Generalizable Address-aware Semantic Prefetching for Scalable Transactional and Analytical Workloads

Farzaneh Zirak, Farhana Choudhury, Renata Borovica-Gajic
University of Melbourne, Melbourne, Australia
{f.zirak, farhana.choudhury, renata.borovica}@unimelb.edu.au

***Abstract*—*Data prefetching* plays a crucial role in reducing I/O overhead and improving the performance of database systems. While traditional prefetchers focus on sequential patterns, recent learning-based approaches, especially those leveraging data semantics, achieve higher accuracy for complex access patterns. However, these methods often struggle with today's dynamic, ever-growing datasets and require frequent, timely fine-tuning. Privacy constraints may also restrict access to complete datasets, necessitating prefetchers that can learn effectively from samples.**

To address these challenges, we present GrASP, a learning-based prefetcher designed for both analytical and transactional workloads. GrASP enhances prefetching accuracy and scalability by leveraging logical block address deltas and combining query representations with result semantics. It frames prefetching as a context-aware multi-label classification task, using multi-layer LSTMs to predict delta patterns from embedded context. This delta modeling approach enables GrASP to generalize predictions from small samples to larger, dynamic datasets without requiring extensive retraining. Experiments on real-world datasets and industrial benchmarks demonstrate that GrASP generalizes to datasets 250× larger than the training data, achieving up to 45% higher hit ratios, 60% lower I/O time, and 55% lower end-to-end query execution latency compared to existing baselines. On average, GrASP attains a 91.4% hit ratio, a 90.8% I/O time reduction, and a 57.1% execution latency reduction.

***Index Terms*—Data-driven prefetching, Semantic prefetching, Access prediction, Delta modeling, Data exploration**

I. Introduction

Data prefetching is a fundamental technique employed by database management systems (DBMS) to improve performance by reducing I/O time. A prefetcher anticipates future accesses and proactively loads relevant data into cache. Prior work spans rule-based heuristics to deep learning methods and shows benefits across diverse workloads [1]–[8].

Traditional prefetchers target sequential and locality based patterns, whereas recent learning-based models capture richer access behaviors [1]–[4]. Notably, semantic-based learning prefetchers further improve accuracy by leveraging data characteristics to capture non-trivial patterns [2]–[4]. However, state-of-the-art (SOTA) learning-based prefetchers often struggle to scale with today's rapidly growing datasets. They fail to generalize to evolving workloads or modified datasets without timely and costly fine-tuning, degrading responsiveness.

This limitation is acute in modern data systems with evolving workloads and rapidly growing datasets that demand scalable prefetching [9], [10]. Timely access to relevant information is vital across applications [11]–[13], requiring prefetchers that scale and adapt with minimal retraining overhead.

These challenges intensify in *data exploration*, where users seek timely insights from (newly ingested) data. Beyond exploration of static data, many tasks require rapid analysis on frequently updated datasets [14]. For example, analysts may monitor recent stock transactions to detect fraud [15], or track social media streams for emerging trends [16]. In such cases, SOTA prefetchers may lack sufficient time to preprocess new batches or update models, limiting their effectiveness.

Another constraint arises when the prefetcher cannot access the full dataset during training due to either constant updates or more often privacy restrictions, as in medical or enterprise environments [17]. Data owners often limit access to complete datasets and query workloads for confidentiality reasons, reducing the effectiveness of deep learning models. This calls for prefetchers that can train on limited samples while effectively generalizing across a much larger, unseen data space.

Given these constraints, we aim to design a prefetcher that accurately anticipates data accesses while satisfying two goals:

i. **Generalizable to larger datasets.** A prefetcher must remain effective even when trained on a much smaller subset of the full deployment dataset. Upon deployment, it should adapt its predictions without requiring extensive retraining.

ii. **Robust across workloads.** Analytical workloads typically exhibit stable semantics and repeated cross-object access patterns, making semantic dependency signals effective. Transactional workloads continuously introduce new or altered blocks and drift semantics, which can invalidate precomputed encodings and break prefetchers that restrict predictions to blocks seen during training [1]–[3]. Since frequent fine-tuning is impractical, the prefetcher must cover the full data space and adapt with minimal adjustment.

Recent memory prefetchers [18]–[21] model differences between successive logical block addresses (LBAs), called *LBA-deltas*, enabling prediction over dynamic and previously unseen regions. This view is also used in previous database prefetchers [1], [5]–[7]. In contrast, semantic-based prefetchers perform well for analytical workloads by capturing inter-object dependencies [2]–[4], but often degrade for transactional workloads as updates render semantics stale or incomplete.

We introduce GrASP, a learning-based prefetcher that bridges this gap by combining LBA-delta (delta in short) modeling with semantic context. GrASP formulates prefetching as contextual multi-label classification: given recent query semantics and LBA signals, a multi-layer LSTM over embedded

contexts predicts the most probable deltas for the next query and maps them to candidate LBAs to prefetch.

GrASP incorporates data semantics by dynamically preprocessing and encoding accessed blocks using feature extraction. To avoid over-reliance on static encodings, it defines *semantic-based context* by combining query result encodings aggregated from accessed blocks and query statement representations capturing query type, tables, join conditions, and filter predicates.

For the *LBA context*, we introduce a table-based LBA abstraction to reduce sensitivity of deltas to database growth. We also define an *order-agnostic* delta to represent the *set* of deltas associated with each query, independent of access order.

GrASP builds its input context by combining semantic and LBA features with metadata such as the last accessed tables and the per-query delta count. To address class imbalance from naturally skewed access patterns, GrASP uses a custom loss and dropout regularization to improve generalization.

In this paper, we make the following contributions:

- We introduce GrASP, to our knowledge the first hybrid prefetcher to fuse semantic context with delta modeling, using a table-based LBA abstraction and an order-agnostic delta.
- We formulate prefetching as a contextual classification problem, leveraging a novel integration of query semantics and LBA information to improve accuracy and generalization.
- GrASP generalizes effectively, transferring learned delta patterns to significantly larger datasets with minimal tuning.
- Extensive experiments on real-world analytical and industrial transactional workloads show that GrASP achieves up to 91.4% hit ratio, 90.8% I/O-time reduction, and 57.1% latency reduction. Compared to SOTA prefetchers, it improves hit ratio/I/O time/latency by up to 17%/36%/28% on analytical and 45%/60%/55% on transactional workloads.

II. Background and Related Work

A. Preliminaries

Prefetching predicts the upcoming block accesses from the recent workload context. The key modeling choices are how to construct the input context based on the accessed LBAs or semantic features, and the output representation, typically either absolute block addresses (LBA prediction) or relative movement between consecutive addresses (delta prediction).

For a query q, let A_q denote its set of accessed data blocks. Assuming an access order is available, sorting the logical addresses associated with the blocks in A_q yields an LBA sequence $L_q = \langle lba_1, \ldots, lba_n \rangle$. The corresponding delta sequence is $\Delta_{q_t} = \langle ld_1, \ldots, ld_{n-1} \rangle$ where:

$$ld_i = lba_{i+1} - lba_i \tag{1}$$

The delta sequence captures how the access stream moves through the address space, with each delta encoding the step from one accessed block to the next (e.g., scans yield small forward steps like $\langle +1, +1, +1 \rangle$, while joins mix back-and-forth bursts with jumps like $\langle +1, -3, +19, -11 \rangle$). This view is often more stable under growth, since inserts and updates may shift absolute LBAs, but local step patterns tend to persist [22], [23]. The following outlines two prefetching formulations.

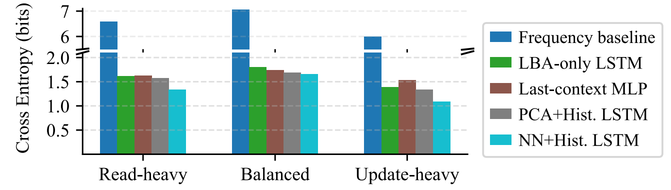

Fig. 1: Cross entropy of delta set prediction across workloads.

1) Address-based Prefetching: These methods predict the next LBA sequence $L_{q_{t+1}}$ from an address-based context, such as recent LBAs L_{q_t} or their delta sequence Δ_{q_t}.

2) Semantic-based Prefetching: Rather than relying on block addresses, semantic prefetchers predict $L_{q_{t+1}}$ by leveraging information from query or block contents, either via semantic similarity [2] or machine learning techniques [3].

B. Problem Definition

In GrASP, the workload context combines delta and semantic information from recent queries. Let c_{q_t} denote the context of query q_t, where $A_{q_t}, \Delta_{q_t} \in c_{q_t}$. Accordingly, the address-based semantic prefetching problem is defined as follows: *Given the l most recent query context, $\langle c_{q_i} \rangle_{i=t-l}^{t}$, predict $\Delta_{q_{t+1}}$ and the next address sequence $L_{q_{t+1}}$ to fetch.*

By treating each delta as a class, a classifier estimates which deltas will appear in $\Delta_{q_{t+1}}$. Although deltas can span a wide range (large label space), in practice, delta frequencies are skewed. Thus, prefetchers restrict predictions to the top-k most frequent deltas and map rare ones to a default class, skipping prefetching when only the default is selected.

Figure 1 shows cross entropy for next delta set prediction across workload types, where lower values mean higher predictability under the given context. It shows that adding richer semantic+LBA context and short access history consistently reduces uncertainty, and that learned context encoders outperform generic dimensionality reduction, motivating our use of a sequence model with learned context for delta prediction.

C. Existing Prefetchers and Limitations

1) Delta-Based Heuristics: A long line of prefetchers model access locality in the delta space, where repeated *movement patterns* can be easier to detect than absolute addresses. Early heuristics exploit fixed small deltas via lookahead and storage readahead [5], [6], stride prefetchers learn stable per-stream deltas [7], [8], and correlation-based designs replay recurring delta sequences from history [23]–[25]. These methods work well for scan-like or repetitive streams but degrade under interleaving, skew, or random accesses.

