SeLeP: Learning Based Semantic Prefetching for Exploratory Database Workloads

Farzaneh Zirak
University of Melbourne
Melbourne, Australia
fzirak@student.unimelb.edu.au

Farhana Choudhury
University of Melbourne
Melbourne, Australia
farhana.choudhury@unimelb.edu.au

Renata Borovica-Gajic
University of Melbourne
Melbourne, Australia
renata.borovica@unimelb.edu.au

ABSTRACT

Prefetching is a crucial technique employed in traditional databases to enhance interactivity, particularly in the context of *data exploration*. Data exploration is a query processing paradigm in which users search for insights buried in the data, often not knowing what exactly they are looking for. Data exploratory tools deal with multiple challenges such as the need for interactivity with no a priori knowledge being present to help with the system tuning. The state-of-the-art prefetchers are specifically designed for navigational workloads only, where the number of possible actions is limited. The prefetchers that work with SQL-based workloads, on the other hand, mainly rely on data logical addresses rather than the data semantics. They fail to predict complex access patterns in cases where the database size is substantial, resulting in an extensive address space, or when there is frequent co-accessing of data. In this paper, we propose SeLeP, a semantic prefetcher that makes prefetching decisions for both types of workloads, based on the encoding of the data values contained inside the accessed blocks. Following the popular path of using machine learning approaches to automatically learn the hidden patterns, we formulate the prefetching task as a time-series forecasting problem and use an encoder-decoder LSTM architecture to learn the data access pattern. Our extensive experiments, across real-life exploratory workloads, demonstrate that SeLeP improves the hit ratio up to 40% and reduces I/O time up to 45% compared to the state-of-the-art, attaining 96% hit ratio and 84% I/O reduction on average.

PVLDB Reference Format:
Farzaneh Zirak, Farhana Choudhury, and Renata Borovica-Gajic. SeLeP: Learning Based Semantic Prefetching for Exploratory Database Workloads. PVLDB, 17(8): 2064 - 2076, 2024.
doi:10.14778/3659437.3659458

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fzirak/SeLeP.git.

1 INTRODUCTION

Exploring massive amounts of data to extract (unknown) information is a query processing paradigm called *data exploration* [3, 15, 16]. The growth in data collection ability in recent decades has led to providing larger and more detailed datasets in both sciences and businesses [12]. Consequently, the popularity of data exploration has significantly increased, giving rise to the need for database systems tailored to its specific requirements, such as responding interactively and adapting to the shifts in the users' workload [12, 19].

It has been shown that during exploratory browsing, the interaction response times should be bounded within 500 ms, since additional delays drastically reduce the rate by which users make observations, draw generalizations, and generate hypotheses [26]. However, given the exponential growth in the amounts of generated data, responding to queries over such large data sets with a subsecond latency has become a tall order for the traditional database management systems (DBMS) [19]. As a result, fetching the data *prior* to the user request is one of the approaches added to the traditional DBMSs to address the issue of non-interactive performance over ever-growing data sets. Usually, an exploratory session contains multiple queries often with pauses between them as users contemplate the next query, since the result of each query affects the formulation of the next one [10, 15, 19]. After each query, while the user is interpreting the recently obtained data and preparing the subsequent request, the unoccupied system can try to predict the data likely to be requested next and fetch it into the cache (if not already there) before the user asks for it. Thus, when the data is requested later, the user perceives a lower response time.

In exploratory tools, prefetching has mainly been used in visual exploration in which the operations are limited to zooming and panning [4, 7, 10, 42, 44]. ForeCache [4] and Scout [42] utilize data characteristics in their prediction algorithm in addition to the spatio-temporal locality of the data and prove that data-driven models can achieve a better access prediction accuracy and cache hit ratio compared to the other action-based prefetchers. However, none of these techniques can work with query workloads comprising full-blown SQL queries since they only examine and pick data in the vicinity of the current accessed area to prefetch.

Recently proposed memory prefetchers [5, 6, 47] have demonstrated that machine learning techniques have great power in automatically learning patterns from a sequence of data addresses. While these prefetchers can be applied to databases, they often overlook the co-accessing of data. In other words, when an application requests n data pages simultaneously, they treat it as a sequence of length n rather than a group. Consequently, prefetching decisions depend on the order of the generated sequence, and the system does not consider the dependencies between all accessed addresses. Furthermore, in our experiments we observe that these models mainly focus on predicting a single data access and do not efficiently handle multiple prefetch decisions in an interactive pace. Additionally, most of these prefetchers lack the utilization of data

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659458

semantics and instead rely on logical block addresses (LBA) or other environmental parameters to learn patterns and make predictions.

Recurrent neural networks (RNNs), and more specifically, Long Short-Term Memory (LSTM) models, are the most popular learning methods used in memory prefetchers [5, 6, 47]. Taking inspiration from the learning-based and the data-driven prefetchers, we propose SeLeP – a semantic-based prefetcher designed for both visual and SQL-driven exploratory workloads. In our approach, we treat the prefetching problem as a time series forecasting problem, where each element in the time series represents a group access to the data. We employ an encoder-decoder LSTM architecture to tackle this problem. However, unlike past efforts that use logical addresses of the accessed data [6, 47], SeLeP learns and predicts the sequence of *semantic-based encodings* of the data values.

SeLeP obtains a vector representation for each physical block by employing deep learning feature extraction methods on its data values. These vector values capture data semantics and serve as the key components of SeLeP, analogous to logical block addresses in memory prefetchers. As demonstrated in [4] and [42], considering data semantics rather than solely addresses can be beneficial since blocks are typically accessed based on the values they contain. For instance, when querying a dataset, the retrieved blocks contain certain values, leading to their co-accessing.

Due to the challenges involved with access prediction among a massive number of blocks, SeLeP groups the blocks that are frequently accessed together into partitions and prefetches data in units of partitions instead of blocks. Additionally, by using these partitions, SeLeP can more effectively prefetch data that are related to each other and are likely to be accessed together, which can enhance the prefetching strategy and improve overall performance.

For each partition, SeLeP creates a matrix representation using the vector encodings of their assigned blocks. It also uses the matrix encodings to represent a query and employs an LSTM model on the sequence of query encodings to predict the subsequent partition accesses. With the query encodings, SeLeP can consider a group of accessed blocks and does not need to define an ordered sequence on the blocks accessed by a single query.

The output of the LSTM is the probability of each partition being accessed subsequently. Using this prediction model, SeLeP selects and brings the partitions most likely to be accessed into the cache. The contribution of this paper are summarized as follows:

- To the best of our knowledge, we propose the first prefetcher that leverages data semantics by encoding data blocks and makes predictions based solely on the encoding sequences.
- We propose a prefetcher that unlike state-of-the-art can cater for both visual and SQL-based exploratory workloads.
- We formalize the prefetching problem as a times series forecasting problem and demonstrate that the prediction models with semantics inputs can outperform the models that utilize logical data addresses.
- We conduct an extensive experimental evaluation across both types of workloads gathered from real-world scientific datasets, as well as a publicly available industrial benchmark, and show that SeLeP outperforms the state-of-the-art prefetchers by up to 40% in hit ratio, achieving 95% hit ratio on average across all experiments.

2 PROBLEM FORMULATION

The core of the semantic prefetching problem is building a prediction model that captures the inter-dependencies of the previously requested data and anticipates future accessing data to fetch it into the cache prior to the request of an upcoming query. We now formulate semantic prefetching as a time-series forecasting problem.

Let set B represents all blocks (i.e., pages of the data) present in the database, $b_n^{tb_i} \in B$ symbolizes n$^{\text{th}}$ block of table tb_i in the database. Consider a sequence of queries $\langle q_i \rangle_{i=1}^{n} = \langle q_1, q_2, \ldots, q_n \rangle$ executed within a session, with each query execution time considered as a distinct timestep. The set $res_{q_i}^{B} \subseteq B$ represents the blocks accessed by query q_i.

Given the sequence of $\langle res_{q_i}^{B} \rangle_{i=1}^{n}$, the non-semantic prefetching problem involves predicting and fetching the $res_{q_{n+1}}^{B}$ regardless of the data values and considering the LBAs and other environmental parameters. However, our objective is to determine the inter-dependencies among query result sets of the n most recent executed queries, and anticipate the data that will be accessed next. Therefore, we define the semantic prefetching as follows:

Definition 2.1 (Semantic Prefetching). Having a sequence of query result blocks corresponding to the n most recently executed queries $\langle res_{q_i}^{B} \rangle_{i=1}^{n}$, capture the connection between their data values to predict and fetch the upcoming block access request $res_{q_{n+1}}^{B}$.

3 SeLeP: LEARNING-BASED SEMANTIC PREFETCHING FRAMEWORK

In this section, we provide an overview of the proposed learning-based semantic prefetcher, followed by an in-depth explanation of its key components. We start by presenting an overview in §3.1. Then, we explore the details of the block encoding process in §3.2, the partitioning process in §3.3, and the semantic learning and prediction model in §3.4, making complete picture of SeLeP operation.

3.1 SeLeP Overview

SeLeP is a learning-based prediction framework that extracts the semantic relationships within the previously requested data and outputs the future data accesses. Due to the small size of blocks (8kB to 32kB) and extensive database sizes, a substantial volume of blocks is present in the system. The numerous number of blocks leads to a considerable number of potential outcomes for the prediction model, which can reduce its accuracy. Hence, we adopt a dynamic clustering approach to group related blocks into more sizable sets, called partitions, and use them instead of blocks in the prefetching process. The impact of partitioning approach is assessed in §4.2.3.

