

Workgroup: QUIC

Internet-Draft: draft-ietf-quic-qpack-17

Published: September 10, 2020

Intended Status: Standards Track

Expires: March 14, 2021

Authors: C. Krasic      M. Bishop      A. Frindell, Ed.

Netflix      Akamai Technologies      Facebook

## **QPACK: Header Compression for HTTP/3**

### **Abstract**

This specification defines QPACK, a compression format for efficiently representing HTTP fields, to be used in HTTP/3. This is a variation of HPACK compression that seeks to reduce head-of-line blocking.

### **Note to Readers**

Discussion of this draft takes place on the QUIC working group mailing list ([quic@ietf.org](mailto:quic@ietf.org)), which is archived at [https://mailarchive.ietf.org/arch/search/?email\\_list=quic](https://mailarchive.ietf.org/arch/search/?email_list=quic).

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-qpack>.

### **Status of This Memo**

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 14, 2021.

### **Copyright Notice**

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

- 1. [Introduction](#)
  - 1.1. [Conventions and Definitions](#)
  - 1.2. [Notational Conventions](#)
- 2. [Compression Process Overview](#)
  - 2.1. [Encoder](#)
    - 2.1.1. [Limits on Dynamic Table Insertions](#)
    - 2.1.2. [Blocked Streams](#)
    - 2.1.3. [Avoiding Flow Control Deadlocks](#)
    - 2.1.4. [Known Received Count](#)
  - 2.2. [Decoder](#)
    - 2.2.1. [Blocked Decoding](#)
    - 2.2.2. [State Synchronization](#)
    - 2.2.3. [Invalid References](#)
- 3. [Reference Tables](#)
  - 3.1. [Static Table](#)
  - 3.2. [Dynamic Table](#)
    - 3.2.1. [Dynamic Table Size](#)
    - 3.2.2. [Dynamic Table Capacity and Eviction](#)
    - 3.2.3. [Maximum Dynamic Table Capacity](#)
    - 3.2.4. [Absolute Indexing](#)
    - 3.2.5. [Relative Indexing](#)
    - 3.2.6. [Post-Base Indexing](#)
- 4. [Wire Format](#)
  - 4.1. [Primitives](#)
    - 4.1.1. [Prefixed Integers](#)
    - 4.1.2. [String Literals](#)
  - 4.2. [Encoder and Decoder Streams](#)
  - 4.3. [Encoder Instructions](#)
    - 4.3.1. [Set Dynamic Table Capacity](#)
    - 4.3.2. [Insert With Name Reference](#)
    - 4.3.3. [Insert Without Name Reference](#)
    - 4.3.4. [Duplicate](#)
  - 4.4. [Decoder Instructions](#)
    - 4.4.1. [Section Acknowledgement](#)
    - 4.4.2. [Stream Cancellation](#)
    - 4.4.3. [Insert Count Increment](#)

4.5.	<a href="#">Field Line Representations</a>
4.5.1.	<a href="#">Encoded Field Section Prefix</a>
4.5.2.	<a href="#">Indexed Field Line</a>
4.5.3.	<a href="#">Indexed Field Line With Post-Base Index</a>
4.5.4.	<a href="#">Literal Field Line With Name Reference</a>
4.5.5.	<a href="#">Literal Field Line With Post-Base Name Reference</a>
4.5.6.	<a href="#">Literal Field Line Without Name Reference</a>
5.	<a href="#">Configuration</a>
6.	<a href="#">Error Handling</a>
7.	<a href="#">Security Considerations</a>
7.1.	<a href="#">Probing Dynamic Table State</a>
7.2.	<a href="#">Applicability to QPACK and HTTP</a>
7.3.	<a href="#">Mitigation</a>
7.4.	<a href="#">Never-Indexed Literals</a>
7.5.	<a href="#">Static Huffman Encoding</a>
7.6.	<a href="#">Memory Consumption</a>
7.7.	<a href="#">Implementation Limits</a>
8.	<a href="#">IANA Considerations</a>
8.1.	<a href="#">Settings Registration</a>
8.2.	<a href="#">Stream Type Registration</a>
8.3.	<a href="#">Error Code Registration</a>
9.	<a href="#">References</a>
9.1.	<a href="#">Normative References</a>
9.2.	<a href="#">Informative References</a>
Appendix A.	<a href="#">Static Table</a>
Appendix B.	<a href="#">Sample One Pass Encoding Algorithm</a>
Appendix C.	<a href="#">Change Log</a>
C.1.	<a href="#">Since draft-ietf-quic-qpack-16</a>
C.2.	<a href="#">Since draft-ietf-quic-qpack-15</a>
C.3.	<a href="#">Since draft-ietf-quic-qpack-14</a>
C.4.	<a href="#">Since draft-ietf-quic-qpack-13</a>
C.5.	<a href="#">Since draft-ietf-quic-qpack-12</a>
C.6.	<a href="#">Since draft-ietf-quic-qpack-11</a>
C.7.	<a href="#">Since draft-ietf-quic-qpack-10</a>
C.8.	<a href="#">Since draft-ietf-quic-qpack-09</a>
C.9.	<a href="#">Since draft-ietf-quic-qpack-08</a>
C.10.	<a href="#">Since draft-ietf-quic-qpack-06</a>
C.11.	<a href="#">Since draft-ietf-quic-qpack-05</a>
C.12.	<a href="#">Since draft-ietf-quic-qpack-04</a>
C.13.	<a href="#">Since draft-ietf-quic-qpack-03</a>
C.14.	<a href="#">Since draft-ietf-quic-qpack-02</a>
C.15.	<a href="#">Since draft-ietf-quic-qpack-01</a>
C.16.	<a href="#">Since draft-ietf-quic-qpack-00</a>
C.17.	<a href="#">Since draft-ietf-quic-qcram-00</a>
<a href="#">Acknowledgments</a>	
<a href="#">Authors' Addresses</a>	

## 1. Introduction

The QUIC transport protocol ([[QUIC-TRANSPORT](#)]) is designed to support HTTP semantics, and its design subsumes many of the features of HTTP/2 ([[RFC7540](#)]). HTTP/2 uses HPACK ([[RFC7541](#)]) for compression of the header and trailer sections. If HPACK were used for HTTP/3 ([[HTTP3](#)]), it would induce head-of-line blocking for field sections due to built-in assumptions of a total ordering across frames on all streams.

QPACK reuses core concepts from HPACK, but is redesigned to allow correctness in the presence of out-of-order delivery, with flexibility for implementations to balance between resilience against head-of-line blocking and optimal compression ratio. The design goals are to closely approach the compression ratio of HPACK with substantially less head-of-line blocking under the same loss conditions.

### 1.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

Definitions of terms that are used in this document:

**HTTP fields:** Metadata sent as part of an HTTP message. The term encompasses both header and trailer fields. Colloquially, the term "headers" has often been used to refer to HTTP header fields and trailer fields; this document uses "fields" for generality.

**HTTP field line:** A name-value pair sent as part of an HTTP field section. See Section 5 of [[SEMANTICS](#)].

**HTTP field value:** Data associated with a field name, composed from all field line values with that field name in that section, concatenated together and separated with commas.

**Field section:** An ordered collection of HTTP field lines associated with an HTTP message. A field section can contain multiple field lines with the same name. It can also contain duplicate field lines. An HTTP message can include both header field and trailer field sections.

**Representation:** An instruction that represents a field line, possibly by reference to the dynamic and static tables.

**Encoder:** An implementation that encodes field sections.

**Decoder:**

An implementation that decodes encoded field sections.

**Absolute Index:** A unique index for each entry in the dynamic table.

**Base:** A reference point for relative and post-base indices.

Representations that reference dynamic table entries are relative to a Base.

**Insert Count:** The total number of entries inserted in the dynamic table.

QPACK is a name, not an acronym.

## 1.2. Notational Conventions

Diagrams use the format described in Section 3.1 of [[RFC2360](#)], with the following additional conventions:

**x (A)** Indicates that x is A bits long

**x (A+)** Indicates that x uses the prefixed integer encoding defined in [Section 4.1.1](#), beginning with an A-bit prefix.

**x ...** Indicates that x is variable-length and extends to the end of the region.

## 2. Compression Process Overview

Like HPACK, QPACK uses two tables for associating field lines ("headers") to indices. The static table ([Section 3.1](#)) is predefined and contains common header field lines (some of them with an empty value). The dynamic table ([Section 3.2](#)) is built up over the course of the connection and can be used by the encoder to index both header and trailer field lines in the encoded field sections.

QPACK defines unidirectional streams for sending instructions from encoder to decoder and vice versa.

