

# Deep Dive into 64-Bit Support for IBM MQ

z/TPF Education

Cameron Doggett

2024 TPF Users Group Conference  
May 05-08, New Orleans, LA

**IBM Z**



# Disclaimer

Any reference to future plans are for planning purposes only. IBM reserves the right to change those plans at its discretion. Any reliance on such a disclosure is solely at your own risk. IBM makes no commitment to provide additional information in the future.



# Agenda

- Memory
  - Defining MQM
  - Creating 64-bit queues
  - Message chains
- Checkpoint
- Sweeper
  - Monitoring MQM usage
- Restart & recovery

# Memory

# 31-Bit IBM MQ Memory

- 31-bit IBM MQ queues on z/TPF are **only capable of supporting less than 2 GB** of message data in memory at any single point
- Messages are stored in system work blocks (SWBs) which if exhausted results in a catastrophic system error
- **SWBs are used for many system functions**, not just IBM MQ, so one function is capable of depriving others of the resource
- As message sizes and volumes continue to trend larger, the likelihood that the 31-bit sweeper needs to be engaged to support the growing amount of data also increases
- The 31-bit sweeper frees SWBs to prevent impacting the broader system, but at the expense of some additional I/O and computational overhead
- When frequently engaged over a longer period of time, this can result in higher CPU usage and volume of I/O

# 64-Bit IBM MQ Memory

- 64-bit IBM MQ queues leverage a new memory type referred to as 64-bit IBM MQ memory, or **MQM**
  - *Note that the IBM MQ product defines MQM as message queue management, but **in this presentation MQM will refer to 64-bit IBM MQ memory alone***
- MQM takes advantage of the 64-bit addressable space and resides above the 2 GB bar
- This means MQM can be defined in capacities on the **order of tens to hundreds of gigabytes**
- At IPL time, the defined MQM size is reserved by the system and later dispensed as 4 KB entries for storing message data
- This **prevents holding up system restart** having to initialize the entirety of a potentially large table

# Defining MQM

# Defining MQM

- The amount of MQM defined to the system is in keypoint A (CTKA)
  - Set and modified online with **ZCTKA DEFINE** and **ZCTKA ALTER** and the **MQM** parameter
  - Ensure to keep the offline copy of the keypoint in sync (**MQM** parameter of **CORREQ** macro)
  - The value specified for **MQM** is in megabytes
  - Can specify a different amount of **MQM** per memory configuration
- As a quick starting point, defining a multiple of the amount of space reserved for SWBs is an option
- To tailor the amount of MQM more specifically to the workload of the system, refer to the following resources:
  - Data reductions to gather message rates and sizes to target *at least* 10 seconds per queue
  - Highwater mark in queue manager display (**ZMQSC DISPLAY QMGR**) after a long period of uptime



# Quickly Determining How Much MQM to Define from SWB Allocation

- Use **ZCTKA DISPLAY** to view the number of defined SWBs
- Divide the number of SWBs by 1,000 to get number of megabytes that the SWBs equate to
- Optionally, multiply the result by a factor to supplement the MQM allocation
  - A factor of 5 should provide a reasonably sized buffer for growth
- The end result is the number of megabytes to allocate for MQM
- An example:
  - A system with 500,000 SWBs equates to 500 MB reserved for those SWBs
  - Multiplying by a factor of 5 results in an MQM allocation of 2,500 MB (2.5 GB)

# Determining How Much MQM to Define with Data Reduction

- Collect a data reduction from the running system
- Look for the mean message rates and message lengths for the queues you intend to migrate to 64-bit
  - In the data reduction, the rates and lengths are broken down by persistence type:
    - ADD PERSISTENT MESSAGE RATE
    - ADD NONPERSISTENT MESSAGE RATE
    - ADDED PERSISTENT MESSAGE LENGTH
    - ADDED NONPERSISTENT MESSAGE LENGTH

# Determining How Much MQM to Define with Data Reduction

- To compute the rate (bytes per second) that data is being added to a given queue:
  - (mean persistent message rate \* mean persistent message length) + (mean nonpersistent message rate \* mean nonpersistent message length)
  - Divide this rate by 1,000,000 (1e6) to get the rate in megabytes per second
- With the computed rate (megabytes per second) for each queue you intend to migrate to 64-bit, sum the rates to obtain the total rate
- Determine the *minimum* number of seconds worth of messages to support in memory
  - We recommend *at least* 10 seconds, but the more the better

# Determining How Much MQM to Define with Data Reduction

- Multiply the total rate by the number of seconds worth of messages to support to get the *minimum* number of megabytes to define for MQM
- Lastly, multiply the obtained figure by a factor to account for growth and to help prevent ever exhausting MQM
  - For example, this factor could be 5 or 10

# Example of Determining How Much MQM to Define with Data Reduction

Suppose these values are gathered from a data reduction in this simple example.

Queue name	Mean added persistent message rate (msg/sec)	Mean added persistent message length (bytes/msg)	Mean added nonpersistent message rate (msg/sec)	Mean added nonpersistent message length (bytes/msg)
Q1	33	64000	650	64000
Q2	1000	12000	100	128000
Q3	1500	10000	0	0

# Example of Determining How Much MQM to Define with Data Reduction

Compute the rate that data is being added to each queue.

Queue name	Mean added persistent message rate (msg/sec)	Mean added persistent message length (bytes/msg)	Mean added <b>nonpersistent</b> message rate (msg/sec)	Mean added <b>nonpersistent</b> message length (bytes/msg)	Mean rate of added message data (MB/sec)
Q1	33	64000	650	64000	43.71
Q2	1000	12000	100	128000	24.8
Q3	1500	10000	0	0	15

# Example of Determining How Much MQM to Define with Data Reduction

Calculate the sum of each of the individual rates to get the total rate of data being added. Suppose we will target supporting 10 seconds of message data in memory.

Queue name	Mean rate of added message data (MB/sec)
Q1	43.71
Q2	24.8
Q3	15

**Total rate of data being added to these queues (MB/sec) =  $43.71 + 24.8 + 15 = 83.51$**

**Target number of seconds worth of messages to support in memory = 10**

# Example of Determining How Much MQM to Define with Data Reduction

Multiply the total rate by the target number of seconds to support to get the minimum amount of MQM required. As suggested, we will multiply by a factor of 5 to account for future growth.

$$\text{Total rate of data being added to these queues (MB/sec)} = 43.71 + 24.8 + 15 = 83.51$$

Target number of seconds worth of messages to support in memory = 10

$$\text{Minimum amount of MQM to define (MB)} = 83.51 * 10 = \mathbf{835.1}$$

$$\text{Recommended amount of MQM to define (MB)} = 835.1 * 5 = \mathbf{4,175.5}$$

(4.17 GB)



# Creating and Migrating to 64-Bit Queues

# Create or Migrate to 64-Bit Queues

- Define a new queue
  - **ZMQSC DEFINE QL-*name* 64BIT-YES**
- Migrate an existing 31-bit queue
  - **ZMQSC ALTER QL-*name* 64BIT-YES**
  - Stages the queue for migration and switches to using 64-bit memory and file structures upon being emptied, seamlessly migrating the queue all without application awareness
  - If necessary, can fallback a queue to 31-bit using the same mechanism by specifying **64BIT-NO** instead

# Considerations for 64-Bit Queues

- The primary use case for 64-bit queues is for **high volume, FIFO queues**
  - For example, business event queues that handle large amounts of data
- The following restrictions are in place for 64-bit queues to enforce sequential (FIFO) access:
  - Browsing is currently not supported
  - Searching by message ID is not supported
  - Searching by correlation ID is not supported
- A 31-bit queue that uses any of the previous functionality **is not a candidate** for migration to 64-bit

# Identifying Candidate Queues for Migration

APAR PJ46881 (Nov 2022) adds the **GETDIAG** option to **ZMQSC DISPLAY QL** to display queues which are using restricted functionality for 64-bit

```
MQSC0278I 14.39.34 LOCAL QUEUE MQGET DIAGNOSTICS DISPLAY
```

Queue Name	SEARCH/ BROWSE	XMITQ GET	LAST PROG
-----	-----	-----	-----
CalculatorQueue	NO	N/A	
CalculatorSyncReplyQueue	NO	N/A	
AsyncCalculatorQueue	NO	N/A	
MY.MEMQ.1	NO	N/A	
<b>MY.MEMQ.2</b>	<b>YES (CMB)</b>	<b>N/A</b>	<b>ABCD</b>
MY.XMITQ.1	NO	NO	

END OF DISPLAY

C = search by correlation ID  
M = search by message ID  
B = browsing

# Message Chains

# MQM Block Chaining

- An individual MQM block is 4 KB, whereas a single SWB is 1 KB
  - The larger block size leads to a smaller number of links to form messages
  - Making the block size too large, however, would increase the chance of wasting memory
    - For example, a block size of 32 KB with an average message size in the range of 3 – 7 KB
- MQM blocks are chained together to form the message as it exists on the queue
  - The first MQM block for a given message has the MQMM eyecatcher
  - Any overflow blocks chained off the MQMM block all have the MQMO eyecatcher

# Single MQM Block Message Example

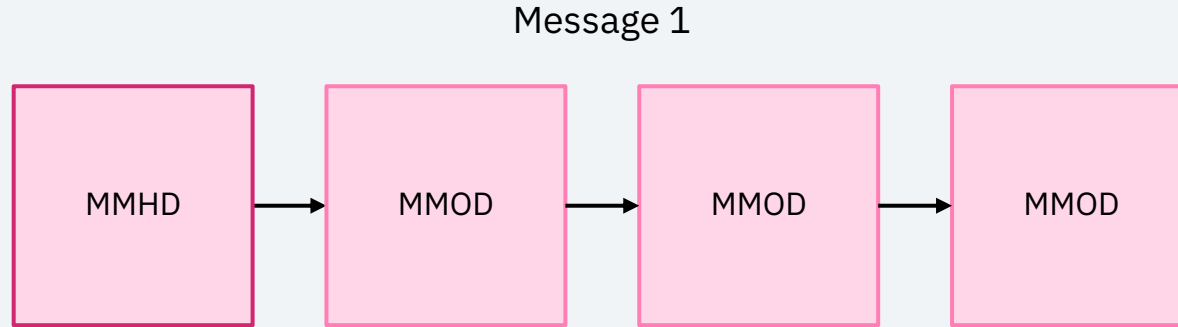
A 3200-byte uncompressed message is MQPUT to a 64-bit local, non-transmission queue. The message fits entirely in the first, single MQMM block.

Message 1



# Multiple SWB Message Example 1

The same 3200-byte uncompressed message placed on a 31-bit queue requires **4 SWBs!**



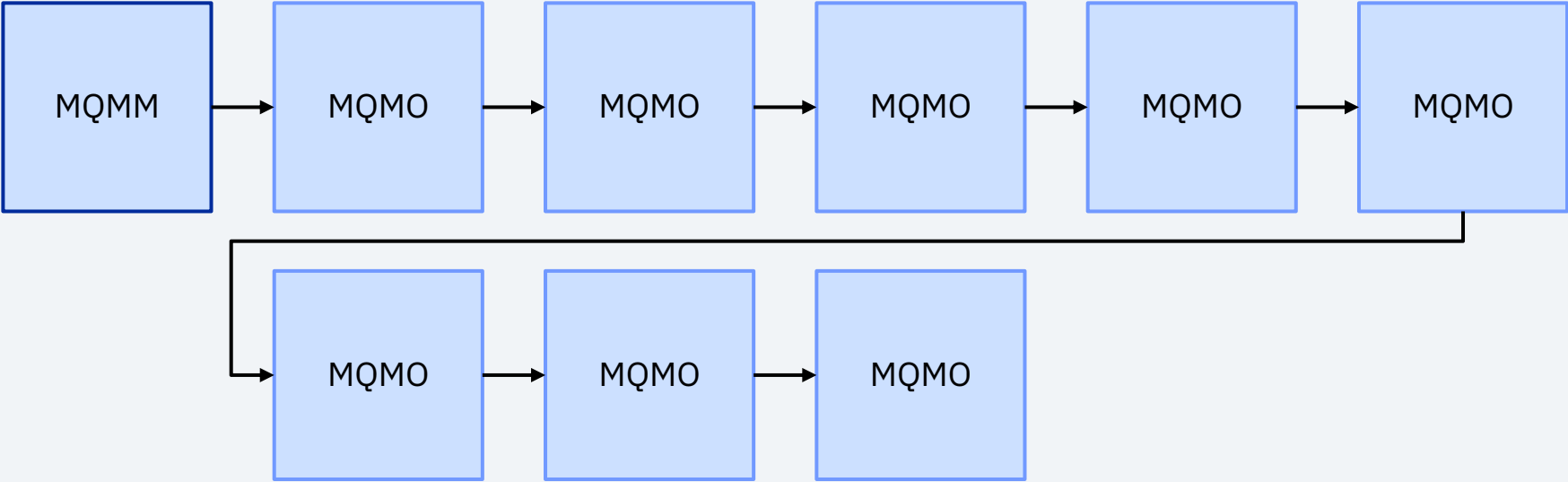


# Multiple MQM Block Message Example

A 34000-byte message is MQPUT to a 64-bit local, non-transmission queue. The message requires eight overflow (MQMO) blocks in addition to the first block (MQMM).