Let P^t denote the set of partitions generated on the blocks at timestep t, where p_i^t and $bp_i^t = \{b | b \in B \wedge b \in p_i^t\}$ represent an individual partition and its set of assigned blocks respectively. To simplify the presentation and since the partitions do not change frequently, we omit the timestep t indices in the reminder of this paper[1]. Using the partitions, we define the set $res_{q_i}^{P} \subseteq P$ as the partitions accessed by query q_i.

To grasp the data semantics, we employ feature extraction methods to encode the data, and express each query by aggregating the

[1]We however do test the impact of partition changes in §4.2.2.

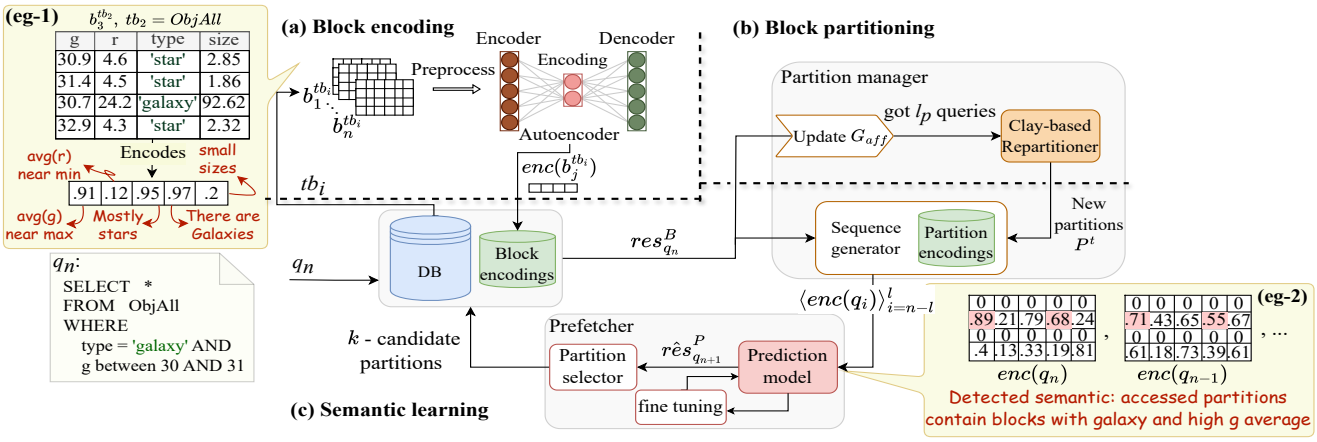

Figure 1: System architecture of SeLeP representing: (a) the block encoding component generating block vector encodings using autoencoders, (eg-1) a simplified example for block encoding, (b) the block partitioning component clustering the blocks, (c) the semantic learning component predicting the partition access pattern based on semantic relationship of query encodings, as illustrated in (eg-2). After initial training, (b) and (c) continuously perform prefetching.

encoding of its result set, represented as $enc(q_i)$. By harnessing these query encodings, the partition level semantic prefetching problem transfers to a time series forecasting problem with query encoding sequence $\langle enc(q_i)\rangle_{i=1}^{n}$ as input and $res_{q_{n+1}}^{P}$ as the output.

The overall framework of our system is depicted in Figure 1. It comprises three distinct components: block encoding as a preprocessing step, and continuous block partitioning and semantic learning. Before the system can utilize these components, a data preparation step is required. During the data preparation step, we execute and collect result of a set of queries q selected for training. These result blocks are used to create the training workloads for warming up the system and training the model.

Within the block encoding module, Figure 1(a), an Autoencoder [36] is employed on each table to generate vector representations for its blocks. Prior to utilizing the Autoencoder for block encodings, the data is preprocessed based on properties of its corresponding table such as the number of columns and range of values (with more details provided in §3.2). This process is executed once, and the block encodings are stored for future reuse. Figure 1(eg-1) depicts a simplified block encoding example, showing that the encoded vector provides a concise representation of the block's characteristics.

The block partitioning component, depicted in Figure 1(b), groups blocks frequently co-accessed by the workload into the same partition. In other words, this component clusters the blocks, and those co-accessed by many queries are more likely to be assigned to the same partition. We extend the Clay partitioning algorithm [37] (details provided in §3.3), which employs an affinity graph to represent the co-access frequency between pairs of blocks. During the initial phase, we apply this algorithm on the queries selected for training the prediction model to create and warm up the partitions. Later, after the complete system is prepared for prefetching, we leverage this component to update the affinity graph and the partitions. We undertake iterative repartitioning following the reception of a certain number of queries to ensure accurate capture of the co-access frequency inherent in the most up-to-date workload.

Once the blocks are encoded, and the partitions are established, we have the required data to train our prediction model in the semantic learning component (Figure 1(c)). To prepare the train and test data, we use the res_q^B generated in data preparation step and P formed in block partitioning to obtain res_q^P and calculate $enc(q)$ for q in the training workloads. We consider each l (named lookback) consecutive query and generate $\langle enc(q_i)\rangle_{i=n-l}^{n}$ as the model input. For the output, we employ a bitmap string where each bit indicates the presence or absence of a partition in $res_{q_{n+1}}^{P}$.

After the model is trained, we can use the system to make prefetch decision. In this stage, when a new query is executed, the system determines its list of accessed partitions, updates the affinity graph, and generates $\langle enc(q_i)\rangle_{i=n-l}^{n}$ using the previous queries to input the model and extract the semantic relation between the queries. Figure 1(eg-2) illustrates a simplified example of semantic relationship between requests. The model output can be interpreted as the probability of accessing each partition in the subsequent request. Consequently, the system identifies the top-k values, which correspond to the partitions most likely to be requested next, and retrieves the blocks associated with these partitions into the cache.

The last two components are employed continuously: the block partitioning component updates the affinity graph based on the received queries. Once a certain number of queries (l_p) are received, i.e., the repartitioning threshold is reached, the partitioning algorithm is applied to the latest version of the affinity graph. During repartitioning, blocks are shuffled between partitions, and the partition encodings are modified. Hence, the accuracy of the prediction model may reduce after repartitioning. To address this, after each repartitioning, we proceed to fine-tune the prediction model. This process involves using the new partition encodings and the set of queries executed since the last repartitioning.

Table 1 lists the frequently used notations in this paper.

3.2 Block Encoding

The purpose of semantic prefetching is to incorporate the actual data values, rather than just block addresses, in access pattern recognition. Since each data block can contain hundreds of values, it is challenging to identify relationships among such vast amounts

Table 1: Frequently Used Notations

Symbol	Definition
$b_n^{tb_i}$	n^{th} block in table tb_i
bp_i	Blocks allocated to partition p_i
res_q^B, res_q^P	Set of blocks and partitions accessed by query q
$enc(b)$, $enc(p)$	Block vector encoding, partition matrix encoding
l_{be}	Length of block encodings
l	Lookback. Sequence length considered by the model for prediction
l_p	Repartitioning threshold
k	Prefetch size in unit of partitions

of data in a series of blocks. Therefore, we need to have a concise representation for each block which captures the essential and distinctive characteristics of its data. To this end, we consider each block of data as a big matrix and encode each matrix to a vector with a block encoding length, denoted as $enc(b_n^{tb_i})$ and l_{be} respectively.

As we lack prior knowledge about the most crucial parts of the data, we address the block encoding problem as an unsupervised feature extraction task, employing an autoencoder model to extract information from the data. Autoencoders [36] are unsupervised neural network architectures specifically designed for feature extraction and representation learning. These models, used for compressing and encoding various data types including images [46], excel at handling large matrices, making them well-suited for our encoding problem. During the block encoding process, we start by preprocessing the data of each table, followed by training an autoencoder to learn the encoding of its blocks.

3.2.1 Data Preprocessing. A key challenge in encoding blocks into fixed-size vectors when using autoencoders is to convert all data types into numeric values usable by the model. To address this, we convert date and time values to Unix timestamp and other non-numeric types to text, which later are encoded to numbers using Word2vec models [28]. We opt for training a Word2vec model for each table, since not all texts, including those originally in character type, are meaningful. For instance, 'B&D_NaD' is a star name in the Sloan Digital Sky Survey (SDSS) [2] DR7 dataset. Hence, each string is treated as a sentence and fed into the Word2vec model, resulting in each column being converted into 8 new numeric columns.

Next challenge is encoding tables with a wide structure (i.e. with hundreds of columns), while typically many columns carry insignificant or repetitive values. For instance, within the SDSS dataset, several tables exceed 500 columns, some filled with default or null values, or displaying variations of one attribute observed at different times. The presence of many such columns can increase the loss value of the encoder and result in a less representative encoding. In addition, columns with vast range of numerical values can hinder the encoding process by causing scaling challenges in the activation functions and loss computation parts [14].

To address these issues, we apply min-max normalization (Equation (1)) on each column individually, ensuring that all values are within the range of -1 to 1. Subsequently, we employ principal component analysis (PCA) [32] to reduce table sizes and filter their columns before feeding them into the autoencoder. PCA identifies a meaningful basis to transform the dataset into a lower dimension by computing new orthogonal axes called principal components.

