Continuous Checkpointing System for Enhanced Model Recovery

by

Dhayaneshwar Balusamy

August 19, 2024

A thesis submitted to the

Faculty of the Graduate School of

the University at Buffalo, The State University of New York

in partial fulfilment of the requirements for the

degree of

Master of Science

Department of Computer Science and Engineering

Acknowledgements

I am immensely grateful for the support and guidance I received during the development of this thesis. Foremost, I extend my deepest gratitude to Dr. Jinjun Xiong for his invaluable supervision throughout my research journey. His expertise and profound insights have been pivotal in shaping this research work.

I also wish to express my sincere appreciation to Amir Nassereldine, whose expert guidance and assistance have been instrumental in my development and the successful completion of this work. Additionally, I am grateful to Jerry Chen, whose technical insights greatly contributed to the research. I am also thankful to Dr. Kaiyi Ji, a member of the thesis defense committee, for his constructive feedback and suggestions.

I owe a debt of gratitude to the Department of Computer Science and Engineering at the University at Buffalo for providing the resources and supportive environment that facilitated my research.

Most importantly, I extend my heartfelt thanks to my family, whose love, support, and encouragement have been my ongoing source of strength and motivation. I dedicate this accomplishment to them.

Table of Contents

List of Tables

List of Figures

Abstract

The training of the Deep Learning Recommendation Model (DLRM) is a computationally demanding task that often takes place across distributed computing environments and utilizes large datasets. Traditional checkpointing methods which periodically store the model's state on disk, are not only inefficient but can also cause potential disruptions to training. This research work focuses on increasing checkpointing frequency while also reducing training interruptions by introducing a novel continuous checkpointing method, especially for DLRMs. The proposed method uses leverages efficient asynchronous operations and Kafka for efficient data streaming of updated embeddings. With this method, real-time model data can be stored without interfering the ongoing training sessions.

The implementation was also extended to support large-scale distributed training and benchmarked on various training infrastructures. Extensive experiments were conducted using both synthetic data and the Criteo Kaggle Display Advertising Challenge Dataset to evaluate the performance of the continuous checkpointing system against traditional method. The data collected on the model checkpointing time, recovery time, and overall runtime demonstrated that continuous checkpointing significantly enhances training performance, and reduces data-loss compared to traditional methods.

This research concludes by successfully developing a complete checkpointing system that includes highly efficient checkpointing and recovery operations. It also establishes a robust framework that ensures data integrity and enables rapid recovery post-failure. By integrating a continuous checkpointing mechanism with DLRM, this work contributes substantially to the field of machine learning, offering a scalable, and efficient method for training large-scale models in distributed environments.

Chapter 1

Introduction

Recommendation systems are a subset of machine learning technology that leverages large-scale data to assist in the prediction and identification of preferences within an ever-expanding array of choices. Recommendation systems utilize machine learning algorithms such as collaborative filtering, clustering, and deep neural networks to analyze datasets containing user behavior, preferences, and interactions. These algorithms enable to predict and provide recommendations that closely align with the user's history such as previous purchases, viewing habits, and search patterns. Recommendation systems are essential in e-commerce and digital media as they allow businesses to enhance customer satisfaction and increase revenues by precisely predicting and catering to customer requirements. Additionally, they help users to navigate through overwhelming choices by providing tailored suggestions, and thereby improving user engagement and retaining customer interest. Major companies like Amazon, Facebook, and YouTube utilize recommendation systems extensively to personalize user experiences. Amazon recommends products based on browsing and purchasing history [1], Facebook customizes the feed to show relevant content [2], and YouTube suggests videos that align with past viewing behaviors [3], driving engagement and content discovery on these platforms.

Recommendation systems have evolved significantly with the advent of deep learning technologies, leading to the development of Deep Learning Recommendation Models (DLRM) [4]. DLRM represents a combination of recommendation systems with deep learning tech-

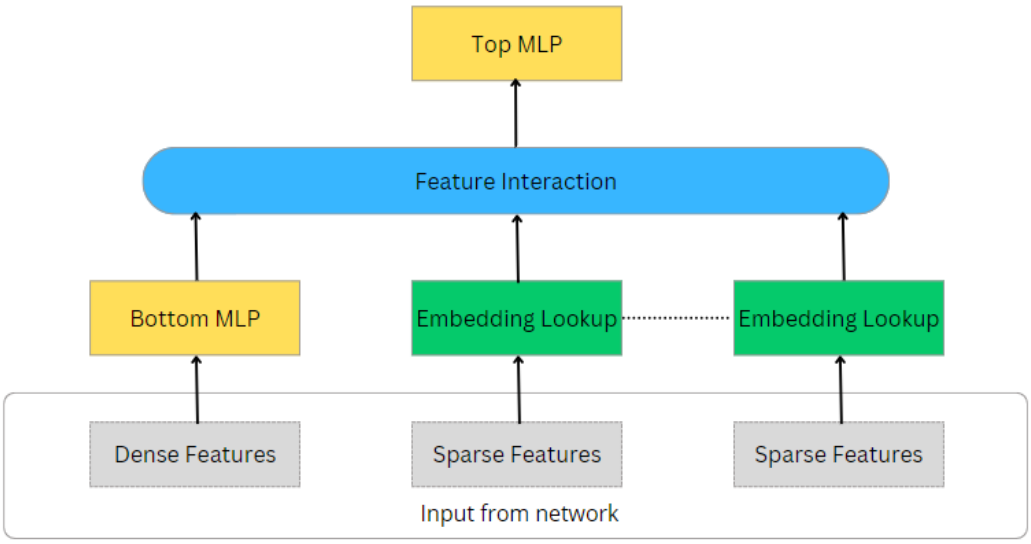

Figure 1.1: DLRM Architecture

niques and designed to use massive amounts of data to predict user preferences with highest accuracy. As shown in Figure 1.1, the DLRM architecture combines categorical data, processed through embedding layers, with numerical features in dense layers in the model to output the probability of click. In DLRM, the categorical data is processed using embeddings, while the continuous data is handled by a bottom MLP (multilayer perceptron). The model then explicitly calculates second-order interactions between these features. These results are subsequently processed by a top MLP, and a sigmoid function is used to estimate the probability of a click.

Training Deep Learning Recommendation Models presents a unique set of challenges due to their complexity and the scale at which they operate. These challenges not only include the computational demands, but also ensuring the training process is reliable and efficient. Common failures during training process include hardware malfunctions such as GPU failures due to overheating, and software issues like memory leaks or data pipeline bottlenecks, which can interrupt the training cycle [5]. The process of traditional checkpointing further exemplifies these difficulties due to the computational intensity and the massive scale of data involved. The traditional checkpointing techniques which periodically save the entire model

state to disk are not only slow and resource-intensive but also disruptive to the training process. These techniques frequently cause interruptions which may lengthen the training period and can potentially affect the model's training process.

Furthermore, in the distributed training of DLRM the need to synchronize state across multiple nodes further exacerbates the issues. The synchronization process is prone to inducing latency, and can complicate the recovery process in the event of any failure. In these scenarios, hardware failures and network outages are also typical problems that can result in partial or total data loss if the checkpointing system fails to precisely and regularly capture the most recent state of every node. The importance of reliable training processes necessitates a more robust, efficient, and less intrusive checkpointing solution to maintain data integrity, reduce training interruptions, and enable swift recovery after any failures.

This thesis introduces a novel continuous checkpointing system specifically designed to address the training challenges of Deep Learning Recommendation Models (DLRM). Traditional checkpointing methods, which periodically save the state of the model to disk have proven to be inefficient and interruptive, particularly in distributed environments handling large-scale data. The proposed system leverages Kafka streaming and asynchronous operations to provide efficient and reliable checkpointing, ensuring that model states specifically embeddings are saved continuously without interrupting the training process. This novel approach significantly enhances the efficiency and reliability of DLRM training.

The main objective of this research is to develop a checkpointing and recovery system that minimizes training disruptions, optimizes resource usage, accelerates recovery operations, and enhances checkpointing frequency to significantly reduce training data loss. By incorporating a continuous checkpointing system, the system ensures high data integrity and resilience against training failures, and thereby supporting the demands of large-scale distributed recommendation model training. The effectiveness of the system was tested using synthetic data and real-world datasets. These tests evaluate the key performance metrics and also the operation of the system in a range of distributed training scenarios.

Chapter 2

Related Work

The foundational paper, Deep Learning Recommendation Model for Personalization and Recommendation Systems [4] by Naumov et al. (2019) introduces the Deep Learning Recommendation Model (DLRM), a state-of-the-art neural network framework designed specifically for recommendation tasks, distinguishing itself by its ability to efficiently process categorical and continuous features. While DLRM provides comprehensive insights recommendation systems, it does not incorporate an efficient checkpointing system. This highlights an opportunity for the development of specialized checkpointing system which addresses the specific needs of large-scale recommendation system training while also enhancing resilience and operational efficiency.

Developing an efficient checkpointing system, especially in distributed training environments requires sophisticated techniques to optimize data handling, enhance fault tolerance, and improve system recovery. The study conducted by Rojas et al. (2021) [6] provides valuable insights into checkpoint-restart mechanisms and emphasizes the need for more sophisticated checkpointing solutions tailored to the complex demands of training large-scale deep neural networks. Continuous checkpointing system significantly enhances fault tolerance without compromising on training efficiency, thereby filling a critical gap identified by Rojas et al. in existing DL frameworks. The study by Tonmoy Dey et al. [7] highlights the use of multi-level checkpointing strategies to reduce I/O traffic and boost efficiency in HPC systems by incorporating AI techniques.

The work by Bogdan Nicolae et al. [8] introduces an efficient asynchronous checkpointing approach in deep learning that uses fine-grained sharding and augmented execution graphs to minimize serialization and I/O overheads on HPC platforms, showing improvements in performance with models like ResNet. However, it does not address all the challenges of checkpointing in real-world operational environments, such as continuous data persistence and real-time model state recovery. Similarly, Trishul Chilimbi et al. [9] discusses the Adam project, which enhances the efficiency and scalability of distributed deep learning systems through optimized balance of computation and communication and asynchronous updates, resulting in improved model accuracy due to the system's ability to handle larger models. Additionally, Aurick Qiao et al. [10] presents a framework that leverages inherent self-correcting properties of machine learning algorithms to provide an fault tolerance strategy, reducing recovery costs and improving reliability in various ML models training. While effective for minor disturbances, this might not support for all error types or high data volumes in real-time. The study by Sze et al. [11] provides a exploration into computational complexities and hardware demands of deep neural networks, emphasizing the need for specialized hardware and algorithmic strategies . This study underscores the importance of efficient processing techniques in systems like continuous checkpointing system, where the rapid processing and storage of model states are vital for maintaining performance without sacrificing accuracy.

ByteCheckpoint [12] presents a PyTorch-native checkpointing system tailored for Large Language Models, addressing prior limitations by incorporating checkpointing resharding for parallel LLM training. It also introduces asynchronous tensor merging, I/O optimizations, and disaggregated data/metadata storage architecture to enhance management across various training frameworks and parallelism strategies. Check-N-Run [13] introduces a checkpointing system optimized for training massive recommendation systems within Facebook's infrastructure. By only checkpointing the changed parts of the model at certain frequency and reducing the data size through quantization, Check-N-Run enhances checkpoint effi-

ciency. ByteCheckpoint and Check-N-Run, while advanced than previous systems, does not address continuous and granular checkpointing to reduce data loss and rapid state recovery during training interruptions, an area where continuous checkpointing excels.

For checkpointing in distributed settings, the work by R. Koo and S. Toueg. [14] details a distributed algorithm for creating consistent checkpoints and a rollback-recovery mechanism. L. Wang et al. [15] explores a coordinated checkpointing protocol for next-generation supercomputers, assessing the scalability and its ability to handle failures during checkpointing, providing insights into fault tolerance at a massive scale. Fault Tolerance in Distributed Systems: A Survey [16] provides a exploration on fault tolerance in distributed systems which is a critical aspect in the continuous checkpointing system.

The research presented by Hestness et al., [17] examines the optimization potential of heterogeneous CPU-GPU processors, emphasizing the role of unified memory architectures in reducing data movement overheads and enhancing compute and cache efficiency. This analysis holds significance for the continuous checkpointing system by highlighting the need for efficient data management, which are essential for reducing checkpointing latency. The effectiveness of using CUDA streams for overlapping data transfers with computations has been demonstrated in various studies, showing substantial performance improvements in both data transfer-intensive and compute-intensive kernels [18, 19, 20]. These studies provide a foundation for integrating similar strategies in the continuous checkpointing, aiming to minimize the latency impacts and enhance the performance of data transfers in checkpointing processes.

The studies by Guozhang Wang et al. [21] and Shubham Vyas et al. [22] emphasize Apache Kafka's robust architecture in stream processing which ensures correctness and high throughput in distributed systems. Regarding the optimization of Apache Kafka configurations specifically the partitioning of topics, studies by Theofanis P. Raptis et al. [23, 24] have highlighted the complexity of achieving optimal partition distribution in high-volume data environments. These works underscores the importance of finely tuned partition strategies to

accommodate the demanding performance and resource utilization requirements of real-time data streaming applications, especially continuous checkpointing system. The study by Han Wu, Zhihao Shang, and Katinka Wolter. [25] introduces a queueing-based model to optimize Kafka configurations for predicting performance impacts from broker and partition settings.

The study presented by Feng He et al. [26] explores advanced techniques in inter-process communication and the optimization of data transfer for computational tasks in multi-core architectures. Dominik Straßel, Philipp Reusch, and Janis Keuper. [27] addresses critical issues related to managing Python workflows on High-Performance Computing systems, particularly in the GPU-based machine learning applications. The continuous checkpointing system leverages and complements these studies by implementing efficient multi-process operations in checkpointing and recovery systems, integrating seamlessly with HPC systems to improve the robustness and efficiency of long-running training processes.

Jackson et al. [28] focuses on enhancing data streaming efficiency across various real-time applications by empirically testing a range of streaming technologies and serialization protocols. This study extensively tests serialization libraries, including MessagePack, ProtoBuf, and Pickle. Their work parallels the developments in the continuous checkpointing system, which utilizes a custom serialization protocol that is optimized for checkpointing large and sparse tensor data more efficiently than standard serialization libraries.

Chapter 3

Checkpointing Strategy

3.1 Motivation

Checkpointing is essential for enhancing the reliability of training large-scale machine learning models, particularly in complex distributed environments. The necessity for implementing a reliable checkpointing system is driven by several factors as described below.

- **Preventing Data Loss**: Training machine learning models in industrial settings can be interrupted by hardware failures, power outages, or other disruptions. This is addressed by checkpointing which saves the model's state data at regular intervals so that training can resume from the most recent stored state rather than from beginning. This preserves both time and computational resources which are vital in real-world industrial settings.