This 34 KB message could either be uncompressed or compressed down from an original size such as 1 MB, for example.

Message 2

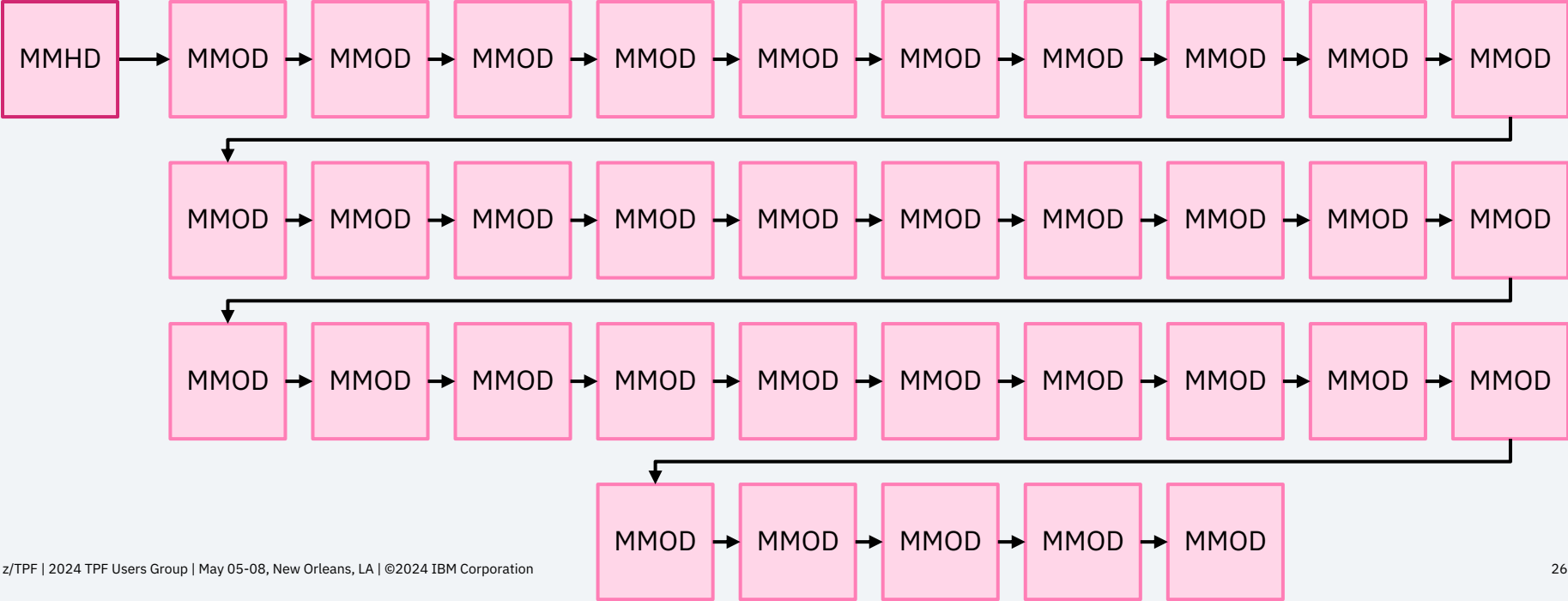


# Multiple SWB Message Example 2

The same 34000-byte message placed on a 31-bit queue requires **36 SWBs**!

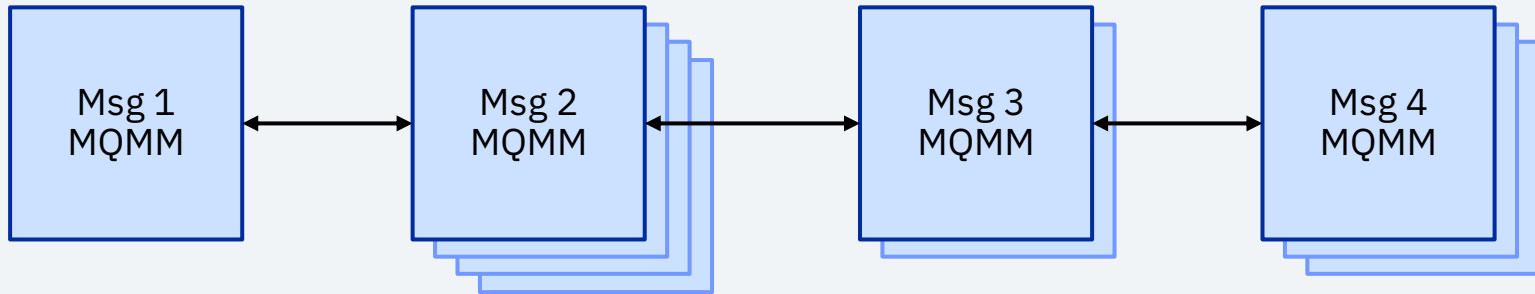
This 34 KB message could either be uncompressed or compressed down from an original size such as 1 MB, for example.

Message 2



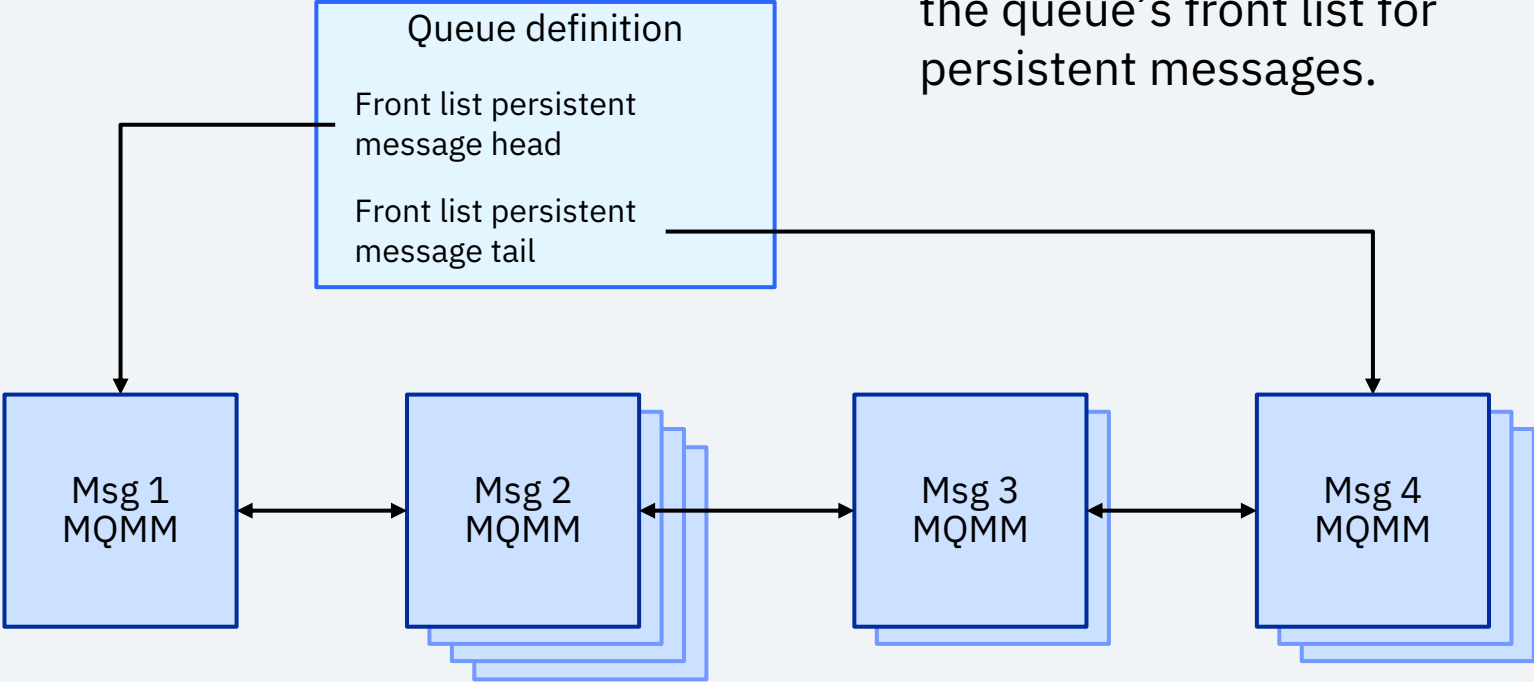
# Chaining Between Messages

Messages placed on the same queue are doubly-linked to each other. This is common between 31-bit and 64-bit queues.



# Queue Message Lists

Assuming messages 1 – 4 are persistent, they are attached to the queue's front list for persistent messages.



# Checkpoint

# Recovery Log and Checkpoint Overview

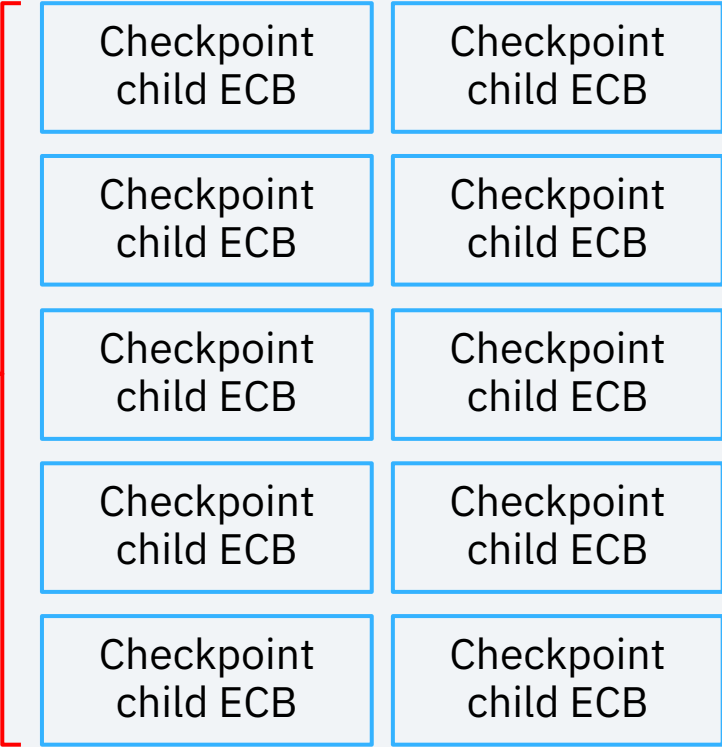
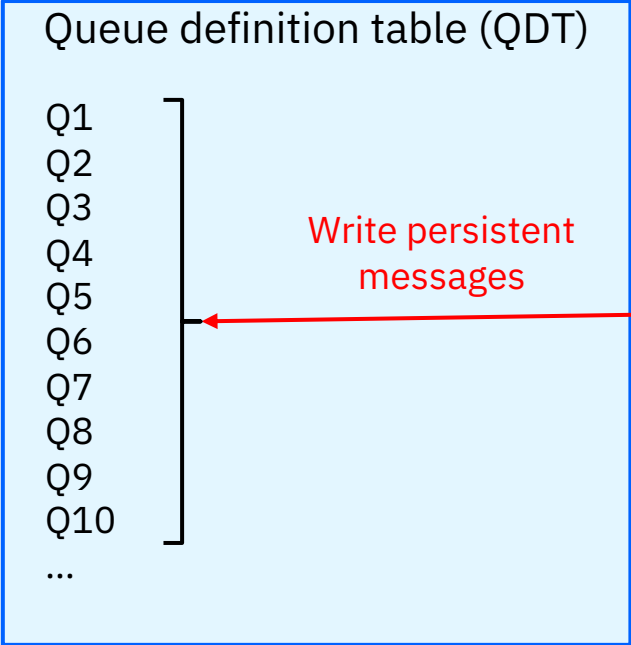
- IBM MQ uses the z/TPF recovery log to record every event, such as an MQPUT or MQGET, relating to a **persistent** message to guarantee its one-time delivery
- The IBM MQ checkpoint process runs at a regular interval to write persistent messages to a separate location so that space on the recovery log can be freed
- In the event of a system outage, persistent messages are restored to their queues from the checkpoint in tandem with the recovery log
- This overall, high-level architecture is shared by both 31-bit and 64-bit IBM MQ
  - However, the actual **implementations differ significantly**

# Checkpoint Process Example

Checkpoint parent ECB

Multiple ECBs are used to enable checkpoint to complete in a timely manner.

