **2011**, *5*, 1488–1493.
29. Maeda, K. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In Proceedings of the 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP), Bangkok, Thailand, 16–18 May 2012; pp. 177–182.
30. Blomer, J. A quantitative review of data formats for HEP analyses. *J. Phys. Conf. Ser.* **2018**, *1085*, 032020. [CrossRef]
31. Peng, Y.; Wang, F. Research on the data format standard of IoT based on XML. *Appl. Mech. Mater.* **2013**, *336*, 1–10. [CrossRef]
32. Wehner, P.; Piberger, C.; Göhringer, D. Using JSON to manage communication between services in the Internet of Things. In Proceedings of the 2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Montpellier, France, 26–28 May 2014, pp. 1–4.
33. Kaur, G.; Fuad, M. An evaluation of Protocol Buffer. In Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon), Concord, NC, USA, 18–21 March 2010; pp. 459–462.
34. Widenius, M.; Axmark, D. *MySQL Reference Manual: Documentation from the Source*; O'Reilly Media: Sebastopol, CA, USA, 2002.
35. Gaffney, K.P.; Prammer, M.; Brasfield, L.; Hipp, D.R.; Kennedy, D.; Patel, J.M. SQLite: Enhancements and Evolutions in Database Management. *Proc. VLDB Endow.* **2022**, *15*, 1234–1245.
36. Bayer, R.; McCreight, E. Organization and Maintenance of Large Ordered Indices. In *SIGFIDET '70, Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*; Association for Computing Machinery: New York, NY, USA, 1970; pp. 107–141. [CrossRef]
37. Douglas, A.S. Techniques for the Recording of, and Reference to data in a Computer. *Comput. J.* **1959**, *2*, 1–9. [CrossRef]
38. Kaur, K.; Rani, R. Modeling and querying data in NoSQL databases. In Proceedings of the 2013 IEEE International Conference on Big Data, Silicon Valley, CA, USA, 6–9 October 2013; pp. 1–7.
39. Aibin, M.; Walkowiak, K. Resource requirements in fixed-grid and flex-grid networks for dynamic provisioning of data center traffic. In Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Vancouver, BC, Canada, 15–18 May 2016; pp. 1–4. [CrossRef]
40. Aibin, M. The Weather Impact on Heating and Air Conditioning with Smart Thermostats. *Can. J. Electr. Comput. Eng.* **2020**, *43*, 190–194. [CrossRef]