- **Minimizing Costs and Time from Training Disruptions**: Restarting the model training from the beginning is expensive and time-consuming, especially when done in large-scale. By enabling the resuming of training process from a previously saved state, checkpointing minimizes the resources and time lost due to interruptions. This is especially crucial in commercial and research domains where efficient resource management is essential for sustaining project viability.

- **Consistency and Fault Tolerance in Distributed Systems**: In distributed settings where multiple nodes are used in training machine learning models, the nodes may not always be in synchronization due to various factors. Checkpointing provides a common recovery point for all the nodes to maintain training consistency and enhance fault tolerance.

3.2 Technical Implementation and Challenges

Traditional checkpointing involves pausing the training process at specified intervals, writing the current model state to persistent storage, and then again resuming training. Traditionally, implementing checkpointing involves using libraries that provide support for these operations. Libraries such as PyTorch offer functions like `torch.save` to write the model or tensor to a file and `torch.load` to retrieve them.

The checkpointing process involves complex technical considerations that must be carefully managed to ensure system efficiency and data integrity. Determining the optimal checkpoint frequency is necessary because more frequent checkpoint operations may impede system performance due to increased overhead of data writing and storage, while infrequent checkpoints may result in data loss in the event of a failure since recent state is not captured. As models size increases, the demand for storage can also cause bottlenecks in write operations and storage capacity.

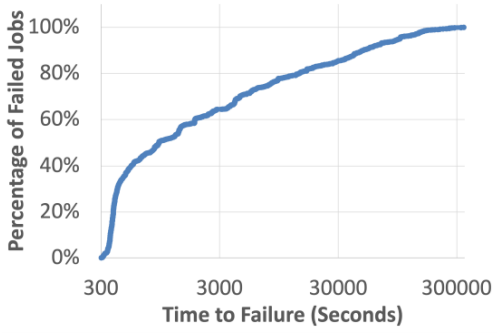

Figure 3.1: Training Job Failure CDF in Meta Cluster

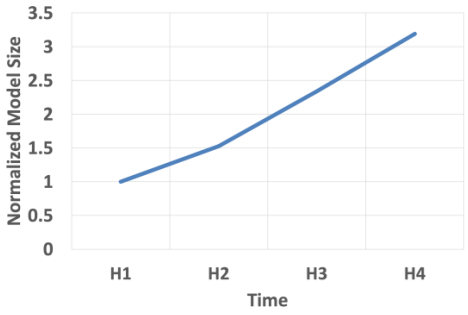

Figure 3.2: Normalized Model Size Over 2 Years in Meta

The Figures 3.1 and 3.2 reflects the work presented by Eisenman et al. [13]. The analysis shows training job failures across 21 clusters over a month, revealing that the longest 10% of failures occurred after at least 13.5 hours, and the top 1% after at least 53.9 hours. These jobs require 128 GPUs spread across multiple nodes and interacts with various systems in the network. A single failure in any interdependent component can significantly disrupt the entire training operation. Moreover, the significant increase in model sizes like tripling over two years as shown in Figure 3.2, presents challenges for checkpointing systems.

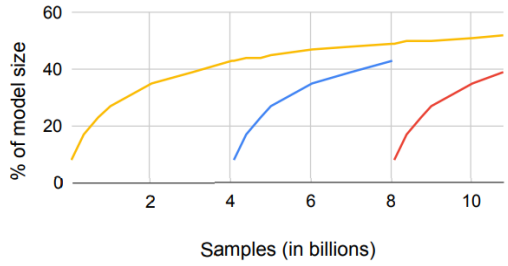

Figure 3.3: Fraction of Model Size Updated

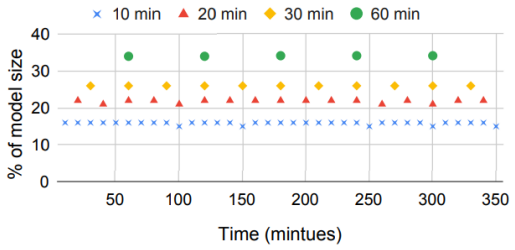

Figure 3.4: Consistency of Model Updates Across Time Intervals

Figures 3.3 and 3.4 continue to explore the dynamics of model training as initially detailed by Eisenman et al. [13]. The first figure shows that the fraction of recommendation model modified during training remains relatively small even when processing up to 11 billion records. The second graph highlights the consistency in the fraction of the model size modified over specific time intervals, emphasizing the sparsity of updates in large-scale models.

These observation presents opportunities for optimizing computation and storage efficiently, and continuous checkpointing system can be instrumental in overcoming these challenges. By focusing on only the updated embeddings of the model, we ensure effective resource utilization using continuous checkpointing. Furthermore, by integrating efficient data streaming and decoupling checkpointing operations from model training can further enhance the system's performance.

Chapter 4

Methodology

The methodology section focuses on the implementation of the continuous checkpointing system tailored for Deep Learning Recommendation Model (DLRM). The section introduces a refined approach to managing large-scale embedding data from DLRM training. By employing a asynchronous process to decouple checkpointing operations from model training and leveraging data streaming platforms like Kafka, the system ensures that updated embeddings from training are efficiently captured and stored. This method not only minimizes the impact on training performance but also significantly reduces the data loss compared to the traditional checkpointing systems by increasing the checkpointing frequency.

Furthermore, a standalone loading service has been introduced to manage the retrieval of the complete model state from the updated embeddings that are streamed during checkpointing. This ensures that in the event of a system failure or training interruption, the model can be quickly restored to a recent state, thereby reducing the training downtime which is essential in real-world industrial settings.

Overall, this dual approach significantly streamline the checkpointing and recovery process in model training, and provides a robust framework for handling large-scale recommendation model training tasks in distributed settings to ensure both efficiency and reliability.

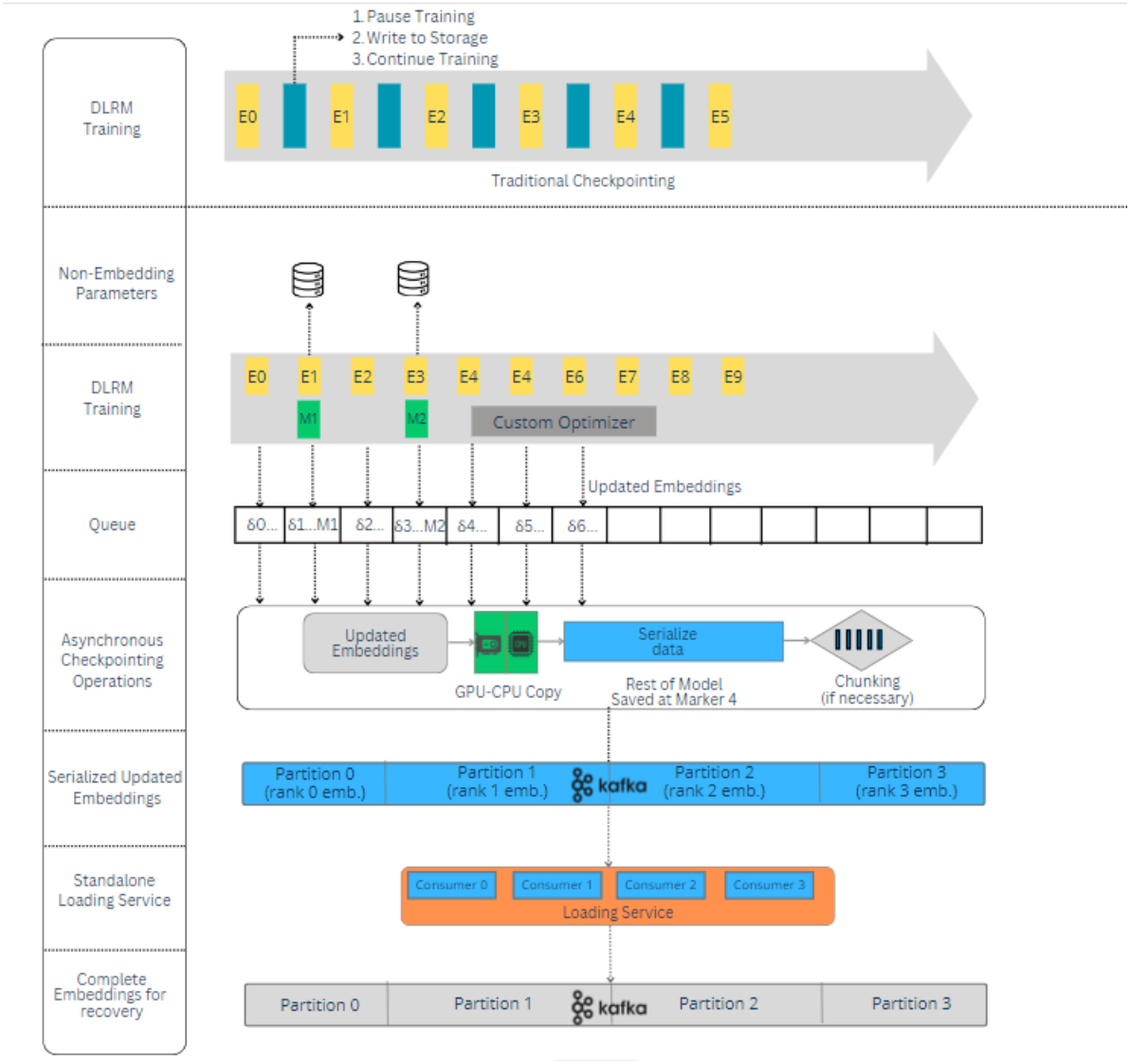

Figure 4.1: Overall Workflow in Continuous Checkpointing

4.1 Checkpointing Operation

The Checkpointing Operation within the continuous checkpointing system is designed to enhance the efficiency and robustness of training Deep Learning Recommendation Models (DLRM). This operation addresses the inefficiencies of traditional checkpointing methods by integrating several novel techniques: efficient extraction of updated embeddings, asynchronous data processing, optimized serialization processes, and robust data streaming mechanisms. The primary focus is the continuous saving of updated embeddings, while the rest

of the model (non-embedding parameters) is saved at required intervals. This approach significantly reduces the interruptions associated with traditional checkpointing activities by eliminating the need to pause the training process for data saving. Additionally, the system's capability to capture data continuously minimizes potential data loss between checkpoints and ensures that the most recent state of the model is preserved in cases of unexpected training failures.

The architecture of the checkpointing operation leverages asynchronous data handling and Kafka for data streaming to ensure that embedding updates are captured and stored with minimal impact on ongoing training operations. The system's design includes:

4.1.1 Updated Embedding Extraction

The updated embedding extraction component of the checkpointing operation is essential for extracting the updated embeddings during model training. This process focuses on precise identification and extraction of embeddings that have been updated during each training cycle. The processes involved in the extraction of updated embeddings are illustrated in Figure 4.2, which provides an overview of the entire mechanism.

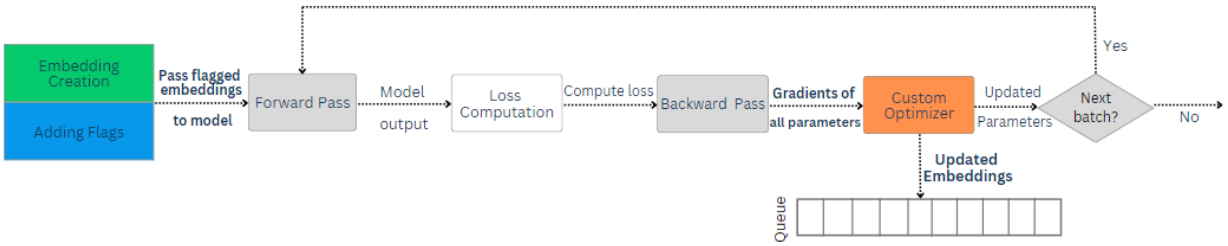

Figure 4.2: Overview of Updated Embedding Extraction

- **Flagging Embeddings During Creation**: During the creation of embedding layers, specific flags are added to identify tensors as embedding among optimizer parameters. This differentiation enables the system to apply specific operations related to continuous checkpointing to these parameters within the optimizer. Embedding layers are

flagged with an attribute (`is_embedding_param`), allowing the optimizer to recognize and handle these parameters separately, such as extracting updated embedding indices or applying standard optimization algorithms.

- **Custom Optimizer**: The optimizer in DLRM has been enhanced with a wrapper to extract updated embeddings efficiently. It can be seen in Figure 4.1 where the custom optimizer component integrates into the checkpointing workflow. Our implementation utilizes a stochastic gradient descent (SGD) optimizer, and this functionality can also be extended to other types of optimizers. The optimizer's step method evaluates each parameter to determine if it is marked as an embedding parameter. For params that are flagged, and where updates has occurred are indicated by non-zero gradients from backward pass, the optimizer efficiently extracts these specific updated embedding indices.

  Identifying these indices allows the system to track the parts of embedding matrix that have changed during training, and significantly reduces the additional computations required to extract the updated embedding. The indices of the embeddings that have updated are compiled into a list (`updated_indices_list`), which records the embeddings indices that need to be updated in the optimization process. Following this, in the (`_single_tensor_sgd`) function, we temporarily store all the values of the embedding tensor along with the updated indices in a tuple, which is later added to multiprocessing queue. We strategically store this information as it enables the extraction and processing of the updated embedding values, such as performing unique operations, and moving from GPU to CPU in later stages through non-blocking asynchronous operations instead of performing them inside the optimizer, which blocks the training process. By separating this data-capturing process from the training computations, we enhance system efficiency by enabling embeddings to be processed independently from the main training workflow.

  This table 4.1 provides a comparison of the optimizer performance between the cus-

Table 4.1: Optimizer Runtime and Total Runtime: Base vs. Continuous DLRM

Description	Base Optimizer (s)	Custom Optimizer (s)
Synthetic Data for 100 Epochs		
Total Time in Optimizer	0.5997	0.7607
Total Training Runtime	1591.08	979.09
Criteo Dataset for 5 Epochs		
Total Time in Optimizer	40.963	43.148
Total Training Runtime	3446.25	3292.27

tom optimizer used in the continuous checkpointing system and the base optimizer in DLRM with traditional checkpointing operations across two different datasets. Despite a slight increase in optimizer runtime, which accommodates additional computations for updated embedding extraction, this is minimal and negligible when compared to the significant benefits obtained in total runtime reduction. This demonstrates the efficiency of the custom optimizer in extracting updated embeddings without adding significant overhead to the training process.