The child ECBs can be filing *either* 31-bit or 64-bit queues, as part of the same process.

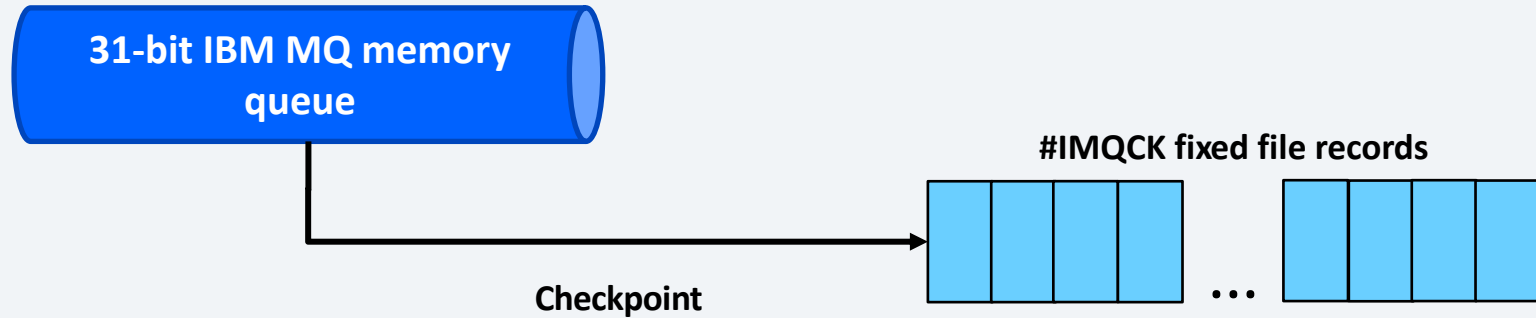


# Checkpointing a 31-Bit Queue

- The checkpoint for 31-bit queues performs a **full replace**, or copy, of the contents of the queues ***each time***
- This is possible because **at most 2 GB of message data** can exist across all the 31-bit queues on the system
- However, if a queue stalls, the same persistent messages are **written repeatedly**
- Messages are written to #IMQCK fixed file records
- Checkpointing ECB will process as many as **175 concurrent I/O operations**
- The **queue lock is held for entire duration** that its messages are being written



# 31-Bit Checkpoint Visual Overview



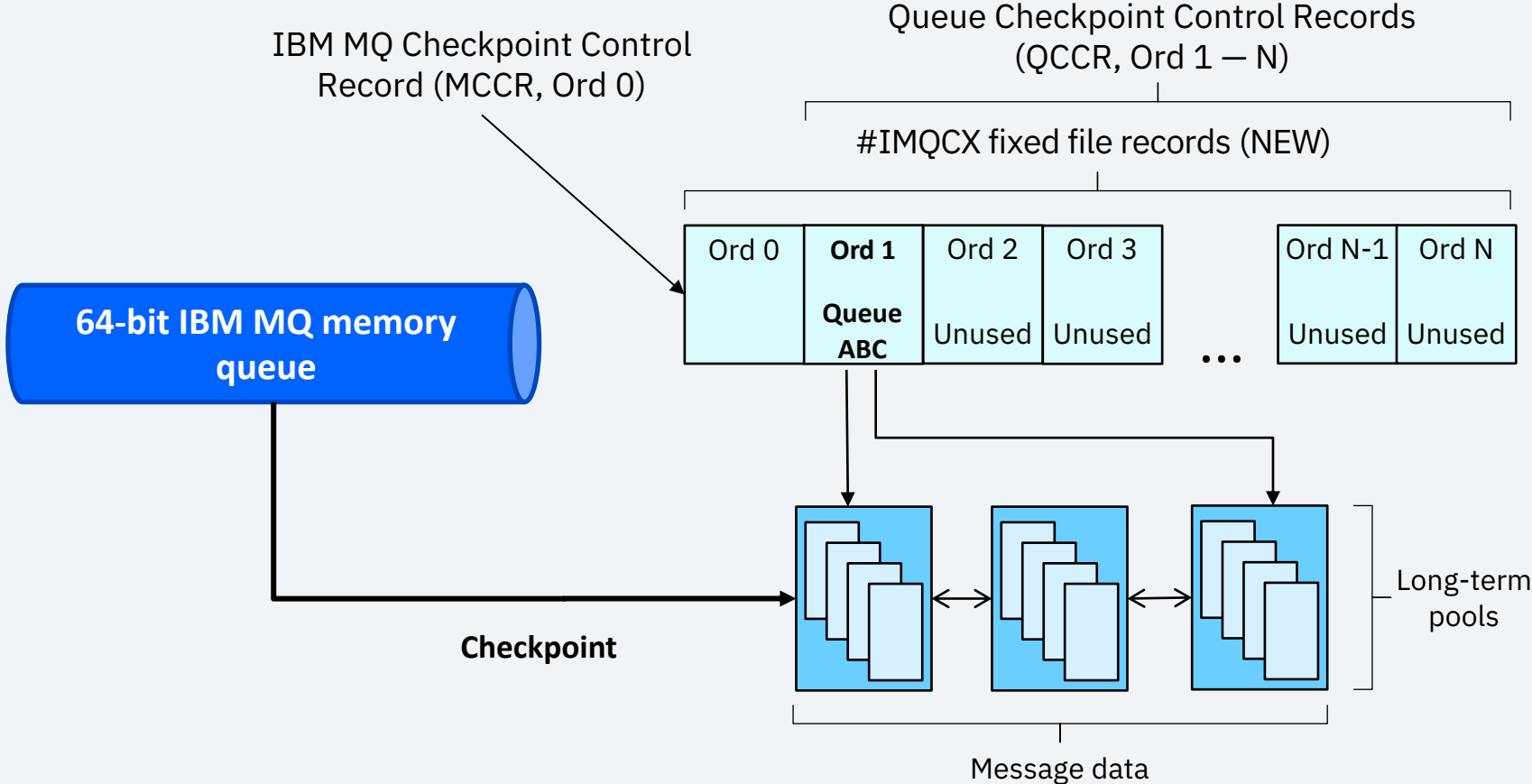
# Checkpointing a 64-Bit Queue

- The checkpoint for 64-bit queues is **incremental**
  - Only new messages created (for example, MQPUT) since the last checkpoint and that still exist at the time of the next checkpoint will be written
  - Previously checkpointed messages which have since been consumed (for example, MQGET) or expired are removed from the checkpoint
- This implies that a persistent message is **written at most one time to checkpoint**
  - Consequently, checkpoint is incredibly efficient when a 64-bit queue stalls
- Messages are **written into pool records** that are then **anchored off an #IMQCX fixed file record** assigned to an individual queue, known as a **QCCR**
- **FARF6 capable** and recommended

# Checkpointing a 64-Bit Queue

- Checkpointing ECB will process as many as **250 concurrent I/O operations**
- The **queue lock is held minimally**, only to copy message data and **never when blocking for I/O**
- This enables messages to be added and removed from the queue while the queue is being actively checkpointed

# 64-Bit Checkpoint Visual Overview



# Summary of Checkpoint Improvements

31-Bit Checkpoint	64-Bit Checkpoint
<ul style="list-style-type: none"><li>• Full-replace each time</li></ul>	<ul style="list-style-type: none"><li>• <b>Incremental</b></li></ul>
<ul style="list-style-type: none"><li>• Redundant I/O with stalled queue</li></ul>	<ul style="list-style-type: none"><li>• <b>No redundant I/O</b>, very efficient with stalled queue</li></ul>
<ul style="list-style-type: none"><li>• 175 concurrent I/Os</li></ul>	<ul style="list-style-type: none"><li>• <b>250</b> concurrent I/Os</li></ul>
<ul style="list-style-type: none"><li>• Queue lock held for entire duration</li></ul>	<ul style="list-style-type: none"><li>• Queue lock held <b>minimally</b></li></ul>
<ul style="list-style-type: none"><li>• Excludes application activity (for example, MQPUT and MQGET)</li></ul>	<ul style="list-style-type: none"><li>• Enables <b>concurrent application activity</b></li></ul>

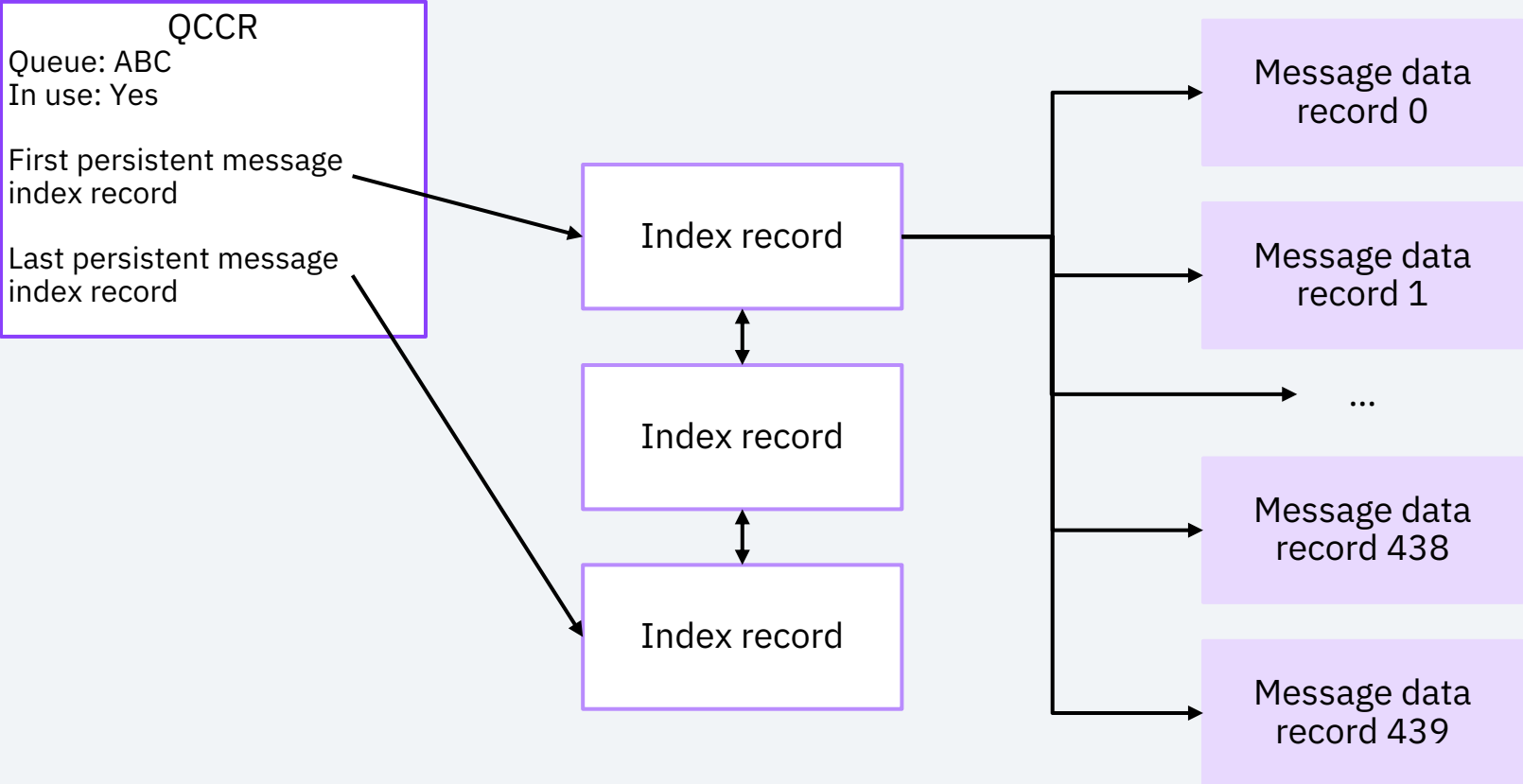
# Queue Checkpoint Control Record (QCCR)