2) Learned Prefetchers: Learning over historical access sequences improves performance on irregular or complex workloads. Some learned database prefetchers directly predict future LBAs. The *address-based prefetcher* in [1] uses a two-level hierarchical model to predict the next LBA, with each level predicted separately. However, assigning a class to every possible LBA is inefficient for large and dynamic databases, as the label space grows with the dataset and limits scalability.

Learned delta models in *memory prefetchers* predict delta classes rather than absolute LBAs, yielding a smaller and more

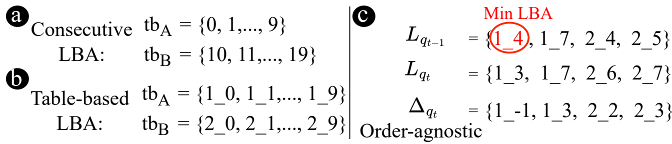

Fig. 2: Examples for (a) consecutive LBA, (b) table-based LBA, and (c) order-agnostic deltas using min_{lba} of prior blocks.

transferable output space [18]–[21]. Starting from [21], most designs often use recurrent neural networks (RNN) to predict the next delta or a short fixed-length delta sequence; longer-horizon prediction relies on rollout or larger one-step models, increasing compute cost and compounding errors. These methods are most effective when accesses remain within regions of low delta diversity, but degrade as delta variability grows.

Semantic prefetchers often outperform address-based methods [2]–[4]. SeLeP [3] encodes each block offline using AutoEncoder-based feature extraction over preprocessed block values [26]. It derives a query encoding for q by aggregating encodings of the blocks in A_q; to capture table-specific semantics, the query encoding is represented as a matrix with one row per table. SeLeP clusters blocks into partitions based on recent co-access ratios, then uses sequences of recent query encodings to estimate future partition access probabilities.

However, SeLeP faces scalability issues akin to direct LBA predictors and requires timely preprocessing and retraining to incorporate newly inserted or modified data, reducing interactivity for workloads with frequent updates or bulk inserts.

D. Query Representation

Query statements and execution plans are key inputs for database tuning tasks such as index tuning [27]–[30], view selection [31], [32], and query optimization [33], [34]. These systems encode query details into analyzable formats using lightweight [29]–[31] or advanced techniques [32]–[34].

Lightweight methods encode query attributes such as accessed tables, query types, and normalized operation costs without plans. Advanced techniques represent execution plans to capture operation hierarchies. For example, Query2vec [35] treats plans or SQL statements as sentences, strips literals and numbers, and encodes them using Doc2Vec [36].

III. Motivation and Challenges

Delta modeling for database prefetching raises two challenges: *(i) assigning a unique LBA to database blocks?* and *(ii) defining Δ_q when q accesses multiple blocks simultaneously?*

Traditional DBMSs store tables as heap files of fixed-size blocks, with indexes steering execution to specific blocks. Query execution reads these blocks into the buffer pool, making runtime highly locality-sensitive: sequential scans traverse adjacent blocks quickly, while index probes and joins often trigger scattered reads with high random latency. Prefetching uses execution slack to issue reads early, and delta modeling is well-suited because it captures movements between blocks, which can often be issued as range-friendly prefetches [22].

Queries access data via DBMS internal identifiers (IIDs) distinct from OS-level physical LBAs (e.g., PostgreSQL CTID and Oracle RowID). IIDs remain valid even when physical placement changes due to maintenance, rewrites, or storage-layer relocation. Therefore, we perform delta modeling in the IID space to keep predictions stable and let the engine reliably translate predictions into physical block fetches.

TABLE I: Total Number of Unique Deltas and Hit Ratio with Different Labeling Methods (best in bold, second best underlined)

Dataset	Hit Rate (%)			Total Delta Count		
Reference LBA	Min	Median	Max	Min	Median	Max
Read-heavy	**91.93**	91.03	<u>91.56</u>	**13992**	18500	<u>18041</u>
Balanced	**94.46**	91.02	<u>92.27</u>	<u>2263</u>	**2132**	2483
Update-heavy	**96.7**	94.86	<u>94.94</u>	<u>5511</u>	5711	**5508**

A. Challenge (i) — LBA Definition

IIDs are composite pointers. Since I/O is block-level, we retain only their block ID, which is unique only within a tablespace and may collide across tables. Thus, raw IIDs cannot serve as LBAs, and we must derive globally unique LBAs.

A naive option is consecutive LBAs, analogous to placing all database blocks in one global order with unique IDs. However, this setup is unstable: inserts shift LBAs and distort delta patterns. It also produces large positive or negative deltas when queries access multiple tables and switch between them.

To address these issues, we implement a table-based LBA scheme with a two-level hierarchy: the block's IID within its table and a table ID that determines the table. Deltas are computed as composite values, including the target table ID and the difference in IID values. For example, the delta from b_i to b_j with LBAs $\mathrm{tb_x_IID_{b_i}}$ and $\mathrm{tb_y_IID_{b_j}}$ is $\mathrm{tb_y_(IID_{b_j} - IID_{b_i})}$, where the table ID indicates the target table after applying the delta. Fig. 2(a) and (b) show consecutive and table-based labeling on two sample tables, each with ten blocks.

B. Challenge (ii) — Delta Computation

To compare LBA schemes, we need a consistent delta definition. In memory prefetchers, CPU instructions typically touch one data page, yielding a clear access order for delta calculation. In contrast, database queries read *sets* of zero to many blocks, where defining a specific order is impractical [3].

A simple workaround is to sort a query's accessed blocks by LBA and apply Eq. (1). However, this requires multiple sequential predictions to generate prefetch decisions, adding latency that is incompatible with interactive workloads, and prediction errors can cascade across steps, reducing accuracy.

To handle ordering and enable collective delta prediction, we compute deltas by subtracting each LBA $\in L_{q_t}$ from a reference LBA in $L_{q_{t-1}}$. We evaluate three reference choices—maximum, minimum, and median of the sorted LBAs—and measure their impact on the number of unique deltas and the hit ratio across three datasets (Datasets and metrics are described in §V-B and §V-D, respectively). As Table I shows, min_{lba} yields fewer unique deltas and better prediction accuracy. Hence, we adopt min_{lba} and compute the order-agnostic delta set using (2). Fig. 2(c) shows the resulting delta set for a sample query using the table-based min_{lba}.

$$\Delta_{q_t} = \left\{ \mathrm{LBA} - \min\left(L_{q_{t-1}}\right) \middle| \mathrm{LBA} \in L_{q_t} \right\} \tag{2}$$

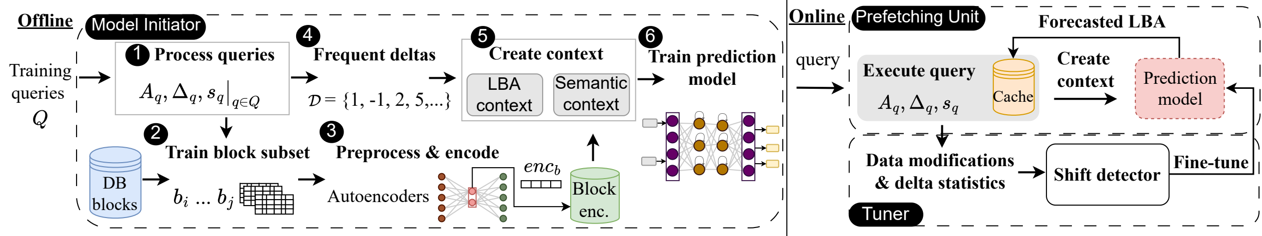

Fig. 3: System architecture of GrASP, representing its three main components: model initiator, prefetching unit and tuner.

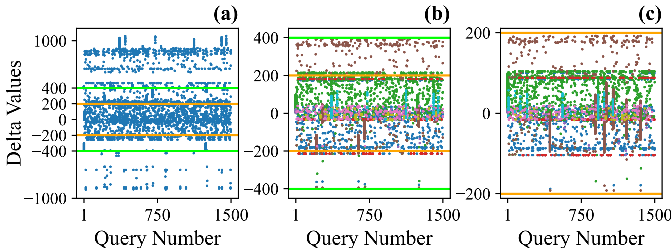

Fig. 4: Delta values of read-update balanced workload under (a) consecutive LBA and (b-c) table-based LBA, colored by table (vertical cluster of deltas in a query indicate a sequential scan). (a-b) 16 GB data; (c) 8 GB. Colored bands aid readability.

C. Delta Analysis

Fig. 4 plots delta values from Eq. (2) under two labeling methods, for 1500 queries from a read-update balanced workload at two dataset sizes. Comparing Fig. 4(a, b) shows that consecutive LBAs yield a much wider delta range, while table-based labeling reduces it by over 60% on the same workload.

Fig. 4 reveals several patterns. First, deltas are concentrated around zero within a bounded range, indicating frequent reuse of a limited set of deltas and supporting delta-based prediction.

Second, delta visualizations reveal noticeable patterns in delta occurrences, especially in Fig. 4(b, c), where deltas are color-coded by table ID. These patterns suggest that deltas are not random and can be modeled for prediction.

Third, comparing the two dataset sizes (Fig. 4(b, c)) shows that the dominant patterns persist for the same schema and query templates, though larger data size may trigger different plans, subtly altering access behavior. As size increases, both the delta range and density expand, with non-linear changes in delta values and per-query delta counts. This indicates that delta patterns scale with dataset size while preserving the underlying structure, motivating a prefetcher that can transfer patterns learned on smaller instances to larger ones.

IV. GrASP Framework

GrASP is a learning-based framework that learns delta patterns from semantic signals, query context, and access history. Since 8–32 kB block size leads to millions of blocks and a large delta space, we reduce prediction complexity by logically grouping lb_{size} consecutive blocks for prediction, while caching stays at native block size. lb_{size} and other parameters introduced in §IV are evaluated in §VI-E.