4.1.2 Asynchronous Operations Framework

The Asynchronous Operations Framework plays a central role in the continuous checkpointing system for efficient data handling and transmission of data streams. This architecture is designed to manage the flow of data between the model and the Kafka messaging system, ensuring that checkpointing operations does not interrupt the training process. An overview of asynchronous processing and operations involved is shown in Figure 4.3 and Figure 4.4. The components of this architecture includes:

- **Dedicated Asynchronous Process**: A dedicated asynchronous process as seen in Figure 4.1 is used to handle the processing of data taken from the queue and to send updated embeddings to Kafka without interrupting the main training process. Figure 4.4 provides an overview of operations within the asynchronous process. This decoupling is essential to ensure that the computation intensive operations in the training process is not being blocked or slowed down by the operations of data handling and

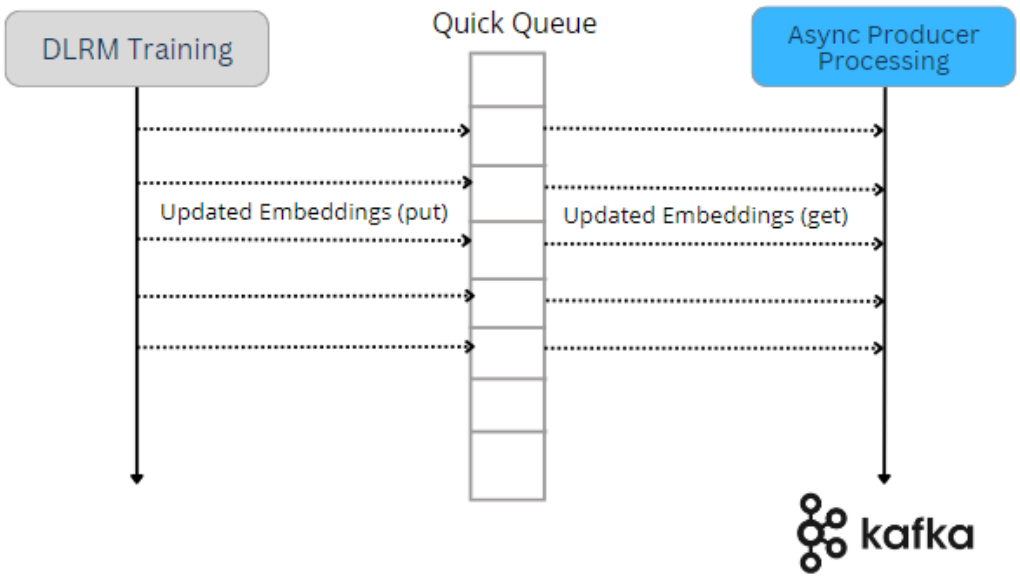

Figure 4.3: Asynchronous Operations Framework Overview

network communication in checkpointing. The asynchronous process is responsible for collecting updated embeddings from the queue, extracting updated values from indices, moving from device to host, serializing, and then producing the data to Kafka.

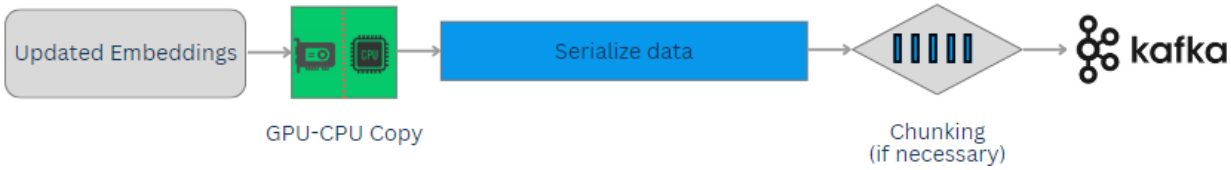

Figure 4.4: Operations in Asynchronous Process

The extraction of updated values from all values as mentioned in custom optimizer section, using the updated embedding indices also leverages a parallel processing mechanism where each embedding layer is processed in parallel to extract the values. By assigning a dedicated process, the system can decouple the computational load related to data handling from the training computations. This separation helps to improve the performance of the continuous checkpointing process.

- **Event-driven Architecture**: In continuous checkpointing system, we use a queue to manage the flow of data between the custom optimizer used in training and an asynchronous checkpointing process. As the training progresses and updated embeddings are generated, they are efficiently transferred to separate processes for additional handling before producing to Kafka without disrupting the ongoing training computations.

  The Quick Multiprocessing Queue enhances this data flow by addressing the bottlenecks in standard multiprocessing queues. The Quick Multiprocessing Queue [29] is specifically designed to optimize data transfer speeds between concurrent processes in a multiprocessing environment. This queue outperforms the standard queue by reducing the overhead associated with putting and getting elements, which is essential for high-throughput data handling in continuous checkpointing system. The key to the Quick Multiprocessing Queue's performance is its ability to minimize the locking and synchronization overhead that typically slows down standard queues, especially in multi-processing environment. By reducing the lock contention, the Quick Queue ensures that data can be enqueued and dequeued with minimal delay. Faster queue operations also facilitates the operations in asynchronous process to spend less time waiting for data availability and be optimized for performing necessary computations.

Table 4.2: Performance Comparison of Multi-Processing Queue vs. Quick Queue

Operation	**Multi Processing Queue (s)**	**Quick Queue (s)**
Put Operation	1.621×10^{-5}	$\mathbf{1.669 \times 10^{-6}}$
Get Operation	0.001159	$\mathbf{1.669 \times 10^{-6}}$

This table highlights the performance of Quick Queue and Multi-Processing Queue, demonstrating faster put and get operations for updated embeddings in continuous checkpointing system. Even though the differences in timings may appear minimal, these advantages accumulate significantly over the course of training process that involves a large number of batches.

4.1.3 Non-blocking Data Handling With Streams

In high-performance computing environments, the ability to manage data operations asynchronously is essential for efficiency. This is achieved by non-blocking data handling with streams, which allows data processing tasks to run concurrently with core computational activities. Using streams, various GPU computational tasks can overlap in execution with data transfer operations, thus maximizing the utilization of GPU resources [18, 19, 20]. For instance, data being moved from device to host can be processed in one stream while the another stream handles computations for the subsequent training batch.

CUDA Streams are used in continuous checkpointing system to manage the frequent transfer of data from the device to host. This operation prepares the data needed for serialization and subsequent transmission to Kafka without interfering the ongoing training processes on the GPU. To initiate the transfer, we create a CUDA Stream and the data transfer commands are issued on this stream. The use of a separate stream for these operations ensures that they are not blocking other computations occurring on the default stream, allowing the GPU to continue executing other tasks concurrently.

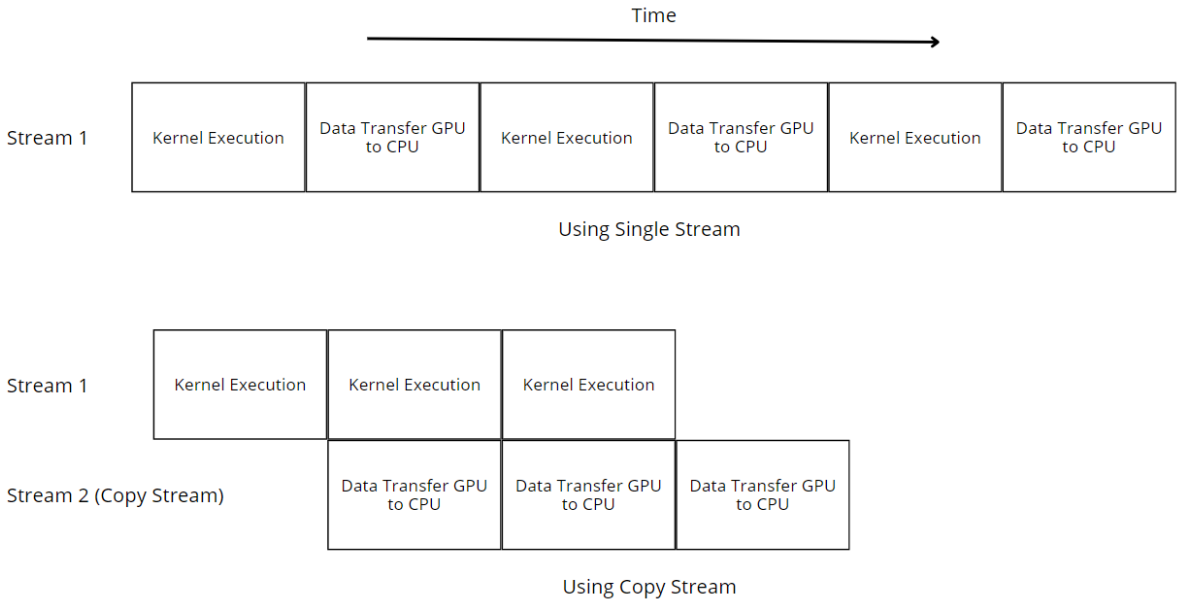

Figure 4.5: Execution Flow with CUDA Copy Stream

Sample execution flow of kernel computations and data transfers with CUDA Streams is shown in Figure 4.5. By facilitating concurrent data operations alongside GPU computations, CUDA Streams ensure that the system can handle frequent device-to-host copy operations efficiently without interruptions.

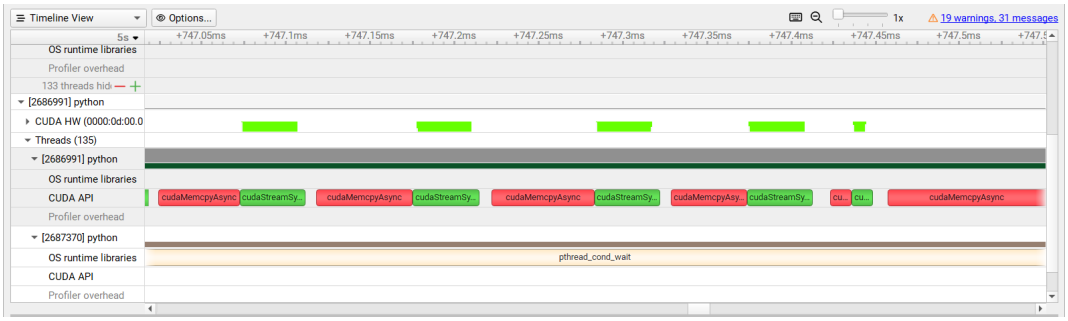

Figure 4.6: Sequential Operations Without CUDA Copy Stream

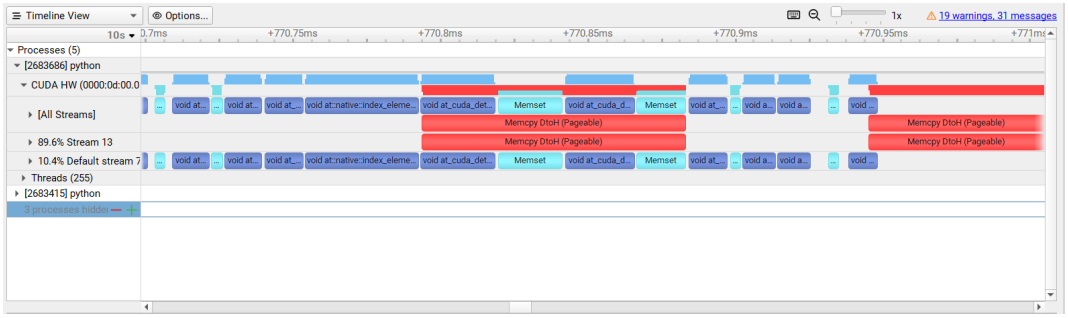

Figure 4.7: Overlapped Operations With CUDA Copy Stream

Figures 4.6 and 4.7 provide a comparison of memory operations in NVIDIA's Nsight profiler [30] for our Continuous Checkpointing system. In Figure 4.7, we demonstrate the use of CUDA streams, which enable the overlapping of memory operations and computational kernels. We leveraged various streams such as Stream 13 (copy stream), and the Default Stream, which are actively engaged in allowing the overlap of multiple tasks. While the Default Stream executes computational kernels, Stream 13 handles memory operations (indicated in red blocks in the figures) like data transfer between host and device (DtoH). In contrast, Figure 4.6 depicts the scenario without the use of CUDA streams, shows a sequential execution of tasks where operations are queued one after the another without overlap,

resulting in reduced efficiency in high-throughput scenarios. We see that the difference in operation handling between the two figures highlights the effectiveness of CUDA streams in enhancing parallel processing capabilities and optimizing the performance of memory-intensive operations.

4.1.4 Data Serialization

Data serialization is required for encoding the tensor embeddings into a format suitable for transmission over networks and for persistent storage. This process coverts the sparse embedding tensors into a compact and manageable format that preserves the integrity of data during storage and retrieval through network. By efficiently packaging the data, serialization reduces I/O overhead involved in transmitting data across a network, and writing and reading from storage devices. Serialization works by encoding the data into a structured format that includes both the raw data and metadata necessary for deserialization. In the continuous checkpointing system, embedding tensor indices and values are serialized using their specific data types optimized for computational performance.

We use a custom data serializer component to enhance this process within the continuous checkpointing system to facilitate faster serialization and reliable data exchange to the streaming platform. This setup helps convert sparse embedding tensors into binary data format that is suitable for network transmission and persistent storage. Serialization tasks are performed inside the asynchronous process such that model training is not interrupted.

- **Custom Binary Serialization**: The system employs a custom binary serialization process tailored to handle the specific requirements of updated embeddings data. This is designed to serialize large sparse data efficiently by encoding them into a binary format. The serialized format includes metadata such as data types and tensor shapes, allowing the consumer or recovery process to deserialize and reconstruct the original embedding data accurately.

- **Header and Metadata**: Each serialized message starts with a header (e.g., 'CSER' for custom serialization) that identifies the format of the data packet as custom serialized, followed by metadata about the number of layers, data types, and other essential parameters. This approach ensures integrity and consistency in data handling.

- **Data Encoding**: The tensor indices and values are encoded using appropriate data types that minimize space while preserving the accurate precision. For example, indices might be stored as int64, and embedding values are stored in formats like float32 or float64 depending on the embedding structure.

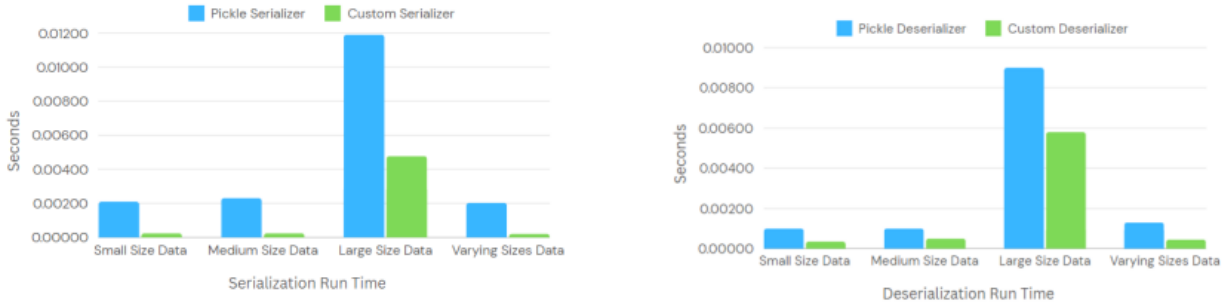

Figure 4.8: Custom Serialization and Deserialization Performance Comparison

The performance data illustrated in the Figure 4.8 highlights the efficacy of the custom serializer compared to the standard Pickle across various data sizes as shown in Table 4.3, specifically when handling sparse data with indices of type torch.int64 and values of type torch.float32. We observed that our custom serializer consistently outperforms the Pickle across various data sizes by a average of 79% faster serialization and 54% faster deserialization performance. It shows improved performance in small and medium data, significant improvement in handling large data, and consistent performance gains in varying data sizes.