Table 2: Average block encoding time, hit ratio at $k = 40$ and ROC AUC score of the system with different block encoders on the SDSS dataset. The best and second-best performance are highlighted in bold and underlined, respectively.

Test		SLP	CNN	MLP(SeLeP)	LSH	AggPCA
s-reg	HR	93.97	<u>94.52</u>	**95.19**	85.90	90.44
	ROC AUC	0.854	<u>0.862</u>	**0.891**	0.835	0.855
s-rand	HR	<u>96.98</u>	96.51	**97.94**	91.06	96.67
	ROC AUC	0.851	0.840	**0.881**	0.841	<u>0.855</u>
m-reg	HR	**99.90**	**99.90**	99.81	**99.90**	<u>99.86</u>
	ROC AUC	0.988	**0.992**	<u>0.991</u>	0.990	<u>0.991</u>
m-rand	HR	<u>99.76</u>	**99.88**	<u>99.76</u>	<u>99.76</u>	**99.88**
	ROC AUC	0.985	0.986	<u>0.987</u>	**0.989**	0.986
mj-reg	HR	<u>96.20</u>	96.02	**96.98**	92.07	89.12
	ROC AUC	0.834	<u>0.839</u>	**0.884**	0.782	0.829
mj-rand	HR	93.09	<u>95.00</u>	**95.67**	91.09	89.57
	ROC AUC	0.859	<u>0.882</u>	**0.893**	0.821	0.840
f-SDSS	HR	78.55	<u>85.95</u>	**86.17**	74.55	81.19
	ROC AUC	0.680	<u>0.759</u>	**0.785**	0.655	0.732
Encoding Time		<u>1868.35</u>	4357.3	2898.17	2983.12	**570.51**

With these axes, we can map the original data to a lower dimension while preserving its underlying pattern.

$$x_{normalized} = \frac{x - min(X)}{max(X) - min(X)} \times 2 - 1 \tag{1}$$

3.2.2 Autoencoders. Autoencoders consist of two main components: the encoder that encodes the data into a compressed size which is called latent space or the encoding, and the decoder that takes the produced encodings and aims to regenerate the original data. By using autoencoders for block encoding two questions arise: *(i) How well does autoencoder perform compared to heuristic-based dimensionality reduction approaches?* and *(ii) Which layer architecture performs better in the autoencoder?*

To answer these questions, we conduct experiments with various prefetch sizes (k) using several block encoders on the SDSS dataset (workloads are explained in §4.1.3). We design three variants of autoencoders: one with a single dense (a.k.a fully connected) layer (SLP), another with two dense layers (MLP), and one with two convolutional neural network layers (CNN) [24], and two heuristic-based alternatives: one utilizing PCA for dimension reduction and row-wise feature aggregation (AggPCA), and the other employing Local Sensitivity Hashing (LSH) [11] for data compression.

Table 2 presents the hit ratio (HR) for each system at $k = 40$. Additionally, the Receiver Operating Characteristic Area Under the Curve (ROC AUC) score is provided to demonstrate each encoder's overall performance across all k values. A higher score indicates a system's capability to achieve a greater hit ratio within the tested k range. We observed that the performance of autoencoders is heavily influenced by the characteristics of the block data.

While heuristic-based methods can effectively compress data with simple patterns, they often fail to extract their semantics which negatively affects the performance in more complex workloads (mj-rand, mj-reg, and f-SDSS). Similarly, SLP struggles with high-dimensional or complex data, but excels at encoding tables with few columns, achieving high performance on workloads accessing these tables (e.g. s-rand). Although CNN and MLP can both handle complex patterns and achieve a high hit ratio, we choose MLP since it offers a simpler model and faster encoding (presented in Table 2).

As depicted in Figure 1(a), an autoencoder is created for each table tb_i, trained with the table's blocks. Subsequently, the trained encoder converts blocks into vectors, represented as $enc(b_n^{tb_i})$, which will be stored in the system for later use.

3.3 Block Partitioning

As mentioned earlier, due to the wide table schema and large numerical values, a single data record could be as large as a few kilobytes. Therefore, prefetching a single 8kB block of data results in caching very few data records. Additionally, the large size of data objects and the extensive size of the database result in a tremendous number of blocks present in the system. This considerable size causes challenges in accurately predicting the subsequent data accesses.

To address these issues, we cluster blocks into partitions, which are more sizeable sets (in particular 128 blocks in our experiments), and use the partition number to refer to a data access. Figure 8(d) depicts the effect of partition size on prefetching performance. We form the initial partitions by grouping the consecutive blocks of the tables. Subsequently, the system periodically performs repartitioning to cluster the blocks that are commonly accessed together in queries. The repartitioning process is done based on the most recently received workload represented by a graph data structure.

3.3.1 Construction of Affinity Graph. To group the frequently co-accessed blocks, we construct a block access graph called affinity graph G_{aff}, with nodes representing the blocks, edges illustrating co-accessing relation and the weight of the edge reflecting the frequency of the corresponding nodes being co-accessed. To create this graph, a node is added for a block only if it is accessed at least once. In data exploration, despite the substantial dataset sizes, only small portions of the data are of interest to the users [3, 15, 16, 19]. In other words, there may be many blocks that never get to be requested or accessed. Thus, though the datasets contain a large number of blocks, the size of the graph can remain relatively small.

The system observes the queries and modifies G_{aff} for a batch of queries with length l_p. For each query q in the batch, it finds res_q^B and passes this list to the partition manager. For every accessed block, the system creates a node in the graph (if it does not already exist) and includes an edge connecting the nodes corresponding to each pair of blocks (if the edge does not already exist). The weight of the edges will increase by $1/l_p$ to demonstrate how frequently a pair has been co-accessed during the current batch. Figure 2 provides an example of the affinity graph modification process with a very small value for $l_p = 10$. Once a complete batch of l_p queries is received, the system triggers a repartitioning process to update the partitions based on the most recent co-accesses represented by G_{aff}.

3.3.2 Clay-based Partitioning. Using G_{aff}, the partitioning problem can be transformed into a graph partitioning problem. Among the methods proposed for graph partitioning, Clay [37] stands out for its application in load balancing distributed databases. It monitors the data accesses and generates a heat map similar to our affinity graph. Its repartitioning module is triggered when a server's load reaches the maximum load threshold θ. At the time of repartitioning, Clay selects a group of data records causing the overload containing the most accessed record and its relative neighbors. It then determines the best server to migrate the chosen records,

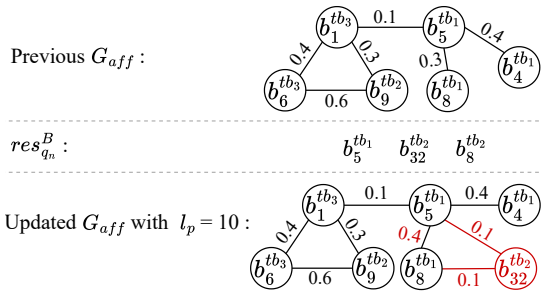

Figure 2: An example of affinity graph modification upon receiving a new query which accesses block $b_{32}^{tb_2}$ for the first time (l_p is set to a very small value to simplify the example)

ensuring the destination server isn't overloaded. If no suitable destination is found, it adds a new empty server to the system.

We have adopted the Clay technique since it focuses on minimizing distributed transactions between partitions, which is different from most graph partitioning techniques that prioritize balancing partition accesses. Also, it is an online partitioning approach that migrates data in dynamically generated groups rather than static sets of blocks. To tailor Clay to our problem, we have performed the following modifications:

- Clay considers the rate of access to individual blocks as well as their co-accessing rate to calculate the load of each partition. However, in our case, we only focus on the co-accessing load and disregard the number of times each individual block has been accessed. Therefore, the load for each partition can be calculated by:

$$load_{p_i} = \sum_{v \in p_i \ \wedge \ u \notin p_i} w(edge(v,u)) \ . \ k_w \qquad (2)$$

  where u and v are two nodes in G_{aff} linked by an edge with weight w, and k_w is a constant weight indicating how much inter-partition accesses are tolerable in the system.
- If a proper destination for migrating the overloading nodes cannot be found, Clay adds a new empty server to the system and reruns the algorithm. However, in our case, the number of partitions is fixed, and new partitions cannot be added to the system. To address this restriction, we made multiple modifications. During the initial partitioning, we ensure that no more than 90% of the partition size is filled. Additionally, after assigning blocks to the partitions, we integrate an additional 5% of the total number of partitions ($|P|$) as empty partitions into the system[2]. Furthermore, we make θ dynamic and increase it when we need to add a new partition to the system. It is important to note that maintaining a fixed θ in the original Clay system can also lead to issues, as a group of data with a size exceeding the maximum server capacity may frequently be accessed.
- Clay discards the graph after each repartitioning, but we take a different method to retain workload history. Instead of discarding the graph entirely, we scale down the weights by a factor less than 1, preserving historical access information with reduced impact on future partition formations.

[2]We measure the impact of each configuration parameter in §4.2.3.