A. An Overview

GrASP consists of three components (Fig. 3). The **model initiator** builds the semantic encoders, prediction model, and delta vocabulary. The **prediction unit** constructs contexts online and issues prefetches using the trained models. The **tuner** adapts these models as data and workloads evolve.

TABLE II: Frequently Used Notations

Symbol	Definition
A_q	Set of data blocks accessed by query q
L_q, Δ_q	Set of LBAs for the blocks in A_q, LBA-delta derived from L_q
r_q, s_q	Result encoding of q, SQL statement representation of q
$\boldsymbol{\mathcal{D}}$	Global set of frequent deltas (delta vocabulary), $\lvert\boldsymbol{\mathcal{D}}\rvert = ds$
$\mathcal{D}_q$	Set of unique deltas extracted from Δ_q (table ID removed)
d_q	Bitmask vector of $\mathcal{D}_q$ over $\boldsymbol{\mathcal{D}}$
τ, table α	table selection threshold and its modifying factor
k_{dc}	Multiplier for predicted query delta count
$n_{lookback}$	Context sequence length used by the model
k	Prefetch size in unit of 128 blocks

During initialization, GrASP processes training queries, collects their results, and selects accessed blocks (and their immediate neighbors) for semantic extraction. For each table, selected blocks are preprocessed and passed through an autoencoder to produce block encodings (Fig. 3 (1–3)). The semantic context generator then forms query semantics by combining a query result encoding r_q with a query statement representation s_q. GrASP also analyzes training-workload deltas, retains the most frequent ones as its delta classes, and represents each query by a binary delta vector d_q and a one-hot encoding of its size. Using sequences of semantic and LBA contexts together with the last accessed table, GrASP trains the prediction model to learn delta patterns (Fig. 3 (4–6)).

At runtime, the prediction unit predicts the next accessed tables, a distribution over delta classes, and the delta count n, then selects the top-n deltas. It combines predicted tables and deltas, filters them using historical frequency, and maps the remaining deltas to candidate LBAs for prefetching. The tuner maintains robustness by updating preprocessing, refreshing frequent deltas, and tuning the autoencoders and predictor; it also encodes newly inserted and first-seen blocks.

B. Block Encoding

Semantic prefetchers leverage the actual data values to make prefetching decisions. Because each block can contain hundreds of values, they require a concise representation that summarizes the block's key characteristics. Since stored data may serve diverse purposes, it is impractical to identify in advance which attributes contain the most critical information. Therefore, block semantics are captured using unsupervised feature extraction methods such as Autoencoders [26].

We enhance SeLeP's block encoding component described in §II-A2. This component encodes each table's blocks into compact representations using table-specific autoencoders im-

plemented as multilayer perceptrons (MLPs). The models are table-specific because differences in schema, size, and semantics make a shared model ineffective.

Before encoding block data, non-numerical values must be converted into numerical representations. The Word2Vec encoding [37] approach that is used by SeLeP cannot handle unseen data, leading us to evaluate two alternatives: FastText [38], which extends Word2Vec by learning embeddings for strings and substrings, and MinHash [39], which is a Locality-Sensitive Hashing technique capable of encoding strings.

Employing these methods resulted in much higher encoding times than Word2Vec, with FastText requiring orders of magnitude longer training and tuning. In contrast, Word2Vec supports incremental updates, where new words refine the existing embedding space instead of rebuilding it. Following the iterative retraining strategy explored in [40]–[42], we retain Word2Vec but add a mechanism to dynamically expand its vocabulary. Textual values from the block are combined into a sentence and fed into the model, enabling it to learn embeddings for new values as they are encountered.

Normalization is the most impactful preprocessing step, as training the Autoencoders on the raw data blocks with a wide range of values will result in a poor block encoding. We retain the min-max normalization method from [3] since, in a dynamic dataset, maintaining minimum and maximum values is simpler and more computationally efficient compared to other statistical metrics, such as mean, standard deviation, or quartiles, used by alternative normalization methods.

To handle tables with a large number of columns, [3] applies PCA [43] after normalization. We make this step more adaptive by replacing PCA with Incremental PCA (IPCA) [44], which updates the transformation as new data arrives, *eliminating the need for a complete re-computation* [45].

After dimensionality reduction, the processed blocks are fed into the table's autoencoder to generate encodings. These encodings are stored for later use in creating query result encodings. In §IV-E we explain how IPCA is used to evaluate whether tuning the autoencoders is necessary.

C. Context Creator

1) Query Semantics Generator: A query's behavior depends not only on the blocks it accesses but also on how it filters, joins, and aggregates data—details that block-level embeddings alone cannot capture. Query statements more fully express user intent and provide richer context, enabling more accurate prediction of future data accesses within a query session (a series of closely timed, goal-aligned queries). This is particularly important in exploratory workloads, where sessions aim to uncover specific insights.

Statement representations capture table interactions in query semantics, which enhance access modeling and reduces reliance on result encodings that may drift as data changes. Thus, GrASP combines the query result encoding (r_q) and statement representation (s_q) into a robust semantic encoding (enc_q).

*a) **Query Result Encoding**:* r_q can be calculated by aggregating enc_b of the blocks in A_q. However, aggregating

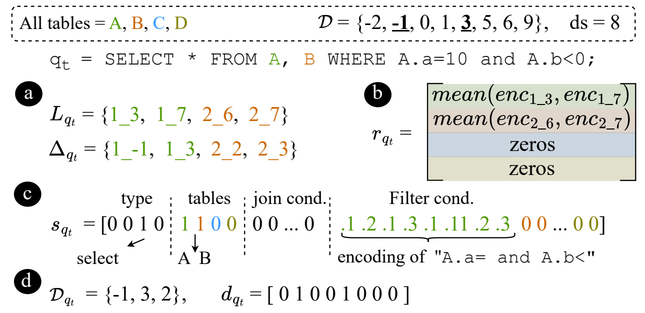

Fig. 5: Examples of (a) result LBAs and *delta*s, (b) result encoding, (c) statement representation, and (d) binary delta.

encoding of blocks from different tables will result in a meaningless representation [3], since the semantic interpretation of individual fields within enc_b varies across tables. Thus, the query result must be encoded as a matrix where encodings of blocks from the same table are aggregated and placed in a single row corresponding to that table. Fig. 5(b) depicts r_q for a sample query accessing four blocks.

*b) **Query Statement Representation**:* Constructing an effective statement representation requires addressing three key questions: *(i) Which query details are most relevant to the task? (ii) Is including additional information from the query plan beneficial? (iii) Should the representation maintain consistent semantic meaning across queries?*

Different systems selectively encode details tailored to their specific tasks. For instance, QTune [30] encodes accessed tables and operation costs, while DBABandit [29] focuses on accessed columns only. In prefetching, the emphasis is on data accessed by a query, which depends on its type, accessed tables, join conditions, accessed columns, and filters. Different query types exhibit distinct block access patterns: modification queries usually access fewer blocks within a single table, whereas selection queries often join multiple tables and access more blocks. Join and filter conditions narrow the query's target, dictating which specific blocks must be read.

We use a compact, structured representation of the statement capturing these details. The query type is one-hot encoded, and since queries may access multiple tables, we represent them with a binary bitmap, setting bits for each referenced table.

Encoding query conditions is more challenging, as multiple conditions can apply to any column within a table. We parse the query execution plan and extract join and filter predicates of each table. Following Query2Vec [35], we remove numeric values and literals from the conditions to improve generalizability. Each table's conditions are then treated as a short document and encoded using a Doc2Vec [36] model. We have separated the join conditions with the filters since they have distinct impact on the accessed blocks.

The final statement representation is composed of 4 bits for the query type, |Tables| bits for accessed tables, and two parts of $8 \times |\text{Tables}|$ bits each capturing encoded joining and filtering conditions applied to each table. This format ensures uniform representations where each field has a consistent and comparable meaning across queries. Fig. 5(c) shows s_q for a

TABLE III: Hit Ratio, Average Recall, and Average Statement Encoding Time of Different Representation Methods

Test		None	Sq2v	Pq2v	Simple	GrASP
Auction	Hit Rate	94.11	94.42	94.66	94.15	**95.26**
	Recall	57.94	71	71.37	71.63	**72.01**
Auction 20% size ratio	Hit Rate	91.22	92.37	92.75	96.11	**96.58**
	Recall	54.56	58.79	58.53	59	**60.01**
SDSS	Hit Rate	97.89	**98.56**	97.6	98.1	97.91
	Recall	73.41	72.87	72.45	**75.2**	74.12
SDSS 10% size ratio	Hit Rate	97.89	98.08	97.51	**98.37**	**98.37**
	Recall	72.56	71.83	70.63	75.52	**76.57**
Avg preparation time/$q(ms)$		NA	2.72	2.69	**0.45**	0.52

given query with two filter conditions on table A.

We focus on table-level details because including every column from all tables in the database results in a sparse high-dimensional query representation, as most queries access only a small subset of columns. We deliberately exclude lower-level plan details such as join strategies or operation order, as they introduce unnecessary specificity and reduce generalizability.

We evaluate our query representation by comparing GrASP to variants with alternative encodings: no statement features (None), SQL-Query2vec (Sq2v), Plan-Query2vec (Pq2v) as in §II-D, and a partial representation that includes query type and accessed tables only (Simple). We assess the prefetcher's hit ratio and average recall across four test datasets (The datasets and the metrics are provided in § V-B and § V-D, respectively).

Table III shows that our plan-agnostic method (GrASP) and its partial version (Simple) achieve the best performance, especially compared to None. In datasets with size ratio, where the prefetcher is trained on a sampled dataset and tested on a larger one, GrASP yields the largest gains over None.