The use of buffer `io.BytesIO` provides a dynamic memory allocation mechanism that supports the serialization of variable-sized data structures without needing predefined memory size. Memory views are employed to create a memory-efficient representation of the tensor indices and values. By using `memoryview`, the serialization process avoids copying

Layer	Small (Bytes)	Medium (Bytes)	Large (Bytes)	Varying (Bytes)
1	1200	12000	1200000	1200
2	1200	12000	1200000	2400
3	1200	12000	1200000	3600
4	1200	12000	1200000	4800
5	1200	12000	1200000	6000
6	1200	12000	1200000	7200
7	1200	12000	1200000	8400
8	1200	12000	1200000	9600

Table 4.3: Memory sizes of sparse layers for various configurations

the data, and creates a view on the existing buffers. These optimizations ensure that the serialization process is efficient in space and time, thus allowing the asyncronous operations to proceed without any major interruption caused by data serialization tasks.

4.1.5 Chunking Mechanism

The chunking mechanism component is designed to handle the serialized data, as shown in Figure 4.4 following serialization step, that exceed the maximum message size threshold of the Kafka producer. This method breaks down large chunks of serialized data into smaller manageable pieces which are sent sent sequentially to ensure the data is transmitted both efficiently and reliably. The chunking mechanism implemented in the `send_large_data_to_kafka` function checks the total size of the binary data to be sent and compares it with a predefined maximum size. If the data exceeds the threshold, the function divides the data into smaller chunks such that each conforms within the size limit.

- **Chunk Size Determination**: The maximum allowable chunk size is set based on Kafka's configuration, ensuring that each chunk can be efficiently managed by Kafka's internal mechanisms without message drop or delay. The `max_size` parameter is configurable, allowing for adjustments based on network conditions or performance requirements.

- **Sequential Data Chunking**: If the data size exceeds the maximum size, the seri-

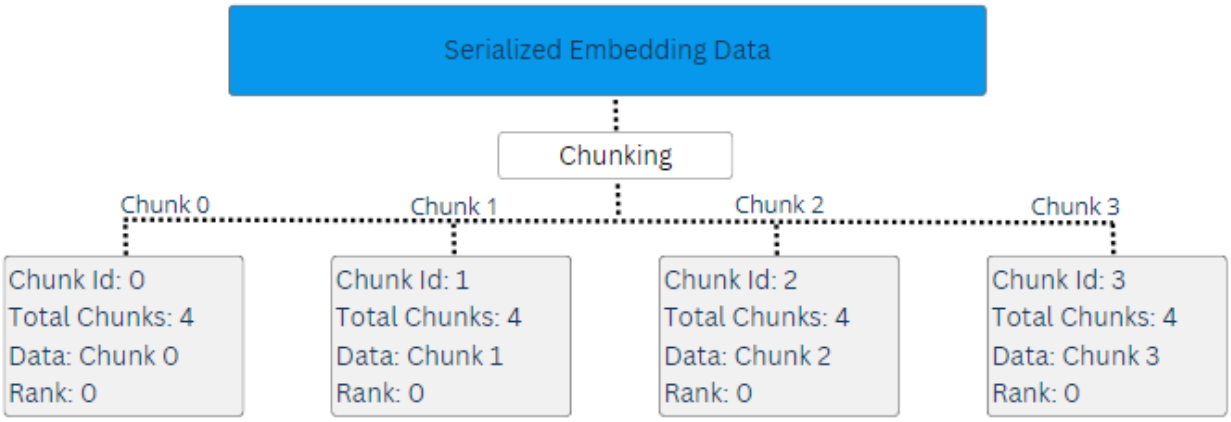

Figure 4.9: Chunking of Serialized Data

alized data is divided into multiple chunks. The number of chunks (`num_chunks`) is determined by dividing the total data length by `max_size` and rounding up to ensure all data chunks is produced to Kafka.

- **Metadata Inclusion**: Along with chunked data, each chunk includes metadata necessary for reassembling the chunks at the consumer. This metadata contains `chunk_id`, `total_chunks`, and `rank` required to correctly reproduce the data again during recovery process. In our implementation, as illustrated in Figure 4.9, we show how each chunk of data includes not only the segment of the serialized data but also the metadata necessary for correct reassembly at the destination. We ensure that every piece of data is partitioned and labeled with its respective metadata, and leveraging this structure to facilitate efficient and accurate data reconstruction during recovery.

- **Asynchronous Sending**: The chunks are then sent sequentially and asynchronously to Kafka, allowing the main training process to continue without interruptions by data transfer to Kafka.

4.1.6 Data Streaming with Kafka

Once the embedding data is processed in the separate asynchronous process, the serialized data is produced to Kafka as illustrated in Figure 4.1. Kafka is specifically employed for continuous checkpointing due to its high throughput, fault tolerance, and distributed nature, which are critical for handling large-scale data streams efficiently. Also, Kafka's partitioning capabilities are essential for distributed checkpointing operations, as they allow for the segregation of data streams based on parameters such as model ranks. The Figure 4.10 shows Kafka deployment configurations with a single node setup for localized operations, which can reduce latency from data transfer over network, and a multi-node setup that enhances scalability and distributes computational load across several nodes.

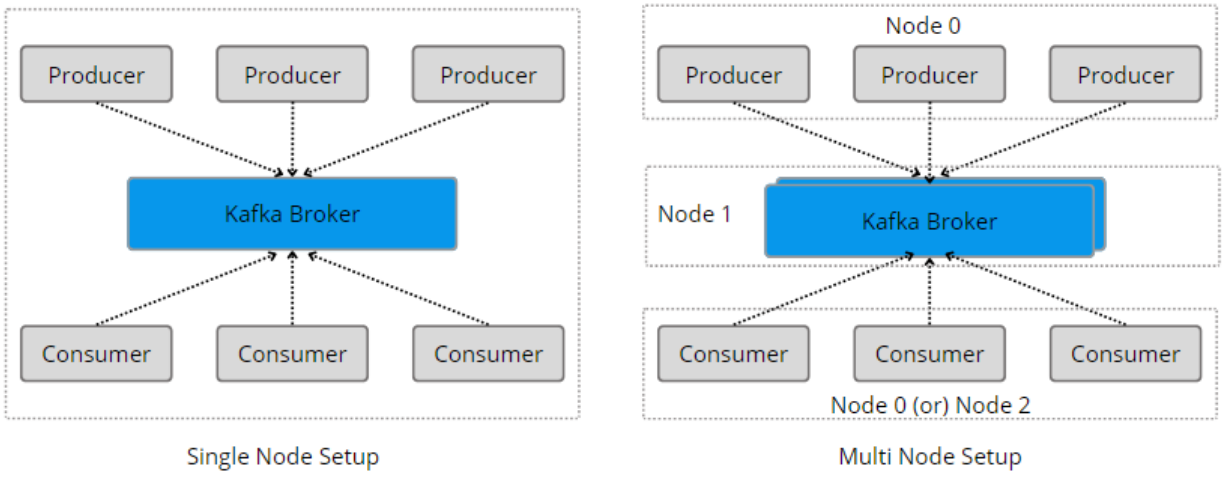

Figure 4.10: Kafka Deployment Configurations

Optimized data streaming with Kafka leverages the capabilities of Confluent Kafka to manage data streaming within the system efficiently. The Kafka producer is configured [25] with custom settings that are optimized for the specific requirements of the embedding data. Key configurations include increasing the `message.max.bytes` setting, which allows the producer to send larger messages than Kafka's default limits. This is essential when dealing with large model states or embeddings that sometimes exceed typical message sizes. Additionally, `queue.buffering.max.kbytes` is increased to provide a larger buffer space on

the producer side, which helps in accommodating bursts of data without dropping messages or causing back pressure in the system.

The Kafka producer uses these settings to efficiently manage the flow of data such that messages are produced with minimal latency. By efficiently managing data streams and utilizing Kafka's partitioning capabilities, the architecture can scale out to handle increasing data in distributed settings without significant reconfiguration of the underlying infrastructure.

4.1.7 Marker-based State Management

In continuous checkpointing system, marker-based state management enables the synchronization between the embedding data in Kafka stream and the non-embedding parameters. Markers are placed within the data stream to indicate the significant checkpoints which are essential for accurate, targeted recovery of model states during the recovery operation when necessary.

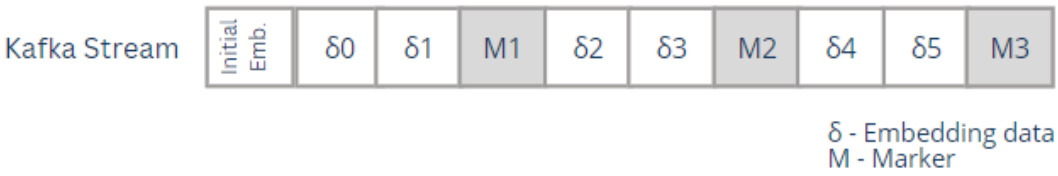

Figure 4.11: Marker placements in the Kafka Stream

Markers serve as indicators within the continuous stream of embedding data being sent to Kafka as shown in Figure 4.11. Their primary purpose is to:

- **Indicate Checkpoints**: Markers signal significant points in the training process, such as the end of an epoch where non-embedding parameters is saved. This signaling also helps us in identifying which portions embeddings in the data stream correspond to specific states of the model.

- **Facilitate Targeted Recovery**: In the event of system failure or when model rollback is required, markers allow for precise restoration of the model at that point. Since at the marker point the non-embedding parameters is saved to disk, and also the complete embedding is available, the model can be restored at that point and the training can be continued.

4.1.8 Saving Non-Embedding Parameters

With the placement of markers within the Kafka stream, the Non-Embedding Parameters (rest of the model with weights of MLPs, model architecture details, optimizer state information, training state like epoch number, etc.,) is concurrently saved to disk. This part of the model typically constitutes less than 1% of the entire model size, but it includes essential components that are not voluminous as embeddings. We save the non-embedding parameters to disk using the normal save operation using `torch.save`. By saving these components concurrently with sending of markers, the system maintains a consistent state across all parts of the model, ensuring that restoration can be performed reliably.

Table 4.4: Non-Embedding Parameters Save Time Compared to Entire Model Save Time

Save Frequency	**Non-Embedding Parameters Save (s)**	**Entire Model Save (s)**
Synthetic Data for 100 Epochs		
Every Epoch	1.029	803.04
Every 2 Epochs	0.532	404.36
Every 3 Epochs	0.350	281.56
Criteo Dataset for 5 Epochs		
Every Epoch	0.016	187.65
Every 2 Epochs	0.007	56.63
Every 3 Epochs	0.004	20.28

The table 4.4 contrasts the save times for non-embedding parameters in continuous checkpointing system against full model save times in a base DLRM implementation across different save frequencies for both Synthetic and Criteo datasets. The results demonstrate that saving non-embedding parameters is significantly faster compared to saving the entire model. This efficiency underscores the practicality of frequent saves of smaller non-embedding model

components which minimizes disruption and enhances the frequency of checkpointing, thus significantly reducing data loss.

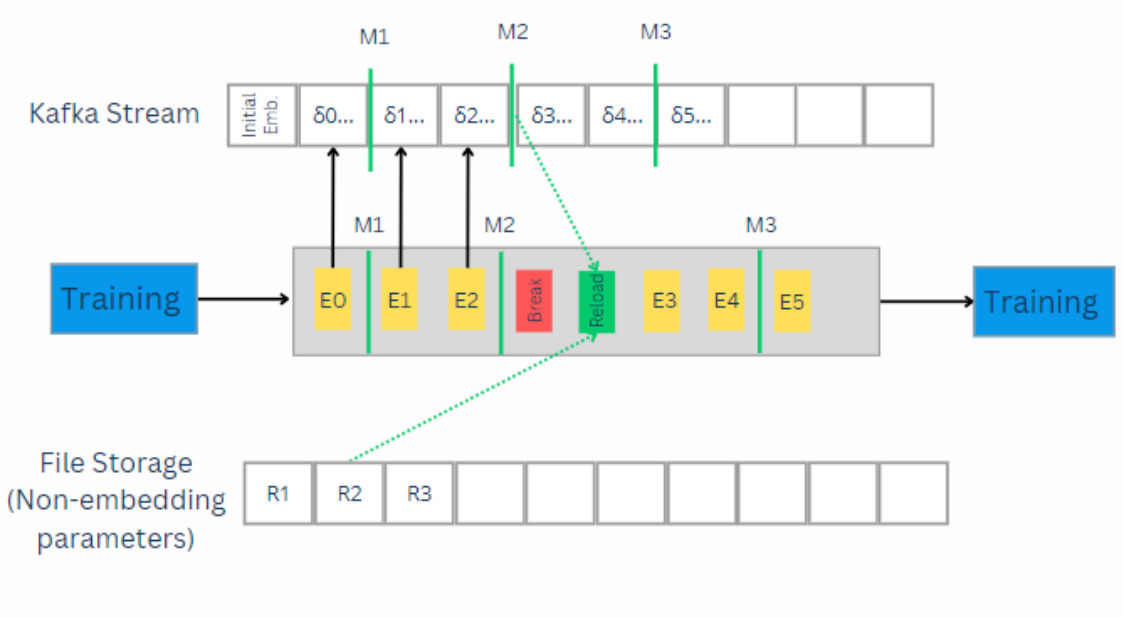

Figure 4.12: Non-Embedding Parameters Saving Workflow

4.1.9 Distributed Scalability of Continuous Checkpointing

Distributed Scalability is essential for continuous checkpointing system to scale for distributed training with mulitple-nodes and it is achieved through partitioned data streaming using Kafka. By directing data streams into Kafka topic partitions based on the rank of embedding processing devices, the system ensures that data handling remains efficient and manageable even if the scale of operations increases.

Partitioned data streaming involves distributing the data across various partitions within a Kafka topic. Each partition is an independent channel that handles a subset of the embedding based on the rank in which they are processed, allowing for parallel processing and data management. Kafka can also be configured to run on any node in the network as required. This strategy is particularly beneficial in distributed environments where embedding data from multiple devices need to be processed, as shown in Figure 4.13.