- #IMQCX ordinals 1 through  $N - 1$ 
  - Where  $N$  is the number of #IMQCX records defined
- Acts as the root record off which all message persistence occurs for the queue
- Assigned to an individual 64-bit queue at the time the queue is defined or altered to 64-bit
- This assignment occurs across the complex in a loosely-coupled environment – it is recommended that the same number of #IMQCX records be defined on every processor

# Queue Checkpoint Control Record (QCCR)

- Points to pool records which index other pools containing message data (two levels of indirection)
  - The index records are doubly-linked to each other, with the first-in-chain record pointing to the oldest messages in the chain
  - Each index record points to 440 pool records which contain the actual message data
- This file structure enables high concurrency for message writes and reads
- Retains up to a single index record dedicated for future use by the queue
  - Known as the *queue reserved pool set*
  - Alleviates pool thrashing, especially in steady state with messages processed regularly

# QCCR Checkpoint Example





# IBM MQ Checkpoint Control Record (MCCR)

- #IMQCX ordinal 0
- Control record reserved by IBM MQ
- Can retain many index records for future use by any of the queues
  - Known as the *subsystem reserved pool set*
  - Alleviates pool thrashing, especially in steady state

# Reusing Obtained Pools

- To help **prevent thrashing** (quickly allocating and releasing) **pool records** in steady state, **a number of pools will be retained** by 64-bit IBM MQ for reuse, both on the individual queue level and the queue manager level
- The individual queue's reserve is known as the ***queue reserved pool set*** (441 pools)
- The queue manager's reserve is known as the ***subsystem reserved pool set*** (up to 88000 pools)
- These **reserves are replenished lazily** as messages are processed and previously used index records are no longer needed
- Before allocating new pools from the system, the queue reserved pool set and subsystem reserved pool set will be interrogated first when pools are needed

# RIAT Definition

- Define in the RIAT table the pool types and options to use for the X'FE1B' record ID, which is the ID used across 64-bit IBM MQ records on z/TPF
- **FARF6** can be used and is recommended
- X'FE1B' pool records should **not** be made VFA candidates
  - During normal operations, message data is only ever written to file and never read
  - X'FE1B' pool records would therefore overwhelm VFA for little to no benefit

# Sweeper

# Sweeper Overview

- The IBM MQ sweeper on z/TPF is responsible for moving messages between memory and DASD to enable a larger number of messages on queues than memory will support
- Sweeping refers to the removal of messages from memory to DASD
  - Potentially avoiding the I/O to file the message if already persisted (64-bit sweeper only)
- Unsweping refers to the restoration of messages into memory from DASD
- Both persistent and nonpersistent messages can be swept
- The exhaustion of either SWBs or MQM entails a catastrophic system error
  - Thus, the sweeper function is critical to ensuring the integrity of the system
  - The sweeper runs at least once a second to see if queues need to be swept

# 31-Bit Sweeper

## Sweeping Messages

- The sweeper for 31-bit queues **writes messages (both persistent and nonpersistent) to z/TPFCS**, which is **separate** from the #IMQCK fixed file records that are used to checkpoint 31-bit queues
- This means the same swept persistent message is rewritten to file **even when it might have already been checkpointed**
- This can result in a large amount of redundant, costly I/O
- The queue lock is held intermittently between I/O to copy message data

# 31-Bit Sweeper

## UnswEEPing Messages

- Swept 31-bit messages are **only unswept back into memory reactively**, or when immediately needed by an application or for transmission
- The **queue lock is held for the entire duration of the unsweep**, even through I/O operations
- This can result in increased latency and reduced throughput
- On an unsweep, the unswept messages are **written to the recovery log**
- If the same message is swept and unswept multiple times, it is **rewritten to z/TPFCS and the recovery log each time**

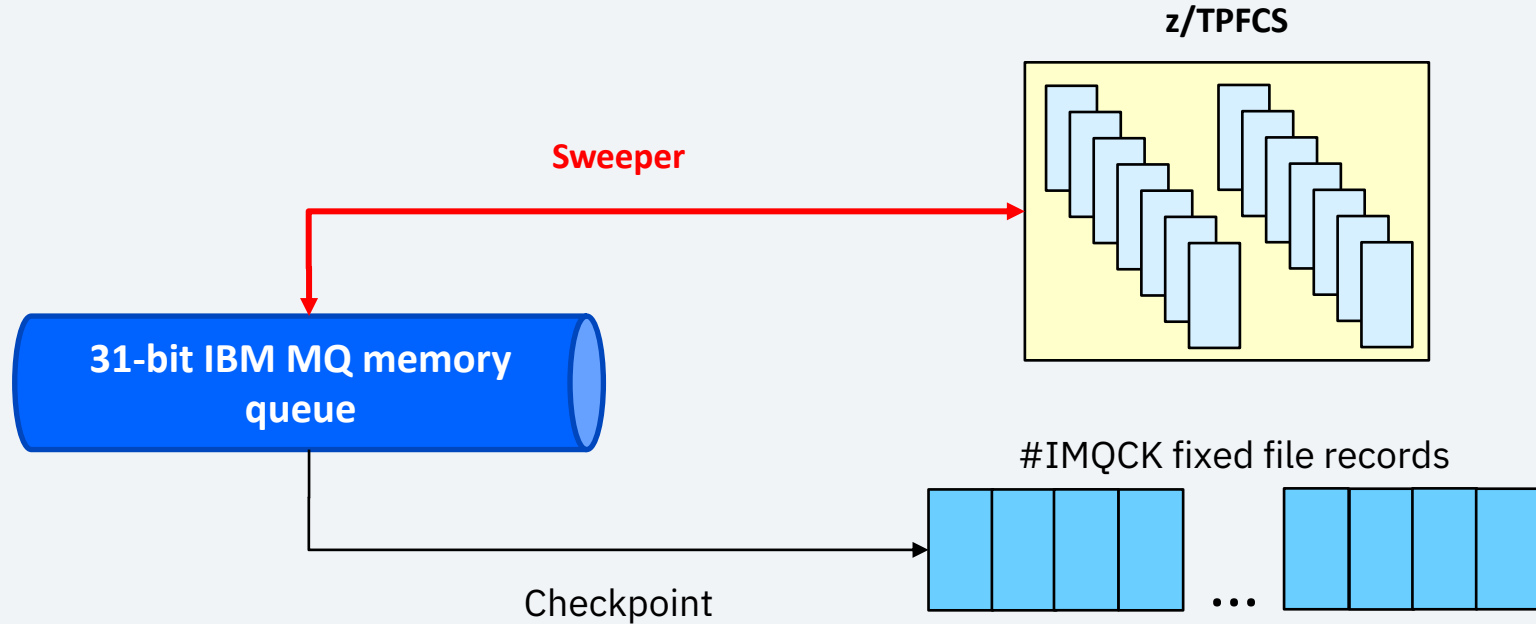
# 31-Bit Sweeper

## Processing

- The 31-bit sweeper sweeps queues **serially**, in the **same order each time**, **exhaustively**, and **unidirectionally** for the same queue (that is, toward the front or rear but not both at the same time)
- With little coordination between sweep and unsweep, there is an **increased risk of message thrashing** for 31-bit queues in certain scenarios
- Message thrashing is sweeping and unsweeping the same messages in a short period of time, potentially repeatedly

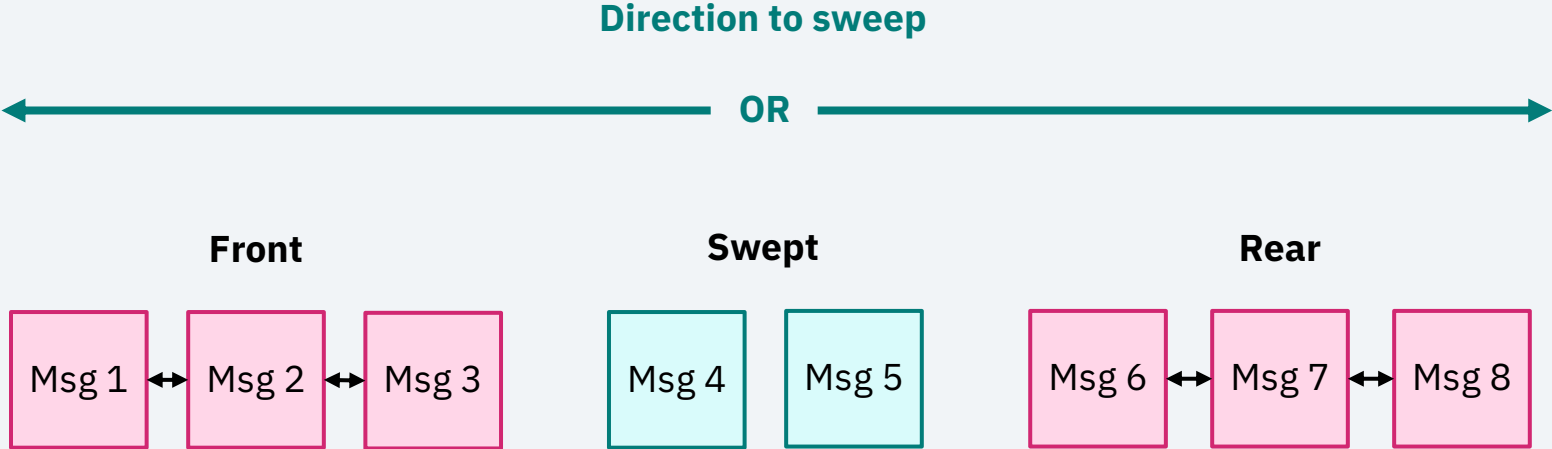


# 31-Bit Sweeper Visual Overview



# Unidirectional Sweeping (31-Bit)

For an already swept queue, the 31-bit sweeper will only sweep messages in a single direction within the same sweep, either toward the front or the rear of the queue but not both.



# 64-Bit Sweeper

## Sweeping Persistent Messages

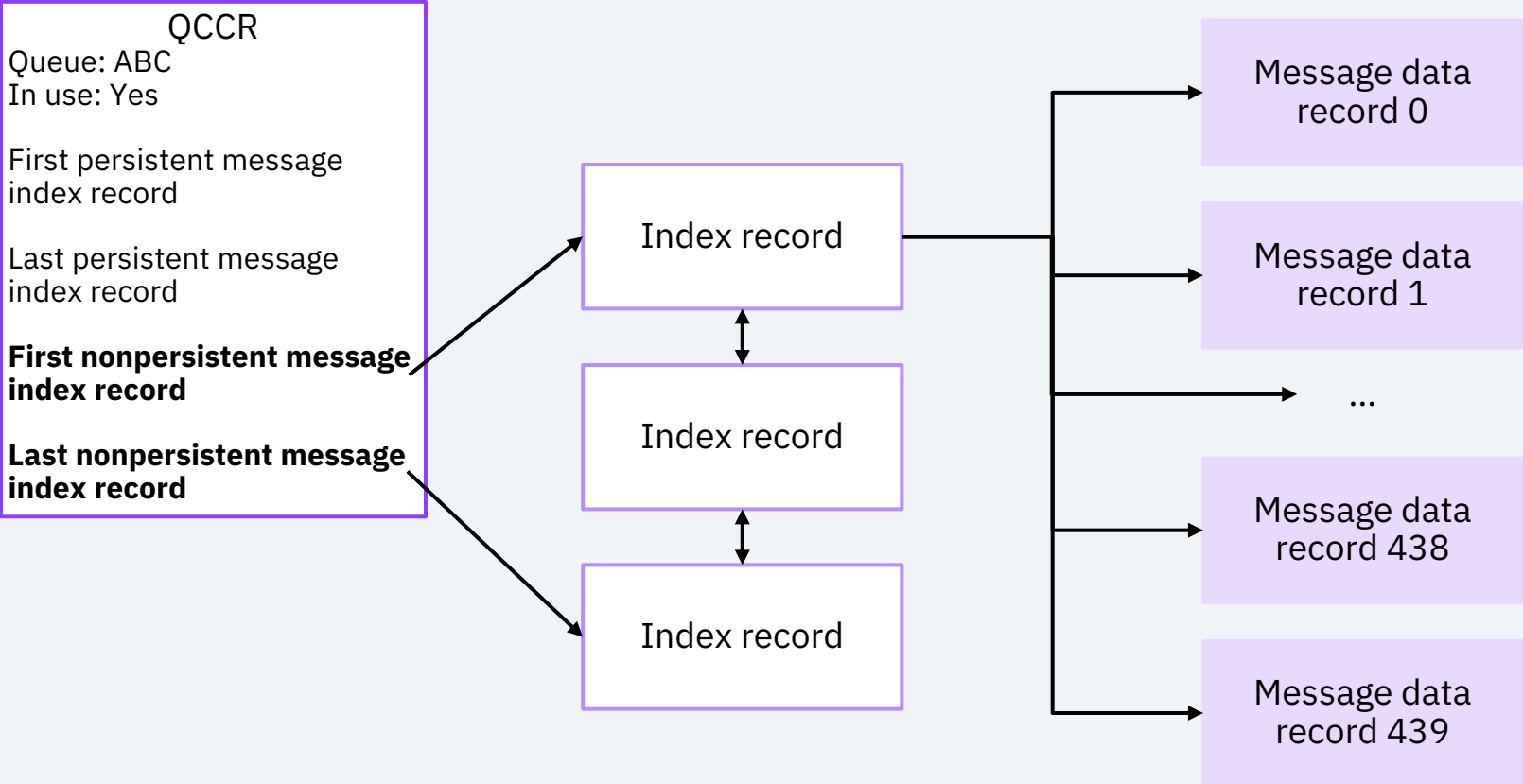
- Persistent messages selected to be swept for a 64-bit queue must have been checkpointed
- This means **persistent messages are swept without incurring any I/O**