3.3.3 Partition Encoding. Once a new partition set P is formed, the system calculates the partition encodings, labelled as $enc(p)$, by aggregating the encodings of their assigned blocks. However, it is inadvisable to aggregate encoding of blocks that belong to different tables. This is because individual fields within the vector encoding of blocks from different tables have distinct semantic interpretations. For instance, in Figure 1(eg-1), the 5th element of block encodings in tb_2 indicates the size of objects, which can be entirely different from the 5th element of encodings in other tables. Hence, we cannot aggregate $enc(b_n^{tb_i})$ and $enc(b_m^{tb_j})$ if $i \neq j$.

To mitigate this issue, we encode each partition as a matrix rather than a vector. This matrix is structured with rows corresponding to individual tables within the database and l_{be} columns. During the data preparation step, we enumerate and assign an integer to each table, indicating their corresponding row in matrix encodings. Consequently, within a given partition, the vector encodings of blocks belonging to each table can be aggregated separately and stored in distinct rows of the matrix corresponding to that table.

The partition encoding process is outlined in Algorithm 1. First, the partition encoding is initiated with an all zero matrix with number of rows equal to the count of tables within the dataset, and l_{be} columns. Subsequently, blocks within each partition are grouped according to their respective tables. For each table tb_j with blocks assigned to the partition, the algorithm calculates an aggregated encoding by averaging the individual block encodings. This aggregated encoding is then stored in the j^{th} row of the matrix.

3.4 Semantic Learning

In this section, we describe the LSTM model used for solving the semantic prefetching problem and compare it against some alternative models. LSTM is one of the models within the family of RNNs specifically designed to learn short-term and long-term dependencies among a sequence of data [13]. Due to its capability to capture temporal patterns, it can be an excellent choice for addressing time-series forecasting problems. An RNN-based alternative to LSTM is the Gated Recurrent Unit (GRU) [9], which is known for its simplicity and fewer trainable parameters compared to LSTM. The question that arise is: *Which model and what architecture perform better as the prediction model of our semantic prefetcher?*

We evaluate various prediction models, composed of different learning models, such as a GRU model, an LSTM model, a two-layer LSTM (ML-LSTM) model, a three-layer MLP model, and an encoder-decoder LSTM (ED-LSTM) model, on the SDSS dataset. The results are shown in Table 3. We observe that the MLP model struggles to effectively capture the query semantics and partition access pattern and performs poorly compared to the other RNN-based models.

In addition, simple models like LSTM and GRU, have restricted model capacity and limited number of learnable parameters, causing difficulties in effectively capturing complex relationships among features [27]. Thus, these models are not well-suited for a multi-variable problem with complex inter-relationships like semantic prefetching. To tackle these defects, the LSTM should be used in a multi-layer structure such as encoder-decoder architecture.

The encoder-decoder LSTM (ED-LSTM) architecture has a superior ability to efficiently grasp the context and the dependencies between variables, making it one of the most popular architectures

Algorithm 1 The Partition Encoding Calculation Algorithm

Input: Partitions P, Block Encodings $\{enc(b)\}$, Count of Tables n_{tb}, Block Encoding Length l_{be}

```
for p_i in P do
    enc(p_i) ← zeros(n_tb, l_be)
    Separate blocks of each table within bp_i
    for tb_j involved in bp_i do
        bp_i^{tb_j} ← blocks of table tb_j in bp_i
        aggregated_enc ← mean(enc(b) for b in bp_i^{tb_j})
        enc(p_i)[j] ← aggregated_enc
    end for
end for
```

Table 3: Best hit ratio of the system with different prediction models on the SDSS dataset.

Test	MLP	GRU	LSTM	ML-LSTM	ED-LSTM(SeLeP)
s-reg	93.61	93.69	89.32	94.00	**95.19**
s-rand	95.87	96.83	97.87	96.35	**97.94**
m-reg	**99.90**	**99.90**	**99.90**	99.86	99.81
m-rand	99.76	**99.88**	99.64	99.76	99.76
mj-reg	89.23	91.08	91.44	90.15	**96.98**
mj-rand	87.35	92.41	91.63	92.44	**94.17**
sdss1	88.96	88.35	87.74	90.36	**91.24**
sdss2	84.11	80.01	81.24	84.76	**86.17**

in natural language processing [41]. Its encoder maps the features into the latent states and inherently captures a hierarchical and high-level representation of the input sequence. Subsequently, the decoder, initialized with the encoder state, receives latent states values as the input and generates the output sequence. We observe that compared to ML-LSTM, ED-LSTM, with its hierarchical representation and state initialization, can be more effective in capturing and leveraging the temporal dependencies in the input sequence.

Figure 3 depicts our prediction model with ED-LSTM architecture. SeLeP generates $enc(q)$ using the *set* of partitions accessed by the queries, which remains invariant to the query plan and access order. The model takes $\langle enc(q_i)\rangle_{i=n-l}^{n}$ as input and compresses them using a time-distributed layer, resulting in a sequence of 128-dimensional vectors. These vectors are then fed into the ED-LSTM, designed with single-layer LSTMs. The decoder's output further passes through two dense layers, each containing $|P|$ units. The final dense layer generates $r\hat{e}s_{q_{n+1}}$, a $|P|$-dimensional bitmap vector, representing the probabilities of each partition getting accessed in the subsequent query. By leveraging these probabilities, we can select top k partitions most likely to be requested next.

With this model, we transform the prefetching problem into a multi-label classification task. Each neuron in the final dense layer corresponds to a specific partition (a.k.a. label), and a sigmoid activation function is applied to each neuron to ensure its output is a probability within the range of 0 to 1. Subsequently, we apply the binary cross-entropy loss function independently to each label and then aggregate them to calculate the total model loss. Considering $y_i \in \{0, 1\}$ and $\hat{y}_i$ as the true and predicted probability of partition p_i being in the $res^P_{q_{n+1}}$, the loss function can be defined as follows:

$$\mathcal{L} = -\sum_i [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \tag{3}$$

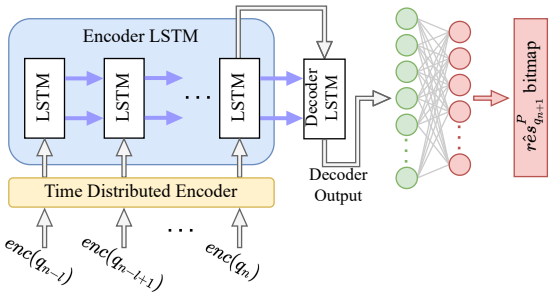

Figure 3: The prediction model encoder-decoder architecture

To fine-tune the model after a repartitioning procedure, we start by freezing all layers up until the first dense layer. Subsequently, we proceed to fine-tune the system on the selected workloads for 15 epochs using a learning rate of 10^{-5}.

4 EVALUATION

We evaluate SeLeP against the state-of-the-art (SOTA) prefetchers across both SQL-based and navigational workloads gathered from SDSS DR7 [2] and SQLShare [17] real-world datasets. For completeness, we extend our evaluation to non-exploratory workloads by testing it on the TPC-DS benchmark. In this section, we first explain the experimental setup, then show the performance of SeLeP against the SOTA prefetchers, and conclude the section with the sensitivity analysis of the parameters of interest.

4.1 Experimental Setup

4.1.1 Implementation and Configurations. SeLeP and the clay-based partitioning are implemented in python and the model is developed using TensorFlow/Keras framework [1]. We configure each LSTM layer with 64 cells and conduct training using batches of size 32. Training continues until either early stopping based on the loss of validation data (10% of the train data) or reaching the maximum number of training epochs, set at 75. For the prediction model, we employ Cross Entropy as the loss function, and for the autoencoder model, Mean Square Error is used. In both models, we utilize the Adam optimizer [21] for the training process.

In our experimental setup, we incorporate SeLeP as a middleware layer on top of the PostgreSQL [40] server. In practical terms, SeLeP makes the prefetch decision and fetches the chosen blocks using the pg_prewarm module within the PostgreSQL server.

Unless mentioned otherwise, the configuration in the experiments are as follows: cache size = 4GB, partition size = 128 block, block size = 32kB, model input sequence length $l = 4$, $k_w = 10$, initial max partition load $\theta = 1$, repartitioning threshold $l_p = 2500$.

4.1.2 Datasets. We use the seventh Data Release of Sloan Digital Sky Survey (SDSS DR7) [2] dataset, which is an open-access database containing digital astronomy data, accessible via various tools including navigation and SQL search tools. Utilizing its publicly available workloads[3], we select two weeks of system logs (86 thousand entries) from WebLogs table for **navigational SDSS workloads**, and a subset of logs spanning three months (200 thousand queries) from SqlLogs table, for **SQL-based SDSS workloads**.

[3]http://cas.sdss.org/runs/en/tools/search/sql.asp

The original DR7 database is hosted on SQL Server and has 20TB of data, comprising 95 tables and 51 views. For our experiments, we first migrate MyBestDR7[4], a 2GB subset of DR7 to the PostgreSQL and extend its size by retrieving more data from the DR7 website, using SciScript[5] python library. The experiments are performed on a 16GB dataset, comprising the same number of components.

Our navigational SDSS database is a 4GB subset of our BestDR7 database, containing tables with attributes that form the data available in the SDSS navigation tool. The data is organized into four levels of granularity, each stored in a separate table.

We also test SeLeP on two additional exploratory datasets gathered by the SQLShare [17] project: **Birds**, containing 8GB of data and 6 tables including textual values in different languages, and **Genomics**, which contains 10GB of data spread across 13 tables.