### 2.1. Encoder

An encoder converts a header or trailer field section into a series of representations by emitting either an indexed or a literal representation for each field line in the list; see [Section 4.5](#). Indexed representations achieve high compression by replacing the literal name and possibly the value with an index to either the static or dynamic table. References to the static table and literal representations do not require any dynamic state and never risk head-of-line blocking. References to the dynamic table risk head-of-

line blocking if the encoder has not received an acknowledgement indicating the entry is available at the decoder.

An encoder MAY insert any entry in the dynamic table it chooses; it is not limited to field lines it is compressing.

QPACK preserves the ordering of field lines within each field section. An encoder MUST emit field representations in the order they appear in the input field section.

QPACK is designed to contain the more complex state tracking to the encoder, while the decoder is relatively simple.

#### **2.1.1.1. Limits on Dynamic Table Insertions**

Inserting entries into the dynamic table might not be possible if the table contains entries that cannot be evicted.

A dynamic table entry cannot be evicted immediately after insertion, even if it has never been referenced. Once the insertion of a dynamic table entry has been acknowledged and there are no outstanding references to the entry in unacknowledged representations, the entry becomes evictable. Note that references on the encoder stream never preclude the eviction of an entry, because those references are guaranteed to be processed before the instruction evicting the entry.

If the dynamic table does not contain enough room for a new entry without evicting other entries, and the entries that would be evicted are not evictable, the encoder MUST NOT insert that entry into the dynamic table (including duplicates of existing entries). In order to avoid this, an encoder that uses the dynamic table has to keep track of each dynamic table entry referenced by each field section until those representations are acknowledged by the decoder; see [Section 4.4.1](#).

##### **2.1.1.1.1. Avoiding Prohibited Insertions**

To ensure that the encoder is not prevented from adding new entries, the encoder can avoid referencing entries that are close to eviction. Rather than reference such an entry, the encoder can emit a Duplicate instruction ([Section 4.3.4](#)), and reference the duplicate instead.

Determining which entries are too close to eviction to reference is an encoder preference. One heuristic is to target a fixed amount of available space in the dynamic table: either unused space or space that can be reclaimed by evicting non-blocking entries. To achieve this, the encoder can maintain a draining index, which is the smallest absolute index ([Section 3.2.4](#)) in the dynamic table that it

will emit a reference for. As new entries are inserted, the encoder increases the draining index to maintain the section of the table that it will not reference. If the encoder does not create new references to entries with an absolute index lower than the draining index, the number of unacknowledged references to those entries will eventually become zero, allowing them to be evicted.

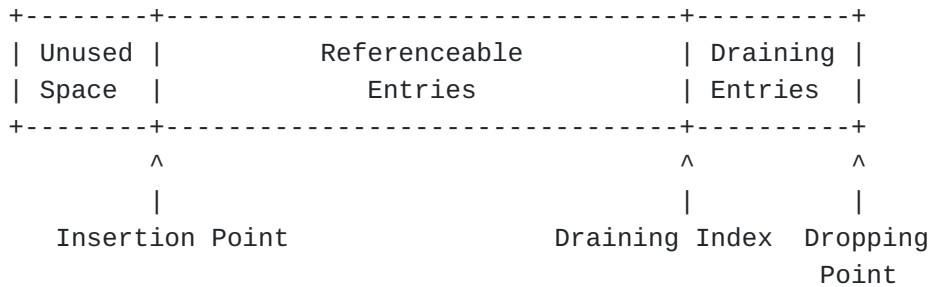


Figure 1: Draining Dynamic Table Entries

### 2.1.2. Blocked Streams

Because QUIC does not guarantee order between data on different streams, a decoder might encounter a representation that references a dynamic table entry that it has not yet received.

Each encoded field section contains a Required Insert Count ([Section 4.5.1](#)), the lowest possible value for the Insert Count with which the field section can be decoded. For a field section encoded using references to the dynamic table, the Required Insert Count is one larger than the largest absolute index of all referenced dynamic table entries. For a field section encoded with no references to the dynamic table, the Required Insert Count is zero.

When the decoder receives an encoded field section with a Required Insert Count greater than its own Insert Count, the stream cannot be processed immediately, and is considered "blocked"; see [Section 2.2.1](#).

The decoder specifies an upper bound on the number of streams that can be blocked using the `SETTINGS_QPACK_BLOCKED_STREAMS` setting; see [Section 5](#). An encoder MUST limit the number of streams that could become blocked to the value of `SETTINGS_QPACK_BLOCKED_STREAMS` at all times. If a decoder encounters more blocked streams than it promised to support, it MUST treat this as a connection error of type `QPACK_DECOMPRESSION_FAILED`.

Note that the decoder might not become blocked on every stream that risks becoming blocked.

An encoder can decide whether to risk having a stream become blocked. If permitted by the value of `SETTINGS_QPACK_BLOCKED_STREAMS`, compression efficiency can often be improved by referencing dynamic table entries that are still in transit, but if there is loss or reordering the stream can become blocked at the decoder. An encoder can avoid the risk of blocking by only referencing dynamic table entries that have been acknowledged, but this could mean using literals. Since literals make the encoded field section larger, this can result in the encoder becoming blocked on congestion or flow control limits.

### **2.1.3. Avoiding Flow Control Deadlocks**

Writing instructions on streams that are limited by flow control can produce deadlocks.

A decoder might stop issuing flow control credit on the stream that carries an encoded field section until the necessary updates are received on the encoder stream. If the granting of flow control credit on the encoder stream (or the connection as a whole) depends on the consumption and release of data on the stream carrying the encoded field section, a deadlock might result.

More generally, a stream containing a large instruction can become deadlocked if the decoder withholds flow control credit until the instruction is completely received.

To avoid these deadlocks, an encoder SHOULD avoid writing an instruction unless sufficient stream and connection flow control credit is available for the entire instruction.

### **2.1.4. Known Received Count**

The Known Received Count is the total number of dynamic table insertions and duplications acknowledged by the decoder. The encoder tracks the Known Received Count in order to identify which dynamic table entries can be referenced without potentially blocking a stream. The decoder tracks the Known Received Count in order to be able to send Insert Count Increment instructions.

A Section Acknowledgement instruction ([Section 4.4.1](#)) implies that the decoder has received all dynamic table state necessary to decode the field section. If the Required Insert Count of the acknowledged field section is greater than the current Known Received Count, Known Received Count is updated to the value of the Required Insert Count.

An Insert Count Increment instruction ([Section 4.4.3](#)) increases the Known Received Count by its Increment parameter. See [Section 2.2.2.3](#) for guidance.



## 2.2. Decoder

As in HPACK, the decoder processes a series of representations and emits the corresponding field sections. It also processes instructions received on the encoder stream that modify the dynamic table. Note that encoded field sections and encoder stream instructions arrive on separate streams. This is unlike HPACK, where encoded field sections (header blocks) can contain instructions that modify the dynamic table, and there is no dedicated stream of HPACK instructions.

The decoder MUST emit field lines in the order their representations appear in the encoded field section.

### 2.2.1. Blocked Decoding

Upon receipt of an encoded field section, the decoder examines the Required Insert Count. When the Required Insert Count is less than or equal to the decoder's Insert Count, the field section can be processed immediately. Otherwise, the stream on which the field section was received becomes blocked.

While blocked, encoded field section data SHOULD remain in the blocked stream's flow control window. A stream becomes unblocked when the Insert Count becomes greater than or equal to the Required Insert Count for all encoded field sections the decoder has started reading from the stream.

When processing encoded field sections, the decoder expects the Required Insert Count to equal the lowest possible value for the Insert Count with which the field section can be decoded, as prescribed in [Section 2.1.2](#). If it encounters a Required Insert Count smaller than expected, it MUST treat this as a connection error of type QPACK\_DECOMPRESSION\_FAILED; see [Section 2.2.3](#). If it encounters a Required Insert Count larger than expected, it MAY treat this as a connection error of type QPACK\_DECOMPRESSION\_FAILED.

### 2.2.2. State Synchronization

The decoder signals the following events by emitting decoder instructions ([Section 4.4](#)) on the decoder stream.

#### 2.2.2.1. Completed Processing of a Field Section

After the decoder finishes decoding a field section encoded using representations containing dynamic table references, it MUST emit a Section Acknowledgement instruction ([Section 4.4.1](#)). A stream may carry multiple field sections in the case of intermediate responses, trailers, and pushed requests. The encoder interprets each Section Acknowledgement instruction as acknowledging the earliest

unacknowledged field section containing dynamic table references sent on the given stream.