# 64-Bit Sweeper

## Sweeping Nonpersistent Messages

- The sweeper for 64-bit queues **writes nonpersistent messages into pool records anchored off the QCCR**, just like the 64-bit checkpoint for persistent messages
- The **queue lock does not need to be held** while nonpersistent messages are filed, because they are detached from the message lists at selection time
- A swept **nonpersistent message is written a single time**
- This means a **previously filed nonpersistent message can be swept without incurring any additional I/O**

# QCCR Sweeper Example



# 64-Bit Sweeper

## Unsweeping Messages

- 64-bit queues are **proactively unswept** ahead of demand, meaning there is **little to no delay** for applications or transmission
- A path still exists to reactively unsweep, but is only needed if there is a surge of activity
- The **queue lock is not held during unsweep**
- On an unsweep, the unswept messages **do not need to be written to the recovery log**

# 64-Bit Sweeper

## Processing Intelligence

- The 64-bit sweeper **performs upfront analysis** of the 64-bit queues **across the entire system**
- **Identifies queues that have the greatest excess of messages** proportional to their dequeue rate (that is, sweep candidates)
- Also **identifies swept queues with an insufficient buffer of messages** present in the front of the queue (that is, proactive unsweep candidates)
- Following the analysis, the sweeper can **determine the optimal set of queues to sweep and unsweep, in which order, and with specific targets** for each queue

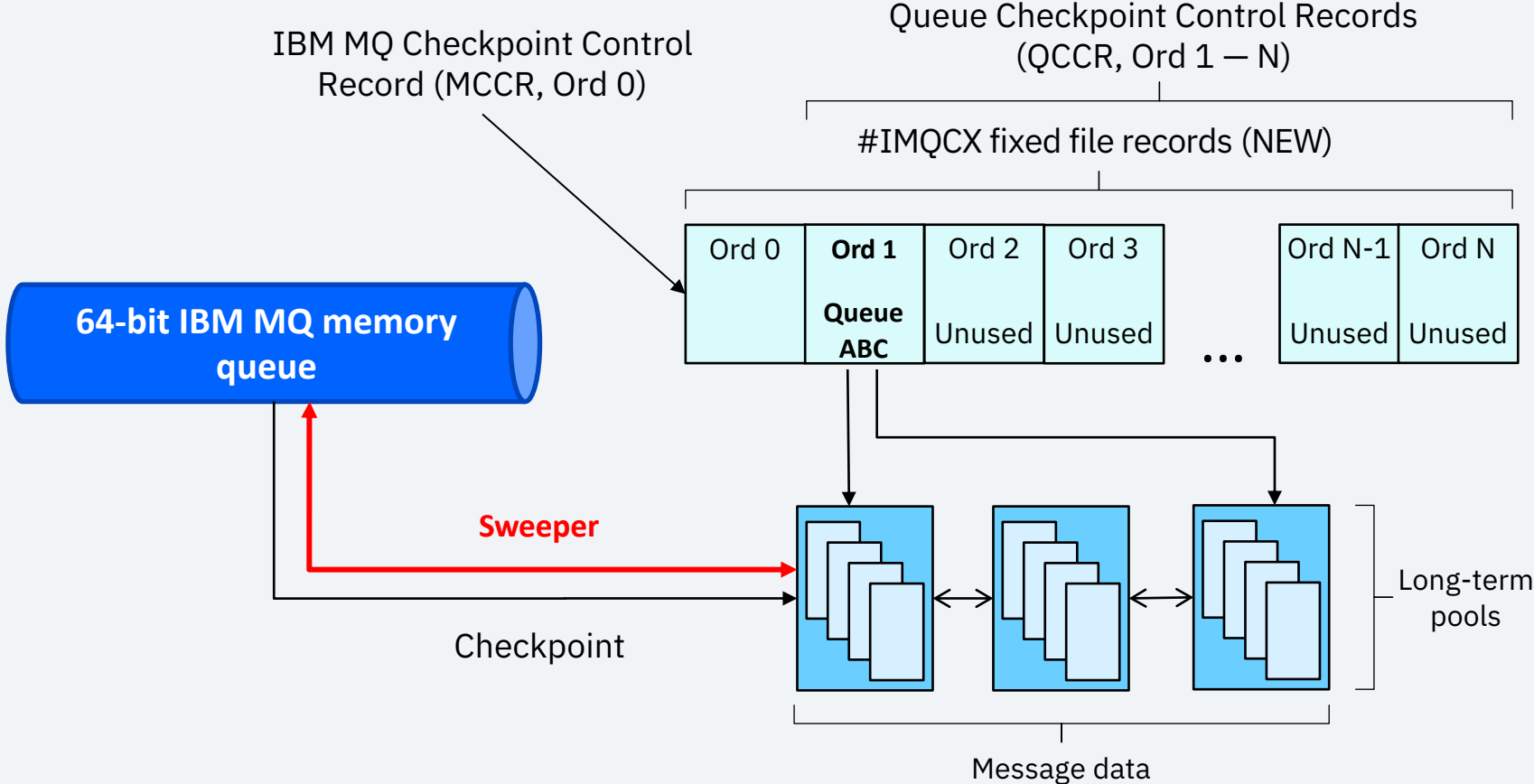
# 64-Bit Sweeper

## Processing Efficiency

- The sweeper then creates several children ECBs to **sweep and unsweep the 64-bit queues concurrently**
- As many as 10 ECBs all sweeping **or** as many as 20 ECBs all unsweeping
- 64-bit queues can be swept **bidirectionally (biased toward the rear of the queue)**
- Coordinating the sweep and unsweep closely **largely eliminates the possibility of message thrashing** for 64-bit queues
- Also enables the same queue to be swept and proactively unswept within the same sweeper run

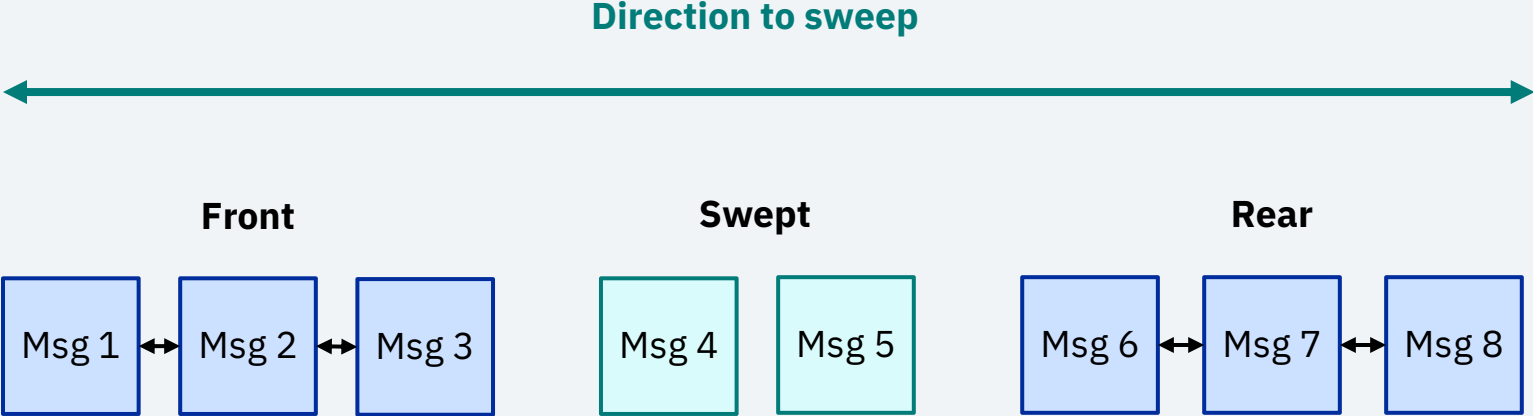


# 64-Bit Sweeper Visual Overview



# Bidirectional Sweeping (64-Bit)

For an already swept queue, the 64-bit sweeper can sweep in both directions, toward the front and rear, if necessary, within the same sweep. Preference is placed on sweeping toward the rear first because those messages will be accessed later than those near the front. Note that if the sweeper sweeps the front of the queue, it will always retain a number of messages in memory proportional to or higher than the dequeue rate.



# Summary of Sweeper Improvements

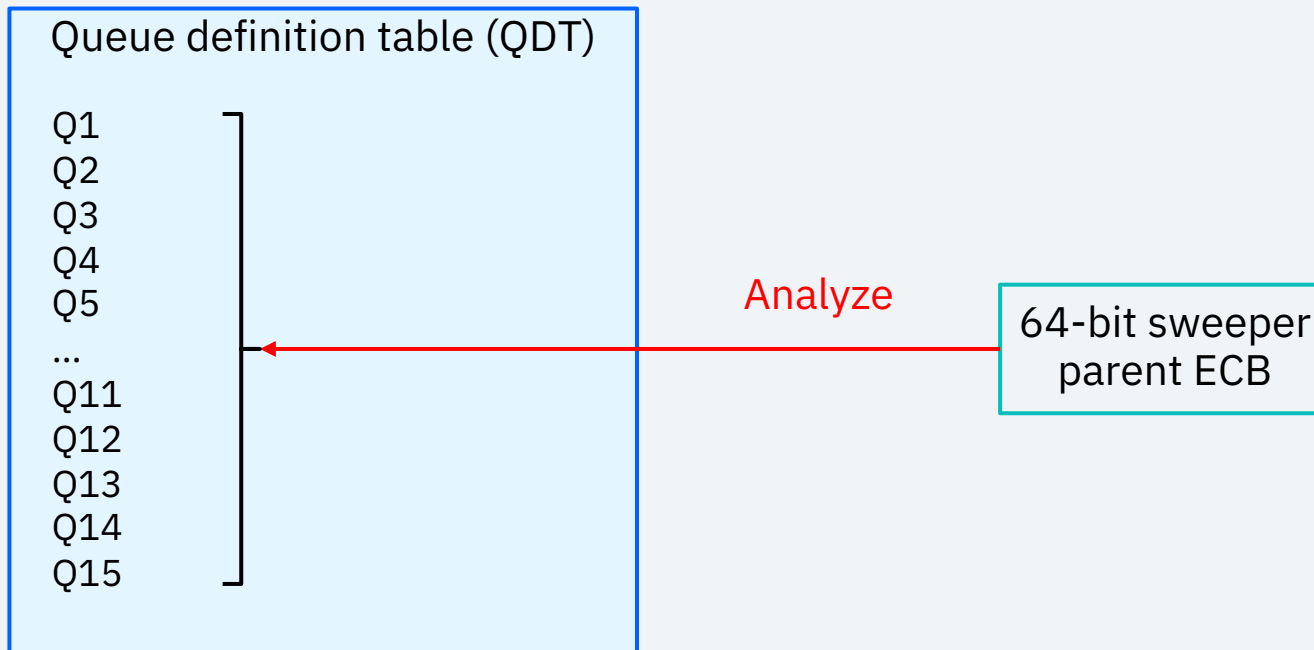
31-Bit Sweeper	64-Bit Sweeper
<ul style="list-style-type: none"><li>• Rewrites persistent messages to z/TPFCS</li></ul>	<ul style="list-style-type: none"><li>• Removes persistent messages from memory <b>without additional I/O</b></li></ul>
<ul style="list-style-type: none"><li>• Unswept messages must be written to the recovery log</li></ul>	<ul style="list-style-type: none"><li>• Unswept messages <b>don't need to be written to the recovery log</b></li></ul>
<ul style="list-style-type: none"><li>• Unsweep done reactively at MQGET or transmission time</li></ul>	<ul style="list-style-type: none"><li>• <b>Unsweps proactively ahead of anticipated demand</b></li></ul>
<ul style="list-style-type: none"><li>• Sweeps queues serially</li></ul>	<ul style="list-style-type: none"><li>• Sweeps and unsweps queues <b>concurrently</b></li></ul>