2) LBA-context Creator: The LBA context is derived from recent queries' Δ_q. Large databases have a vast set of possible delta values, increasing model complexity and reducing accuracy if all are included. To address this, we select a subset of these deltas to define the model output and the LBA context.

GrASP predicts future accesses by modeling deltas. Because deltas define the input features, the output classes, and the overall model complexity, their selection is central to the design. Since most queries use a small set of frequent deltas (§III-C), using those frequent values simplifies the prediction and ensures the model focuses on the most impactful values.

To construct the *global delta vocabulary* $\boldsymbol{\mathcal{D}}$, we analyze Δ_q from the training workload, discard table identifiers, and retain IID deltas, denoted as $\mathcal{D}_q$. We compute delta frequencies and select the top ds most frequent values where $ds = |\boldsymbol{\mathcal{D}}|$ is the *delta vocabulary size* (i.e., the number of delta classes). The impact of ds values is evaluated in §VI-E, with 1500 chosen as the optimal setting, which performs well on a 155GB database.

To handle infrequent deltas not included in $\boldsymbol{\mathcal{D}}$, we introduce a default class to ensure full coverage. $\mathcal{D}_q$ is then encoded as a binary vector, denoted as d_q, of length ds, where $d_q[i] = 1$ if $\boldsymbol{\mathcal{D}}[i] \in \mathcal{D}_q$. If no value of $\mathcal{D}_q$ is in $\boldsymbol{\mathcal{D}}$, only the default class is set. Fig. 5(d) shows d_q of a sample query with $ds = 8$.

d_q describes recent delta patterns to the model, computed relative to a min_{lba} value using (2). To enrich the LBA-based context and enhance delta modeling, we also one-hot encode $|\mathcal{D}_q|$ and include the table ID of the min_{lba}.

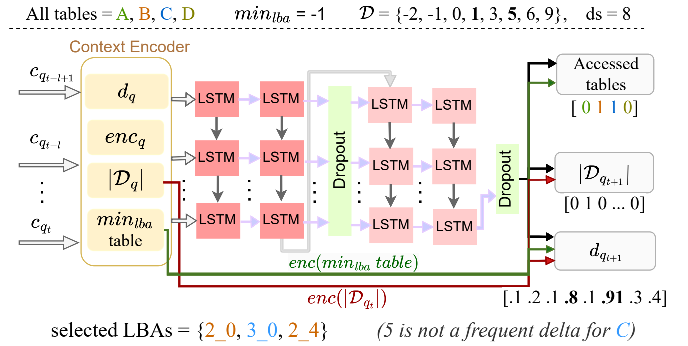

Fig. 6: Predictor architecture and sample outputs. Context features are encoded separately and merged into the LSTM layers.

D. Delta Modeling

1) Input: GrASP uses the last $n_{lookback}$ query contexts to capture temporal dependencies across queries. This accounts for the impact of one query's results on subsequent queries formation, particularly in exploratory workloads [10], [46].

2) Output: The multi-task model predicts three aspects of the next query q_{t+1} to form $L_{q_{t+1}}$: the accessed tables, the delta classes for values in $\mathcal{D}_{q_{t+1}}$, and their count.

a) Accessed Tables: We encode accessed tables as a bitmap where each field corresponds to a specific table and bit j=1 iff the query touches table j. GrASP estimates the probability of each table being accessed in q_{t+1}, and converts these probabilities into binary values using a threshold τ.

Since workloads evolve, a static threshold is unsuitable. GrASP updates τ at each step using the minimum predicted probability of accessed tables and the count of false negatives, as defined in (3) with $\alpha = 0.01$ (evaluated in §VI-E). This adaptation prioritizes recall to capture all positive tables.

$$\tau = \begin{cases} \tau - \alpha \times |\text{False Negatives}|, & \text{if } \tau < \min(\text{P(Accessed Tables)}) \\ \tau + \alpha/10, & \text{otherwise} \end{cases} \tag{3}$$

b) Delta Count: SOTA prefetchers prefetch a fixed number of blocks after each access, such as 9 blocks per access [1], [2] or 40 partitions (128 blocks each) per query [3]. In contrast, GrASP predicts the delta count ($|\mathcal{D}_q|$) for the next query and dynamically adjusts the prefetch size to better accommodate variable query result sizes.

Accurately predicting $|\mathcal{D}_q|$, is challenging as it depends on factors such as data distribution and query predicates. Moreover, since delta predictions are not perfectly precise, some extra blocks must be prefetched to ensure a high performance. To address this, GrASP scales the predicted delta count by a factor k_{dc} and prefetches $\widehat{|\mathcal{D}_q|} \times k_{dc}$ blocks.

c) Delta Values: The final output predicts the likelihood of each delta class in $d_{q_{t+1}}$. The predicted deltas are combined with the predicted tables to generate candidate table-based deltas. However, using all such combinations is inefficient, and complex filtering is impractical within the limited time before the next query. Thus, GrASP filters deltas based on their occurrence frequency in the system history. It maintains a per-table lookup of frequent deltas, dynamically updated with

recent queries, ensuring that only commonly observed deltas are used to construct LBAs for prefetching.

3) Model Architecture: To capture temporal dependencies in delta patterns, GrASP employs an LSTM-based architecture. LSTM networks are a type of RNNs capable of learning sequential dependencies in data by maintaining an internal state over time. This makes them particularly suitable for modeling sequences in prediction tasks, including prefetching [1], [3], [18], [21]. We also evaluated other sequence modules (GRU, TCN, and bidirectional variants) across diverse workloads and found that LSTM consistently delivers the best, or among the best, performance.

Although more complex models like Transformers have become popular, LSTMs remain a strong choice for systems requiring faster and simpler training and inference. In addition to the LSTM-based model, we implemented and evaluated GrASP using two other models, a three-layer MLP network and a two-layer CNN. Results show that the LSTM architecture achieves comparable or better recall while prefetching up to 18% fewer blocks, demonstrating its ability to capture temporal patterns effectively for accurate and efficient prefetching.

The prediction model in Fig. 6 takes $\langle c(q_i) \rangle_{i=t-l}^{t}$ and compresses each context component with time-distributed Dense layers. The resulting embeddings are concatenated into a sequence of 128-dimensional vectors and passed to the LSTM layers. The LSTM output summarizes the recent workload and serves as a shared representation for the Dense heads, each producing a certain output with task-specific inputs.

The delta count is produced by a Dense layer with Softmax activation, which additionally takes in the encoding of the last $|\mathcal{D}_q|$. Accessed tables and delta values are predicted by Dense layers with sigmoid activation that additionally incorporate the embedding of the min_{lba} table ID and d_q, respectively.

4) Training Configuration: Binary cross-entropy (BCE) suits multi-label classifications such as our prefetching problem since it treats each label as an independent binary decision, optimizing predictions for each label separately [1], [3], [19]. However, the prefetching task usually faces significant class imbalance that necessitates adjustments to this loss function.

a) Class Imbalance Challenge: In query workloads, the blocks are accessed unevenly. Our delta analysis (§III-C) reveals that even among the frequent deltas, certain values occur more often. This leads to an uneven class distribution, where majority classes are overrepresented and minority classes are sparse. Predicting frequent classes becomes easy, while minority classes are treated as harder cases.

To address this imbalance, we employ Focal Binary Cross-Entropy Loss (*FL*), which extends standard BCE by adding a modulating factor that down-weights easy examples and focuses training on hard, misclassified ones. *FL* is defined as:

$$FL(\hat{y}) = -\alpha(1-\hat{y})^{\gamma} log(\hat{y}) \tag{4}$$

$\hat{y}$ is the predicted probability for the true class; $\gamma > 1$ reduces the loss contribution from easy cases, focusing on harder ones; α adjusts the overall loss contribution of the class, with higher α increasing the weight of the minority class in the total loss.

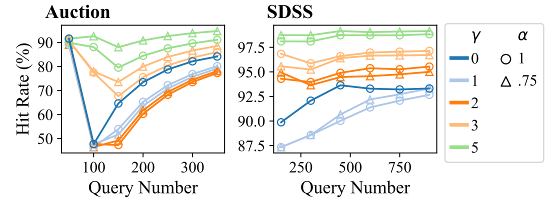

Fig. 7: *FL* setup impact on hit rate of 100-query windows.

Fig. 7 presents the hit ratio under different *FL* setup, where α=1, γ=0 shows plain binary cross-entropy (BCE) loss. Poor BCE performance stresses the need for better handling of class imbalance. Increasing γ improves hit ratio but also causes greater training and prediction instability. Although γ=5 has the highest hit ratio, we select γ=3 for a better trade-off between accuracy and stability.

b) Overfitting Challenge: Imbalanced classes can cause overfitting by making the model overly biased toward the frequent classes, resulting in poor generalization for the minority classes. Additionally, using *FL* loss, the model may overemphasize the minority classes and overfit to rare cases, reducing overall performance. To mitigate this, we incorporate dropout layers to regularize learning, use a low learning rate for stable convergence, and employ a large batch size to reduce gradient noise and enhance generalization.

E. Tune and Generalize

System tuning is crucial in dynamic environments where both data and workloads evolve. GrASP adapts its components to accommodate these changes, ensuring stable performance.

1) Block Encoding: The IPCA module identifies shifts in data distribution by comparing the cosine similarity of principal components before and after fitting new data. Small data batches typically leave similarities greater than 0.8. For similarities below 0.8, GrASP fine-tunes the table's autoencoder on the new data and re-encodes only the new blocks, leaving prior encodings intact to save time. Query semantics help mitigate any inconsistencies and maintain performance.

2) Deltas: Frequent deltas change with workload shifts. The tuner tracks delta frequencies and updates the lookup tables. After l_{tune} queries, GrASP refreshes $\mathcal{D}$ with the most recent frequent deltas, while keeping the vocabulary size fixed at ds. Since modifying $\mathcal{D}$ alters the model's output, GrASP fine-tunes the model by freezing all layers but the final dense layers and retraining on the recent workload using 15 epochs, with a low learning rate.