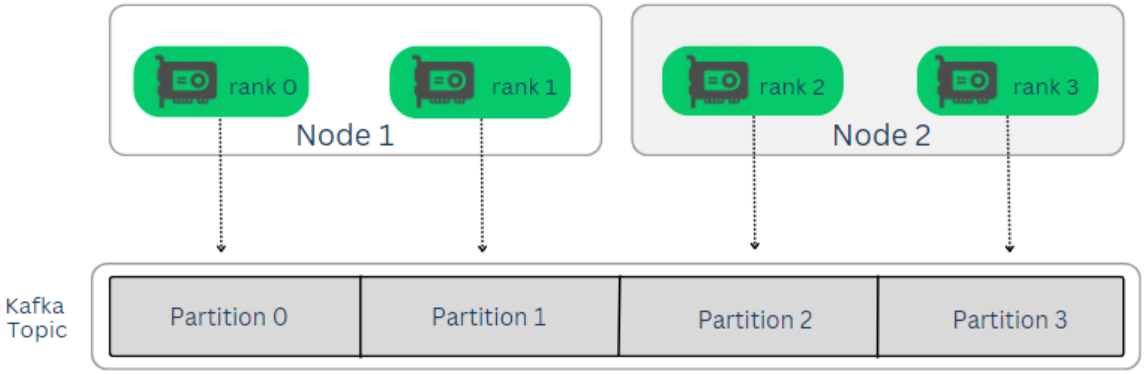

Figure 4.13: Distributed Data Streaming with Kafka Partitions

The integration of this partitioning mechanism into the system is achieved as follows:

- **Rank-Based Partition Assignment**: Each device in the distributed system is assigned a unique rank. We use this rank in the data streaming process, as it determines the partition to which the device's embedding data is sent. By mapping the rank to a specific partition, the system ensures that all data of that device process is streamed consistently to the same partition, as depicted in Figure 4.13. This consistency is necessary for maintaining integrity of the data, which is particularly necessary during the recovery operation to recover the embeddings to their respective processing ranks.

- **Kafka Producer Configuration**: The Kafka producer is configured to recognize the rank of the data and use this information to route the data to the appropriate partition. This is done by specifying the partition value in the producer's send method, ensuring that each data is produced to the correct partition based on its rank.

4.1.10 Hybrid Parallelism in DLRM

The complexity of DLRM necessitates a sophisticated approach to parallelism, especially due to memory-intensive nature of embeddings and computationally intensive characteristics of the multi-layer perceptrons (MLPs). To efficiently manage these aspects, DLRM employs a

hybrid parallel approach [4]:

- **Model Parallelism for Embeddings**: Due to the large size of embeddings which can require several gigabytes of memory, model parallelism is used. This type of parallelism distributes parts of the model (particularly the embeddings) across multiple devices to fit within memory constraints without needing to replicate large embeddings on every device.

- **Data Parallelism for MLPs**: The MLP components, which are smaller in memory but substantial in computational demands benefit from data parallelism. This approach allows for the concurrent processing of samples on different devices, with communication only required for accumulating updates during the backward pass.

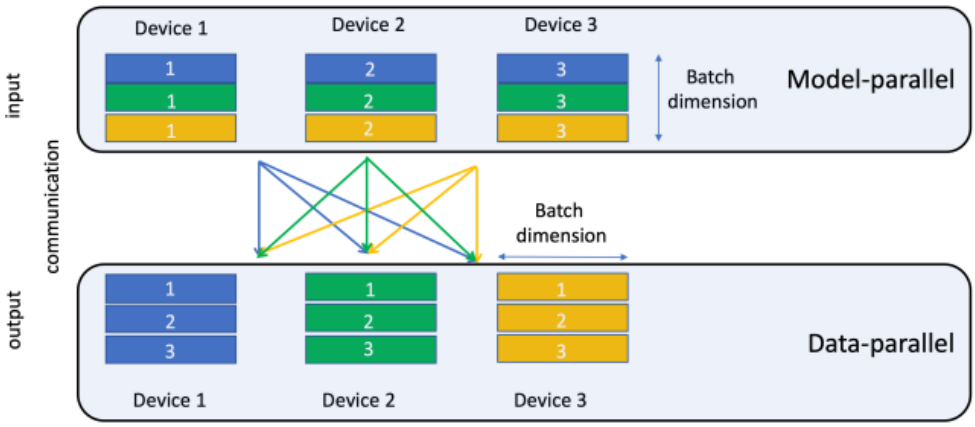

Figure 4.14: Hybrid Parallel Training Scheme in DLRM [4]

The rank-based handling of embeddings provides an efficient way to manage embeddings in the distributed training process, based on the hybrid parallel approach described for DLRM. In the context of model parallelism of embeddings, embeddings are divided and distributed across multiple devices where each device is responsible for a portion of the embeddings. The rank-based system aligns each portion of the embeddings with a specific device to organize the storage, retrieval, and processing of this data.

As we discussed earlier, data for each rank is produced to the corresponding partitions of a Kafka Topic, facilitating efficient and targeted data management. When a system recovery or rollback is needed, embeddings can be precisely retrieved from their designated partitions and restored to the appropriate device, ensuring that training can resume without inconsistencies or data integrity issues. This embedded handling strategy guarantees logical and effective data operations in distributed settings.

4.2 Recovery Operation

The recovery operation refers to the process of restoring a model's state from previously saved checkpoints. These checkpoints represent snapshots of the model's parameters and state at specific intervals during training. This is crucial for recovery and continuity purposes, enabling the system to revert to a known state in the event of failures and to resume training.

In continuous checkpointing system, a standalone loading service is designed to manage the complexities associated with the recovery operation efficiently. The loading service continuously builds the complete embeddings using updated embeddings on top of initial embedding. This service ensures that the process of retrieving, and integrating the embeddings does not interfere with the ongoing training operations, and enables faster model recovery by continuously preparing embeddings for instant training recovery when necessary.

- **Immediate Availability for Recovery**: The system ensures that a complete version of the embeddings is always available in case of a failure or need to revert to a previous state by constantly preparing and updating the embeddings. This availability reduces the recovery time and complexity, as the system does not need to reconstruct embeddings from the beginning of the training process every time from Kafka stream.

- **Reduction of Recovery Time**: Continuously preparing complete embeddings mean that we can efficiently load the most recent state directly, which is readily available, rather than processing all historical data from Kafka from the start of the training.

This approach significantly speeds up the recovery process, making it both faster and less resource-intensive.

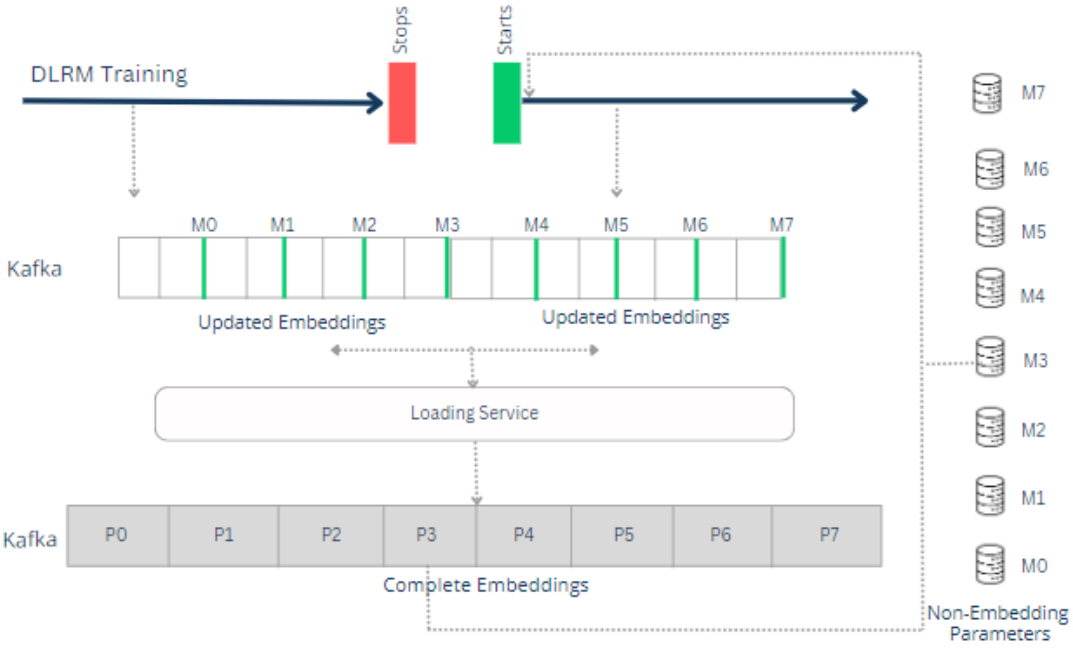

Figure 4.15: Recovery Operation Workflow Overview

It is also essential that the embeddings loaded from the checkpoints are synchronized with the non-embedding parameters. The loading service by utilizing markers in data stream as shown in Figure 4.12, ensures that the embeddings align with the specific training epochs or iterations.

4.2.1 Asynchronous Data Recovery

Asynchronous data recovery operations ensures that the process of updating the embeddings is both efficient and scalable. Using a multi-process approach, this method spawns multiple Kafka consumers where each consumer is dedicated to a specific partition that corresponds to a model rank. We implemented this strategy for enhancing the performance of the recovery operation across the distributed system with multiple producers. The asynchronous recovery

process is designed to handle multiple data streams concurrently, which is achieved through the following mechanisms:

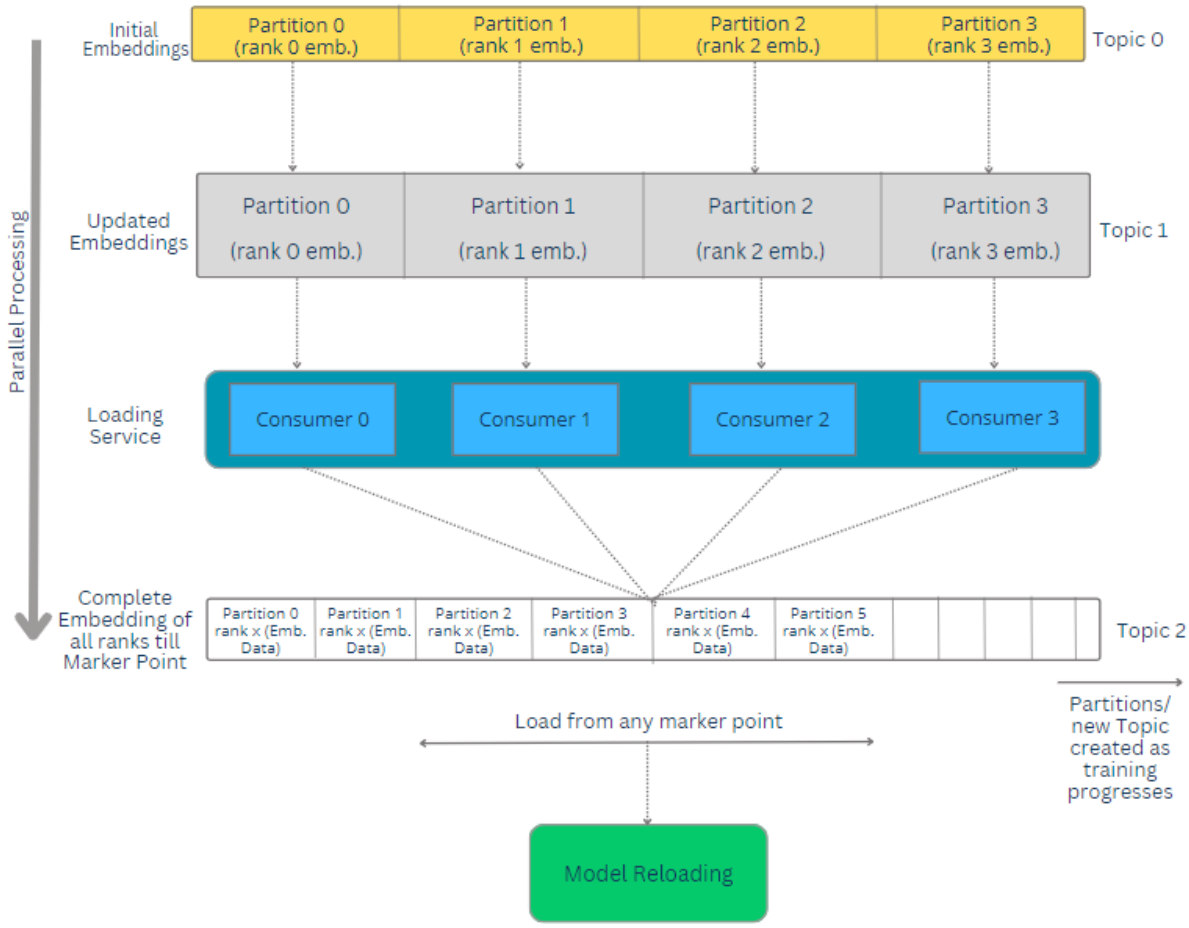

Figure 4.16: Loading Service Operation Architecture

- **Dedicated Kafka Consumers**: In this setup, multiple Kafka consumers are spawned and each consumer is assigned to a specific partition corresponding to the model's rank. This assignment means that each consumer handles data exclusively from its designated partition. As illustrated in Figure 4.17 and Figure 4.16, this focused approach minimizes unnecessary data processing. Each consumer processes only the data relevant to their operation, reducing the overhead and potential errors that could occur when dealing with unrelated information like embeddings from other ranks which may vary

in structure or layers. This targeted method of handling data not only enhances the speed of data processing but also decreases the potential of data mismatches when reconstructing the complete embedding's state.