#### **2.2.2.2. Abandonment of a Stream**

When an endpoint receives a stream reset before the end of a stream or before all encoded field sections are processed on that stream, or when it abandons reading of a stream, it generates a Stream Cancellation instruction; see [Section 4.4.2](#). This signals to the encoder that all references to the dynamic table on that stream are no longer outstanding. A decoder with a maximum dynamic table capacity ([Section 3.2.3](#)) equal to zero MAY omit sending Stream Cancellations, because the encoder cannot have any dynamic table references. An encoder cannot infer from this instruction that any updates to the dynamic table have been received.

The Section Acknowledgement and Stream Cancellation instructions permit the encoder to remove references to entries in the dynamic table. When an entry with absolute index lower than the Known Received Count has zero references, then it is considered evictable; see [Section 2.1.1](#).

#### **2.2.2.3. New Table Entries**

After receiving new table entries on the encoder stream, the decoder chooses when to emit Insert Count Increment instructions; see [Section 4.4.3](#). Emitting this instruction after adding each new dynamic table entry will provide the timeliest feedback to the encoder, but could be redundant with other decoder feedback. By delaying an Insert Count Increment instruction, the decoder might be able to coalesce multiple Insert Count Increment instructions, or replace them entirely with Section Acknowledgements; see [Section 4.4.1](#). However, delaying too long may lead to compression inefficiencies if the encoder waits for an entry to be acknowledged before using it.

#### **2.2.3. Invalid References**

If the decoder encounters a reference in a field line representation to a dynamic table entry that has already been evicted or that has an absolute index greater than or equal to the declared Required Insert Count ([Section 4.5.1](#)), it MUST treat this as a connection error of type QPACK\_DECOMPRESSION\_FAILED.

If the decoder encounters a reference in an encoder instruction to a dynamic table entry that has already been evicted, it MUST treat this as a connection error of type QPACK\_ENCODER\_STREAM\_ERROR.

### 3. Reference Tables

Unlike in HPACK, entries in the QPACK static and dynamic tables are addressed separately. The following sections describe how entries in each table are addressed.

#### 3.1. Static Table

The static table consists of a predefined list of field lines, each of which has a fixed index over time. Its entries are defined in [Appendix A](#).

All entries in the static table have a name and a value. However, values can be empty (that is, have a length of 0). Each entry is identified by a unique index.

Note that the QPACK static table is indexed from 0, whereas the HPACK static table is indexed from 1.

When the decoder encounters an invalid static table index in a field line representation it MUST treat this as a connection error of type QPACK\_DECOMPRESSION\_FAILED. If this index is received on the encoder stream, this MUST be treated as a connection error of type QPACK\_ENCODER\_STREAM\_ERROR.

#### 3.2. Dynamic Table

The dynamic table consists of a list of field lines maintained in first-in, first-out order. Each HTTP/3 endpoint holds a dynamic table that is initially empty. Entries are added by encoder instructions received on the encoder stream; see [Section 4.3](#).

The dynamic table can contain duplicate entries (i.e., entries with the same name and same value). Therefore, duplicate entries MUST NOT be treated as an error by the decoder.

Dynamic table entries can have empty values.

##### 3.2.1. Dynamic Table Size

The size of the dynamic table is the sum of the size of its entries.

The size of an entry is the sum of its name's length in bytes, its value's length in bytes, and 32. The size of an entry is calculated using the length of its name and value without Huffman encoding applied.

### 3.2.2. Dynamic Table Capacity and Eviction

The encoder sets the capacity of the dynamic table, which serves as the upper limit on its size. The initial capacity of the dynamic table is zero. The encoder sends a Set Dynamic Table Capacity instruction ([Section 4.3.1](#)) with a non-zero capacity to begin using the dynamic table.

Before a new entry is added to the dynamic table, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to (table capacity - size of new entry). The encoder MUST NOT cause a dynamic table entry to be evicted unless that entry is evictable; see [Section 2.1.1](#). The new entry is then added to the table. It is an error if the encoder attempts to add an entry that is larger than the dynamic table capacity; the decoder MUST treat this as a connection error of type QPACK\_ENCODER\_STREAM\_ERROR.

A new entry can reference an entry in the dynamic table that will be evicted when adding this new entry into the dynamic table. Implementations are cautioned to avoid deleting the referenced name or value if the referenced entry is evicted from the dynamic table prior to inserting the new entry.

Whenever the dynamic table capacity is reduced by the encoder ([Section 4.3.1](#)), entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to the new table capacity. This mechanism can be used to completely clear entries from the dynamic table by setting a capacity of 0, which can subsequently be restored.

### 3.2.3. Maximum Dynamic Table Capacity

To bound the memory requirements of the decoder, the decoder limits the maximum value the encoder is permitted to set for the dynamic table capacity. In HTTP/3, this limit is determined by the value of SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY sent by the decoder; see [Section 5](#). The encoder MUST NOT set a dynamic table capacity that exceeds this maximum, but it can choose to use a lower dynamic table capacity; see [Section 4.3.1](#).

For clients using 0-RTT data in HTTP/3, the server's maximum table capacity is the remembered value of the setting, or zero if the value was not previously sent. When the client's 0-RTT value of the SETTING is zero, the server MAY set it to a non-zero value in its SETTINGS frame. If the remembered value is non-zero, the server MUST send the same non-zero value in its SETTINGS frame. If it specifies any other value, or omits SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY from

SETTINGS, the encoder must treat this as a connection error of type QPACK\_DECODER\_STREAM\_ERROR.

For HTTP/3 servers and HTTP/3 clients when 0-RTT is not attempted or is rejected, the maximum table capacity is 0 until the encoder processes a SETTINGS frame with a non-zero value of SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY.

When the maximum table capacity is zero, the encoder MUST NOT insert entries into the dynamic table, and MUST NOT send any encoder instructions on the encoder stream.

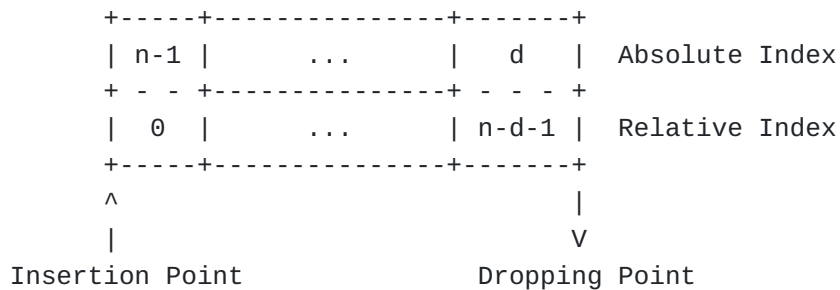
### 3.2.4. Absolute Indexing

Each entry possesses an absolute index that is fixed for the lifetime of that entry. The first entry inserted has an absolute index of "0"; indices increase by one with each insertion.

### 3.2.5. Relative Indexing

Relative indices begin at zero and increase in the opposite direction from the absolute index. Determining which entry has a relative index of "0" depends on the context of the reference.

In encoder instructions ([Section 4.3](#)), a relative index of "0" refers to the most recently inserted value in the dynamic table. Note that this means the entry referenced by a given relative index will change while interpreting instructions on the encoder stream.



n = count of entries inserted  
d = count of entries dropped

Figure 2: Example Dynamic Table Indexing - Encoder Stream

Unlike in encoder instructions, relative indices in field line representations are relative to the Base at the beginning of the encoded field section; see [Section 4.5.1](#). This ensures that references are stable even if encoded field sections and dynamic table updates are processed out of order.

In a field line representation, a relative index of "0" refers to the entry with absolute index equal to Base - 1.

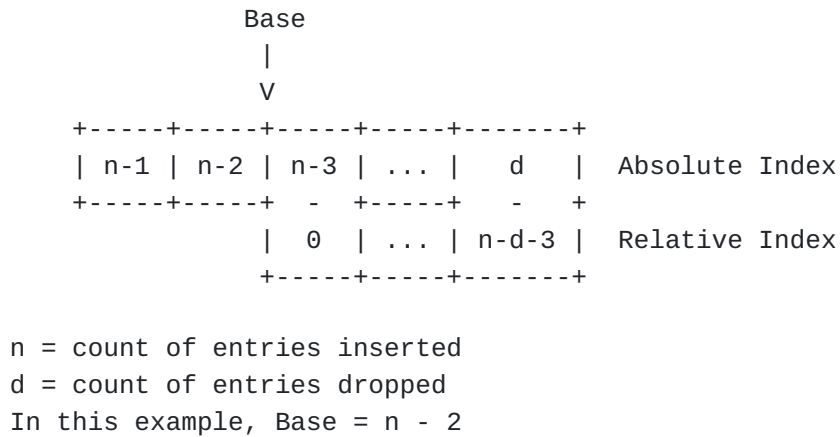


Figure 3: Example Dynamic Table Indexing - Relative Index in Representation

### 3.2.6. Post-Base Indexing

Post-Base indices are used in field line representations for entries with absolute indices greater than or equal to Base, starting at 0 for the entry with absolute index equal to Base, and increasing in the same direction as the absolute index.