3) Generalizability: GrASP's tuning capabilities allow it to be trained on a smaller dataset and deployed on a much larger one. It gradually adjusts the IPCA and autoencoders to the new data and updates the prediction model to handle new delta values. Larger databases typically have a wider range of frequent deltas (§ III-C), requiring a larger output size for the prediction model compared to the training dataset. To address this, GrASP includes ds void classes, initially unassigned, during training so newly detected deltas can be mapped to these classes after deployment, enabling faster fine-tuning and better adaptation to the larger database.

F. Putting It All Together

Consider query q_t in Fig. 5. GrASP collects L_{q_t} and computes Δ_{q_t} using $\min(L_{q_{t-1}})$ (Fig. 2-c). It generates r_{q_t} by aggregating the embeddings of the blocks from tables A and B (Fig. 5-b). For s_{q_t}, it activates bits for the query type and the accessed tables, encodes the filter conditions after stripping literals, and keeps the join segments zero (Fig. 5-c). d_{q_t} is constructed by marking the deltas present in $\mathcal{D}$ (Fig. 5-d), and the table of $\min(L_{q_{t-1}})$ is encoded as a one-hot vector.

Using these, GrASP predicts the next tables {B, C} and deltas {1, 5} (Fig. 6), combines them, filters low-frequency deltas, and fetches candidate LBAs into the cache in advance.

V. Experiment Settings

GrASP is evaluated across a wide range of real-world and benchmark datasets with analytical read-only workloads and mixed read/write query streams (hybrid). This section explains the experimental setting and the used databases.

A. Implementation and Configurations

GrASP is implemented in python using TensorFlow/Keras framework [47]. LSTMs are configured with 64 cells and trained in batches of 128, using early stopping on delta prediction loss (validated on 10% of the training data) or a maximum of 25 epochs. The prediction model is trained using FL loss ($\alpha = 0.75$, $\gamma = 3$), while the autoencoders use mean squared error; both are optimized with Adam [48], using learning rates of 0.0001 and 0.001, respectively.

GrASP is deployed on PostgreSQL, and uses pg_prewarm to fetch blocks by their CTID as a background task. After each query, it selects candidate blocks, computes contiguous CTID ranges, and issues prewarm commands. It terminates early if a new query arrives to avoid interfering with its I/O. Table-based LBAs translate directly to CTID ranges usable by pg_prewarm. For example, to fetch LBA 2_4 in Fig. 6, we issue `pg_prewarm(B, from=4, to=5)`.

Unless stated otherwise, experiments are configured as follows: cache size = 8GB for datasets larger than 21 GB and 4 GB otherwise, $ds = 1500$, block size = 32 kB, $lb_{size} = 32$ blocks, $n_{lookback} = 2$, table $\alpha = 0.01$, and $k_{dc} = 25$. All values are selected based on our sensitivity analysis in §VI-E.

We evaluate GrASP on single-node PostgreSQL, and leave distributed integration to future work. In distributed deployments, the core prediction task remains the same: inference and prefetching can run locally per node, with periodic synchronization of model parameters and delta statistics.

TABLE IV: Datasets and Workload Summary

Property	SDSS	Genomes	Birds	TPC-H Skew	TPC-C	Auction	Wiki
Scale Factor	-	-	-	30	250	50	100
Size (GB)	155	10	8	70	25	16	21
Tables	95	13	6	8	9	16	9
Read-only queries	100%	100%	100%	100%	8%	55%	92%
Avg($\lvert L_q \rvert$)	9.74	20.7	7.3	29699	2.44	42.9	48.5
Min($\lvert L_q \rvert$)	1	1	1	11	1	1	1
Max($\lvert L_q \rvert$)	643	607	201	125000	108	12500	3171
Train workload	220k	10k	10k	10k	15k	100k	15k
Test workload	1000	450	300	50	500	400	600

Hardware. Experiments run on an Ubuntu server equipped with 48 Core at 2.4GHz, 1.1TB RAM, 10K RPM disk, and one 16GB NVIDIA V100 GPU. To isolate prefetching effect, the operating system cache is flushed after each query.

B. Datasets and Workloads

Table IV lists our datasets and their workload characteristics.

1) Analytical: Three real-world datasets are used for analytical tests: SDSS, Birds, and Genomes. **SDSS (read-heavy)** is a subset of the seventh Data Release (DR7) of Sloan Digital Sky Survey [49] extended from MyBestDR7 [50] using SciScript library [51]. **Birds** and **Genomes** are datasets from the SQLShare project [52], containing primarily textual data on bird species and genomic information, respectively.

2) Hybrid: We use Benchbase [53] to generate three benchmarks: **TPC-C (update-heavy)**, AuctionMarket (**Auction** or **read-update balanced**), and Wikipedia (**Wiki**). To get their test workloads, we run Benchbase with its default settings for 2 minutes and collect all executed queries.

3) Generalized: To evaluate generalization, we use the databases in Table IV as test targets and train on smaller versions of each dataset. Training sizes are: 16GB and 90GB for SDSS, scale factors (SF) 1, 10, 50, 100, 150, 200 for TPC-C, SF 1, 10, 25 for Auction, and SF 1, 10, 25, 50 for Wiki.

4) Skewed: For completeness, we evaluate GrASP on skewed data using **TPC-H Skew** benchmark [54]. We generate four datasets with SF = 30 and zipf factors from 0.5 to 3.

C. Baselines

GrASP is compared against traditional prefetchers used in mainstream DBMS and SOTA learning-based data prefetchers.

- **Lookahead (LA)** [5]: A simple prefetcher, used in many DBMSs, sequentially fetches blocks after the accessed ones.
- **Random Readahead (RandR)** [6]: If a predefined number (l_{RR}) of an extent blocks are accessed within its LBA trace window, the prefetcher fetches the entire extent.
- **Naïve prefetcher** [8]: Fetches blocks by repeatedly adding the most frequent delta to the last accessed LBA.
- **SGDP** [19]: This SOTA prefetcher models interactions among delta streams with a weighted graph and learns delta patterns using a gated graph network (GGNN).
- **SeLeP** [3]: This SOTA database prefetcher partitions and fetches blocks based on data semantics.

Since GrASP's prefetch size is dynamic, we bound it to k blocks for fair comparison. LA, Naïve, and SGDP are extended to prefetch k blocks instead of one. The RandR model, originally implemented in MySQL, uses default settings. SeLeP fetches $k/pSize$ partitions, where *pSize* is the partition size; we use *pSize*=128, reported to balance cache use and performance.

D. Metrics

We evaluate three metrics. Hit ratio (6) is the proportion of cache hits to total accesses, reflecting overall cache effectiveness. Prefetch recall (7) assesses the accuracy of immediate predictions (blocks prefetched for the next time step). Since a system with no prefetcher (NP) achieves some hits through

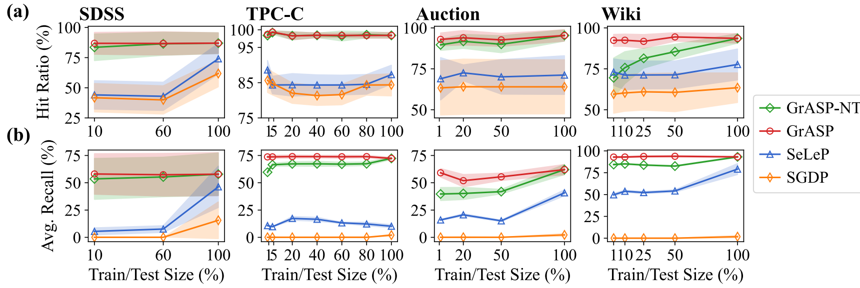

Fig. 8: (a) Hit ratio and (b) average recall with 95% confidence interval in generalization tests. X-axis represents the size ratio of the train and test datasets.

TABLE V: Average Miss Coverage of GrASP and Baselines on Generalized Datasets and Two Selected Size Ratio.

Dataset	size rate	GrASP	GrASP-NT	SeLeP	SGDP
SDSS	10%	**79.41**	75.42	9.59	7.23
	60%	**78.93**	78.66	7.6	4.6
TPC-C	20%	86.35	**87.1**	1.33	1.03
	60%	**87.62**	87.6	1.16	1.56
Auction	20%	**77.97**	69.77	36.88	2.75
	50%	**71.42**	63.59	18.88	2.7
Wiki	10%	**84.18**	55.42	40.31	3.74
	50%	**88.27**	70.68	33.5	2.1

block reaccess, miss coverage (8) measures the fraction of NP cache misses eliminated by the prefetcher, isolating its impact.

$$\text{Hit Ratio} = \frac{\text{Hits}}{\text{Hits} + \text{Misses}} \quad (6) \qquad \text{Recall} = \frac{\text{Correct Prefetches}}{\text{Accessed Blocks}} \quad (7)$$

$$\text{Miss Coverage} = \frac{\text{Misses}_{NP} - \text{Misses}}{\text{Misses}_{NP}} \quad (8)$$

We also evaluate the prefetcher's runtime impact by comparing total execution time, total I/O time, throughput, and 95th percentile latency across workloads with simulated delays.

VI. Experimental Results

We evaluate GrASP through the following key questions:

- How does GrASP generalize its prediction on an enlarged dataset compared to other learning-based methods?(§VI-A)
- How does GrASP improve database performance compared to traditional and SOTA baselines?(§VI-B)
- How does GrASP perform on skewed datasets and shifting workloads?(§VI-C1)
- What is the time complexity of GrASP?(§VI-D)
- How do hyperparameters affect the performance?(§VI-E)

A. Generalization Experiment

We assess generalization of learning-based prefetchers on four datasets by training them on smaller databases and testing on larger ones (§V-B3). After training, GrASP and SeLeP are fine-tuned on 5000 queries from the target database; GrASP-NT (no tuning) is included to isolate the impact of tuning.