Lastly, the **TPC-DS** database used in our evaluation is generated with a scale factor of 10. We exclude the query templates that take more than ten minutes to complete and generate our test and train queries using the remaining 93 templates.

4.1.3 Workloads.

Exploratory SQL-based Workloads. We classify the query workloads into three broad categories and within each category, considered two types: one with a recognizable pattern in the accessed data *(regular)*, and the other without such patterns *(random)*. The recognition of patterns is based on the sequence of accessed tables and the sequence of differences in the accessed block addresses (LBA-delta). In addition to these six categories, all of which consist of single-user sessions, our model is also tested against a full workload, encompassing queries from all categories executed by one or more users. The resulting types are as follows:

- **Single-user, single table access**: All queries in the *single-regular (s-reg)* and *single-random (s-rand)* workloads access just one particular table. Usually, workloads in *s-reg* category are sequential table accesses.
- **Single-user, multiple table access without join**: Queries in the workload accesses a single table, while multiple tables are accessed in total across the entire workload. Workload is *multiple-regular (m-reg)* if a recurrent pattern exist in the accessed LBA-delta, else *multiple-random (m-rand)*.
- **Single-user, multiple table access with join**: Queries in this category of workloads mostly contain up to 4 join operations. If there is a pattern in the LBA-delta accessed by the queries, the workload is in type *multiple-join-regular (mj-reg)*, and *multiple-join-random (mj-rand)* otherwise.
- **Multi-user, full workload**: A *full* session of a dataset (*f-<dataset>*) with all types of queries, executed by one or more users . This mostly mimics real-life exploratory workloads.

Exploratory Navigational Workloads. We use three types of exploration based on the gaps between consecutive viewed area [4, 10].

- **Smooth exploration**: The user observes adjacent areas by panning to the neighboring region or zooming in and out. There is no significant gap between consecutive locations.
- **Jumping navigation**: This type involves *Jumps* in the actions, meaning that the user requests an area that is not

[4]http://www.skyserver.org/myskyserver/
[5]https://github.com/sciserver/SciScript-Python

in the vicinity of the current viewport. In this type, the user observes some adjacent area and then jumps into another location to do the same.
- **Random navigation**: The user randomly probes data space without adhering to specific patterns in spots coordinates.

Non-exploratory SQL-based Workloads over TPC-DS. We generate a test workload, using 93 templates of this benchmark which 25% templates are not used in generating training workloads.

4.1.4 Baselines. We compare our proposed method on query workloads against traditional prefetching methods implemented in most mainstream DBMSs, and SOTA learning-based data prefetchers.

- **Lookahead** [39]: One block lookahead algorithm is the simplest prefetcher which is used in several DBMSs. When an address is accessed, it simply prefetches the next one.
- **Random Readahead (Rand-Readahead)** [30]: Stores a window trace of recent demanded addresses. If within a window, a predefined number (l_{RR}) of blocks of an extent[6] get accessed, the prefetcher retrieves the entire extent.
- **Naïve prefetcher**: It calculates the LBA-delta and selects its prefetching candidate blocks by recursively adding the most frequent LBA-delta to the last accessed address.
- **SGDP** [47]: This SOTA prefetcher represents interactive relations among LBA-delta streams using a weighted directed graph structure and learns the pattern of LBA-deltas using a gated graph neural network.

In SeLeP, we prefetch k partitions with a maximum size of *MaxParSize*. Thus, we extend Lookahead and Naïve models to prefetch $k\times$ *MaxParSize* blocks instead of one. In the Rand-Readahead model, we configured extents to be as large as twice *MaxParSize* and l_{RR} to 13 which is its default value in MySQL Server. In SGDP, after each query, we recursively predict the next LBA-delta for $k\times$ *MaxParSize* times to get the proper number of candidate partitions.

On navigational workloads, we compare our method against a SOTA prefetcher, **ForeCache** [4], which is a hybrid prefetcher working with both the history of the movements and the data semantic similarities of the adjacent area.

4.1.5 Metrics. As performance metrics, we employ hit ratio, presented in Equation (4) and prefetch coverage presented in Equation (5). The prefetch coverage is determined by subtracting the number of cache misses in a system without a prefetcher (NP) from the number of cache misses in a system with a prefetcher. This metric quantifies the percentage of cache misses eliminated after implementing a prefetcher in the system (the higher the better).

$$Hit\ Ratio = \frac{Hits}{Hits + Misses} \tag{4}$$

$$Coverage = \frac{Misses_{NP} - Misses}{Misses_{NP}} \tag{5}$$

Additionally, we measure the I/O time to illustrate the impact of each model on reducing query response time. Query execution time can be divided into I/O time and computation time, with prefetching techniques only affecting the former and having no impact on the computational overhead of the queries. Hence, prefetchers can improve the query response time to a limit, which is when it gets 100% hit ratio. To better demonstrate the I/O time improvement, we report the relative I/O time calculated as follows:

$$Relative\ t\_io_{pr} = \frac{t\_io_{pr}}{t\_io_{NP}} \tag{6}$$

where t_io_{pr} is the I/O time of the workload when employing pr prefetcher, and t_io_{NP} stands for the I/O time of the NP system.

4.1.6 Hardware. All experiments are conducted on a server running Ubuntu 18, equipped with 48 Core at 2.4GHz, 1.1TB RAM, 50TB disk (10K RPM), and one NVIDIA V100 GPU with 16GB memory.

4.2 Results

4.2.1 SeLeP versus the SOTA. We test SeLeP on different workloads and compare its performance against the baselines.

SQL-based workloads. In this subsection, we report the results of the tests conducted on nine SQL-based workloads (§4.1.3) to evaluate both SeLeP and the baselines (§4.1.4). Figure 4(a) shows the hit ratio of SeLeP and other baselines at various k, informing our choice of $k = 42$. In Table 4 and Figure 4(b), we present the miss coverage and I/O time results for this selected k, respectively.

In the simplest workload, *s-reg*, SGDP initially achieves the highest hit ratio near 80%. However, on more complex yet regular workloads, SeLeP outperformed SGDP by a significant margin. SGDP could perform better in smaller k due to its individual block selection, where subsequent requested blocks, even in different partitions, can be found among the first few *MaxParSize* prefetched blocks. However, for larger k values, SeLeP proves more effective in bringing more accurate data to the cache and SGDP is incapable of predicting diverse block addresses and has a constant hit ratio.

SGDP halts prefetching upon detecting uncertainty in the output predictions, which typically occurs when it encounters irregular LBA-deltas. Irregular LBA-deltas refer to values other than the top 1000 frequently occurring LBA-delta in the training workload. The presence of irregular LBA-delta is a common phenomenon in data exploration with large datasets and random data access. Furthermore, SGDP frequently predicts zero as the next LBA-delta multiple times in a row. This leads to numerous duplicates LBAs among the selected prefetch candidates, which is the primary reason why SGDP achieves near zero miss coverage in multiple tests.

Conversely, Naïve prefetcher, which has a similar logic to select prefetch candidates, continues to generate and prefetch block addresses without considering memory consumption or any uncertainty. Consequently, in larger k, its hit ratio decreases and it attains negative miss coverage in *m-reg* and *mj-rand* tests. This occurs since the previously accessed blocks that are still needed are evicted from the cache to make space for the prefetched blocks.

In workloads with *random* access pattern, which is more common in data exploration, SeLeP always attains the best hit ratio, while SGDP and most of the traditional prefetchers could not bring any particular advantages to the system. Rand-Readahead has a high hit ratio in *s-rand* since the data is randomly accessed from a single table, making it an ideal scenario for this model which does not rely on patterns to trigger prefetching. Nevertheless, in *m-rand* and *mj-rand*, where each query accesses data randomly from multiple tables, there are not enough common extent blocks in the window trace so the model triggers the prefetching process.