Post-Base indices allow an encoder to process a field section in a single pass and include references to entries added while processing this (or other) field sections.

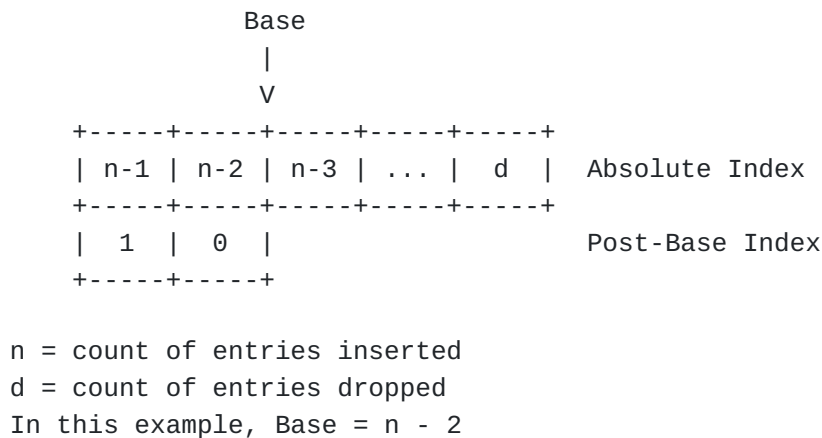


Figure 4: Example Dynamic Table Indexing - Post-Base Index in Representation

## 4. Wire Format

### 4.1. Primitives

#### 4.1.1. Prefixed Integers

The prefixed integer from Section 5.1 of [\[RFC7541\]](#) is used heavily throughout this document. The format from [\[RFC7541\]](#) is used unmodified. Note, however, that QPACK uses some prefix sizes not actually used in HPACK.

QPACK implementations MUST be able to decode integers up to and including 62 bits long.

#### 4.1.2. String Literals

The string literal defined by Section 5.2 of [\[RFC7541\]](#) is also used throughout. This string format includes optional Huffman encoding.

HPACK defines string literals to begin on a byte boundary. They begin with a single bit flag, denoted as 'H' in this document (indicating whether the string is Huffman-coded), followed by the Length encoded as a 7-bit prefix integer, and finally Length bytes of data. When Huffman encoding is enabled, the Huffman table from Appendix B of [\[RFC7541\]](#) is used without modification.

This document expands the definition of string literals by permitting them to begin other than on a byte boundary. An "N-bit prefix string literal" begins with the same Huffman flag, followed by the length encoded as an (N-1)-bit prefix integer. The prefix size, N, can have a value between 2 and 8 inclusive. The remainder of the string literal is unmodified.

A string literal without a prefix length noted is an 8-bit prefix string literal and follows the definitions in [\[RFC7541\]](#) without modification.

### 4.2. Encoder and Decoder Streams

QPACK defines two unidirectional stream types:

\*An encoder stream is a unidirectional stream of type 0x02. It carries an unframed sequence of encoder instructions from encoder to decoder.

\*A decoder stream is a unidirectional stream of type 0x03. It carries an unframed sequence of decoder instructions from decoder to encoder.

HTTP/3 endpoints contain a QPACK encoder and decoder. Each endpoint MUST initiate at most one encoder stream and at most one decoder stream. Receipt of a second instance of either stream type MUST be treated as a connection error of type H3\_STREAM\_CREATION\_ERROR. These streams MUST NOT be closed. Closure of either unidirectional stream type MUST be treated as a connection error of type H3\_CLOSED\_CRITICAL\_STREAM.

An endpoint MAY avoid creating an encoder stream if it will not be used (for example if its encoder does not wish to use the dynamic table, or if the maximum size of the dynamic table permitted by the peer is zero).

An endpoint MAY avoid creating a decoder stream if its decoder sets the maximum capacity of the dynamic table to zero.

An endpoint MUST allow its peer to create an encoder stream and a decoder stream even if the connection's settings prevent their use.

### 4.3. Encoder Instructions

An encoder sends encoder instructions on the encoder stream to set the capacity of the dynamic table and add dynamic table entries. Instructions adding table entries can use existing entries to avoid transmitting redundant information. The name can be transmitted as a reference to an existing entry in the static or the dynamic table or as a string literal. For entries that already exist in the dynamic table, the full entry can also be used by reference, creating a duplicate entry.

#### 4.3.1. Set Dynamic Table Capacity

An encoder informs the decoder of a change to the dynamic table capacity using an instruction that begins with the '001' three-bit pattern. This is followed by the new dynamic table capacity represented as an integer with a 5-bit prefix; see [Section 4.1.1](#).

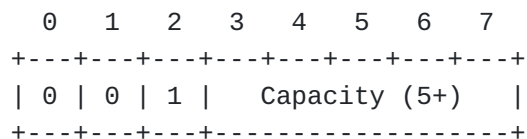


Figure 5: Set Dynamic Table Capacity

The new capacity MUST be lower than or equal to the limit described in [Section 3.2.3](#). In HTTP/3, this limit is the value of the SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY parameter ([Section 5](#)) received from the decoder. The decoder MUST treat a new dynamic table



capacity value that exceeds this limit as a connection error of type QPACK\_ENCODER\_STREAM\_ERROR.

Reducing the dynamic table capacity can cause entries to be evicted; see [Section 3.2.2](#). This MUST NOT cause the eviction of entries that are not evictable; see [Section 2.1.1](#). Changing the capacity of the dynamic table is not acknowledged as this instruction does not insert an entry.

#### 4.3.2. Insert With Name Reference

An encoder adds an entry to the dynamic table where the field name matches the field name of an entry stored in the static or the dynamic table using an instruction that starts with the '1' one-bit pattern. The second ('T') bit indicates whether the reference is to the static or dynamic table. The 6-bit prefix integer ([Section 4.1.1](#)) that follows is used to locate the table entry for the field name. When T=1, the number represents the static table index; when T=0, the number is the relative index of the entry in the dynamic table.

The field name reference is followed by the field value represented as a string literal; see [Section 4.1.2](#).

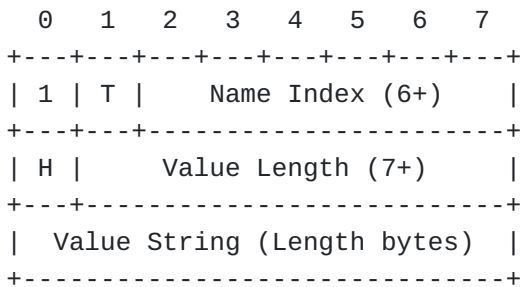


Figure 6: Insert Field Line -- Indexed Name

#### 4.3.3. Insert Without Name Reference

An encoder adds an entry to the dynamic table where both the field name and the field value are represented as string literals using an instruction that starts with the '01' two-bit pattern.

This is followed by the name represented as a 6-bit prefix string literal, and the value represented as an 8-bit prefix string literal; see [Section 4.1.2](#).

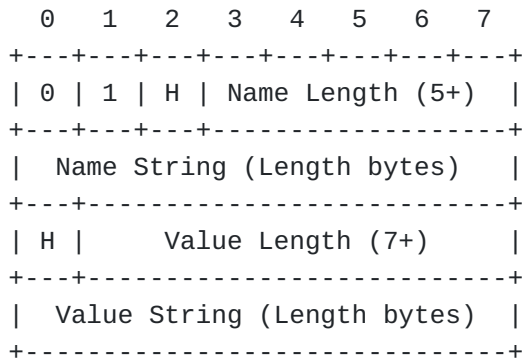


Figure 7: Insert Field Line -- New Name

#### 4.3.4. Duplicate

An encoder duplicates an existing entry in the dynamic table using an instruction that begins with the '000' three-bit pattern. This is followed by the relative index of the existing entry represented as an integer with a 5-bit prefix; see [Section 4.1.1](#).

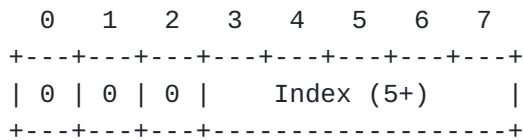


Figure 8: Duplicate

The existing entry is re-inserted into the dynamic table without resending either the name or the value. This is useful to avoid adding a reference to an older entry, which might block inserting new entries.