# Summary of Sweeper Improvements

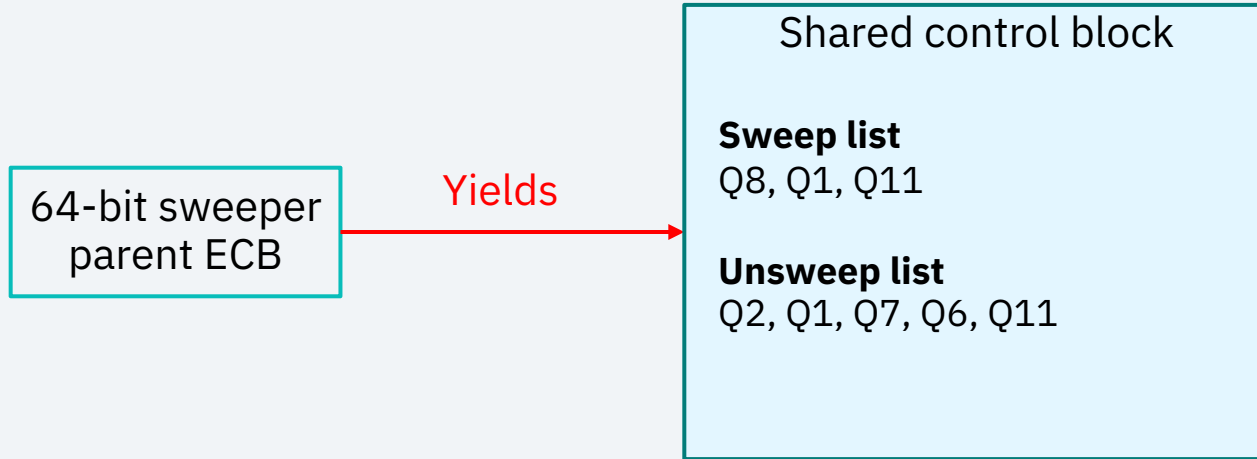
31-Bit Sweeper	64-Bit Sweeper
<ul style="list-style-type: none"><li>• Sweeps queues in the same order each time</li></ul>	<ul style="list-style-type: none"><li>• <b>Sweeps queues intelligently</b>, setting targets and priorities for each queue</li></ul>
<ul style="list-style-type: none"><li>• Queue lock held for entire duration of unsweep, preventing all other queue activity</li></ul>	<ul style="list-style-type: none"><li>• <b>Queue lock not held during unsweep</b></li></ul>
<ul style="list-style-type: none"><li>• Little coordination between sweep and unsweep, increasing chances of message thrashing</li></ul>	<ul style="list-style-type: none"><li>• <b>Coordinates sweep and unsweep for the same queue</b> to help prevent thrashing</li></ul>