[6] https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos

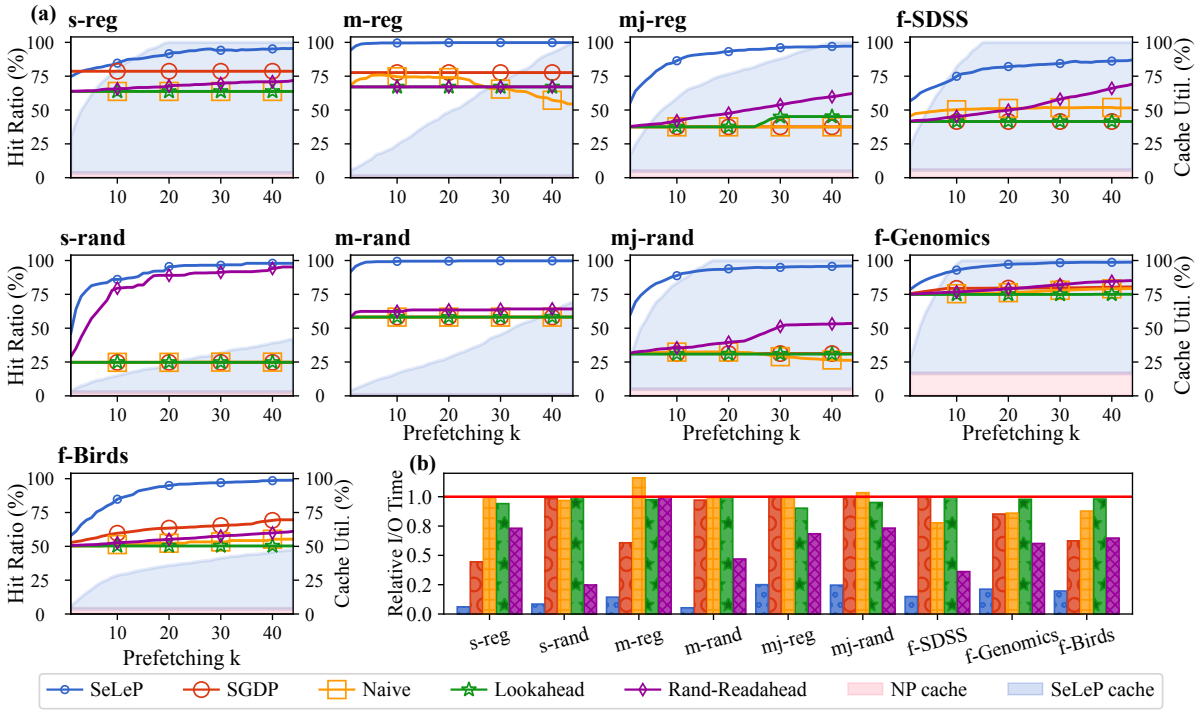

Figure 4: (a) Hit ratio in SQL-based exploratory workloads. The shaded area shows the portion of cache utilized by the workload (pink) and the additional portion consumed by SeLeP prefetcher (light blue). (b) Relative I/O time with k=42 for same workloads.

Table 4: Miss Coverage in SQL-based Workload with $k = 42$.

Test	Naïve	Lookahead	Rand-Readahead	SGDP	SeLeP
s-reg	0.0	1.25	31.97	49.31	**79.28**
s-rand	1.01	0.0	93.67	0.71	**97.26**
m-reg	-33.62	0.54	0.14	32.16	**99.42**
m-rand	0.0	0.0	14.41	1.28	**99.42**
mj-reg	0.0	12.17	37.58	0.0	**95.51**
mj-rand	-6.54	2.41	32.38	0.53	**91.8**
f-SDSS	17.17	0.0	44.61	0.0	**69.22**
f-Genomics	16.88	0.86	39.84	16.93	**94.82**
f-Birds	1.01	0.37	20.33	40.0	**97.61**

It is evident that SeLeP can benefit different types of exploratory workloads, in particular the ones consisting of random access patterns or queries with joins. Workloads falling under the *s-reg* and *m-reg* categories mainly consist of sequential data accesses. The lower hit rate of SeLeP in sequential accesses at small k values is primarily due to its shuffling of blocks in partitions based on workloads. Therefore, if certain types of workloads, such as sequential accesses, are less frequent compared to others, it is likely that their order is not preserved effectively in the partitions.

Figure 4 also provides insights into the portion of the cache populated by the workload (pink) and the additional portion consumed by SeLeP (blue). From this data, we conclude that even when the cache is full, e.g. in *f-SDSS* and *f-Genomics*, the prefetcher can effectively populate the cache and enhance the hit ratio.

Table 4 signifies that in all workloads, SeLeP has the highest coverage, meaning that it is more capable than the other models to eliminate cache misses from an NP system. In cases where coverage is zero, it means that the prefetcher could not accurately predict any upcoming access that is not already in the cache and therefore its performance is the same as an NP system. For instance, Lookahead hit ratio is equal to the NP hit ratio and it has almost zero miss coverage in all workloads except the *mj-reg* test.

Figure 4(b) depicts the relative I/O time, calculated using Equation (6). In this plot, 1 implies the maximum I/O time, equivalent to NP system that solely utilizes the LRU cache to store previously accessed blocks. Conversely, reaching 0 means the lowest possible I/O time, similar to a model that attains a 100% hit rate on every query of the workload. If the relative I/O time of a system is 0.2, it conveys that the system was able to reduce the I/O time by 80%. Thus, in this plot, a lower value signifies better performance.

While the hit ratios in workloads such as *f-Genomics* workload, are close, there is a significant variance in the I/O times of each prefetcher. Across all tests, SeLeP has the lowest relative I/O time, yielding a reduction of up to 95% in I/O time for single-table workloads and approximately 70% in more complex multi-join workloads. The Naïve's negative coverage results in a relative I/O time exceeding 1, indicating it would actually increase the I/O time of the workload. Lookahead has a negligible impact on I/O time, same as SGDP in most workloads except *s-reg* and *m-reg*. Among the baselines, Rand-Readahead performs the best, but it still falls short of achieving more than a 75% reduction.

It is worth noting that the access time of a group of blocks does not only depend on the quantity of the blocks; it is also influenced by the specific locations of these blocks on the disk. For instance, accessing ten blocks stored sequentially on the disk is significantly faster than retrieving ten blocks from various tables located in different parts of the disk. Conversely, the hit ratio is calculated based

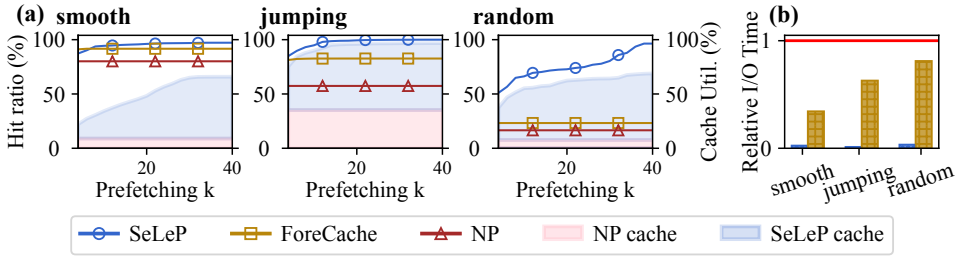

Figure 5: (1) Hit ratio in navigational workloads. The pink area represents portion of the 500MB cache, utilized by the workload, and the light blue area is the additional portion consumed by SeLeP, (2) Relative I/O time of SeLeP and ForeCache for the same workloads with $k = 36$.

Table 5: Miss Coverage of SeLeP and ForeCache in Navigational Workloads with $k = 36$.

Test	ForeCache	SeLeP
smooth	58.21	**85.88**
jumping	59.26	**99.93**
random	8.16	**85.44**

on the count of the presence or absence of the accessed blocks in the cache. Hence, achieving a hit ratio does not linearly translate into the same reduction in I/O time. It is also possible that systems with the same hit ratio have different I/O times, as the specific blocks that are missing and their location can significantly impact performance. This effect becomes particularly evident in complex workloads, such as *full* and *multi-table-joins*, which demand accessing blocks from various tables scattered across the disk.

Note that prefetching occurs during idle periods between user requests, terminating upon receiving a new query to avoid delaying execution. However, in the SDSS workload, the average prediction and prefetching process of SeLeP is 64(ms) which is far less than the average delay between user actions, ensuring completion before new requests arrive. All time overheads are reported in Table 6.

Navigational Workloads. We compare SeLeP and ForeCache using the mentioned performance metrics on navigational workloads. We treat each navigational query as a simple SQL query selecting data points within the viewport's boundary, and use these converted queries to train and test SeLeP. Due to the smaller size of this database, we adjust some parameter values, setting *MaxParSize* to 64, the block size to 16kB, and the cache size to 500MB.

Figure 5(a) shows the hit ratio of the models on the navigational workloads. ForeCache performs well in smooth navigation where users mostly move to adjacent areas. However, in random navigation where users select different areas with random coordinates, it struggles to predict accesses accurately, leading to a low hit ratio. Since ForeCache only examines the adjacent area of the current viewport, increasing k does not notably improve its performance.

Conversely, SeLeP detects the potential semantic relationship among the observed areas even if they have randomly changing coordinates. It achieves a high hit ratio in all workloads and outperforms ForeCache in almost all cases, except for when k is small in smooth navigation, which can be attributed to the same reason it does not perform optimally in sequential SQL-based workloads.

Using Figure 5(a) we select $k = 36$ to conduct other tests. Table 5 represents the coverage of SeLeP and ForeCache in the navigational workloads with $k = 36$, in which is evident that SeLeP can more effectively eliminate cache misses from an NP system.

Figure 5(b) depicts relative I/O time for SeLeP with $k = 36$ and ForeCache. While ForeCache reduces I/O time by about 60% in smooth navigation and less than 40% in random and jumping scenarios, SeLeP achieves over 90% reduction in all workloads. The converted navigational queries are structurally simple with no joins, often accessing adjacent blocks, leading to faster disk read for missing blocks compared to the SQL-based exploratory workloads. This clarifies Forecache's 20% reduction in I/O time with a 25% hit ratio.

Non-exploratory Workloads. Though SeLeP is designed based on characteristics of exploratory workloads, we conduct tests on non-exploratory workloads to underscore its effectiveness in such scenarios. We specifically choose TPC-DS benchmark, which features the largest schema and the most complex queries among other benchmarks, such as TPC-H. Given that these benchmarks share similarities in data and certain workload characteristics, we expect similar performance for SeLeP across the other benchmarks.

The key distinction between TPC-DS and exploratory workloads is the number of blocks accessed by each query. On average, each query in TPC-DS accesses about 40,000 blocks, while this number for SDSS is about 47. This is why k has larger values on this test. Also, accessing a large number of blocks increases the likelihood of block reaccess, resulting in a high hit ratio for the NP system.