#### 4.4. Decoder Instructions

A decoder sends decoder instructions on the decoder stream to inform the encoder about the processing of field sections and table updates to ensure consistency of the dynamic table.

##### 4.4.1. Section Acknowledgement

After processing an encoded field section whose declared Required Insert Count is not zero, the decoder emits a Section Acknowledgement instruction. The instruction begins with the '1' one-bit pattern, followed by the field section's associated stream ID encoded as a 7-bit prefix integer; see [Section 4.1.1](#).

This instruction is used as described in [Section 2.1.4](#) and in [Section 2.2.2](#).

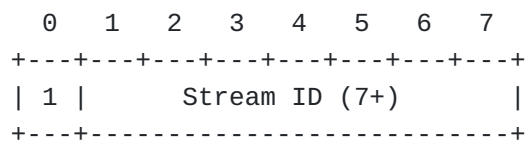


Figure 9: Section Acknowledgement

If an encoder receives a Section Acknowledgement instruction referring to a stream on which every encoded field section with a non-zero Required Insert Count has already been acknowledged, this MUST be treated as a connection error of type QPACK\_DECODER\_STREAM\_ERROR.

The Section Acknowledgement instruction might increase the Known Received Count; see [Section 2.1.4](#).

#### 4.4.2. Stream Cancellation

When a stream is reset or reading is abandoned, the decoder emits a Stream Cancellation instruction. The instruction begins with the '01' two-bit pattern, followed by the stream ID of the affected stream encoded as a 6-bit prefix integer.

This instruction is used as described in [Section 2.2.2](#).

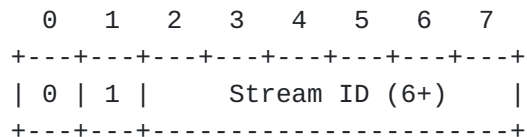


Figure 10: Stream Cancellation

#### 4.4.3. Insert Count Increment

The Insert Count Increment instruction begins with the '00' two-bit pattern, followed by the Increment encoded as a 6-bit prefix integer. This instruction increases the Known Received Count ([Section 2.1.4](#)) by the value of the Increment parameter. The decoder should send an Increment value that increases the Known Received Count to the total number of dynamic table insertions and duplications processed so far.

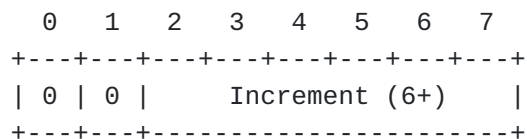


Figure 11: Insert Count Increment

An encoder that receives an Increment field equal to zero, or one that increases the Known Received Count beyond what the encoder has sent MUST treat this as a connection error of type QPACK\_DECODER\_STREAM\_ERROR.

## 4.5. Field Line Representations

An encoded field section consists of a prefix and a possibly empty sequence of representations defined in this section. Each representation corresponds to a single field line. These representations reference the static table or the dynamic table in a particular state, but do not modify that state.

Encoded field sections are carried in frames on streams defined by the enclosing protocol.

### 4.5.1. Encoded Field Section Prefix

Each encoded field section is prefixed with two integers. The Required Insert Count is encoded as an integer with an 8-bit prefix using the encoding described in [Section 4.5.1.1](#). The Base is encoded as a sign bit ('S') and a Delta Base value with a 7-bit prefix; see [Section 4.5.1.2](#).

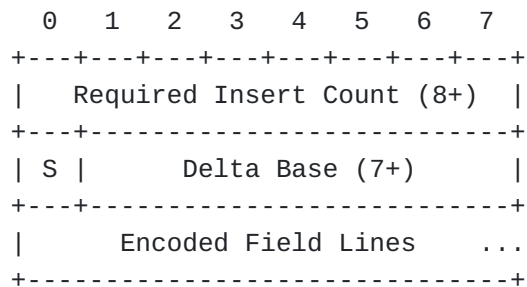


Figure 12: Encoded Field Section

#### 4.5.1.1. Required Insert Count

Required Insert Count identifies the state of the dynamic table needed to process the encoded field section. Blocking decoders use the Required Insert Count to determine when it is safe to process the rest of the field section.

The encoder transforms the Required Insert Count as follows before encoding:

```

if ReqInsertCount == 0:
    EncInsertCount = 0
else:
    EncInsertCount = (ReqInsertCount mod (2 * MaxEntries)) + 1

```

Here MaxEntries is the maximum number of entries that the dynamic table can have. The smallest entry has empty name and value strings and has the size of 32. Hence MaxEntries is calculated as

```
MaxEntries = floor( MaxTableCapacity / 32 )
```

MaxTableCapacity is the maximum capacity of the dynamic table as specified by the decoder; see [Section 3.2.3](#).

This encoding limits the length of the prefix on long-lived connections.

The decoder can reconstruct the Required Insert Count using an algorithm such as the following. If the decoder encounters a value of EncodedInsertCount that could not have been produced by a conformant encoder, it MUST treat this as a connection error of type QPACK\_DECOMPRESSION\_FAILED.

TotalNumberOfInserts is the total number of inserts into the decoder's dynamic table.

```
FullRange = 2 * MaxEntries
if EncodedInsertCount == 0:
    ReqInsertCount = 0
else:
    if EncodedInsertCount > FullRange:
        Error
    MaxValue = TotalNumberOfInserts + MaxEntries

    # MaxWrapped is the largest possible value of
    # ReqInsertCount that is 0 mod 2*MaxEntries
    MaxWrapped = floor(MaxValue / FullRange) * FullRange
    ReqInsertCount = MaxWrapped + EncodedInsertCount - 1

    # If ReqInsertCount exceeds MaxValue, the Encoder's value
    # must have wrapped one fewer time
    if ReqInsertCount > MaxValue:
        if ReqInsertCount <= FullRange:
            Error
        ReqInsertCount -= FullRange

    # Value of 0 must be encoded as 0.
    if ReqInsertCount == 0:
        Error
```

For example, if the dynamic table is 100 bytes, then the Required Insert Count will be encoded modulo 6. If a decoder has received 10 inserts, then an encoded value of 4 indicates that the Required Insert Count is 9 for the field section.

#### 4.5.1.2. Base

The Base is used to resolve references in the dynamic table as described in [Section 3.2.5](#).

To save space, the Base is encoded relative to the Required Insert Count using a one-bit sign ('S') and the Delta Base value. A sign bit of 0 indicates that the Base is greater than or equal to the value of the Required Insert Count; the decoder adds the value of Delta Base to the Required Insert Count to determine the value of the Base. A sign bit of 1 indicates that the Base is less than the Required Insert Count; the decoder subtracts the value of Delta Base from the Required Insert Count and also subtracts one to determine the value of the Base. That is:

```
if S == 0:
    Base = ReqInsertCount + DeltaBase
else:
    Base = ReqInsertCount - DeltaBase - 1
```

A single-pass encoder determines the Base before encoding a field section. If the encoder inserted entries in the dynamic table while encoding the field section and is referencing them, Required Insert Count will be greater than the Base, so the encoded difference is negative and the sign bit is set to 1. If the field section was not encoded using representations that reference the most recent entry in the table and did not insert any new entries, the Base will be greater than the Required Insert Count, so the delta will be positive and the sign bit is set to 0.

An encoder that produces table updates before encoding a field section might set Base to the value of Required Insert Count. In such case, both the sign bit and the Delta Base will be set to zero.

A field section that was encoded without references to the dynamic table can use any value for the Base; setting Delta Base to zero is one of the most efficient encodings.

For example, with a Required Insert Count of 9, a decoder receives an S bit of 1 and a Delta Base of 2. This sets the Base to 6 and enables post-base indexing for three entries. In this example, a relative index of 1 refers to the 5th entry that was added to the table; a post-base index of 1 refers to the 8th entry.

#### 4.5.2. Indexed Field Line

An indexed field line representation identifies an entry in the static table, or an entry in the dynamic table with an absolute index less than the value of the Base.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+
| 1 | T |           Index (6+)      |
+---+---+---+---+---+---+

```

Figure 13: Indexed Field Line

This representation starts with the '1' 1-bit pattern, followed by the 'T' bit indicating whether the reference is into the static or dynamic table. The 6-bit prefix integer ([Section 4.1.1](#)) that follows is used to locate the table entry for the field line. When T=1, the number represents the static table index; when T=0, the number is the relative index of the entry in the dynamic table.

#### 4.5.3. Indexed Field Line With Post-Base Index

An indexed field line with post-base index representation identifies an entry in the dynamic table with an absolute index greater than or equal to the value of the Base.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 |   Index (4+)      |
+---+---+---+---+---+---+

```

Figure 14: Indexed Field Line with Post-Base Index

This representation starts with the '0001' 4-bit pattern. This is followed by the post-base index ([Section 3.2.6](#)) of the matching field line, represented as an integer with a 4-bit prefix; see [Section 4.1.1](#).

