HexaURL Character Code

Yota Inomoto

“HexaURL is a high-performance, fixed-length, case-insensitive, URL-safe encoding scheme designed to accelerate search operations. Leveraging a streamlined character set and efficient 6-bit encoding, HexaURL standardizes identifier formats to enable rapid, constant-time comparisons across systems. This approach is particularly well-suited for URL slugs and database index aliases where speed and consistent indexing are critical. Additionally, its fixed-length structure and URL-safe properties ensure robust and unified handling of identifiers.”

February 18, 2025

Contents

1. Overview

HexaURL is an encoding scheme that converts URL-safe characters into a fixed-length, bit-packed format. By representing each allowed character using 6 bits, the system reduces storage overhead and enables rapid comparisons. In practice, a database lookup becomes as simple as comparing fixed-size keys in memory. The approach is inspired by historical techniques (e.g., DEC SIXBIT) and optimizes common operations in search-intensive systems.

For example, when converting a user-supplied identifier alias such as "Tyler-Durden" (with uppercase normalized to lowercase), the encoding:

- Ensures the output is always of constant size regardless of input quirks
- Allows byte-level comparisons without extra normalization during lookup
- Streamlines indexing in systems handling hundreds of millions of records

2. Goals and Non-Goals

2.1. Goals

- Provide a fixed-length, URL-safe character encoding optimized for search performance.
- Achieve fast constant-time comparisons through uniform, predictable, and compact key representations.
- Lower CPU overhead by eliminating the need for runtime normalization and complex string manipulations.

2.2. Non-Goals

- Improving compression ratios.
- Replacing rich, variable-length encodings for human readability (HexaURL is strictly for internal identifiers).
- Implementing encryption or security features beyond basic input validation.

3. Context

Modern systems rely on fast and efficient identifier comparisons—for example, when conducting database searches or comparing short URLs. Traditional representations (like UTF-8 or standard ASCII°) may require additional per-character checks or suffer from case sensitivity issues, which incur extra processing overhead.

The HexaURL approach builds on a reduced character set that includes:

- 26 letters (normalized to lowercase for case-insensitivity)
- 10 digits (0-9)
- 2 special characters (hyphen and underscore)

Each character is represented by exactly 6 bits. As a result, comparisons become direct bit-level operations, reminiscent of legacy schemes such as DEC SIXBIT°, but updated for modern architectures and use cases.

4. Existing Solution Overview

Before HexaURL, many systems relied on variable-length encodings (e.g., ASCII) which are case-sensitive and can lead to inefficient comparisons due to extra normalization steps. For instance:

- Standard ASCII comparisons may require converting uppercase letters.
- Variable lengths increase the complexity of indexing and caching.

These challenges motivated the creation of HexaURL, which standardizes identifiers into fixed, compact keys.

5. Proposed Solution

HexaURL transforms textual identifiers into a fixed-length byte array using three main steps:

- Character Validation:
  - Validate that the input contains only the supported 38 characters.
  - Immediately normalize any uppercase letters to lowercase.
- Encoding Process:
  - Map each valid character to a 6-bit value using a precomputed lookup table (or subtract 32 from the ASCII byte representation and truncate the upper two bits to 6 bits).
  - Pack every four 6-bit values (totaling 24 bits) into three consecutive bytes.
  - The relationship between the string represented and the byte size is (N input chars $\rightarrow$ $\text{ceil}\left(\frac{N \times 3}{4}\right)$ bytes).
  - Pad data if necessary to maintain fixed-length output, ensuring predictable key sizes.
- Decoding Process:
  - Reverse the operation by unpacking three bytes into four 6-bit values.
  - Use a reverse lookup table to reassemble the original (normalized) characters.
  - Validate the decoded string to ensure conformity with HexaURL's specifications.

The above approach guarantees that all processed keys have a constant length, allowing for direct, byte-wise key comparisons without additional runtime overhead.

6. Alternatives and Trade-Offs

- Alternative: Using Variable-Length Encodings
  - Pros: More compact identifiers when characters are sparse.
  - Cons: Leads to unpredictable key sizes and extra normalization overhead, impacting search performance.
- Alternative: Leveraging Full ASCII with Additional Normalization
  - Pros: Minimal changes to existing systems.
  - Cons: Requires runtime case conversion and adds processing delays in high-frequency search environments.

The fixed-size, bit-level solution offered by HexaURL was selected for its simplicity and superior constant-time comparison performance.

7. Implementation Details

7.1. Encoding Optimizations

- Use of Direct Lookup Tables:
  - Fast conversion from character to 6-bit value helps reduce CPU cycles.
- Efficient Bitwise Operations:
  - Streamlined packing and unpacking reduce the overhead associated with shifting and masking operations.
- Fixed Stack Allocation:
  - By avoiding heap allocations, we ensure a predictable memory footprint across search operations.

7.2. Performance Considerations

1. Constant-Time Comparisons:
   - Fixed-length outputs allow the use of $O(1)$ comparisons, critical for real-time search functions.
2. Memory and Cache Efficiency:
   - Storing identifiers in a compact, bit-packed format improves cache utilization and lowers memory usage.
3. Benchmarking:
   - Preliminary metrics show that HexaURL achieves approximately 25% lower processing overhead compared to standard ASCII operations.

7.3. Safety & Testability

- Validation Modes:
  - Comprehensive mode for untrusted inputs (with detailed checks).
  - Quick mode for trusted inputs, bypassing redundant validations.
  - Integration with Zod for JavaScript/TypeScript: Leverage Zod in the frontend to perform robust, schema-based validations that complement the back-end checks.
- Logging (Future Work):
  - Integrate logging around conversions to catch edge-case errors in production environments.

8. Open Questions

- Are there edge cases where input normalization might inadvertently reject valid identifiers?
- Should we support future expansion of the character set?
- Is there potential for further hardware acceleration using SIMD instructions for bulk conversions?

9. Conclusion

HexaURL offers a targeted solution for systems requiring rapid searches and direct key comparisons. By employing a fixed-length, 6-bit representation for a specific set of URL-safe characters, the design reduces processing overhead and ensures predictable performance improvements for search-intensive environments.