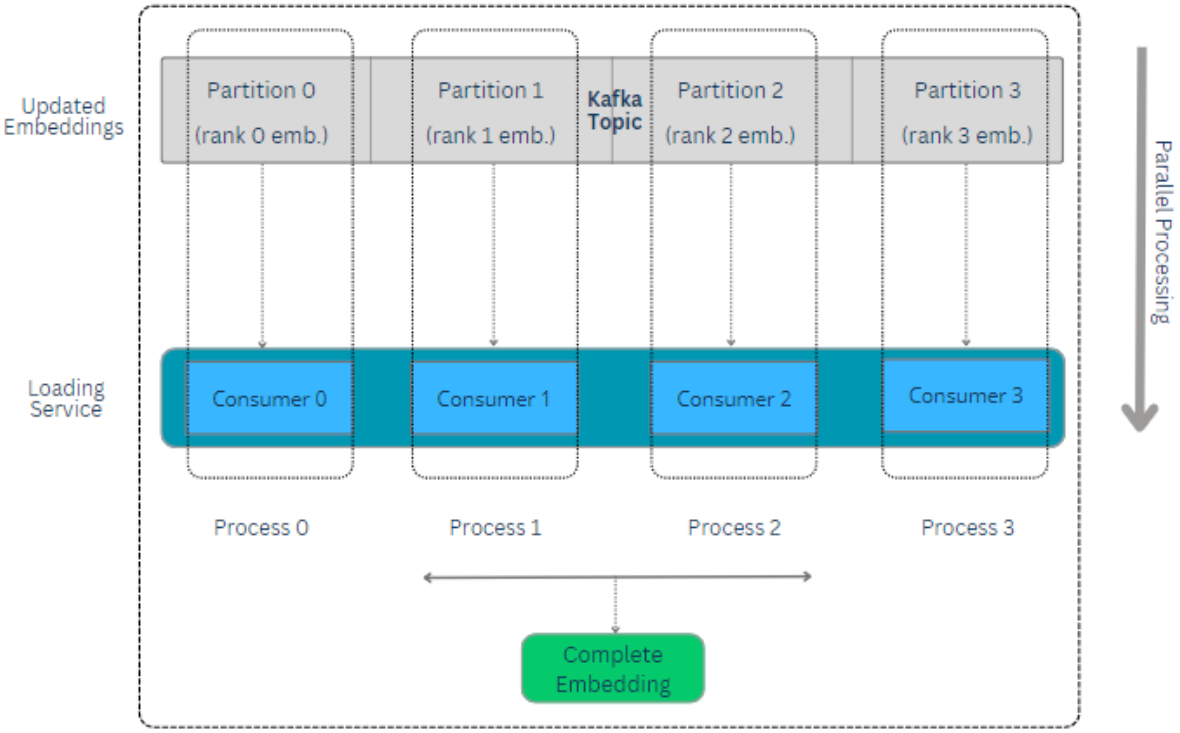

Figure 4.17: Parallel Data Retrieval inside Loading Service

- **Parallel Data Retrieval**: The multi-process approach which allows for parallel data retrieval operations from Kafka consumer, as described above, is key to handling large volumes of data efficiently, and it reduces the time required to load the necessary data from Kafka. Parallel processing as depicted in Figure 4.17 ensures that while one consumer is retrieving and processing data from its partition, other consumers can continue processing their partition data simultaneously. This overlap in processes significantly reduces the overall processing time of the loading service.

- **Optimized Service Deployment**: Optimizing recovery operations is crucial for distributed training, especially regarding the deployment of the loading service relative to

model training nodes as shown in Figure 4.18. Given that local data retrieval is typically faster than cross-network operations, deploying the loading service on the same node as the model training can reduce latency and data transfer times. Alternatively, deploying the loading service on a separate node can enhance system scalability and distribute the computational load. This also reduces the risk of a single point of failure impacting both training and recovery processes.

Figure 4.18: Deployment Configurations for DLRM Training and Loading Service

4.2.2 Model State Reconstruction

The reconstruction of the complete embedding involves continuously updating embeddings by deserializing and integrating the updated embedding data. Continuous embedding preparation keeps the model's embeddings up-to-date by applying incremental updates up to the latest checkpoint embedding. By utilizing technique similar to log compaction as shown in Figure 4.19, we enhance system efficiency for rapid recovery when required.

The deserialization process transforms serialized binary data back into a usable model state. This involves custom deserialization methods that similar the serialization process, using the meta-data like number of layers, data types, and other essential parameters which helps to maintain data integrity and reassembles the structure correctly. Once the data has been deserialized, it is integrated to reconstruct complete embeddings.

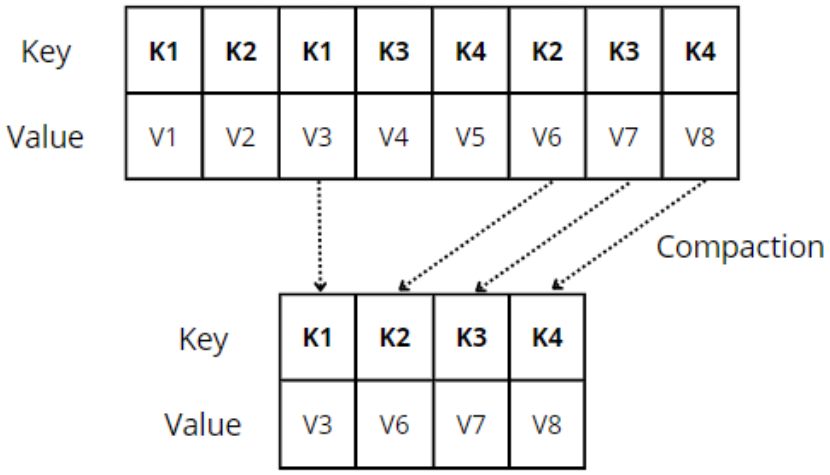

Figure 4.19: Before and After Compaction Visualization

Therefore, by continuously updating and integrating the updated embeddings on top of initial embedding as illustrated in Figure 4.20, the system enables the ability to quickly recover from disruptions. This capability is essential for maintaining high performance and lowering down-time in real-time training scenarios.

4.2.3 Dynamic Data Production to Kafka

In distributed training, managing the complete updated embeddings requires integration with Kafka. This integration involves producing updated embeddings to topic partitions and leveraging marker points for accurate recovery.

Complete embeddings are updated and sent to a Kafka topic that is organized into partitions based on specific marker points, as represented in Figure 4.16. Each partition is dedicated to storing a complete snapshot of embeddings that align with particular marker in the training process. This partitioning ensures efficient data organization and helps in the accurate retrieval of model states when necessary by synchronizing complete embedding in partition with non-embedding parameters.

When the system needs to recover or revert to a previous state, it uses these markers to

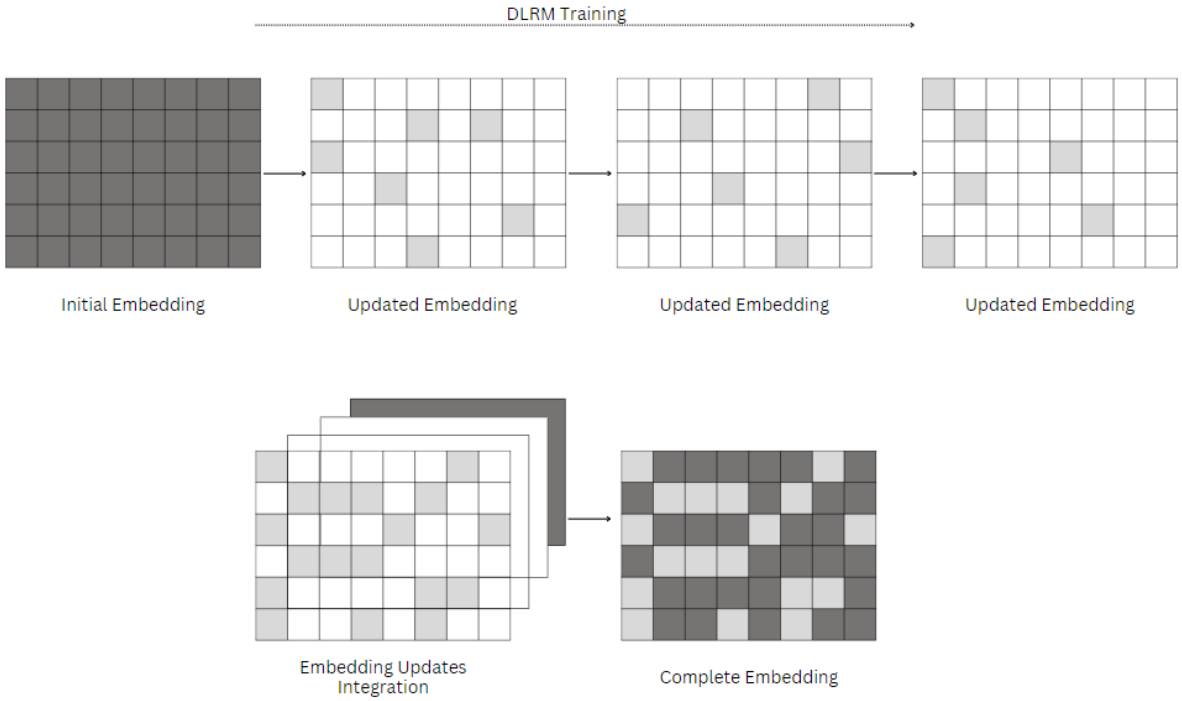

Figure 4.20: Overview of Updated Embeddings Integration

find and load the correct embeddings from the designated partition. This approach makes sure that the embeddings and other model parameters are synchronized and match the state recorded at the checkpoint. The system is also designed to handle storage efficiently, giving the option to either retain or delete old embeddings from Kafka depending on storage availability and operational needs.

This Figure 4.21 illustrates the process of restoring a model from a specific snapshot (Marker 4). It demonstrates the selective retrieval of data that are necessary to reconstruct a complete model state for a particular rank (Rank 0 in this case).

The Figure 4.21 shows 'Partition 4', which contains complete embedding data for multiple ranks at that marker point. For the restoration of the model at Rank 0, only the embedding data corresponding to Rank 0 is retrieved from this partition. This selective retrieval ensures that only relevant data is loaded into the model's state. Along with the embeddings, the non-embedding parameters saved at Marker 4 are retrieved from file system. This includes

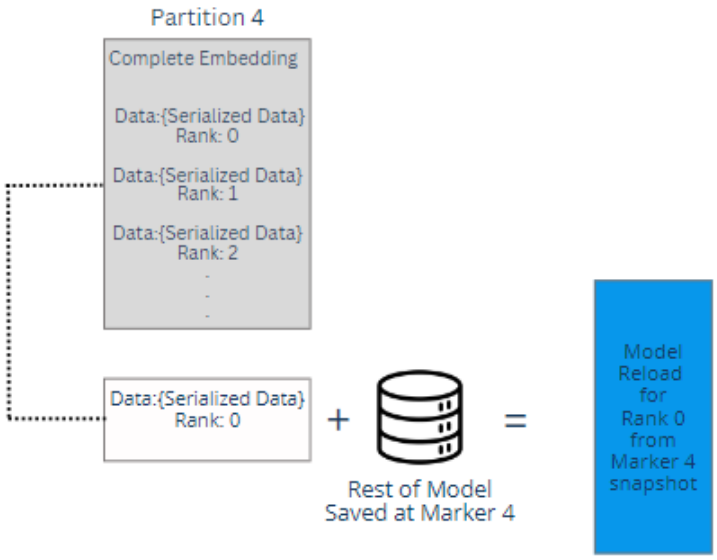

Figure 4.21: Sample Recovery Operation from a Partition

non-embedding parameters (rest of model) that are essential for the complete reconstruction of the model state. The combined data forms the complete model state for Rank 0, enabling the model to be reloaded to the state it was in at the snapshot taken at Marker 4.

Chapter 5

Datasets and Experimentation

5.1 Datasets Overview

In the development and testing of the continuous checkpointing system for Deep Learning Recommendation Model, two types of datasets were utilized to validate the system's performance across a range of training configurations. The datasets include the Criteo Display Ad Challenge Dataset and DLRM-generated synthetic and random data tailored for various training contexts.

5.1.1 Criteo Display Advertising Challenge Dataset

The Criteo Display Advertising Challenge Dataset [31] is an extensively utilized resource for benchmarking click-through rate (CTR) prediction models. This dataset consists of seven days of click-through data, providing a substantial amounts of real-world interactions for recommendation model training and testing.

The dataset is composed of both categorical and continuous values structured as follows:

- **Label**: This is the target variable that indicates whether an ad received a click (1) or not (0).
- **Integer Features (I1-I13)**: There are 13 columns of integer features in the dataset consisting of count data. These features provide quantitative insight into various at-

tributes associated with the ads, such as the number of times an ad was shown, interaction rates, and other relevant metrics for CTR predictions.

- **Categorical Features (C1-C26)**: The dataset includes 26 categorical features, which are anonymized and hashed into 32 bits to protect user privacy. These features encode aspects of the ad's context such as device type, user demographic, ad category, and other factors that could influence an ad being clicked.

In the context of the continuous checkpointing system described in this thesis, the dataset provides a testing ground to validate the system's performance under real-world conditions.

5.1.2 Synthetic Data

To complement the real-world data, random and synthetic generated datasets generated by the Deep Learning Recommendation Model (DLRM) were also used. We crafted these datasets to simulate a variety of training environments and stress tests, offering a controlled setting to evaluate the system's performance across different data characteristics and scales. The generation of this synthetic data is facilitated through scripts available in the DLRM code base [4]. We conducted the tests with the following architectural parameters, along with varying data sizes for synthetic data.

Table 5.1: DLRM Model Configuration Parameters for Synthetic Data

Parameter Description	**Configuration Description**
Bottom MLP architecture	3 layers with 512, 512, and 64 neurons
Top MLP architecture	4 layers with 1024, 1024, 1024, and 1 neurons
Sizes of Embedding Tables	8 embedding tables with 80,000 entries
Sparse Feature Size	Embedding dimensionality of 64
Interaction Operation	Dot product
Mini-batch Size	2048 samples per mini-batch
Number of Indices per Embedding Lookup	100 indices
Synthetic Data Sizes	Data sizes of 10,000, 25,000, and 50,000

Utilizing both the Criteo dataset and synthetic data enables thorough examination of the system's operational capabilities. With this method, the checkpointing system is tested under realistic conditions provided by the Criteo dataset, while also being validated in a range of configurations through synthetic data, thereby ensuring operability across various environments.

5.2 Experimentation

To evaluate the continuous checkpointing system performance, experiments were conducted across multiple server configurations each equipped with state-of-the-art GPUs and large memory capacities. This setup ensured that the system's capabilities were tested under conditions that resembled industry-level data processing demands.

5.2.1 Hardware Configuration

The development and experimentation utilized four servers for various training configurations like Single-Node Single-GPU, Single-Node Multi-GPU and Multi-Node Multi-GPU:

- **Servers with NVIDIA RTX A6000 GPUs**: These servers are each equipped with 4 NVIDIA RTX A6000 GPUs, providing 48 GB of memory per GPU for a total of 192 GB per server. This setup is particularly suited for processing extensive computations and large datasets efficiently.

- **Servers with NVIDIA A100 GPUs**: Each of these servers features 2 NVIDIA A100 GPUs with 80 GB of memory per GPU, amounting to 160 GB per server.

5.2.2 Distributed Backend

In distributed training of Deep Learning Recommendation Model (DLRM), NVIDIA NCCL (NVIDIA Collective Communications Library) is leveraged as the distributed backend. This

library enables multi-GPU and multi-node communications required for managing data transfers during distributed training. NCCL offers specialized communication protocols for computation needs of large-scale recommendation systems training.