#### 4.5.4. Literal Field Line With Name Reference

A literal field line with name reference representation encodes a field line where the field name matches the field name of an entry in the static table, or the field name of an entry in the dynamic table with an absolute index less than the value of the Base.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+
| 0 | 1 | N | T | Name Index (4+) |
+---+---+---+---+---+---+
| H |           Value Length (7+)  |
+---+---+---+---+---+---+
| Value String (Length bytes)      |
+---+---+---+---+---+---+

```

Figure 15: Literal Field Line With Name Reference

This representation starts with the '01' two-bit pattern. The following bit, 'N', indicates whether an intermediary is permitted to add this field line to the dynamic table on subsequent hops. When the 'N' bit is set, the encoded field line MUST always be encoded with a literal representation. In particular, when a peer sends a field line that it received represented as a literal field line with the 'N' bit set, it MUST use a literal representation to forward this field line. This bit is intended for protecting field values that are not to be put at risk by compressing them; see [Section 7](#) for more details.

The fourth ('T') bit indicates whether the reference is to the static or dynamic table. The 4-bit prefix integer ([Section 4.1.1](#)) that follows is used to locate the table entry for the field name. When T=1, the number represents the static table index; when T=0, the number is the relative index of the entry in the dynamic table.

Only the field name is taken from the dynamic table entry; the field value is encoded as an 8-bit prefix string literal; see [Section 4.1.2](#).

#### 4.5.5. Literal Field Line With Post-Base Name Reference

A literal field line with post-base name reference representation encodes a field line where the field name matches the field name of a dynamic table entry with an absolute index greater than or equal to the value of the Base.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | N |NameIdx(3+)|
+---+---+---+---+---+---+-----+
| H |      Value Length (7+)      |
+---+-----+
| Value String (Length bytes) |
+-----+

```

Figure 16: Literal Field Line With Post-Base Name Reference

This representation starts with the '0000' four-bit pattern. The fifth bit is the 'N' bit as described in [Section 4.5.4](#). This is followed by a post-base index of the dynamic table entry ([Section 3.2.6](#)) encoded as an integer with a 3-bit prefix; see [Section 4.1.1](#).

Only the field name is taken from the dynamic table entry; the field value is encoded as an 8-bit prefix string literal; see [Section 4.1.2](#).



#### 4.5.6. Literal Field Line Without Name Reference

The literal field line without name reference representation encodes a field name and a field value as string literals.

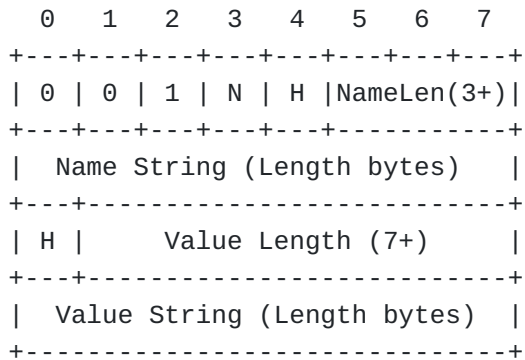


Figure 17: Literal Field Line Without Name Reference

This representation begins with the '001' three-bit pattern. The fourth bit is the 'N' bit as described in [Section 4.5.4](#). The name follows, represented as a 4-bit prefix string literal, then the value, represented as an 8-bit prefix string literal; see [Section 4.1.2](#).

## 5. Configuration

QPACK defines two settings for the HTTP/3 SETTINGS frame:

**SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY (0x1):** The default value is zero.

See [Section 3.2](#) for usage. This is the equivalent of the SETTINGS\_HEADER\_TABLE\_SIZE from HTTP/2.

**SETTINGS\_QPACK\_BLOCKED\_STREAMS (0x7):** The default value is zero.

See [Section 2.1.2](#).

## 6. Error Handling

The following error codes are defined for HTTP/3 to indicate failures of QPACK that prevent the stream or connection from continuing:

**QPACK\_DECOMPRESSION\_FAILED (0x200):**

The decoder failed to interpret an encoded field section and is not able to continue decoding that field section.

**QPACK\_ENCODER\_STREAM\_ERROR (0x201):** The decoder failed to interpret an encoder instruction received on the encoder stream.

**QPACK\_DECODER\_STREAM\_ERROR (0x202):** The encoder failed to interpret a decoder instruction received on the decoder stream.

## 7. Security Considerations

This section describes potential areas of security concern with QPACK:

- \*Use of compression as a length-based oracle for verifying guesses about secrets that are compressed into a shared compression context.

- \*Denial of service resulting from exhausting processing or memory capacity at a decoder.

### 7.1. Probing Dynamic Table State

QPACK reduces the encoded size of field sections by exploiting the redundancy inherent in protocols like HTTP. The ultimate goal of this is to reduce the amount of data that is required to send HTTP requests or responses.

The compression context used to encode header and trailer fields can be probed by an attacker who can both define fields to be encoded and transmitted and observe the length of those fields once they are encoded. When an attacker can do both, they can adaptively modify requests in order to confirm guesses about the dynamic table state. If a guess is compressed into a shorter length, the attacker can observe the encoded length and infer that the guess was correct.

This is possible even over the Transport Layer Security Protocol (TLS, see [RFC5246](#)), because while TLS provides confidentiality protection for content, it only provides a limited amount of protection for the length of that content.

**Note:** Padding schemes only provide limited protection against an attacker with these capabilities, potentially only forcing an increased number of guesses to learn the length associated with a given guess. Padding schemes also work directly against compression by increasing the number of bits that are transmitted.

Attacks like CRIME ([\[CRIME\]](#)) demonstrated the existence of these general attacker capabilities. The specific attack exploited the fact that DEFLATE ([\[RFC1951\]](#)) removes redundancy based on prefix matching. This permitted the attacker to confirm guesses a character at a time, reducing an exponential-time attack into a linear-time attack.

## 7.2. Applicability to QPACK and HTTP

QPACK mitigates but does not completely prevent attacks modeled on CRIME ([\[CRIME\]](#)) by forcing a guess to match an entire field line, rather than individual characters. An attacker can only learn whether a guess is correct or not, so is reduced to a brute force guess for the field values associated with a given field name.

The viability of recovering specific field values therefore depends on the entropy of values. As a result, values with high entropy are unlikely to be recovered successfully. However, values with low entropy remain vulnerable.

Attacks of this nature are possible any time that two mutually distrustful entities control requests or responses that are placed onto a single HTTP/3 connection. If the shared QPACK compressor permits one entity to add entries to the dynamic table, and the other to access those entries, then the state of the table can be learned.

Having requests or responses from mutually distrustful entities occurs when an intermediary either:

- \*sends requests from multiple clients on a single connection toward an origin server, or
- \*takes responses from multiple origin servers and places them on a shared connection toward a client.

Web browsers also need to assume that requests made on the same connection by different web origins ([\[RFC6454\]](#)) are made by mutually distrustful entities.

## 7.3. Mitigation

Users of HTTP that require confidentiality for header or trailer fields can use values with entropy sufficient to make guessing infeasible. However, this is impractical as a general solution because it forces all users of HTTP to take steps to mitigate attacks. It would impose new constraints on how HTTP is used.

Rather than impose constraints on users of HTTP, an implementation of QPACK can instead constrain how compression is applied in order to limit the potential for dynamic table probing.

An ideal solution segregates access to the dynamic table based on the entity that is constructing the message. Field values that are added to the table are attributed to an entity, and only the entity that created a particular value can extract that value.

To improve compression performance of this option, certain entries might be tagged as being public. For example, a web browser might make the values of the Accept-Encoding header field available in all requests.

An encoder without good knowledge of the provenance of field values might instead introduce a penalty for many field lines with the same field name and different values. This penalty could cause a large number of attempts to guess a field value to result in the field not being compared to the dynamic table entries in future messages, effectively preventing further guesses.

**Note:** Simply removing entries corresponding to the field from the dynamic table can be ineffectual if the attacker has a reliable way of causing values to be reinstalled. For example, a request to load an image in a web browser typically includes the Cookie header field (a potentially highly valued target for this sort of attack), and web sites can easily force an image to be loaded, thereby refreshing the entry in the dynamic table.

This response might be made inversely proportional to the length of the field value. Disabling access to the dynamic table for a given field name might occur for shorter values more quickly or with higher probability than for longer values.

