Tiresias: Enabling Predictive Autonomous Storage and Indexing

Michael Abebe, Horatiu Lazu, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{mtabebe,hslazu,kdaudjee}@uwaterloo.ca

ABSTRACT

To efficiently store and query a DBMS, administrators must select storage and indexing configurations. For example, one must decide whether data should be stored in rows or columns, in-memory or on disk, and which columns to index. These choices can be challenging to make for workloads that are mixed requiring hybrid transactional and analytical processing (HTAP) support. There is growing interest in system designs that can adapt how data is stored and indexed to execute these workloads efficiently. We present **Tiresias**, a predictor that learns the cost of data accesses and predicts their latency and likelihood under different storage scenarios. Tiresias makes these predictions by collecting observed latencies and access histories to build predictive models in an online manner, enabling autonomous storage and index adaptation. Experimental evaluation shows the benefits of predictive adaptation and the trade-offs for different predictive techniques.

PVLDB Reference Format:
Michael Abebe, Horatiu Lazu, Khuzaima Daudjee. Tiresias: Enabling Predictive Autonomous Storage and Indexing. PVLDB, 15(11): 3126 - 3136, 2022.
doi:10.14778/3551793.3551857

PVLDB Artifact Availability:
The source code and other artifacts have been made available at https://github.com/mtabebe/Adaptive-Storage-Tiresias-and-Proteus.

1 INTRODUCTION

Organizations rely on large amounts of data to make data-driven decisions [10, 48, 49, 54]. To store and query this data efficiently, administrators must make physical design decisions such as whether data should be stored in rows or columns, in-memory or on disk, and which columns to index.

These storage and indexing choices are difficult to make when executing mixed or hybrid transactional *and* analytical (HTAP) workloads because data is accessed concurrently by both transactional (OLTP) and analytical (OLAP) operations [10, 29, 46, 62]. For example, storing data using a row layout provides low latency OLTP operations but results in significantly degraded performance for OLAP queries. This outcome is reversed when using a column layout of data [7, 10].

Consider the *new order* table from the CH-BenCHmark [14]. OLAP queries scan the table to find popularly ordered items, while OLTP transactions insert new orders and update the delivery status of recent orders. Over time, the row representing a placed order goes from likely to be updated (insertion and subsequent update in delivery status) to scan-heavy. Using appropriate tools, the system administrator may discern this general pattern, but they would not know when the shifts in access patterns will occur. Hence, if a system *learns* these different access costs and *predicts* the workload patterns, then it could *predictively adapt* its layout and reduce access latencies. For example, the system could initially store data in a row layout and predictively change it to a columnar layout when it expects updates to that data are unlikely. Similarly, the system could predictively load data items from disk prior to executing scan queries on them. Such predictive changes reduce access latency as the data is in an access-optimized layout.

As another example, consider database cracking that incrementally adapts indexes to the workload by reorganizing data based on accesses. Cracking amortizes the upfront costs of indexing the entire database to indexing data as it is accessed at the expense of increased query latency. If the database system could predict which data would be accessed and when, then by predictively indexing that data, the system could achieve the best of both traditional indexing and cracking through low upfront costs and query latency.

As the above examples illustrate, a database management system (DBMS) needs to be predictive to benefit from adaptive storage changes. In particular, the system should predict (i) when a data item will be accessed, and (ii) the latency (cost) of a given data item access under a specific storage layout. While achieving the goal of being predictive is challenging, the resulting performance benefits from system storage adaptations are highly desirable.

Analytical and transaction processing workloads often exhibit both temporal and spatial locality in data accesses [3, 37, 50, 52, 56, 57]. These trends are typically cyclic and hence predictable. For example, a trace of requests to Wikipedia (Figure 2a) [59] shows that there are *periodic* daily and weekly trends, as well as *short-term* effects such as load spikes. Data accesses exhibit a time-varying skew that follows the sun: workload demands shift following daylight hours when most people are awake. Such time-varying skew is typical in transactional workloads, while users often schedule analytical query workloads to run at regular intervals to generate hourly reports or refresh dashboards every minute [52]. In contrast to temporal locality, spatial locality in a workload arises from accesses to regions of data varying over time. For example in the SkyServer workload [3, 56], users periodically focus on different areas of the sky (Figure 6a). Consequently, a system must model and learn workload patterns to predict future data accesses.

Predicting the latency or cost of data access under different storage layouts poses a challenge. When considering factors such as sort order, data size, selectivity, or columns indexed, the system may have limited data-specific history of prior observed latencies under a particular storage layout. Moreover, to understand the effects of storage layouts on performance, the system must attribute latency

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551857

to specific operations, and not the entire transaction.