# 64-Bit Sweeper Example

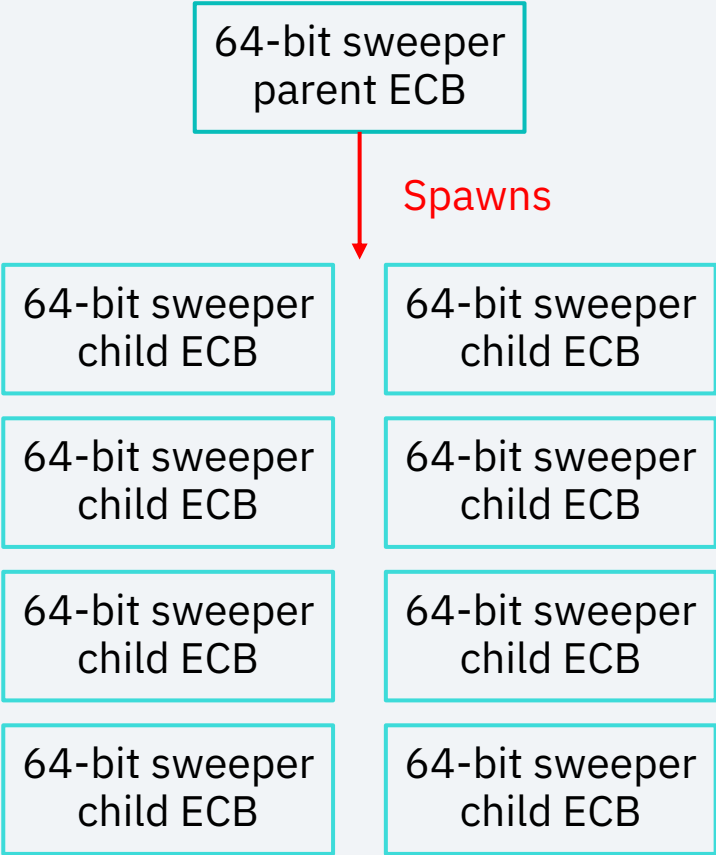
# 64-Bit Sweeper Example



# 64-Bit Sweeper Example



# 64-Bit Sweeper Example





# 64-Bit Sweeper Example

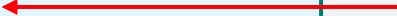
64-bit sweeper parent ECB

Shared control block

**Sweep list**  
Q8, Q1, Q11

**Unswep list**  
Q2, Q1, Q7, Q6, Q11

Grab first item



64-bit sweeper child ECB

64-bit sweeper child ECB

64-bit sweeper child ECB

64-bit sweeper child ECB

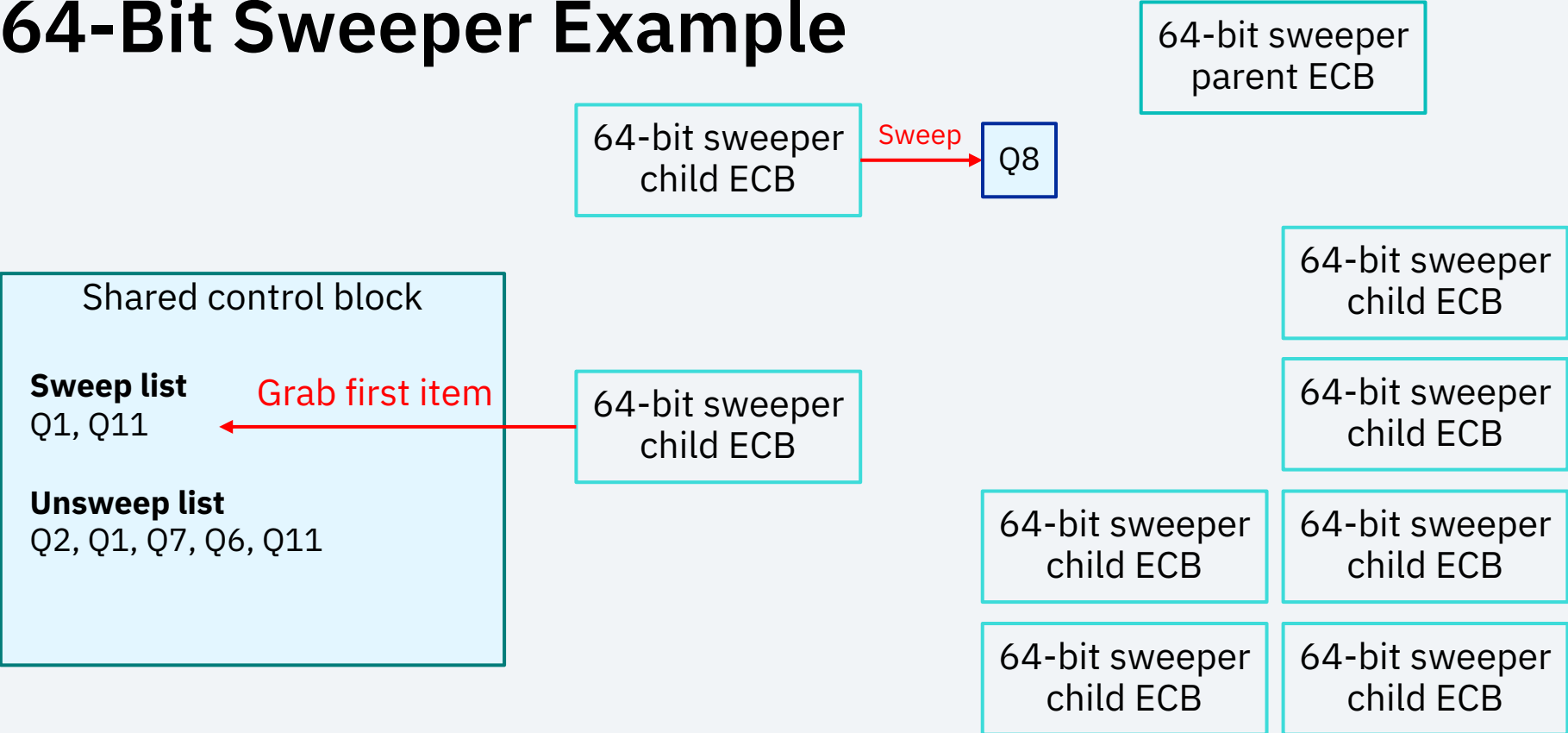
64-bit sweeper child ECB

64-bit sweeper child ECB

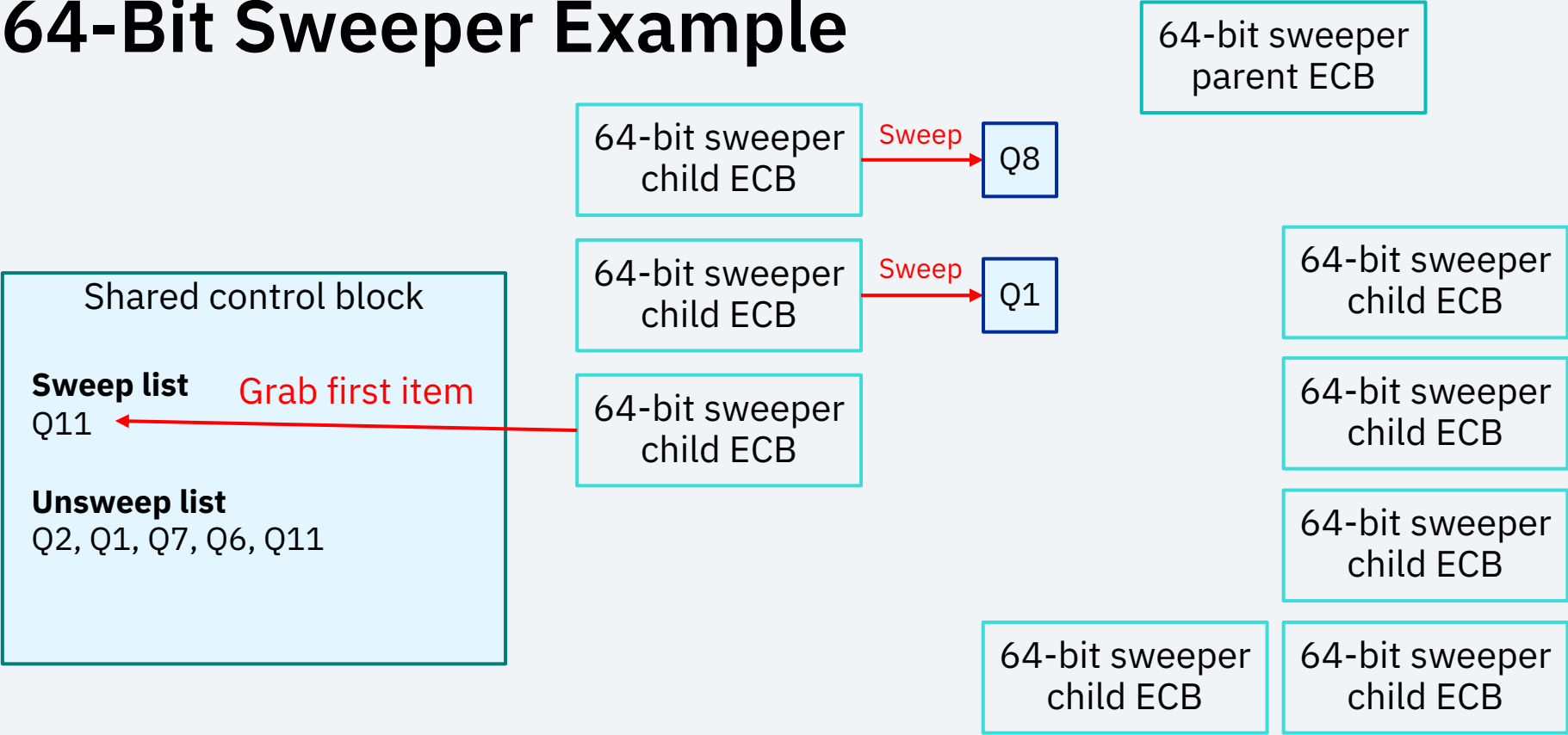
64-bit sweeper child ECB

64-bit sweeper child ECB

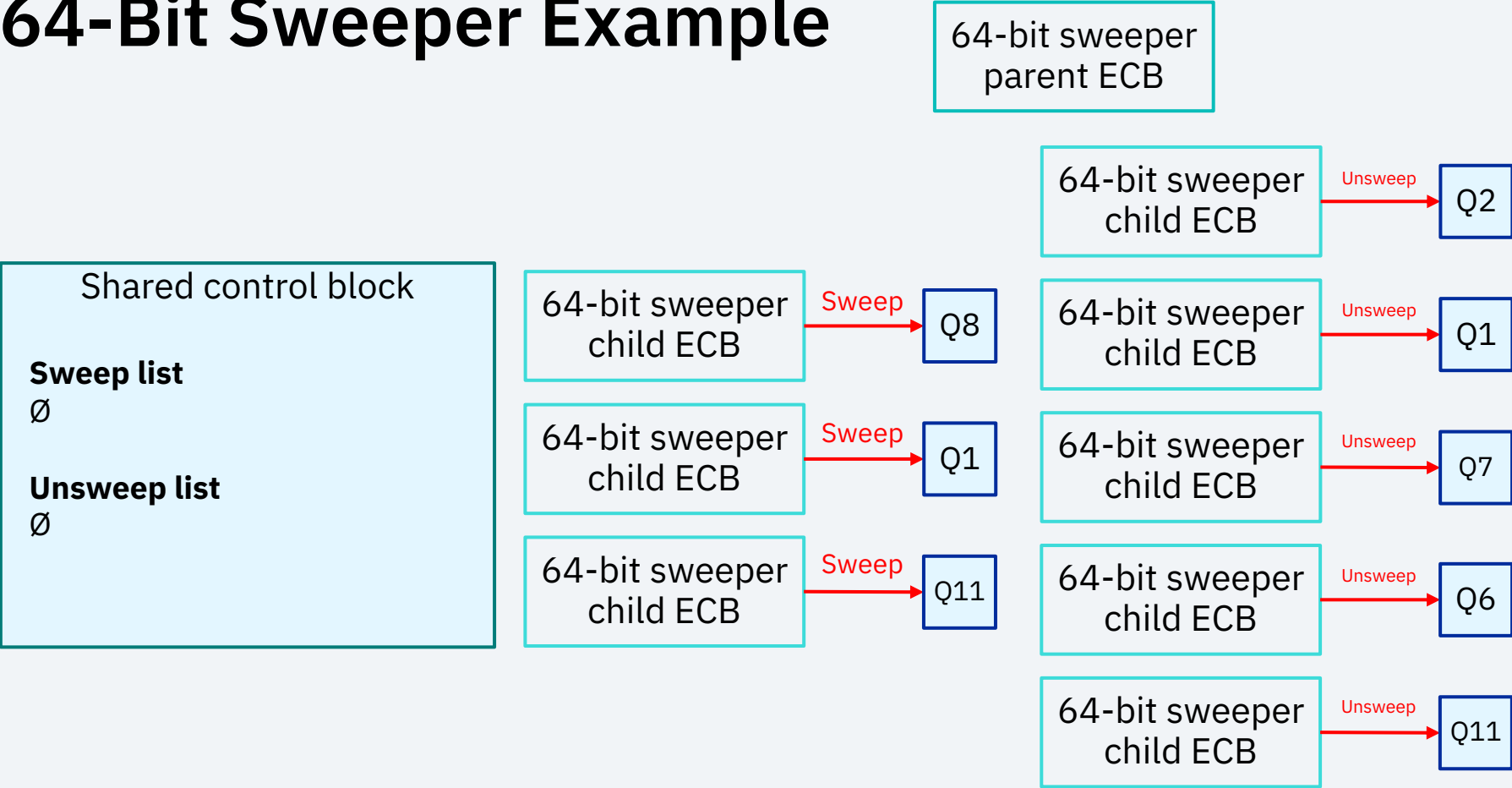
# 64-Bit Sweeper Example



# 64-Bit Sweeper Example

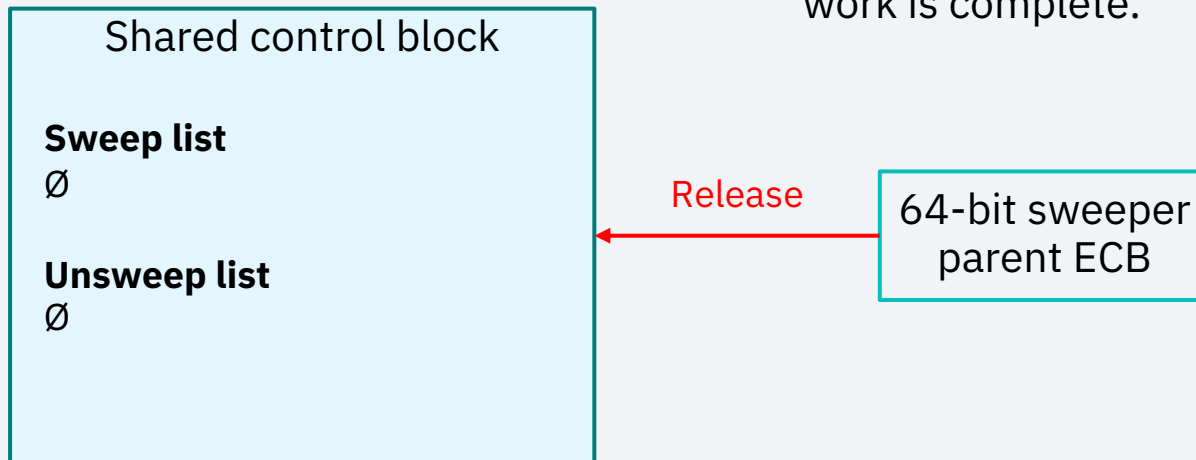


# 64-Bit Sweeper Example



# 64-Bit Sweeper Example

The parent ECB is woken when all sweep and unsweep work is complete.



# 64-Bit Sweeper Settings

# Parameters

- **MQMSWPL** and **MQMSWPT** have been added to CTKA
- **MQMSWPL** is the in-use percentage of MQM at which point 64-bit queues will be swept
  - The default value is 80%
- **MQMSWPT** is the target in-use percentage of MQM that the 64-bit sweeper will work to achieve
  - The default value is 60%.
- Modify with **ZCTKA ALTER**
- Display with **ZCTKA DISPLAY** or **ZMQSC DISPLAY QMGR**

# Queue Manager Display

ZMQSC DISPLAY QMGR

```
MQSC0283I 15.55.27 QUEUE MANAGER DISPLAY
          QMNAME - TPFQM
          ...
          MQMSWPL (%) - 80
          MQMSWPT (%) - 60
          END OF DISPLAY+
```



# Parameters

- With correctly tuned **MQMSWPL** and **MQMSWPT** parameters and a sufficient MQM allocation on the system for the workload, **sweeping 64-bit queues can be largely avoided altogether under normal circumstances**
- A remote system outage, for example, might still cause a transmission queue to grow to the point at which sweeping is necessary

# Ignored Parameters

- The **SWEEP** and **SWEEPSWB** parameters for the queue manager (**ZMQSC ALTER QMGR**) are ignored by the 64-bit sweeper
- The **SWEEP** and **SWEEPDEPTH** parameters for the local queue definition (**ZMQSC ALTER QL**) are ignored by the 64-bit sweeper
- Therefore, **the 64-bit sweeper is always enabled** for every 64-bit queue
- **These parameters remain in use by the 31-bit sweeper** the same as before

# Tuning MQMSWPL

- Generally, the default **MQMSWPL** of 80% should be sufficient
  - For example, with **MQMSWPL** at 80% on a system with 50 GB or 100 GB of MQM, queues will be swept at 40 GB and 80 GB of MQM in use, respectively
- If your MQM allocation falls near the extremes of either very small or very large, it should be set proportionally
  - For example, a system with only 250 MB of MQM should consider setting **MQMSWPL** lower, such as to 50%
  - For example, a system with 1 TB of MQM should consider setting **MQMSWPL** higher, such as to 90%

# Tuning MQMSWPT

- The **MQMSWPT** should be set according to how much the sweeper should try to sweep out in a single run
- The larger the difference between **MQMSWPL** and **MQMSWPT**, the greater the amount of message data targeted to be swept
- For example, with 50 GB of MQM, **MQMSWPL** at 80%, and **MQMSWPT** at 78%, the sweeper will target 1 GB of data to sweep

# System Resource Minimums

- The 64-bit sweeper, 31-bit sweeper, and checkpoint can all run concurrently
- Altogether, these three processes can use **as many as 11024 IOBs at a single time**
- With multiple release detection (MRD) enabled for record ID X'FE1B', the 64-bit checkpoint cleanup process can use **as many as 5000 system frames at a single time**
- While we recommend these minimums for production and stress test environments, unit test systems (for example, VPARS) and smaller native test systems are unlikely to approach the message volume needed to peak these resources' utilizations

# Monitoring MQM Usage

# Monitoring MQM Usage

- Display the total allocated, current in-use, and highwater in-use number of MQM blocks with **ZMQSC DISPLAY QMGR**
- Display the number of MQM blocks in use by a particular queue with **ZMQSC DISPLAY QL** and the **STAT** parameter specified
- Also shown are the timestamps of the last sweep, proactive unsweep, and reactive unsweep for the queue

# Queue Manager Display

ZMQSC DISPLAY QMGR

```
MQSC0283I 16.53.20 QUEUE MANAGER DISPLAY
          QMNAME - TPFQM
          ...
          Total MQM - 127997
          In Use MQM - 18300
          HW In Use Count MQM - 61000
          HW In Use Time MQM - 2024-03-27 16:53:04
          ...
          END OF DISPLAY+
```



# Queue Display

ZMQSC DISPLAY QL with STAT specified

```
MQSC0285I 15.34.48 LOCAL QUEUE STATISTICS DISPLAY: -
rcvry_driver_64bit_1
    Current Depth          - 10
...
    Num of SWBs in use    - NONE
Num of MQMs in use    - 10 _
...
Time Last 64Bit Swp          - 2023/11/12 15.34
Time Last 64Bit Proactive UnSwp - 2023/11/12 15.34
Time Last 64Bit Reactive UnSwp - 2023/11/12 10.32
...
END OF DISPLAY+
```

# Identifying an Undersized MQM Allocation

- An undersized MQM for the system's workload will result in the 64-bit sweeper *frequently* running
- To quickly identify if the 64-bit sweeper has likely run at all, view the **ZMQSC DISPLAY QMGR** display's highwater in-use mark and total allocated for MQM
- If the highwater mark as a percentage of the total allocated is greater than or equal to the **MQMSWPL**, the 64-bit sweeper has run at least once
- When it has been determined that the 64-bit sweeper has run at least once, examine the console for the presence of **MQSC0801I** messages, which indicate sweep activity for the named queue
- If **lots** of these messages exist on the console, the 31-bit sweeper, 64-bit sweeper, or both must be frequently active

# Identifying an Undersized MQM Allocation

- The next step after seeing many **MQSC0801I** messages on the console is to determine whether they pertain to 64-bit queues
- Take the most prevalent queues named in the messages and do **ZMQSC DISPLAY QL** to confirm if they are using MQM

# Identifying an Undersized MQM Allocation Example

## ZMQSC DISPLAY QMGR

This queue manager display reveals the highwater mark has likely triggered the 64-bit sweeper to sweep queues at some point.

```
MQSC0283I 14.50.21 QUEUE MANAGER DISPLAY
QMNAME - TPFQM
```

...