Fig. 8 shows (a) hit ratio and (b) average recall across varying train-to-test dataset size ratios, with k=50. A 100% size ratio indicates training and testing on the same database, where prefetchers achieve their best performance. Results are averaged over 8 sessions with 95% confidence interval (CI).

Fig. 8(a) shows that GrASP consistently achieves the highest hit ratio across datasets, reaching over 92% on hybrid workloads even with minimal training. Its CI remains below 10% in all cases except SDSS, where greater variability in query templates and access patterns leads to wider variation.

While collecting block requests from the training datasets, we observed that the DBMS generates different execution plans for similar queries across dataset scales, even with identical indexes and schema. For instance, the Auction access patterns in size ratios 20% and 100% are similar, while size ratios 50% and 100% differ. Despite these variations, GrASP maintains robust performance across scales.

GrASP-NT attains hit ratios close to GrASP in most tests, as frequent deltas in smaller training datasets often overlap with those in the target dataset, enabling effective prefetching. Also, similar data distributions and access patterns also preserve query representations and block encodings, supporting generalization without fine-tuning. Consequently, GrASP shows stable performance and narrow CIs, largely unaffected by size ratio. By contrast, GrASP-NT varies more and has wider CIs—especially on Wiki—where diverging delta distributions and high access rates make prediction without tuning harder.

Despite its adaptability, SeLeP underperforms relative to GrASP, especially in hybrid workloads. Even at a 100% size ratio, some test blocks are absent from the training set, limiting SeLeP's ability to generalize and achieve high accuracy. This limitation persists across all ratios, contributing to consistently low performance and a similar CI. The 5000 tuning queries are also insufficient to capture complex access patterns, especially in SDSS. In contrast, GrASP effectively uses these queries to refine delta values and adjust predictions.

SGDP, based on delta modeling, is designed to generalize as the dataset size increases. However, it consistently fails to scale and struggles with accuracy even at a 100% size ratio due to its reluctance to prefetch under uncertainty. This highlights the limitation of relying solely on LBA-based information for access prediction in complex workloads. Additionally, we observed its recursive delta prediction approach is inefficient and significantly increases prediction time.

Fig. 8(b) shows that GrASP achieves the highest average recall with a narrow CI. At a prefetch size of k=50, this metric reflects the average per-query hit ratio in a 400MB cache. Thus, higher recall is critical when the memory budget for prefetching is limited. Achieving a high hit ratio with low recall means that cache hits are primarily due to blocks prefetched in previous steps that were not promptly accessed.

Table V shows miss coverage of prefetchers in two different size ratios. GrASP consistently outperforms the baselines, achieving an average miss coverage of 83.75% with fine-tuning and 71.25% without it. In contrast, SeLeP and SGDP fail to surpass 40% miss coverage. A low miss coverage indicates an inability to anticipate and prefetch upcoming accesses not already present in the cache.

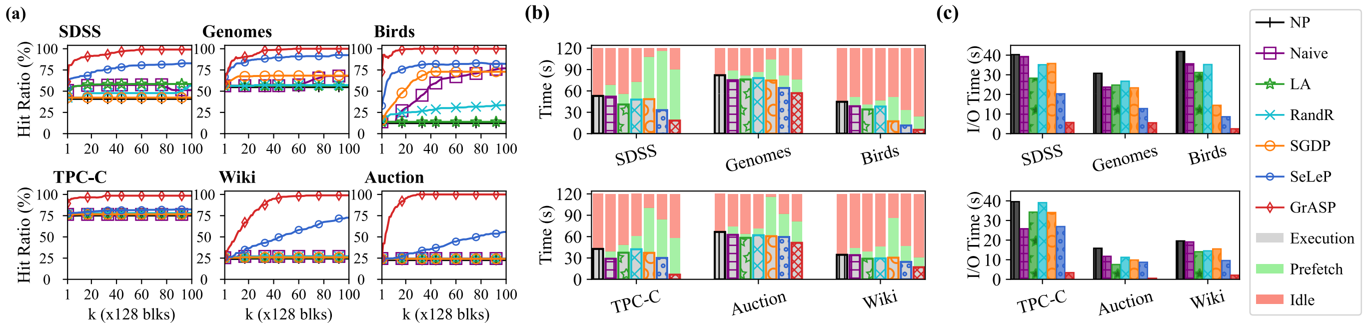

Fig. 9: (a) Hit ratio across different prefetch sizes, k; (b) breakdown of execution, prefetch, and idle time in simulations at 5 qps with $k = 50$; and (c) I/O time for the same simulations. Analytical workloads are on top and hybrid ones are below.

TABLE VI: Average Recall (Rec) and Miss Coverage (MC) in Analytical and Hybrid Workloads With $k = 50$.

Method	SDSS		Genomes		Birds		TPC-C		Auction		Wiki	
	Rec	MC	Rec	MC	Rec	MC	Rec	MC	Rec	MC	Rec	MC
Naïve	2.4	15.3	1.2	26.4	1.12	41.7	2.6	8.1	0	0.2	0	1.02
LA	9.6	27.9	1.3	3.3	1.9	12.1	0.8	1.02	2.2	3.9	7.9	2.24
RandR	3.8	9.3	1.18	2.06	27.6	15.3	0	0.4	0	0.6	8.23	2.3
SGDP	1.5	1.97	1.7	27.5	4.1	66	1.4	1.97	1.6	1.3	0.9	0.6
SeLeP	38.3	60.3	82.8	76.65	16.8	78.7	7.7	17	42.1	11.4	77.9	17.2
GrASP	**90.8**	**91.5**	**83.6**	**89.7**	**99.1**	**99.8**	**72.2**	**87.4**	**61.9**	**95.3**	**91.9**	**91.2**

B. Analytical and Transactional Experiment

This section compares GrASP with baselines on analytical and hybrid workloads over various performance metrics.

1) ***Correctness and Coverage:*** Fig. 9(a) shows hit ratios across prefetch size k, guiding our choice of k for other tests. Table VI reports average recall and miss coverage at k=50.

Analytical workloads, which do not modify data, are generally easier to predict. In smaller datasets like Birds, LBA-based prefetchers perform well. With larger datasets or higher block access rates, LBA-based and traditional methods struggle to fill the cache effectively. In contrast, semantic prefetchers excel, with GrASP consistently outperforming all baselines.

Hybrid workloads challenge prefetchers, especially LBA-based and traditional ones. While SeLeP struggles with dominant transactional queries, GrASP performs well across all workload types. It achieves its best hit ratio near k=20 for low-access workloads (Birds, TPC-C) and k=40 for high-access ones (Wiki, Auction, Genomes). Since other baselines perform similarly around k=50, we use this value in all evaluations.

Table VI highlights the superiority of semantic prefetchers in terms of prefetching recall and miss coverage, where GrASP is always the best and SeLeP ranks second. However, GrASP shows significantly better performance than SeLeP in most datasets, by up to 83% in recall and 84% in miss coverage.

2) ***Runtime Impact:*** Fig. 9(b) reports simulation time breakdown at 5 queries per second (qps) rate, corresponding to a maximum query interarrival delay of 250 ms. It shows total execution time (patterned gray), combined prediction and prefetch time (green), and system idle time (pink). The simulations run until the full workload is processed, while prefetching is *non-blocking* and stops as soon as the next query arrives.

GrASP yields the largest execution time reduction across all datasets. Compared to NP, it saves over 85% on Birds and TPC-C, 65% on SDSS, 50% on Wiki, and up to 30% on others. It outperforms SeLeP by 9–55%, with the largest gains on TPC-C (55%), SDSS (28%), Wiki (22%), and Birds (14%).

GrASP's prefetch time is lower than other learning-based methods, as it estimates and adjusts prefetch size (bounded by k) rather than using a fixed size. In our tests, GrASP prefetched the same or fewer blocks per query than SeLeP—On average 19.13% fewer overall and up to 93.42% fewer on some queries—reducing prefetch time by 37% on Wiki, 19% on SDSS, and up to 14% on other datasets.

Since a non-blocking prefetcher utilizes idle time *without* adding overhead, the true end-to-end latency excludes prefetch time. Fig. 9(b) shows GrASP effectively using idle periods to prefetch relevant blocks, resulting in lower end-to-end latency.

Prefetching reduces query response time by reducing I/O delays, as computation time is unaffected. To isolate the impact on I/O overhead, we also report I/O time for the same simulations in Fig. 9(c), where a value of zero indicates the ideal case for I/Os with all data served from the cache.

Across all tests, GrASP achieves the lowest I/O time, reducing I/O delays of an NP system by up to 96% in analytical workloads and up to 94% in hybrid ones. SeLeP ranks second but struggles with hybrid workloads, achieving less than a 51% I/O reduction even in the ones with mainly analytical queries. Traditional prefetchers perform close to NP, offering at most a 48% improvement in I/O time.

Although higher miss coverage generally lowers I/O time, absolute gains depend on where blocks reside. Some blocks need more movement or processing, leading to variation in I/O times for systems with similar statistics.

Moreover, prefetchers primarily reduce cache misses, so relative improvement trends are expected to be consistent across storage backends, though absolute runtimes vary with device characteristics. We confirmed this by re-running NP and GrASP on both HDD and NVMe under same workloads.

Throughput and latency. We extend our runtime analysis by evaluating the prefetcher impact on system throughput and 95th percentile query latency across varying query arrival rates. Fig. 10 shows results from 120s simulations on the Auction hybrid high-access workload and the SDSS analytical workload (Table IV). The tested rates correspond to maximum interarrival delays (d) ranging from 25 to 250 ms, with actual delays sampled from $[d/2, d]$ and biased toward d, averaging $7d/8$—similar to the skewed arrival in [55].