In this paper, we present **Tiresias**, a predictor designed to make decisions to predictively adapt storage in DBMSs. We demonstrate Tiresias' benefits for three systems: (i) adapting storage layouts for HTAP workloads in Proteus [7], a distributed HTAP DBMS, (ii) enable predictive cracking in an OLAP DBMS [24], and (iii) enabling automatic indexing in PostgreSQL. We conduct a detailed experimental study to evaluate end-to-end performance and show the benefits of predictive storage adaptation while understanding the trade-offs in using Tiresias.

2 RELATED WORK

Next, we discuss how Tiresias relates to prior work that predicts access arrivals, access costs, and uses predictive DBMS techniques.

2.1 Predicting Access Arrivals

Prophet [58] and QB5000 [37] predict access frequencies using an ensemble learning approach, decomposing predictions into periodic and trend components. These systems learn the periodic component and do not require user configuration. QB5000 combines a linear regression trend predictor with a recurrent neural network (RNN) periodic predictor. Prophet learns the periodic component using Fourier series and the trend using a linear trend with change points. Prophet also incorporates user-defined holiday list to account for irregular recurring trends. Tiresias' hybrid-ensemble access arrival predictor combines a linear predictor with an RNN (Section 5.1.2) and holiday list to adapt storage in a distributed HTAP DBMS, predictively crack data and modify indexes.

In the context of elastically adding or removing database nodes based on predicted future query load, prior work models shifts in query workloads using Fourier transforms to detect peaks and interval analysis to detect periods [27]. P-Store [57] uses SPAR to predict load; however, as we discuss in Section 5, SPAR requires prior knowledge of the workload periodicity.

Some systems use Markov models to predict the set of queries that will be executed in the near future based on the set of recently executed queries [21, 28]. However, these approaches do not predict the number of times a query will execute in a time period.

2.2 Predicting Operation Costs

There are many proposed techniques for predicting the cost of an operation within a database system as part of cost models in query planners. The traditional approach of a cost model relies on a user-defined notion of the relative costs of different operations within the system [11]. Offline learning-based approaches execute operations with different parameters to collect training data, which are used to infer the costs of database operations [33]. Prior work has considered both regression-based techniques and similarity-based clustering to predict operation time [20, 53, 55]. By contrast, the online learning approach continuously learns by updating its models based on observed costs as the system executes transactions [36]. MB2 [38] extends these ideas by decomposing database operations, e.g., flushing a log record and query operators, into independent units to predict their run time.

Black box techniques aim to learn the costs of operations or the cardinality of their outputs without explicit knowledge of the different components of operations that compose their costs [40–42].

Table 1: Tiresias' API for prediction and training.

Tiresias API
predictLatency(*costFunction, costArguments*) → latency // Short form for a given cost function: $F(a_1, \ldots, a_n)$
recordLatency(*costFunction, costArguments, latency*)
predictAccessCount(*partition, accessType, timeWindow*) → count // Short form for partition/accessType T, timeWindow τ: $\delta(T, \tau)$
recordAccessCount(*partition, accessType, timeWindow, count*)
train()

Deep reinforcement learning-based approaches have been explored in this domain [22, 35, 43, 45], and consume the plan tree structure as input, including join predicate information, while relying on the hidden layers to capture and learn the relevant information. The complexity of the model vastly increases training time, precluding its use in an online environment [43]. Recently, zero-shot cost models [25] employ transferrable graph representations of queries to predict latency for queries that execute over data sets not included in offline training sets.

2.3 Predictive Adaptation in DBMSs

DBMSs utilize predictive components and models to alter their design based on the workload. The commonality in these systems is that they perform a cost-driven "what-if" analysis to quantify the effect of the proposed action on system performance [12]. Automatic knob-tuning systems select knobs to tune based on an expected reward (tail latency) using machine learning algorithms [18, 39, 60]. Auto-indexing tools [8, 12, 16, 32, 44] identify the set of indexes that should be used and automatically generate, deploy, or tune them. Similarly, tools select the set of views to materialize based on the workload [23]. In these scenarios, dependencies among the decisions and delayed effects of changes make it difficult to attribute performance effects to any one change. However, these systems do not predict the upcoming workload. Tiresias provides an interface that allows different DBMS components to leverage both cost and access arrival predictions to improve system performance.

3 TIRESIAS OVERVIEW AND DEPLOYMENT

Tiresias presents an API (Table 1) to **predict** data access latencies using cost functions (Section 4), predict upcoming workload using access arrival estimators (Section 5), **record** collected observations and periodically **train** the models. Using these predictions, systems can use Tiresias to make cost-driven decisions to adapt their storage and indexing based on predictions of the workload (Section 6).