Figure 6(a) depicts hit ratio of SeLeP and other baselines and Figure 6(b) represents their Relative I/O time for k=900. Similar to the exploratory workloads, Lookahead prefetcher fails to cover any misses and essentially has the hit ratio and I/O time equivalent to the NP system. The performance of other baselines are similar, improving the hit rate and I/O time up to 6% and 20% respectively.

In contrast, SeLeP can more effectively prefetch the required blocks and reduce the I/O time. It consistently achieves the highest hit ratio across nearly all values of k, except in certain cases where SGDP attains a higher hit ratio. However, as k grows larger, SGDP maintains a constant hit rate, while SeLeP achieves a 93% hit ratio and 70% I/O time reduction. Unlike Rand-Readahead and other approaches, SeLeP enhances I/O time efficiency by prefetching data from various tables located across diverse disk sections, rather than just focusing on blocks local to the recently accessed ones.

Impact of physical design. We explore the impact of physical database design by indexing the TPC-DS database and retesting with SeLeP. For index tuning, we utilize HMAB [34], a SOTA database tuning tool, which generates new indices adaptive to the workload. Performance of this new prefetcher, named SeLeP-indexed, is depicted in Figure 6. With the help of indices, the number of blocks getting accessed is reduced, which increases the chance of accurate access prediction, causing SeLeP-index to have a higher hit ratio. For the same result, the hit ratio of the NP system on the enhanced database is 3.5% higher than the NP hit rate on the original database. Additionally, in Figure 6(b), we have calculated the SeLeP-index I/O

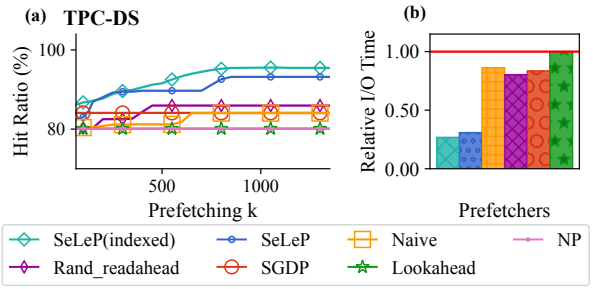

Figure 6: Hit ratio and relative I/O time for TPC-DS non-exploratory workload with 40% novel templates and $k = 900$. The workload cache utilization is 100% and is not shown.

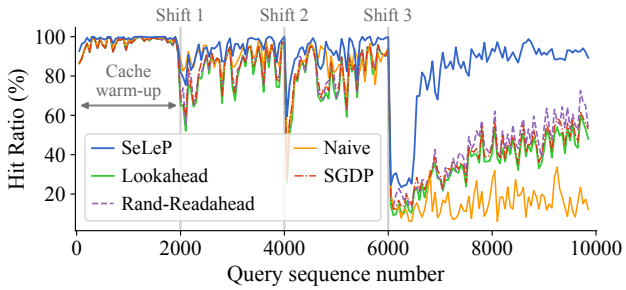

Figure 7: Hit ratio of prefetchers for each 50 consecutive queries in a shifting workload.

time relative to the I/O time of the NP system in the same database. The results in Figure 6 demonstrate that SeLeP can further enhance performance even when there are changes in the database indices.

4.2.2 Adaptivity test. As mentioned earlier, exploratory workloads can alter due to the users' evolving knowledge or changing interests and an effective exploratory tool must be able to adapt to these shifts. While it is uncommon for a large portion of the exploratory workload to undergo abrupt shifts very frequently, we conduct an adaptivity test to demonstrate SeLeP can adjust to such workload shifts. In this test, after every $l_p = 500$ queries, system repartitions, taking 102.3(s) on average, and the prediction model fine-tunes with the recent queries which, due to small l_p, takes 3.61(s) on average.

The initial batch of queries, starting from sequence number (SN) 0, constitutes a simple session without unseen blocks (blocks never accessed before), or new templates. In the next batch, SN=2000, 80% of the accessed tables change, accompanied by 20% unseen blocks. At SN=4000, 75% of the requested tables change, and 33% unseen blocks are accessed using 66% new templates. The most challenging batch starts at SN=6000, where all accessed tables change, and 85% unseen blocks are accessed using completely new query templates.

Figure 7 depicts the hit ratio of the prefetchers for each 50 consecutive queries. It is evident that SeLeP swiftly adapts to workload shifts, and maintains a more consistent hit rate compared to other approaches. Upon receiving completely new workloads in the third shift, its hit rate drops less steeply than other approaches, and after receiving l_p=500 queries and completing repartitiong and fine-tuning, its hit rate improves by 50%. Same as other tests, Lookahead and SGDP exhibit nearly identical behaviour to the NP system. Random and Naïve models occasionally achieve better hit ratios in certain batches. Following the last shift, the Naïve prefetcher however overfills the cache with incorrect prefetch decisions achieving the lowest hit ratio. Due to stochastic nature of the last query batch, neither of the prefetchers could converge to a steady hit ratio.

4.2.3 Sensitivity analysis. SeLeP has several configuration parameters mostly related to its partitioning algorithm. To address the challenges associated with these parameters, we run clay-based partitioning and the prediction model with varying configurations, evaluating their impact on the hit ratio of the SQL-based workloads.

Partitioning parameters. The initial value for θ should be a small number to enforce stricter repartitioning in the beginning that the partitions are a group of sequential blocks. As the repartitioning procedure progresses, this value gradually increases. As explained in §3.3, we have restricted the partitioning algorithm to not creating new partitions, and instead increase the θ value in case adding a new partition is the only option. A large l_p results in multiple partitions having high loads and sticking in the situation that θ should be modified. Such increases in the θ value reduce the effectiveness of the partitioning and lead to irrelevant blocks grouping together.

Figure 8(a) illustrates how the hit ratio changes as each of clay-partitioning parameters is modified. Based on these results, we selected the following parameter values: $l_p = 2500$, $k_w = 10$, and $\theta = 1$. In the initial partition generation phase, each partition should be filled to 95% of its capacity, and an extra $0.05 \times |P|$ empty partitions should be added to the system. After each repartitioning, the weights of the affinity graph should be reduced by 0.25.

Lookback. The line charts in Figure 8(b) display the hit ratio of SeLeP as the prediction model is trained with varying lengths of query encoding sequences for input. This parameter serves as a history size, and from the results, we can infer that larger history sizes can have a negative impact on the system's performance. The optimal choice for this parameter is found to be 3 or 4 and we decided to set this variable to 4.

Cache size. To show effectiveness of SeLeP with different cache sizes, we have conducted a test with all exploratory workloads with $k = 24$ and cache sizes varying from 500MB to 6GB. Results in 8(c) illustrate that hit ratio is lower for smaller cache sizes while the system could make improvement with a 1GB or larger cache. However, based on the workload data footprint, the system can achieve a high hit ratio even with 500MB cache. For instance, in *s-rand*, *m-reg*, *m-rand* and *f-Birds* tests with 500MB cache, the system yields 40% improvement in hit ratio on average over the NP.

Maximum partition size. To provide a more comprehensive representation of the impact of partitioning, we conducted tests with SeLeP using variable partition sizes. We adjust the value of k based on the partition size to prefetch an equivalent number of blocks in all systems. Figure 8(d) illustrates the hit ratio relative to the maximum partition size. With larger partitions, usually, there is a greater likelihood of accurately predicting subsequent accesses, even when prefetching the same number of blocks. This could also be observed in Figure 4 where heuristic methods such as Naïve and Rand-Readahead prefetch the same number of blocks but achieve significantly lower hit ratios compared to SeLeP. It is worth noting that despite prefetching a larger amount of data with larger partition sizes, the prefetch time remains negligible.

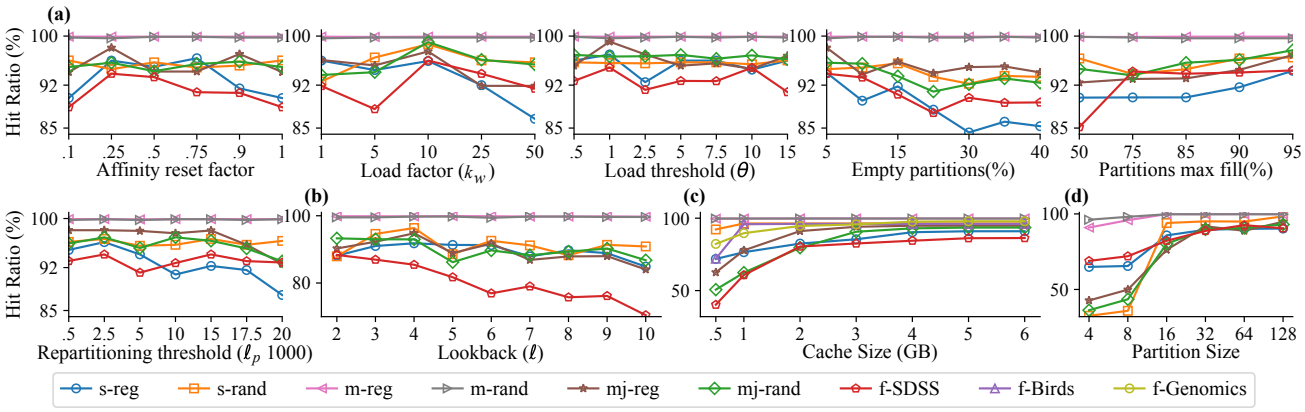

Figure 8: Effect of (a) partitioning config, (b) lookback l, (c) cache size, and (d) *MaxParSize* on hit ratio of SQL-based workloads.