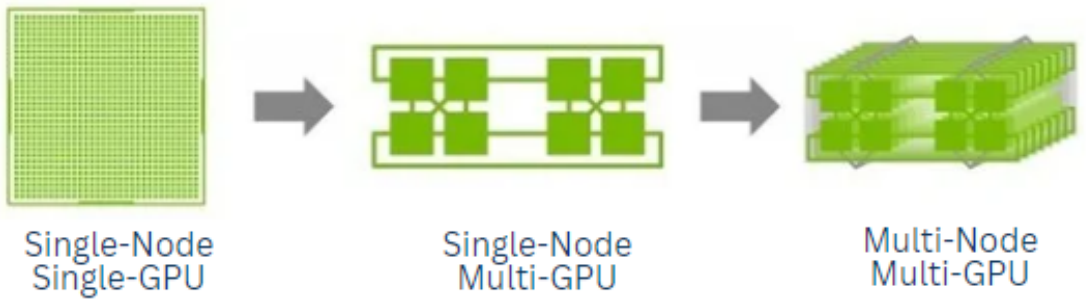

Figure 5.1: Scaling Up GPU Configurations with NCCL

Additionally, DLRM's scalability and synchronous training across multiple GPUs and nodes is supported by PyTorch's distributed framework. This framework accommodate wide range of deployment scenarios from single-node, multi-GPU setups to configurations involving multiple nodes and GPUs. Leveraging libraries such as `nn.DistributedDataParallel` and `nn.DataParallel`, PyTorch facilitates data parallelism by replicating the model on each device and managing the communication necessary for synchronizing updates across devices.

5.2.3 Baseline and Benchmark Comparisons

In evaluation of continuous checkpointing system, its performance was systematically benchmarked against base Deep Learning Recommendation Model (DLRM) setup which uses standard PyTorch checkpointing methods for checkpointing and recovery operations. This comparative analysis is necessary for demonstrating the specific efficiency gains and performance enhancements achieved by the continuous checkpointing system.

- **Comparison Focus**: Performance benchmarks were set against the base DLRM configuration which uses `torch.save` for checkpointing and `torch.load` recovery oper-

ations. Comparing against the baseline version demonstrates the efficiencies gained with the continuous checkpointing system.

- **Experimental Conditions**: Benchmarking tests were performed with a variety of data sizes, embedding sizes, and training durations (number of epochs) to ensure the system was assessed under comprehensive configurations.

- **Datasets**: The experimentation utilized both synthetic data generated to simulate various training scenarios, and real-world data from the Criteo Display Advertising Challenge Dataset.

Chapter 6

Results

This section details the results of runtime evaluation of checkpointing and recovery operations with continuous checkpointing system, compared to traditional PyTorch checkpointing in various configurations. We conducted tests across different environments including single node with single GPU, single node with multiple GPUs, and multiple nodes with multiple GPUs using both synthetic data for 100 epochs and the Criteo dataset for 5 epochs. The experiments focused on various data sizes and checkpointing frequencies to comprehensively assess the performance enhancements achieved with continuous checkpointing. These results are crucial for understanding how continuous checkpointing can optimize the checkpointing process in terms of speed and efficiency in diverse computational setups.

6.1 Checkpointing Operation

6.1.1 Single-Node Single-GPU

Tables 6.1 and 6.2 show that continuous checkpointing significantly reduces save times compared to base checkpointing across both synthetic and Criteo datasets on a single-node single-GPU setup. The improvement is more noticeable with larger data sizes and more frequent checkpointing, highlighting continuous checkpointing's effectiveness in minimizing overhead in training processes.

Table 6.1: Checkpoint Runtime using Synthetic Data in Single-Node Single-GPU

Save Frequency	**Data Size**	**Base Checkpointing (seconds)**	**Continuous Checkpointing (seconds)**
Every Epoch	10,000	336.49	**229.41**
	25,000	808.83	**534.75**
	50,000	1591.08	**979.09**
Every 2 Epochs	10,000	250.67	**220.23**
	25,000	598.97	**507.24**
	50,000	1194.72	**969.09**
Every 3 Epochs	10,000	239.10	**216.65**
	25,000	544.54	**506.76**
	50,000	1118.78	**954.92**

Figure 6.1: Single-Node Single-GPU with Synthetic Data

Table 6.2: Checkpoint Runtime using Criteo Dataset in Single-Node Single-GPU

Save Frequency	Base Checkpointing (s)	Continuous Checkpointing (s)
Every Epoch	3446.25	**3292.27**
Every 2 Epochs	3368.87	**3287.07**
Every 3 Epochs	3342.20	**3283.90**

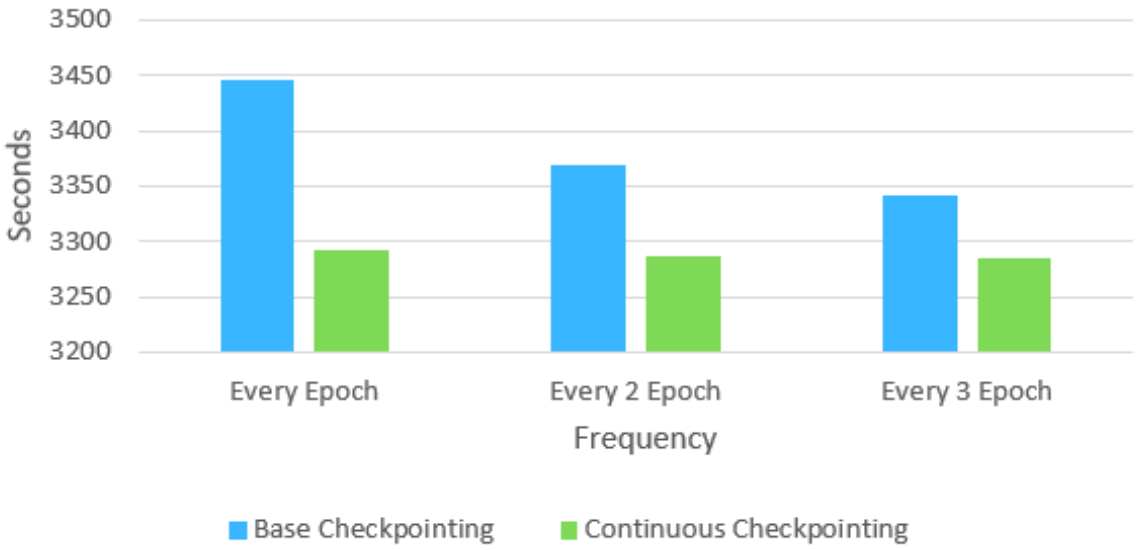

Figure 6.2: Single-Node Single-GPU with Criteo Data

Figures 6.1 and 6.2 illustrate the runtime advantages of continuous checkpointing over traditional methods using both synthetic and real-world Criteo dataset. We can see that continuous checkpointing consistently outperforms the base checkpointing method in terms of runtime by approximately 30% in synthetic and 5% in Criteo data. The improvement over the base method appears modest in the Criteo dataset is due to substantial volume of data being saved with continuous checkpointing.

6.1.2 Single-Node Multi-GPU

Tables 6.3 and 6.4 demonstrate that continuous checkpointing is significantly faster than base checkpointing in save times on a single-node with multiple GPUs setup across both synthetic

and Criteo datasets. The data shows marked time reductions for various checkpointing frequencies and data size with continuous checkpointing compared to base checkpointing.

Table 6.3: Checkpoint Runtime using Synthetic Data in Single-Node Multi-GPU

Save Frequency	**Data Size**	**Base Checkpointing (seconds)**	**Continuous Checkpointing (seconds)**
Every Epoch	10,000	328.93	**203.68**
	25,000	801.57	**463.48**
	50,000	1671.36	**880.66**
Every 2 Epochs	10,000	251.29	**198.43**
	25,000	602.42	**452.92**
	50,000	1193.89	**865.13**
Every 3 Epochs	10,000	242.59	**196.35**
	25,000	541.24	**446.68**
	50,000	1062.61	**861.19**

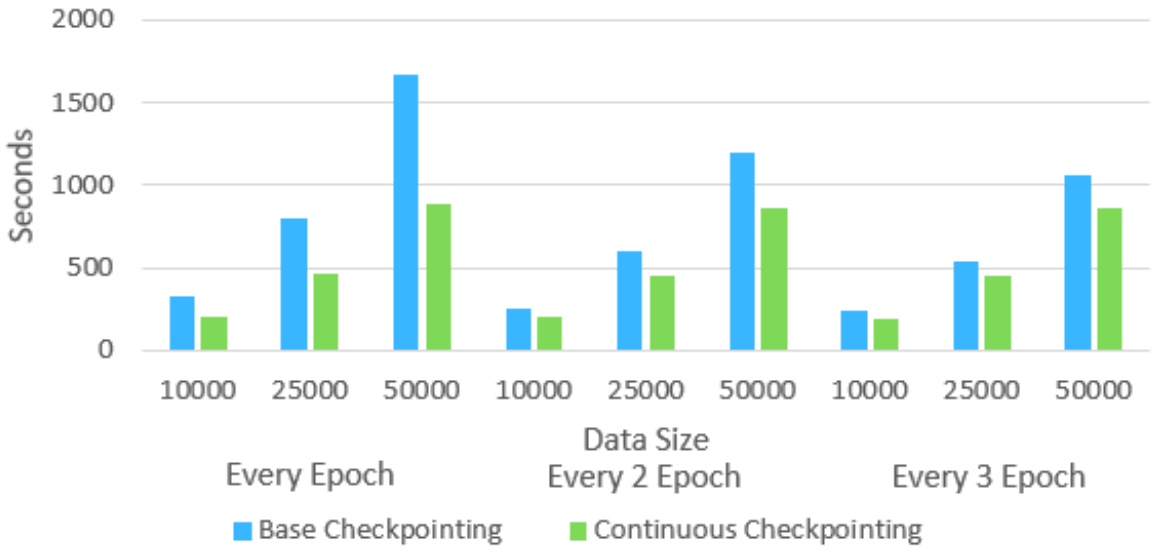

Figure 6.3: Single-Node Multi-GPU with Synthetic Data

Table 6.4: Checkpoint Runtime using Criteo Dataset in Single-Node Multi-GPU

Save Frequency	Base Checkpointing (s)	Continuous Checkpointing (s)
Every Epoch	3574.49	**3302.60**
Every 2 Epochs	3496.51	**3288.73**
Every 3 Epochs	3471.98	**3282.77**

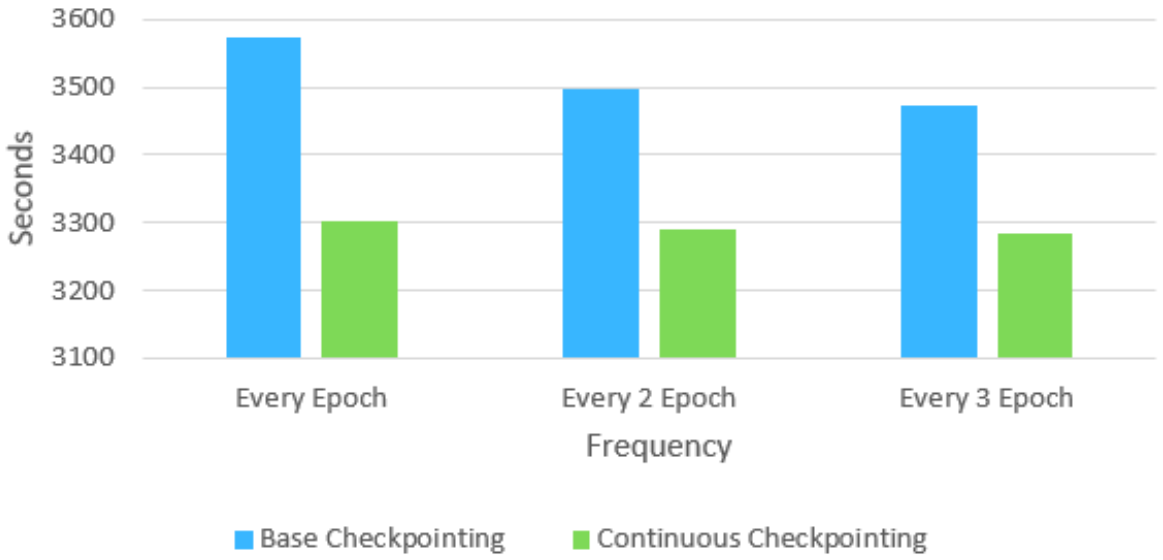

Figure 6.4: Single-Node Multi-GPU with Criteo Data

6.1.3 Multi-Node Multi-GPU

Tables 6.5 and 6.6 illustrate the performance of continuous checkpointing over base checkpointing in a multi-node environment with multiple GPUs using synthetic and Criteo datasets respectively. For the synthetic dataset, continuous checkpointing consistently achieves significant time savings across various data sizes and save frequencies.

Table 6.5: Checkpoint Runtime using Synthetic Data in Multi-Node Multi-GPU

Save Frequency	**Data Size**	**Base Checkpointing (seconds)**	**Continuous Checkpointing (seconds)**
Every Epoch	10,000	374.27	**171.76**
	25,000	757.87	**435.70**
	50,000	1426.05	**856.32**
Every 2 Epochs	10,000	249.52	**173.15**
	25,000	571.58	**430.70**
	50,000	1083.65	**863.29**
Every 3 Epochs	10,000	225.20	**170.13**
	25,000	513.09	**429.59**
	50,000	970.40	**869.15**

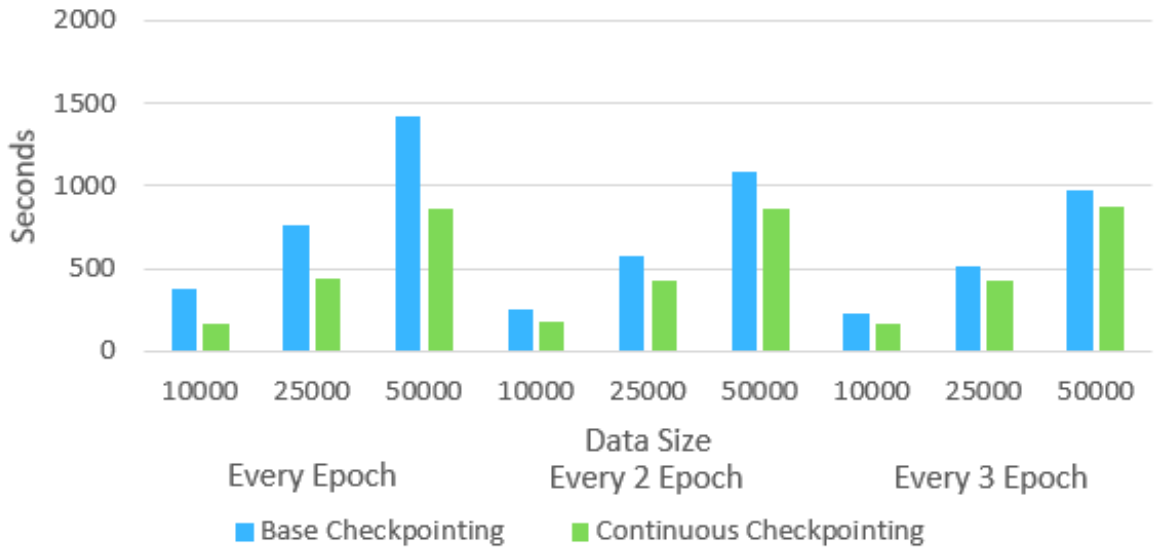

Figure 6.5: Multi-Node Multi-GPU with Synthetic Data

Table 6.6: Checkpoint Runtime using Criteo Dataset in Multi-Node Multi-GPU

Save Frequency	Base Checkpointing (s)	Continuous Checkpointing (s)
Every Epoch	3709.15	**3398.47**
Every 2 Epochs	3619.03	**3391.73**
Every 3 Epochs	3587.94	**3385.56**