#### 7.4. Never-Indexed Literals

Implementations can also choose to protect sensitive fields by not compressing them and instead encoding their value as literals.

Refusing to insert a field line into the dynamic table is only effective if doing so is avoided on all hops. The never-indexed literal bit (see [Section 4.5.4](#)) can be used to signal to intermediaries that a particular value was intentionally sent as a literal.

An intermediary **MUST NOT** re-encode a value that uses a literal representation with the 'N' bit set with another representation that would index it. If QPACK is used for re-encoding, a literal representation with the 'N' bit set **MUST** be used. If HPACK is used

for re-encoding, the never-indexed literal representation (see Section 6.2.3 of [[RFC7541](#)]) MUST be used.

The choice to mark that a field value should never be indexed depends on several factors. Since QPACK does not protect against guessing an entire field value, short or low-entropy values are more readily recovered by an adversary. Therefore, an encoder might choose not to index values with low entropy.

An encoder might also choose not to index values for fields that are considered to be highly valuable or sensitive to recovery, such as the Cookie or Authorization header fields.

On the contrary, an encoder might prefer indexing values for fields that have little or no value if they were exposed. For instance, a User-Agent header field does not commonly vary between requests and is sent to any server. In that case, confirmation that a particular User-Agent value has been used provides little value.

Note that these criteria for deciding to use a never-indexed literal representation will evolve over time as new attacks are discovered.

## **7.5. Static Huffman Encoding**

There is no currently known attack against a static Huffman encoding. A study has shown that using a static Huffman encoding table created an information leakage, however this same study concluded that an attacker could not take advantage of this information leakage to recover any meaningful amount of information (see [[PETAL](#)]).

## **7.6. Memory Consumption**

An attacker can try to cause an endpoint to exhaust its memory. QPACK is designed to limit both the peak and stable amounts of memory allocated by an endpoint.

The amount of memory used by the encoder is limited by the protocol using QPACK through the definition of the maximum size of the dynamic table, and the maximum number of blocking streams. In HTTP/3, these values are controlled by the decoder through the settings parameters `SETTINGS_QPACK_MAX_TABLE_CAPACITY` and `SETTINGS_QPACK_BLOCKED_STREAMS`, respectively (see [Section 3.2.3](#) and [Section 2.1.2](#)). The limit on the size of the dynamic table takes into account the size of the data stored in the dynamic table, plus a small allowance for overhead. The limit on the number of blocked streams is only a proxy for the maximum amount of memory required by the decoder. The actual maximum amount of memory will depend on how much memory the decoder uses to track each blocked stream.

A decoder can limit the amount of state memory used for the dynamic table by setting an appropriate value for the maximum size of the dynamic table. In HTTP/3, this is realized by setting an appropriate value for the `SETTINGS_QPACK_MAX_TABLE_CAPACITY` parameter. An encoder can limit the amount of state memory it uses by signaling a lower dynamic table size than the decoder allows (see [Section 3.2.2](#)).

A decoder can limit the amount of state memory used for blocked streams by setting an appropriate value for the maximum number of blocked streams. In HTTP/3, this is realized by setting an appropriate value for the `QPACK_BLOCKED_STREAMS` parameter. Streams which risk becoming blocked consume no additional state memory on the encoder.

An encoder allocates memory to track all dynamic table references in unacknowledged field sections. An implementation can directly limit the amount of state memory by only using as many references to the dynamic table as it wishes to track; no signaling to the decoder is required. However, limiting references to the dynamic table will reduce compression effectiveness.

The amount of temporary memory consumed by an encoder or decoder can be limited by processing field lines sequentially. A decoder implementation does not need to retain a complete list of field lines while decoding a field section. An encoder implementation does not need to retain a complete list of field lines while encoding a field section if it is using a single-pass algorithm. Note that it might be necessary for an application to retain a complete list of field lines for other reasons; even if QPACK does not force this to occur, application constraints might make this necessary.

While the negotiated limit on the dynamic table size accounts for much of the memory that can be consumed by a QPACK implementation, data that cannot be immediately sent due to flow control is not affected by this limit. Implementations should limit the size of unsent data, especially on the decoder stream where flexibility to choose what to send is limited. Possible responses to an excess of unsent data might include limiting the ability of the peer to open new streams, reading only from the encoder stream, or closing the connection.

## **7.7. Implementation Limits**

An implementation of QPACK needs to ensure that large values for integers, long encoding for integers, or long string literals do not create security weaknesses.

An implementation has to set a limit for the values it accepts for integers, as well as for the encoded length; see [Section 4.1.1](#). In the same way, it has to set a limit to the length it accepts for string literals; see [Section 4.1.2](#).

## 8. IANA Considerations

### 8.1. Settings Registration

This document specifies two settings. The entries in the following table are registered in the "HTTP/3 Settings" registry established in [\[HTTP3\]](#).

Setting Name	Code	Specification	Default
QPACK_MAX_TABLE_CAPACITY	0x1	<a href="#">Section 5</a>	0
QPACK_BLOCKED_STREAMS	0x7	<a href="#">Section 5</a>	0

Table 1

### 8.2. Stream Type Registration

This document specifies two stream types. The entries in the following table are registered in the "HTTP/3 Stream Type" registry established in [\[HTTP3\]](#).

Stream Type	Code	Specification	Sender
QPACK Encoder Stream	0x02	<a href="#">Section 4.2</a>	Both
QPACK Decoder Stream	0x03	<a href="#">Section 4.2</a>	Both

Table 2

### 8.3. Error Code Registration

This document specifies three error codes. The entries in the following table are registered in the "HTTP/3 Error Code" registry established in [\[HTTP3\]](#).

Name	Code	Description	Specification
QPACK_DECOMPRESSION_FAILED	0x200	Decoding of a field section failed	<a href="#">Section 6</a>
QPACK_ENCODER_STREAM_ERROR	0x201	Error on the encoder stream	<a href="#">Section 6</a>
QPACK_DECODER_STREAM_ERROR	0x202	Error on the decoder stream	<a href="#">Section 6</a>

Table 3

## 9. References

### 9.1. Normative References

[\[HTTP3\]](#)

Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-30, September 10, 2020, <<https://tools.ietf.org/html/draft-ietf-quic-http-30>>.

**[QUIC-TRANSPORT]**

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-30, September 10, 2020, <<https://tools.ietf.org/html/draft-ietf-quic-transport-30>>.

**[RFC2119]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

**[RFC7541]** Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.

**[RFC8174]** Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

**[SEMANTICS]** Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-11, August 27, 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-11.txt>>.

## 9.2. Informative References

**[CRIME]** Wikipedia, "CRIME", May 2015, <<http://en.wikipedia.org/w/index.php?title=CRIME&oldid=660948120>>.

**[PETAL]** Tan, J. and J. Nahata, "PETAL: Preset Encoding Table Information Leakage", April 2013, <<http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-13-106.pdf>>.

**[RFC1951]** Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/



RFC1951, May 1996, <<https://www.rfc-editor.org/info/rfc1951>>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <<https://www.rfc-editor.org/info/rfc2360>>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/info/rfc6454>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

## Appendix A. Static Table

This table was generated by analyzing actual Internet traffic in 2018 and including the most common header fields, after filtering out some unsupported and non-standard values. Due to this methodology, some of the entries may be inconsistent or appear multiple times with similar but not identical values. The order of the entries is optimized to encode the most common header fields with the smallest number of bytes.

Index	Name	Value
0	:authority	
1	:path	/
2	age	0
3	content-disposition	
4	content-length	0
5	cookie	
6	date	
7	etag	
8	if-modified-since	
9	if-none-match	
10	last-modified	
11	link	
12	location	
13	referer	
14	set-cookie	
15	:method	CONNECT

Index	Name	Value
16	:method	DELETE
17	:method	GET
18	:method	HEAD
19	:method	OPTIONS
20	:method	POST
21	:method	PUT
22	:scheme	http
23	:scheme	https
24	:status	103
25	:status	200
26	:status	304
27	:status	404
28	:status	503
29	accept	*/*
30	accept	application/dns-message
31	accept-encoding	gzip, deflate, br
32	accept-ranges	bytes
33	access-control-allow-headers	cache-control
34	access-control-allow-headers	content-type
35	access-control-allow-origin	*
36	cache-control	max-age=0
37	cache-control	max-age=2592000
38	cache-control	max-age=604800
39	cache-control	no-cache
40	cache-control	no-store
41	cache-control	public, max-age=31536000
42	content-encoding	br
43	content-encoding	gzip
44	content-type	application/dns-message
45	content-type	application/javascript
46	content-type	application/json
47	content-type	application/x-www-form-urlencoded
48	content-type	image/gif
49	content-type	image/jpeg
50	content-type	image/png
51	content-type	text/css
52	content-type	text/html; charset=utf-8
53	content-type	text/plain
54	content-type	text/plain; charset=utf-8
55	range	bytes=0-
56	strict-transport-security	max-age=31536000