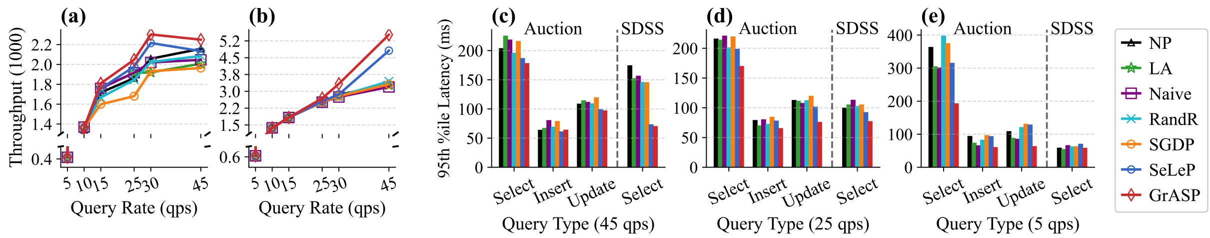

Fig. 10: Throughput on (a) Auction and (b) SDSS; 95th percentile latency per query type at (c) 45 qps (25 ms max delay), (d) 25 qps (50 ms max delay), and (e) 5 qps (250 ms max delay). Higher throughput and lower latency are better.

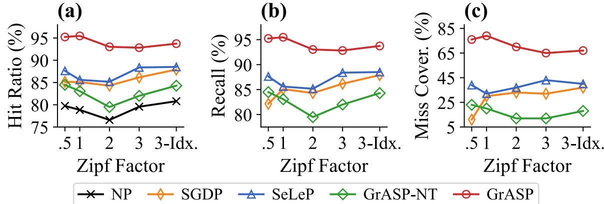

Fig. 11: (a) Hit ratio, (b) recall, and (c) miss coverage in original and indexed TPC-H Skew with variable z and k=1500.

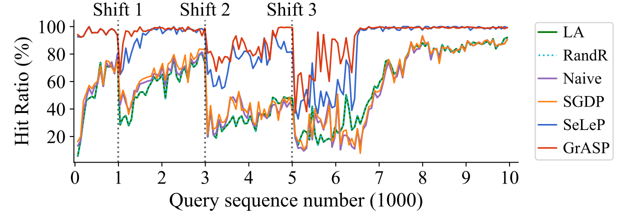

Fig. 12: Hit ratio for consecutive non-overlapping 60-query batches in a shifting SDSS workload with k=50.

Fig. 10(a) shows Auction throughput. At low query rates, methods achieve similar throughput because simple or repeated queries complete quickly, balancing more complex or longer-running ones. Nonetheless, Fig. 9(b) reveals large execution time differences for the same query count, with GrASP completing them up to 85% faster. As load increases, some prefetchers degrade throughput by fetching irrelevant blocks, falling below NP, while GrASP maintains the highest throughput. At 45 qps, the system saturates and throughput drops due to minimal available time (<25 ms) for prefetching.

The throughput results of the SDSS in Fig. 10(b) follow a similar trend. However, due to the lower block access rate in SDSS, GrASP leads even at 45 qps, executing 20–85% more queries. At higher qps however, all prefetchers fail to keep up.

Fig. 10(c-e) reports 95th percentile latency per query type (Auction has few DELETEs) at 45, 25, and 5 qps, matching d values of 25, 50, and 250 ms. GrASP consistently yields lower latency with up to 57% for selection and 32% for transactional queries, even under high loads. In contrast, SeLeP's latency is less stable, and some methods degrade performance by polluting the cache. Note that at higher qps, prefetchers execute different numbers of queries, which can affect their latency.

C. Adaptivity Experiment

Real-world database workloads often exhibit non-uniform data distributions and changing query patterns, as user interests shift over time [10], [56]. In this section, we evaluate how GrASP performs under these realistic and dynamic conditions.

*1) **Skewed Dataset**:* This subsection evaluates the prefetchers on skewed TPC-H datasets with varying Zipf factors (z). Models are trained on a dataset with SF=10 and z=0.5, and tested on datasets with SF=30 and z={0.5, 1, 2, 3}. In TPC-H Skew, higher z values correspond to more pronounced skewness, where some customers place more orders and certain parts are ordered more frequently. The test workloads run similar queries across datasets, with GrASP and SeLeP fine-tuned using 500 queries from the target dataset.

Fig. 11 shows (a) hit ratio, (b) recall, and (c) miss coverage of the skew tests. Since TPC-H queries access significantly more blocks than other workloads, achieving high performance demands prefetching many more blocks (k =1500).

Fig. 11 shows that GrASP, after tuning on a few queries, achieves over 93% hit ratio and recall, and up to 80% miss coverage. Data skewness does not drastically impact GrASP as it includes LBA details in its input context, while SeLeP, relying solely on semantics, is more affected and performs similarly to the LBA-based SGDP. Since both semantic and LBA contexts shift in these tests, GrASP-NT performs significantly worse and fine-tuning is critical.

For completeness, we evaluate the methods under a modified physical schema using TPC-H Skew (z=3), augmented with indexes derived from HMAB [31] ("3-Idx." in Fig. 11). By reducing the number of blocks touched, indexes raise block reaccess rates and improve prediction accuracy, enhancing performance for all methods (NP included). GrASP remains the best and stays stable, owing to its plan-agnostic encoding.

*2) **Shifting Workload**:* To evaluate adaptivity, we simulate evolving workloads on the SDSS dataset. The cache is first warmed up, followed by three staged shifts: at sequence number (SN)=1k, 25% of blocks are unseen; at SN=3k, 40% of blocks, 50% of tables, and 50% of templates are new; and at SN=5k, all tables change and unseen blocks are accessed using entirely new templates. GrASP updates delta classes every l_{tune} = 0.5k queries, fine-tuning only if deltas change, with 5.54s average overhead. SeLeP also tunes every l_{tune} queries, taking 46.42s due to the costly repartitioning.

Fig. 12 plots hit ratios over batches of 60 consecutive queries up to SN=10k. GrASP drops less sharply and recovers faster, maintaining a relatively consistent performance—especially in the first shift, where it improves even before tuning. After the final shift, the fully unseen workload increases uncertainty, leading GrASP to skip prefetching for some queries, while SeLeP issues inefficient random

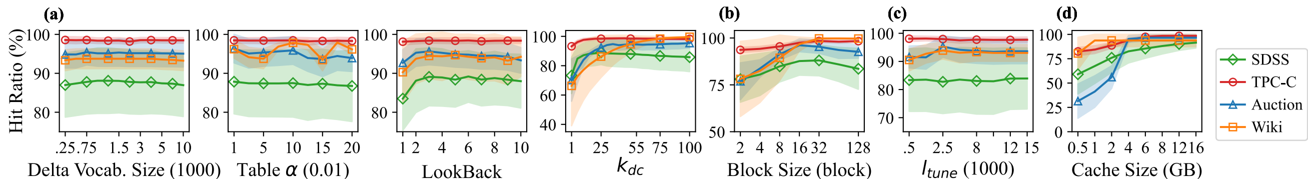

Fig. 13: Effect of (a) model config, (b) block size, (c) tuning query count, (d) cache size on GrASP's hit ratio with 95% CI.

TABLE VII: Time overheads of model initialization (split into sub-operations), block encoding, model training, delta prediction, block fetching, and tuning with $l_{tune} = 5000$.

Operation		SDSS	Wiki	Auction	TPC-C	TPC-H Skew
One-off model initialization	Block encoding of 100 blocks (s)	20.71	3.6	59.17	49.04	75.89
	Model training (s)	570.17	27.98	37.41	250.86	27.78
	Other operations (s)	561.94	27.96	17.78	144.63	132.82
Delta prediction per query (ms)		3.06	6.57	4.57	2.4	27.29
Block prefetching per query (s)		0.25	0.06	0.36	0.15	2.2
Fine-tuning (s)		23.25	27.14	21.6	21.8	40.21

prefetches that occasionally succeed by chance. Upon tuning at SN=5.5k, GrASP's average hit rate rises from 53% to 81%, while SeLeP peaks below 75% despite tuning. Due to the stochastic nature of the workload, all prefetchers struggle to stabilize until SN=7k, where only GrASP and SeLeP converge.

D. Time Analysis

Table VII reports GrASP's time overhead across datasets.

*1) **One-off Model Initialization**:* This phase includes all one-time setup steps before deployment, such as block encoding, delta set collection, context generator setup, and model training. Table VII lists them separately to highlight the main costs of encoding and training. Initialization time mainly scales with training workload size (Table IV) and query complexity, with SDSS and TPC-H Skew showing the highest overheads.

Block encoding cost depends on schema characteristics. Datasets with fewer columns (Wiki) or mostly numeric values (SDSS) incur lower cost. **Training** cost reflects model complexity: compared to GrASP, SeLeP's deeper model with extra Dense layers, and SGDP's complex graph-based model, increase training time by up to 9.7× and 418.4×, respectively. For reference, SeLeP's training times are 5573, 60, 30, 1369, and 31 s, while SGDP's are 4340, 147, 1245, 449, and 11715 s.

*2) **Delta Prediction**:* This involves context creation and model inference, both influenced by query complexity and block selectivity. Thus, TPC-H Skew, which accesses 30k blocks with complex queries, exhibits the highest prediction time. However, this time stays within the millisecond range and does not impact database interactivity.

*3) **Block Prefetching**:* It computes LBAs from predicted deltas and retrieves them from storage. Latency decreases if the block is already cached and grows with the number of fetches. Except for TPC-H Skew with high block access rate, prefetching completes in under 400 ms, ensuring interactivity.

*4) **Tuning**:* Fine-tuning time includes delta set adjustment and prediction model tuning. TPC-H Skew, with its higher query encoding time, has the longest tuning time. However, tuning takes under a minute and can be run asynchronously.