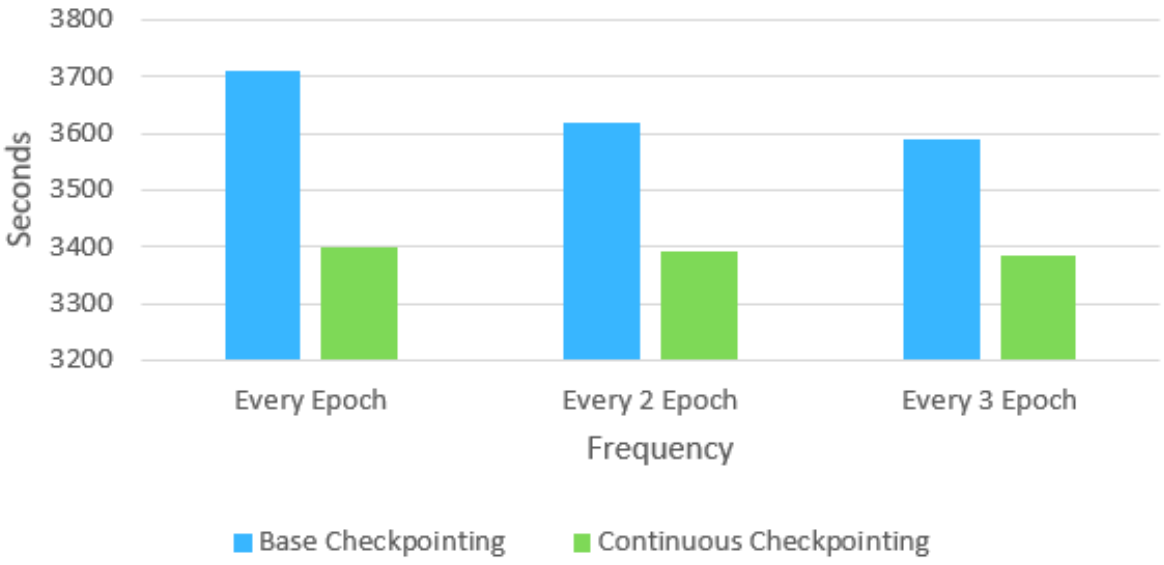

Figure 6.6: Multi-Node Multi-GPU with Criteo Data

Figures 6.5 and 6.6 demonstrate the performance in a multi-node multi-GPU environment with continuous checkpointing in reducing runtimes compared to base checkpointing by approximately 40% for synthetic data and 5% for Criteo dataset. The results highlight continuous checkpointing's efficiency in reducing checkpointing times and scalability across different data sizes and checkpoint frequencies even in multi-node distributed setup.

6.2 Recovery Operation

6.2.1 Single-Node Single-GPU

Tables 6.7 and 6.8 show differences in recovery operation times between continuous and base checkpointing methods on a single node with a single GPU. While continuous checkpointing

generally exhibits slightly more load time compared to the base method, the differences are minimal and unlikely to impact overall system performance significantly. In practical scenarios, this slight increase in load times of less than a second with continuous checkpointing is offset by its substantial improvements in checkpointing operation efficiency.

Table 6.7: Recovery Runtime using Synthetic Data in Single-Node Single-GPU

Data Size	**Base Checkpointing (s)**	**Continuous Checkpointing (s)**
10,000	0.09976	0.63970
25,000	0.09981	0.50413
50,000	0.09857	0.66877

Table 6.8: Recovery Runtime using Criteo Dataset in Single-Node Single-GPU

Base Checkpointing (s)	**Continuous Checkpointing (s)**
1.20537	2.28374

6.2.2 Single-Node Multi-GPU

Tables 6.9 and 6.10 shows the load times for continuous and base checkpointing on a single node with multiple GPUs. Continuous checkpointing shows slightly longer load times compared to the base method but the differences are minimal in practical applications.

Table 6.9: Recovery Runtime using Synthetic Data in Single-Node Multi-GPU

Data Size	**Base Checkpointing (s)**	**Continuous Checkpointing (s)**
10,000	0.09384	0.54548
25,000	0.09631	0.65086
50,000	0.09107	0.61471

Table 6.10: Recovery Runtime using Criteo Dataset in Single-Node Multi-GPU

Base Checkpointing (s)	**Continuous Checkpointing** (s)
1.31874	2.48593

6.2.3 Multi-Node Multi-GPU

Tables 6.11 and 6.12 shows that continuous checkpointing has slightly longer recovery times than base checkpointing on multi-node with multiple GPUs, but the differences are minimal. This is due to the continuous checkpointing's process of loading data from Kafka partitions, but the impact on practical applications remains negligible.

Table 6.11: Recovery Runtime using Synthetic Data in Multi-Node Multi-GPU

Data Size	**Base Checkpointing (s)**	**Continuous Checkpointing (s)**
10,000	0.08921	0.67014
25,000	0.08736	0.66839
50,000	0.08791	0.62437

Table 6.12: Recovery Runtime using Criteo Dataset in Multi-Node Multi-GPU

Base Checkpointing (s)	**Continuous Checkpointing** (s)
1.45381	2.89582

Chapter 7

Conclusion and Future Work

The implementation and deployment of the continuous checkpointing system, as discussed in this thesis, bring significant improvements to the large-scale training of recommendation models. This architecture not only improves the robustness but also streamlines the management of checkpointing process across diverse computational settings.

The system efficiently minimizes training disruptions by increasing the frequency and precision of checkpoints, and utilizing standalone loading service to decrease downtime. This enhancement allows for a seamless progression in recommendation model training. Its effectiveness is further proven through extensive testing with both synthetic and real datasets, confirming the checkpointing system's functionality across various training conditions and its efficiency in real-world deployment scenarios.

Additionally, the system demonstrates its scalability through successful implementation in diverse environments ranging from single to multi-node distributed configurations. This flexibility helps the system to scale according to the infrastructure requirements. The checkpointing system along with the loading service facilitates quicker restoration from failures which is essential for operational efficiency. Quick restoration of model states from checkpoints minimizes interruptions to ongoing activities, which is invaluable in time-sensitive and resource-limited real-world settings.

Lastly, the system presents better performance than traditional methods in run-time efficiency, delivering quicker training times and more efficient bandwidth usage. These im-

provements not only accelerate the model training but also reduce computational demands, resulting in more efficient resource utilization. Additionally, by performing more frequent checkpointing operations compared to traditional methods, the system significantly reduces data loss.

In future works, we will focus on expanding our system's effectiveness across a broader range of machine learning models, particularly those where embeddings play a major role, such as in natural language processing. This includes exploring compatibility with models like BERT [32], known for their complex embedding layers. We plan to implement quantization techniques as discussed in [33] and reduce embedding sizes using methods like Plug-in Embedding Pruning [34]. These steps are essential to decrease model size and speed up data streaming over networks. Additionally, we will develop more robust fault recovery protocols to increase the system's resilience against various types of failures. These enhancements will ensure higher system availability and reliability, which are critical for continuous operations in real-time industrial settings. Furthermore, we aim to develop advanced data compression techniques to improve serialization efficiency, enabling faster data transmission. These improvements will help minimize latency and maximize throughput during data exchanges, which are crucial for keeping pace with evolving technological demands and further optimizing machine learning workflows.

In conclusion, the continuous checkpointing system has potential to advance the reliable large-scale training for recommendation models. By enhancing efficiency, scalability, and robustness, it has not only improved over current checkpointing techniques but also paved the way for further developments. The ongoing enhancements and future expansions are expected to enhance the system's applicability, contributing greatly to the field of machine learning.

Bibliography

[1] Brent Smith and Greg Linden. "Two Decades of Recommender Systems at Amazon.com". In: *IEEE Internet Computing* 21.3 (2017), pp. 12–18. DOI: 10.1109/MIC.2017.72.

[2] Udit Gupta et al. *The Architectural Implications of Facebook's DNN-based Personalized Recommendation*. 2020. arXiv: 1906.03109 [cs.DC]. URL: https://arxiv.org/abs/1906.03109.

[3] Paul Covington, Jay Adams, and Emre Sargin. "Deep Neural Networks for YouTube Recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys '16. Boston, Massachusetts, USA: Association for Computing Machinery, 2016, 191–198. ISBN: 9781450340359. DOI: 10.1145/2959100.2959190. URL: https://doi.org/10.1145/2959100.2959190.

[4] Maxim Naumov et al. *Deep Learning Recommendation Model for Personalization and Recommendation Systems*. 2019. arXiv: 1906.00091 [cs.IR]. URL: https://arxiv.org/abs/1906.00091.

[5] Devesh Tiwari et al. "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 331–342. DOI: 10.1109/HPCA.2015.7056044.

[6] Elvis Rojas et al. *A Study of Checkpointing in Large Scale Training of Deep Neural Networks*. 2021. arXiv: 2012.00825 [cs.DC]. URL: https://arxiv.org/abs/2012.00825.

[7] Tonmoy Dey et al. "Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 1036–1043. DOI: 10.1109/IPDPSW50202.2020.00174.

[8] Bogdan Nicolae et al. "DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 172–181. DOI: 10.1109/CCGrid49817.2020.00-76.

[9] Trishul Chilimbi et al. "Project adam: Building an efficient and scalable deep learning training system". In: *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 2014, pp. 571–582.

[10] Aurick Qiao et al. "Fault Tolerance in Iterative-Convergent Machine Learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5220–5230. URL: https://proceedings.mlr.press/v97/qiao19a.html.

[11] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. 2017. arXiv: 1703.09039 [cs.CV]. URL: https://arxiv.org/abs/1703.09039.

[12] Borui Wan et al. *ByteCheckpoint: A Unified Checkpointing System for LLM Development*. 2024. arXiv: 2407.20143 [cs.AI]. URL: https://arxiv.org/abs/2407.20143.

[13] Assaf Eisenman et al. *Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models*. 2021. arXiv: 2010.08679 [cs.IR]. URL: https://arxiv.org/abs/2010.08679.

[14] R. Koo and S. Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems". In: *IEEE Transactions on Software Engineering* SE-13.1 (1987), pp. 23–31. DOI: 10.1109/TSE.1987.232562.

[15] L. Wang et al. "Modeling coordinated checkpointing for large-scale supercomputers". In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 2005, pp. 812–821. DOI: 10.1109/DSN.2005.67.

[16] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. "Fault Tolerance in Distributed Systems: A Survey". In: *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. 2018, pp. 1–5. DOI: 10.1109/PAIS.2018.8598484.

[17] Joel Hestness, Stephen W. Keckler, and David A. Wood. "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors". In: *2015 IEEE International Symposium on Workload Characterization*. 2015, pp. 87–97. DOI: 10.1109/IISWC.2015.15.

[18] N. V. Sunitha, K. Raju, and Niranjan N. Chiplunkar. "Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead". In: *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2017, pp. 211–215. DOI: 10.1109/ICICCT.2017.7975190.

[19] B. van Werkhoven et al. "Performance Models for CPU-GPU Data Transfers". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 11–20. DOI: 10.1109/CCGrid.2014.16.

[20] Burak Bastem et al. “Overlapping Data Transfers with Computation on GPU with Tiles”. In: *2017 46th International Conference on Parallel Processing (ICPP)*. 2017, pp. 171–180. DOI: 10.1109/ICPP.2017.26.

[21] Guozhang Wang et al. “Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: Association for Computing Machinery, 2021, 2602–2613. ISBN: 9781450383431. DOI: 10.1145/3448016.3457556. URL: https://doi.org/10.1145/3448016.3457556.

[22] Shubham Vyas et al. “Literature Review : A Comparative Study of Real Time Streaming Technologies and Apache Kafka”. In: *2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)*. 2021, pp. 146–153. DOI: 10.1109/CCICT53244.2021.00038.

[23] Theofanis P. Raptis, Claudio Cicconetti, and Andrea Passarella. “Efficient topic partitioning of Apache Kafka for high-reliability real-time data streaming applications”. In: *Future Generation Computer Systems* 154 (2024), pp. 173–188. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.12.028. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23004892.

[24] Theofanis P. Raptis and Andrea Passarella. “On Efficiently Partitioning a Topic in Apache Kafka”. In: *2022 International Conference on Computer, Information and Telecommunication Systems (CITS)*. 2022, pp. 1–8. DOI: 10.1109/CITS55221.2022.9832981.

[25] Han Wu, Zhihao Shang, and Katinka Wolter. “Performance Prediction for the Apache Kafka Messaging System”. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 154–161. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00036.

[26] Feng He et al. “Algorithm for Improving Processor Utilization in Multi-core Processor Environment by Python Language”. In: *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. Vol. 4. 2021, pp. 775–779. DOI: 10.1109/IMCEC51613.2021.9481962.

[27] Dominik Straßel, Philipp Reusch, and Janis Keuper. “Python Workflows on HPC Systems”. In: *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. 2020, pp. 32–40. DOI: 10.1109/PyHPC51966.2020.00009.

[28] Samuel Jackson, Nathan Cummings, and Saiful Khan. *Streaming Technologies and Serialization Protocols: Empirical Performance Analysis*. 2024. arXiv: 2407.13494 [cs.SE]. URL: https://arxiv.org/abs/2407.13494.

[29] Ramon Invarato Menendez. *Quick Multiprocessing Queue*. https://pypi.org/project/quick-queue/. Accessed: 2024-08-09. 2021.

[30] NVIDIA Corporation. *NVIDIA Nsight Systems*. https://developer.nvidia.com/nsight-systems. Version 2023.4.1. Accessed: 2023-11-22. 2023.

[31] Criteo AI Lab. *Criteo Dataset*. https://ailab.criteo.com/ressources/. Accessed: 2024-08-09. 2024.

[32] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[33] Qijiong Liu et al. *Vector Quantization for Recommender Systems: A Review and Outlook*. 2024. arXiv: 2405.03110 [cs.IR]. URL: https://arxiv.org/abs/2405.03110.

[34] Siyi Liu et al. *Learnable Embedding Sizes for Recommender Systems*. 2021. arXiv: 2101.07577 [cs.LG]. URL: https://arxiv.org/abs/2101.07577.