Index	Name	Value
57	strict-transport-security	max-age=31536000; includesubdomains
58	strict-transport-security	max-age=31536000; includesubdomains; preload
59	vary	accept-encoding
60	vary	origin
61	x-content-type-options	nosniff
62	x-xss-protection	1; mode=block
63	:status	100
64	:status	204
65	:status	206
66	:status	302
67	:status	400
68	:status	403
69	:status	421
70	:status	425
71	:status	500
72	accept-language	
73	access-control-allow-credentials	FALSE
74	access-control-allow-credentials	TRUE
75	access-control-allow-headers	*
76	access-control-allow-methods	get
77	access-control-allow-methods	get, post, options
78	access-control-allow-methods	options
79	access-control-expose-headers	content-length
80	access-control-request-headers	content-type
81	access-control-request-method	get
82	access-control-request-method	post
83	alt-svc	clear
84	authorization	
85	content-security-policy	script-src 'none'; object-src 'none'; base-uri 'none'
86	early-data	1
87	expect-ct	
88	forwarded	
89	if-range	

Index	Name	Value
90	origin	
91	purpose	prefetch
92	server	
93	timing-allow-origin	*
94	upgrade-insecure-requests	1
95	user-agent	
96	x-forwarded-for	
97	x-frame-options	deny
98	x-frame-options	sameorigin

Table 4

## Appendix B. Sample One Pass Encoding Algorithm

Pseudo-code for single pass encoding, excluding handling of duplicates, non-blocking mode, available encoder stream flow control and reference tracking.

```

base = dynamicTable.getInsertCount()
requiredInsertCount = 0
for line in field_lines:
    staticIndex = staticTable.findIndex(line)
    if staticIndex is not None:
        encodeIndexReference(streamBuffer, staticIndex)
        continue

    dynamicIndex = dynamicTable.findIndex(line)
    if dynamicIndex is None:
        # No matching entry. Either insert+index or encode literal
        staticNameIndex = staticTable.findName(line.name)
        if staticNameIndex is None:
            dynamicNameIndex = dynamicTable.findName(line.name)

        if shouldIndex(line) and dynamicTable.canIndex(line):
            encodeInsert(encoderBuffer, staticNameIndex,
                        dynamicNameIndex, line)
            dynamicIndex = dynamicTable.add(line)

    if dynamicIndex is None:
        # Could not index it, literal
        if dynamicNameIndex is not None:
            # Encode literal with dynamic name, possibly above base
            encodeDynamicLiteral(streamBuffer, dynamicNameIndex,
                                base, line)
            requiredInsertCount = max(requiredInsertCount,
                                    dynamicNameIndex)
        else:
            # Encodes a literal with a static name or literal name
            encodeLiteral(streamBuffer, staticNameIndex, line)
    else:
        # Dynamic index reference
        assert(dynamicIndex is not None)
        requiredInsertCount = max(requiredInsertCount, dynamicIndex)
        # Encode dynamicIndex, possibly above base
        encodeDynamicIndexReference(streamBuffer, dynamicIndex, base)

# encode the prefix
if requiredInsertCount == 0:
    encodeIndexReference(prefixBuffer, 0, 0, 8)
    encodeIndexReference(prefixBuffer, 0, 0, 7)
else:
    wireRIC = (
        requiredInsertCount
        % (2 * getMaxEntries(maxTableCapacity))
    ) + 1;
    encodeInteger(prefixBuffer, 0x00, wireRIC, 8)
    if base >= requiredInsertCount:

```

```
        encodeInteger(prefixBuffer, 0, base - requiredInsertCount, 7)
    else:
        encodeInteger(prefixBuffer, 0x80,
                      requiredInsertCount - base - 1, 7)

return encoderBuffer, prefixBuffer + streamBuffer
```

## **Appendix C. Change Log**

**RFC Editor's Note:** Please remove this section prior to publication of a final version of this document.

### **C.1. Since draft-ietf-quic-qpack-16**

Editorial changes only

### **C.2. Since draft-ietf-quic-qpack-15**

No changes

### **C.3. Since draft-ietf-quic-qpack-14**

Added security considerations

### **C.4. Since draft-ietf-quic-qpack-13**

No changes

### **C.5. Since draft-ietf-quic-qpack-12**

Editorial changes only

### **C.6. Since draft-ietf-quic-qpack-11**

Editorial changes only

### **C.7. Since draft-ietf-quic-qpack-10**

Editorial changes only

### **C.8. Since draft-ietf-quic-qpack-09**

\*Decoders MUST emit Header Acknowledgements (#2939)

\*Updated error code for multiple encoder or decoder streams (#2970)

\*Added explicit defaults for new SETTINGS (#2974)

### **C.9. Since draft-ietf-quic-qpack-08**

\*Endpoints are permitted to create encoder and decoder streams even if they can't use them (#2100, #2529)

\*Maximum values for settings removed (#2766, #2767)

**C.10. Since draft-ietf-quickpack-06**

\*Clarify initial dynamic table capacity maximums (#2276, #2330, #2330)

**C.11. Since draft-ietf-quickpack-05**

\*Introduced the terms dynamic table capacity and maximum dynamic table capacity.

\*Renamed SETTINGS\_HEADER\_TABLE\_SIZE to SETTINGS\_QPACK\_MAX\_TABLE\_CAPACITY.

**C.12. Since draft-ietf-quickpack-04**

\*Changed calculation of Delta Base Index to avoid an illegal value (#2002, #2005)

**C.13. Since draft-ietf-quickpack-03**

\*Change HTTP settings defaults (#2038)

\*Substantial editorial reorganization

**C.14. Since draft-ietf-quickpack-02**

\*Largest Reference encoded modulo MaxEntries (#1763)

\*New Static Table (#1355)

\*Table Size Update with Insert Count=0 is a connection error (#1762)

\*Stream Cancellations are optional when SETTINGS\_HEADER\_TABLE\_SIZE=0 (#1761)

\*Implementations must handle 62 bit integers (#1760)

\*Different error types for each QPACK stream, other changes to error handling (#1726)

\*Preserve header field order (#1725)

\*Initial table size is the maximum permitted when table is first usable (#1642)

**C.15. Since draft-ietf-quickpack-01**

\*Only header blocks that reference the dynamic table are acknowledged (#1603, #1605)



#### **C.16. Since draft-ietf-quick-qpack-00**

- \*Renumbered instructions for consistency (#1471, #1472)
- \*Decoder is allowed to validate largest reference (#1404, #1469)
- \*Header block acknowledgments also acknowledge the associated largest reference (#1370, #1400)
- \*Added an acknowledgment for unread streams (#1371, #1400)
- \*Removed framing from encoder stream (#1361, #1467)
- \*Control streams use typed unidirectional streams rather than fixed stream IDs (#910, #1359)

#### **C.17. Since draft-ietf-quick-qcram-00**

- \*Separate instruction sets for table updates and header blocks (#1235, #1142, #1141)
- \*Reworked indexing scheme (#1176, #1145, #1136, #1130, #1125, #1314)
- \*Added mechanisms that support one-pass encoding (#1138, #1320)
- \*Added a setting to control the number of blocked decoders (#238, #1140, #1143)
- \*Moved table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

#### **Acknowledgments**

The IETF QUIC Working Group received an enormous amount of support from many people.

The compression design team did substantial work exploring the problem space and influencing the initial draft. The contributions of design team members Roberto Peon, Martin Thomson, and Dmitri Tikhonov are gratefully acknowledged.

The following people also provided substantial contributions to this document:

- \*Bence Beky
- \*Alessandro Ghedini
- \*Ryan Hamilton

\*Robin Marx

\*Patrick McManus

\* (Kazuho Oku)

\*Lucas Pardue

\*Biren Roy

\*Ian Swett

This draft draws heavily on the text of [[RFC7541](#)]. The indirect input of those authors is also gratefully acknowledged.

Buck's contribution was supported by Google during his employment there.

A portion of Mike's contribution was supported by Microsoft during his employment there.

#### **Authors' Addresses**

Charles 'Buck' Krasic  
Netflix

Email: [ckrasic@netflix.com](mailto:ckrasic@netflix.com)

Mike Bishop  
Akamai Technologies

Email: [mbishop@evequefou.be](mailto:mbishop@evequefou.be)

Alan Frindell (editor)  
Facebook

Email: [afrind@fb.com](mailto:afrind@fb.com)