Unlike SeLeP, GrASP does not require encoding new blocks during fine-tuning. SeLeP must encode all newly inserted blocks and assign them to partitions at each tuning event, incurring overheads orders of magnitude higher than GrASP.

E. Sensitivity Analysis

To assess parameter impact, we evaluate GrASP under various settings on the SDSS and hybrid workloads. The hit ratios shown in Fig. 13 confirm that the settings in §V-A ensure stable and robust performance across datasets and workloads.

Model Parameters impact is shown in Fig. 13(a). Small *delta vocabulary size* (ds) fails to cover all frequent deltas, reducing performance, while large ds adds too many classes, making accurate predictions harder. Higher *table* α values distort τ, which degrades the performance. Using query history (i.e., $n_{lookback>1}$) improves predictions, but long histories increase model complexity and reduce performance. For datasets with high block access rates, a larger k_{dc} boosts performance, but overly high values risk selecting irrelevant blocks.

Block size (lb_{size}). Increasing block size reduces the number of deltas, improving prediction accuracy (Fig. 13(b)). However, very large lb_{size} populate the cache with unused data.

Tuning query count (l_{tune}). Fig. 13(c) shows GrASP adapts deltas and predictions after tuning on at least 2500 queries, or as few as 500 queries for datasets with low access like TPC-C.

Cache Size is evaluated in Fig. 13(d). Larger caches preserve longer access histories, increasing block reaccess chances and extending the impact of prefetches. GrASP maintains a high hit ratio with a 2GB cache across datasets from 16GB to 155GB.

VII. Conclusion

This paper presents GrASP, a learning-based semantic prefetcher designed to enhance database interactivity by leveraging both LBA patterns and data semantics. GrASP combines queries LBA-Delta with their encoded semantics to predict future delta values and optimize prefetching hit ratio across a diverse range of workloads. Our evaluation on analytical and transactional workloads demonstrates that GrASP significantly improves performance, outperforming SOTA methods with up to 45% higher hit ratio, 60% lower I/O time, and 55% lower execution latency. Additionally, our experiments on enlarged datasets demonstrate that GrASP, through delta modeling and lightweight fine-tuning, generalizes its high performance to datasets up to 250× larger, and with different skewed data distributions—capabilities not achievable by SOTA prefetchers.

Acknowledgment

This work is partially supported by the Australian Research Council (ARC) via Discovery Early Career Researcher Award DE230100366, and Google Foundational Science 2025 Award.

References

[1] Y. Chen, Y. Zhang, J. Wu, J. Wang, and C. Xing, "Revisiting data prefetching for database systems with machine learning techniques," in *37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2165–2170.

[2] L. Battle, R. Chang, and M. Stonebraker, "Dynamic prefetching of data tiles for interactive visualization," in *Proceedings of the International Conference on Management of Data, SIGMOD Conference 2016, USA, 2016*. ACM, 2016, pp. 1363–1375.

[3] F. Zirak, F. M. Choudhury, and R. Borovica-Gajic, "Selep: Learning based semantic prefetching for exploratory database workloads," *Proceedings of the VLDB Endowment*, vol. 17, no. 8, pp. 2064–2076, 2024.

[4] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki, "SCOUT: prefetching for latent feature following queries," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1531–1542, 2012.

[5] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems (TODS)*, vol. 3, no. 3, pp. 223–247, 1978.

[6] M. Opdenacker and F. Electrons, "Readahead: time-travel techniques for desktop and embedded systems," in *Proc. of the 2007 Ottawa Linux Symposium*, vol. 2, 2007, pp. 97–106.

[7] A. Ki and A. E. Knowles, "Stride prefetching for the secondary data cache," *Journal of systems architecture*, vol. 46, no. 12, pp. 1093–1102, 2000.

[8] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, 1994, p. 223–232.

[9] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber, "Scientific data management in the coming decade," *SIGMOD Record*, vol. 34, no. 4, pp. 34–41, 2005.

[10] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou, "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds," *VLDB*, vol. 4, no. 12, pp. 1474–1477, 2011.

[11] A. Hospital, F. Battistini, R. Soliva, J. L. Gelpí, and M. Orozco, "Surviving the deluge of biosimulation data," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 10, no. 3, p. e1449, 2020.

[12] P. Kakria, N. Tripathi, and P. Kitipawang, "A real-time health monitoring system for remote cardiac patients using smartphone and wearable sensors," *International journal of telemedicine*, vol. 2015, no. 1, p. 373474, 2015.

[13] R. Guha, D.-T. Nguyen, N. Southall, and A. Jadhav, "Dealing with the data deluge: handling the multitude of chemical biology data sources," *Current protocols in chemical biology*, vol. 4, no. 3, pp. 193–209, 2012.

[14] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 66–76.

[15] A. Tamersoy, B. Xie, S. L. Lenkey, B. R. Routledge, D. H. Chau, and S. B. Navathe, "Inside insider trading: Patterns & discoveries from a large scale exploratory analysis," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2013, pp. 797–804.

[16] F. Atefeh and W. Khreich, "A survey of techniques for event detection in twitter," *Computational Intelligence*, vol. 31, no. 1, pp. 132–164, 2015.

[17] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. Association for Computing Machinery, 2015, p. 1310–1321.

[18] C. Chakraborttii and H. Litz, "Learning i/o access patterns to improve prefetching in ssds," in *ECML/PKDD*, 2020, pp. 427–443.

[19] Y. Yang, R. Li, Q. Shi, X. Li, G. Hu, X. Li, and M. jie Yuan, "Sgdp: A stream-graph neural network based data prefetcher," *2023 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2023.

[20] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 861–873. [Online]. Available: https://doi.org/10.1145/3445814.3446752

[21] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1924–1933. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[22] M. Grannæs, M. Jahre, and L. Natvig, "Storage efficient hardware prefetching using delta-correlating prediction tables," *J. Instr. Level Parallelism*, vol. 13, 2011. [Online]. Available: http://www.jilp.org/vol13/v13paper2.pdf

[23] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015, pp. 141–152.

[24] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*. ACM, 1997, pp. 252–263.

[25] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 18–27, 2005.

[26] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning internal representations by error propagation," 1985.

[27] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: leveraging query executions to improve index recommendations," in *Proceedings of the International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 1241–1258.

[28] R. M. Perera, B. Oetomo, B. I. P. Rubinstein, and R. Borovica-Gajic, "No dba? no regret! multi-armed bandits for index tuning of analytical and HTAP workloads with provable guarantees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 855–12 872, 2023.

[29] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic, "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, 2021, pp. 600–611.

[30] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2118–2130, 2019.

[31] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic, "Hmab:self-driving hierarchy of bandits for integrated physical database design tuning," *Proceedings of VLDB Endowment*, vol. 16, no. 2, pp. 216–229, 2022.

[32] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1501–1512.

[33] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," *SIGMOD Rec.*, vol. 51, no. 1, pp. 6–13, 2022.

[34] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, 2019.

[35] S. Jain, B. Howe, J. Yan, and T. Cruanes, "Query2vec: An evaluation of nlp techniques for generalized workload analytics," *CoRR*, vol. abs/1801.05613, 2018. [Online]. Available: http://arxiv.org/abs/1801.05613

[36] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, vol. 32. JMLR.org, 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html

[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, pp. 3111–3119, 2013.

[38] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[39] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences 1997*. IEEE, 1997, pp. 21–29.

[40] Y. Kim, Y. Chiu, K. Hanaki, D. Hegde, and S. Petrov, "Temporal analysis of language through neural language models," in *Proceedings of the Workshop on Language Technologies and Computational Social Science@ACL 2014*. Association for Computational Linguistics, 2014, pp. 61–65.

[41] H. He, S. Chen, K. Li, and X. Xu, "Incremental learning from stream data," *IEEE Transactions on Neural Networks*, vol. 22, no. 12, pp. 1901–1914, 2011.

[42] R. Bamler and S. Mandt, "Dynamic word embeddings," in *International conference on Machine learning*. PMLR, 2017, pp. 380–389.

[43] K. Pearson, "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*, vol. 2, no. 11, pp. 559–572, 1901.

[44] C. M. Bishop, *Pattern Recognition and Machine Learning*, 1st ed. New York: Springer, 2007.

[45] T. Halpern and S. Toledo, "Advances in incremental PCA algorithms," in *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017, Part I*, vol. 10777. Springer, 2017, pp. 3–13.

[46] S. Idreos, *Big Data Exploration*. Taylor and Francis, 2013.

[47] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings*, 2015.

[49] K. N. Abazajian, J. K. Adelman-McCarthy, Agüeros *et al.*, "The seventh data release of the sloan digital sky survey," *The Astrophysical Journal Supplement Series*, vol. 182, no. 2, p. 543, 2009.

[50] SkyServer, "MySkyServer: Interactive Astronomy Portal," http://www.skyserver.org/myskyserver/, 2010, accessed: 2023-06-01.

[51] SciScript-Python, "SciServer SciScript-Python Library," https://github.com/sciserver/SciScript-Python, 2016, accessed: 2023-07-01.

[52] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska, "Sqlshare: Results from a multi-year sql-as-a-service experiment," in *Proceedings of the International Conference on Management of Data, SIGMOD Conference 2016*. ACM, 2016, pp. 281–293.

[53] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[54] "TPC-H Skew Dataset for Linux Repository," https://github.com/qwertyfz/tpch-skew-linux.git, 2025, accessed: 2025-02-01.

[55] C. Nuessle, O. Kennedy, and L. Ziarek, "Benchmarking pocket-scale databases," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2019, pp. 99–115.

[56] C. Anneser, A. Kipf, H. Zhang, T. Neumann, and A. Kemper, "Adaptive hybrid indexes," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1626–1639.