Table 6: Average time of block encoding, data preparation (including block encoding), access prediction, prefetching selected blocks, and repartitioning in different datasets.

Action	Navigational	SDSS	Birds	Genomics	TPC-DS
Block encoding (s) one-off	779.6	2898.17	3672.62	1997.86	4978.88
Data preparation (s) one-off	1108.11	3848.83	8548.37	3220.22	7358.85
Prediction (ms)	2.16	8.28	11.08	18.44	356.15
Prefetching (ms)	20.87	55.81	44.94	52.05	511.52
Repartitioning (s)	17.31	142.81	228.26	456.92	1901.83

4.2.4 Time overheads. Table 6 represents the time overhead of different components of the system for each dataset. The data preparation as a one-off task typically requires the longest time, while partition access prediction and data prefetching are done at the level of milliseconds. Therefore, the prefetching process can lower the I/O time without delaying execution of the user's next query.

5 RELATED WORK

Visual tools prefetchers. are utilized to prefetch spatial data observed through a fixed-size viewport, such as a laptop screen [4, 7, 10, 18, 42, 42, 44]. Their primary assumption is that users navigate smoothly through the data space, and select prefetch candidates from the neighboring area of the recently accessed region. These prefetchers make decision by repeating user's previous movements (*action-based*)[7, 10], or learning and predicting movement or location sequences patterns (*learning-based*)[42, 44], or, akin to semantic prefetchers, exploit the similarity in data properties among data tiles to make predictions *(characteristic-based)* [4, 44].

SQL-based tools prefetchers. are mostly traditional prefetchers such as Random Readahead [30] and One Block Lookahead [39] which struggle with complex data access patterns. The only learning-based database prefetcher [8] applies offline training on a single set of data and falls short in meeting the adaptivity demands of exploratory tools. Memory prefetchers, extracting LBAs [20] and LBA-deltas [6, 47] sequece patterns, may find utility in SQL-based tools. SGDP [47] is the SOTA learning based memory prefetcher, demonstrating superior performance compared to other models. These prefetchers excel in memory systems where applications access data within their limited allocated memory portion and the LBA-deltas are less diverse, but struggle in data exploration due to the complexity and diversity of data access patterns.

Workload-driven approaches for latency reduction. Physical design tuning is a well-studied area where tuning decisions are made to optimize system performance based on resource constraints and data access patterns. These decisions involve adaptively creating or altering indices to reduce the query execution time [22, 23, 33–35] or selecting which views to materialize or which results to cache to prevent repeated computation in the future [25, 34, 38]. We view such approaches as complementary, as they mainly optimize computation part of execution (rather than I/O).

Data partitioning. Partitioning is a technique to improve manageability of the data or load balancing in distributed systems [43, 45]. Combined with other techniques such as index tuning for individual partitions [29] or locating partitions of frequently co-accessed data sequentially on the disk [31], this approach can also be effective in reducing system latency and improving performance.

6 CONCLUSION

In this work, we present SeLeP, a learning based semantic prefetcher to enhance interactivity of the exploratory workloads. Unlike the majority of SOTA prefetchers, SeLeP utilizes data semantics acquired through encoding data blocks rather than relying on logical data addresses. Our experimental results on several SQL-based, navigational and non-exploratory workloads indicate that prefetching based on data semantics can yield comparable or even superior performance (up to 40% higher hit ratio in challenging workloads) when compared to prefetchers that rely solely on data addresses. Additionally, decision making based on the encoded query results calculated using data block encodings allows our prefetcher to cater to both types of workloads effectively. Furthermore, our system outperforms traditional address-based prefetchers in terms of speed (up to a 45% greater reduction in I/O time), enabling faster and more responsive query execution, crucial for exploratory systems.

ACKNOWLEDGMENTS

We gratefully acknowledge support from the Australian Research Council Discovery Early Career Researcher Award DE230100366, and L'Oreal-UNESCO For Women in Science'23 Fellowship.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Kevork N Abazajian, Jennifer K Adelman-McCarthy, Agüeros, et al. 2009. The seventh data release of the Sloan Digital Sky Survey. *The Astrophysical Journal Supplement Series* 182, 2 (2009), 543.

[3] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*. 241–252.

[4] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data*. 1363–1375.

[5] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1121–1137.

[6] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *ECML/PKDD*. 427–443.

[7] Sye-Min Chan, Ling Xiao, John Gerth, and Pat Hanrahan. 2008. Maintaining interactivity while exploring massive time series. In *2008 IEEE Symposium on Visual Analytics Science and Technology*. IEEE, 59–66.

[8] Yu Chen, Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. 2021. Revisiting data prefetching for database systems with machine learning techniques. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2165–2170.

[9] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. (2014), 103–111. https://doi.org/10.3115/V1/W14-4012

[10] Punit R Doshi, Elke A Rundensteiner, and Matthew O Ward. 2003. Prefetching for visual data exploration. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings*. IEEE, 195–202.

[11] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[12] Jim Gray, David T. Liu, María A. Nieto-Santisteban, Alexander S. Szalay, David J. DeWitt, and Gerd Heber. 2005. Scientific data management in the coming decade. *SIGMOD Record* 34, 4 (2005), 34–41.

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. 2023. Normalization techniques in training dnns: Methodology, analysis and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2023), 10173–10196.

[15] Stratos Idreos. 2013. *Big Data Exploration*. Taylor and Francis.

[16] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of data exploration techniques. In *SIGMOD*. 277–281.

[17] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data*. 281–293.

[18] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2014. Interactive data exploration using semantic windows. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 505–516.

[19] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. 2011. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *VLDB* 4, 12 (2011), 1474–1477.

[20] Ando Ki and Alan E Knowles. 2000. Stride prefetching for the secondary data cache. *Journal of systems architecture* 46, 12 (2000), 1093–1102.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings*.

[22] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proc. ACM Manag. Data* 1, 2 (2023), 139:1–139:22. https://doi.org/10.1145/3589284

[23] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-disk Updatable Learned Index. In *40th IEEE International Conference on Data Engineering (ICDE)*. IEEE.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[25] Xi Liang, Aaron J Elmore, and Sanjay Krishnan. 2019. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint arXiv:1903.01363* (2019).

[26] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.

[27] Holger R Maier and Graeme C Dandy. 1998. The effect of internal parameters and geometry on the performance of back-propagation neural networks: an empirical study. *Environmental Modelling & Software* 13, 2 (1998), 193–209.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013), 3111–3119.

[29] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1106–1117.

[30] Michael Opdenacker and Free Electrons. 2007. Readahead: time-travel techniques for desktop and embedded systems. In *Proc. of the 2007 Ottawa Linux Symposium*, Vol. 2. 97–106.

[31] Mirjana Pavlovic, Eleni Tzirita Zacharatou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2016. Space odyssey: efficient exploration of scientific data. In *Proceedings of the Third International Workshop on Exploratory Search in Databases and the Web*. 12–18.

[32] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2, 11 (1901), 559–572.

[33] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 600–611.

[34] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2022. HMAB: self-driving hierarchy of bandits for integrated physical database design tuning. *Proceedings of the VLDB Endowment* 16, 2 (2022), 216–229.

[35] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2023. No DBA? No Regret! Multi-Armed Bandits for Index Tuning of Analytical and HTAP Workloads With Provable Guarantees. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12855–12872. https://doi.org/10.1109/TKDE.2023.3271664

[36] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. 1985. Learning internal representations by error propagation.

[37] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[38] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J Elmore, Sanjay Krishnan, and Michael J Franklin. 2020. CrocodileDB: Efficient Database Execution through Intelligent Deferment. In *CIDR*.

[39] Alan Jay Smith. 1978. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)* 3, 3 (1978), 223–247.

[40] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014), 3104–3112.

[42] Farhan Tauheed, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. 2012. SCOUT: Prefetching for Latent Feature Following Queries. *Proc. VLDB Endow.* 5, 11 (2012), 1531–1542.

[43] Hoang Vo, Ablimit Aji, and Fusheng Wang. 2014. SATO: a spatial data partitioning framework for scalable query processing. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*. 545–548.

[44] Ran Wan, Roman Garnett, and Alvitta Ottley. 2018. Learning and Anticipating Future Actions During Exploratory Data Analysis. *arXiv preprint arXiv:1809.09664* (2018).

[45] Eugene Wu and Samuel Madden. 2011. Partitioning techniques for fine-grained indexing. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1127–1138.

[46] Fei Yang, Luis Herranz, Joost Van De Weijer, José A Iglesias Guitián, Antonio M López, and Mikhail G Mozerov. 2020. Variable rate deep image compression with modulated autoencoder. *IEEE Signal Processing Letters* 27 (2020), 331–335.

[47] Yiyuan Yang, Rongshang Li, Qiquan Shi, Xijun Li, Gang Hu, Xing Li, and Min jie Yuan. 2023. SGDP: A Stream-Graph Neural Network Based Data Prefetcher. *2023 International Joint Conference on Neural Networks (IJCNN)* (2023), 1–8.