**Total MQM - 127997**

In Use MQM - 83326

**HW In Use Count MQM - 102724**

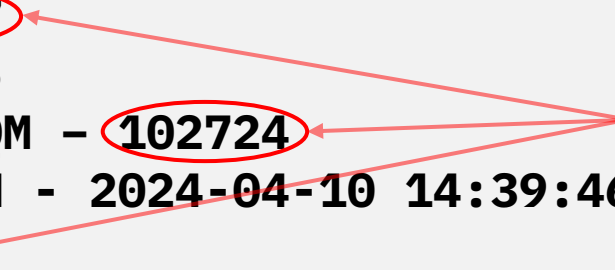
**HW In Use Time MQM - 2024-04-10 14:39:46**

**MQMSWPL (%) - 80**

MQMSWPT (%) - 60

END OF DISPLAY+

80% of the total  
MQM was in use



# Identifying an Undersized MQM Allocation Example

Looking at the console, the presence of **MQSC0801I** messages indicates sweep activity for the named queue MYQ.

```
MQSC0801I 14.40.27 SWEEP FOR MYQ COMPLETED+  
...  
MQSC0801I 14.44.02 SWEEP FOR MYQ COMPLETED+  
...  
MQSC0801I 14.45.08 SWEEP FOR MYQ COMPLETED+  
...  
MQSC0801I 14.46.15 SWEEP FOR MYQ COMPLETED+
```

# Identifying an Undersized MQM Allocation Example

## ZMQSC DISPLAY QL

This local queue display of queue MYQ confirms that it is using MQM.

```
MQSC0282I 15.04.57 LOCAL QUEUE DISPLAY:
```

```
Queue Name - MYQ
```

```
...
```

```
64BIT - YES
```

```
MQM - YES
```

```
64CHKPTORD - 1
```

```
END OF DISPLAY+
```

# Identifying an Undersized MQM Allocation

- If the conclusion from your investigation is that 64-bit queues are being **swept hundreds or thousands of times a day on a recurring basis**, the MQM allocation for the system is too small for the workload
- *Note that the example just covered was for the purpose of exemplifying how to approach this investigation, and does not necessarily depict a system with too little MQM – it would need much more frequent, regular activity to lead to such a conclusion*

# Identifying an Oversized MQM Allocation

- An oversized MQM for the system's workload will be apparent from a highwater in-use mark that is only a small fraction of the total allocated
- Note, however, that **the highwater mark is not persisted across an IPL** and therefore should only be referenced for this purpose of sizing the MQM after a long period of uptime



# Identifying an Oversized MQM Allocation Example

## ZMQSC DISPLAY QMGR

This queue manager display shows that the highwater mark of in-use MQM has only reached about 10% of the total allocated.

```
MQSC0283I 18.20.15 QUEUE MANAGER DISPLAY
QMNAME - TPFQM
```

...

**Total MQM - 127997**

In Use MQM - 5000

**HW In Use Count MQM - 12250**

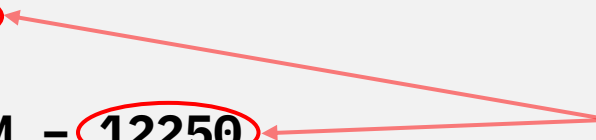
**HW In Use Time MQM - 2024-04-08 14:35:30**

MQMSWPL (%) - 80

MQMSWPT (%) - 60

END OF DISPLAY+

10% of the total  
MQM was in use



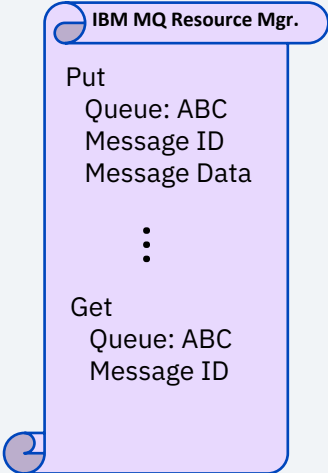
# Restart & Recovery

# Restart & Recovery Overview

- Restart & recovery is the process in IBM MQ responsible for ensuring that no persistent messages are lost across an outage (IPL)
- This involves processing the recovery log, merging events from the log with the checkpoint, and then rebuilding persistent messages onto their queues
- The number of persistent messages rebuilt for a given queue in restart is dependent upon whether the queue is 31-bit or 64-bit (discussed soon)

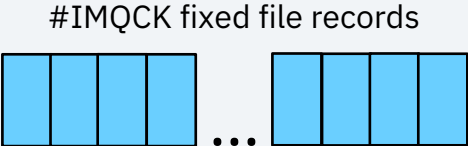
# Rebuilding Queues After An Outage

## Recovery log

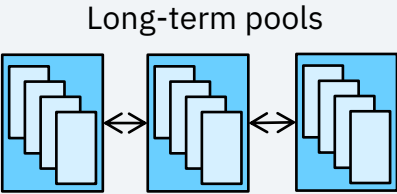


+

## Checkpoint

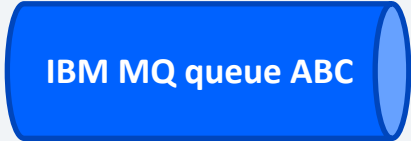


OR



=

## Rebuilt queue



Note that 31-bit queues are rebuilt entirely in restart, while 64-bit queues might have some message restoration deferred to when the system comes up (especially if the queue is large)

# 31-Bit Restart & Recovery

- The 31-bit restart & recovery rebuilds queues **serially, entirely,** and **synchronous** to the overall system restart process
- With large 31-bit queues with many persistent messages, this can represent a significant bottleneck on the ability to quickly bring a system back up
- If the system was in a low SWB condition prior to an outage, there is a real possibility that the system comes back up in that same state or even in input list shutdown, because 31-bit queues are rebuilt in their entirety according to what was in memory prior

# 64-Bit Restart & Recovery

- The 64-bit restart & recovery rebuilds queues **concurrently, dynamically,** and **asynchronous** to the overall system restart process past a fixed interval
- The number of restoration workers is a function of the available system resources (that is, i-streams, ECBs, IOBs)
- If **system recovery boost** is enabled, 64-bit restart & recovery can **take advantage of fenced i-streams** to supplement restoration
- The target number of persistent messages to restore for a single queue is **proportional to the queue's dequeue rate** before the outage
- Queues with large or many messages will continue to be rebuilt as system restart continues, **reducing overall time in bringing the system back up**
- Restoring only what will be immediately needed according to anticipated demand **greatly reduces the likelihood that the system will be burdened with sweeping** upon coming back up to NORM

# 64-Bit Restart & Recovery

- Any persistent messages which haven't been rebuilt as part of restart & recovery will be rebuilt (unswept) as needed by the 64-bit sweeper

# Summary of Restart & Recovery Improvements

31-Bit Restart & Recovery	64-Bit Restart & Recovery
<ul style="list-style-type: none"><li>• Queues are rebuilt serially</li></ul>	<ul style="list-style-type: none"><li>• Queues are rebuilt <b>concurrently</b></li></ul>
<ul style="list-style-type: none"><li>• Every message in checkpoint is rebuilt into memory</li></ul>	<ul style="list-style-type: none"><li>• A <b>dynamic number of messages are rebuilt</b> according to the queue's dequeue rate</li></ul>
<ul style="list-style-type: none"><li>• All rebuild activity occurs synchronous to the overall system restart</li></ul>	<ul style="list-style-type: none"><li>• <b>Rebuild activity occurs asynchronously</b> after a period of time elapses</li></ul>



# Restart Reserved Pool Set & VPARS

- Special ***restart reserved pool set*** of index records stored in the MCCR for use by 64-bit IBM MQ restart & recovery only
  - Critical because new pools are unobtainable in system restart, and we need them to write messages from the recovery log to the checkpoint
  - This reserve is equivalent in size to 90% of the system's recovery log allocation
- In a VPARS test environment, we **highly** recommend allocating the restart reserved pool set on the base packs before turning over
  - This avoids consuming individuals' databases and time to allocate
- The restart reserve is allocated on queue manager start
  - Therefore, must bring up the base VPARS system to NORM state and make sure the queue manager is started before turning the system over for use by developers

# Conclusion

# Summary of Settings for 64-Bit Queues

- Define **#IMQCX fixed file records in FACE table**
  - Number should be *at least* equal to the number of 64-bit queues you plan to define on your system, but defining a surplus will allow for future growth
  - In a loosely coupled complex, be sure to define the same number of these records on every processor
- Define the **X'FE1B' record ID to the RIAT table** to specify the pool settings
- **MQM** allocation for the system defined in CTKA
- **MQMSWPL** and **MQMSWPT** defined in CTKA control the 64-bit sweeper's sweep level and target

# Scaling with Your Growing Business

- As message data volumes increase, you can grow the amount of **MQM** on your system to support more data in memory at once
- As part of such an increase, also consider adjusting the **MQMSWPL** and **MQMSWPT** if the new allocation is very large
- Should you need to define more 64-bit queues, add additional **#IMQCX fixed file records** across your complex
- Consider also increasing **MQM** on the system if these queues will potentially represent a large increase in message volumes

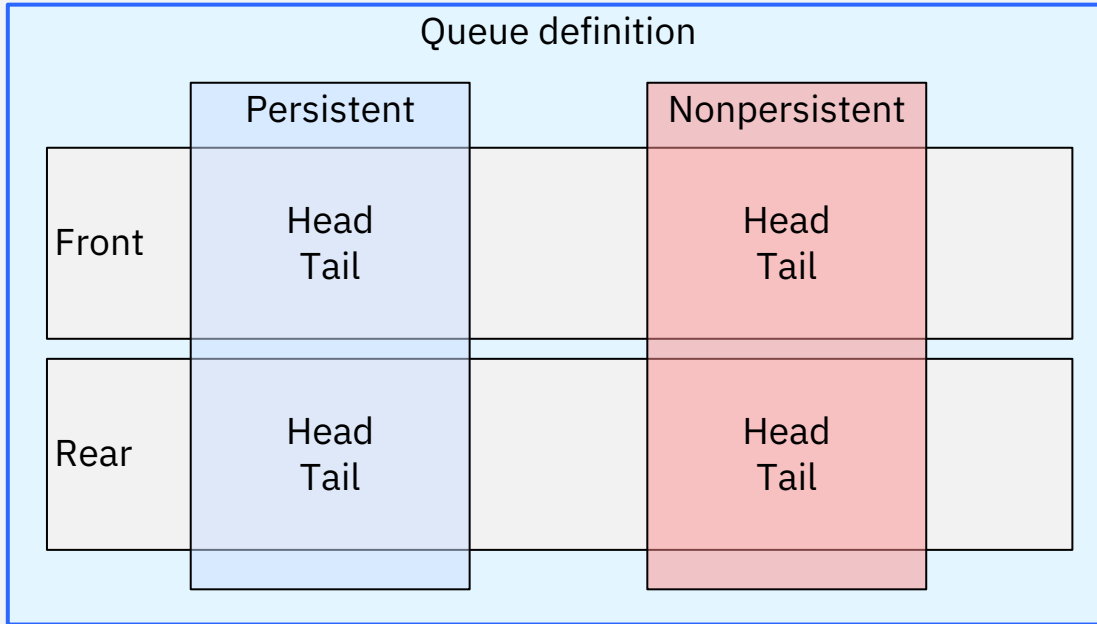
# Questions?

# Thank you

© Copyright IBM Corporation 2024. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. Any statement of direction represents IBM's current intent, is subject to change or withdrawal, and represent only goals and objectives. IBM, the IBM logo, and ibm.com are trademarks of IBM Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available at [Copyright and trademark information](#).



# Queue Message Lists



Unique to 64-bit queues, nonpersistent messages are separated into their own lists apart from persistent messages. There are also lists to separate the front of the queue from the rear, which are used specifically in the case of sweeping.

Altogether, for 64-bit queues, there are four message lists, each constituted by a head and tail message pointer.

Separating out these lists based on message type helps with efficiently checkpointing and sweeping 64-bit queues.

**Most importantly, use of these lists is completely transparent to applications.**