3.1 Deployment

We used Tiresias' flexible and general API to deploy predictive adaptation of storage and indexing in three systems. We describe Tiresias' integration with these systems next.

3.1.1 Adaptive Storage for HTAP We integrated Tiresias into Proteus [7], a distributed HTAP DBMS. Proteus stores and distributes relational data among data sites in multiple – row or column – storage formats across multiple – memory or disk – storage tiers. Proteus also supports storage layout optimizations, such as maintaining data in sorted order or a compressed form [4–6, 30, 31, 63]. Proteus selects a master (or primary) node (or site) for each data

item where update transactions execute and a storage layout for that node. Proteus replicates data selectively, and it decides which sites store data item replicas and the associated layout for each replica. Proteus makes these decisions dynamically at the granularity of a data partition.

To integrate Tiresias into Proteus, we: (i) decomposed transactional latency predictions into storage-specific operators that Proteus calls on Tiresias to predict (e.g., latency of a scan over row format data), (ii) call Tiresias to record operator latency, and data access history, (iii) use Tiresias to predict when data accesses will arrive and with what frequency, and (iv) use Tiresias to compute the expected benefit of a storage layout change for future and upcoming transactions, performing the adaptation if Tiresias deems it beneficial to do so.

3.1.2 Predictive Cracking We incorporate Tiresias into an OLAP DBMS that performs cracking [24]. By integrating Tiresias' capabilities, we enable the DBMS to perform cracking predictively. If Tiresias predicts that an area of data will be accessed in the future, it reorganizes (cracks) that area prior to the access occurring. To integrate Tiresias into the OLAP DBMS, we: (i) record query arrivals and latency along with statistics to indicate whether the data accessed was sorted, (ii) call Tiresias to predict which areas of data will be accessed in an upcoming window, and (iii) execute predictive data cracking if the system has free cycles.

3.1.3 Automatic Indexing Finally, we integrated Tiresias into PostgreSQL to automatically add and remove indexes. To enable this automatic index addition and removal, we integrate Tiresias into the PostgreSQL client to (i) record query arrivals and latency along with whether a secondary index is used on the table, (ii) call Tiresias to predict the query access type and latency of the query, with and without secondary indexes, and (iii) add or remove secondary indexes based on Tiresias' predictions, using the SQL interface.

4 PREDICTING DATA ACCESS LATENCY

Tiresias uses cost functions to predict the latency of transactions and storage adaptations by decomposing each transaction into storage operators that are chained together to execute the transaction. Tiresias estimates the transaction's latency by summing each operator's predicted latency. To do so, Tiresias learns a cost function F for each storage operator parameterized by arguments $(a_1, a_2, ..., a_n)$ determined by the storage layout and workload statistics. For example, to predict the latency of a transaction that updates a partition in a row layout in Proteus, Tiresias combines the following predictions: (i) sending a network request to the data site (ii) acquiring a lock (iii) performing the update on the row, and (iv) committing. Each of these predictions is parameterized; for example, the latency of performing the update is parameterized by: (i) the number of cells accessed, and (ii) the total average size of each cell updated. Decomposing latency into different storage operators, as opposed to end-to-end black-box models, allows Tiresias' models to quickly converge on the order of minutes (Section 7.2). Tiresias' fast convergence occurs because it observes similar input for individual operators as training data, despite differing overall transaction properties.

Using storage specific cost functions and parameterizing them accordingly allows Tiresias to predict the latency of transactions under different storage layouts and index selections. Consider the previous example of an update transaction in Proteus for which Tiresias wishes to compute the latency under a columnar layout. In this case, Tiresias replaces the row-format predictor of performing an update with the column-format predictor of performing that update. Similarly, if data is vertically partitioned, then cost function parameters are altered as (i) the size of stored partition data decreases, which decreases update or read latency, and (ii) contention on the partition's lock decreases.

4.1 Predictors

Tiresias learns the cost function F using three different learning algorithms: (i) linear regression, (ii) non-linear regression and (iii) a neural network model (Section 4.1). As we show in Section 7.2, these algorithms have significantly different latencies for both inference (making a prediction) and training (building the model), as well as accuracy differences. Tiresias uses the linear regression algorithm, which has the lowest inference latency when predicting in latency-sensitive situations such as generating an execution plan. In contrast, the non-linear regression and neural network algorithms are more accurate than the linear regression algorithm but take longer to train and converge.

Each algorithm takes n arguments, as in $F(a_1, ..., a_n)$, and returns a scalar prediction y. Tiresias also records the true observed value for a given estimated cost, o, which is the latency of the specific operation (Section 4.2). Periodically (by default every 15 seconds), with each observation o, a function's corresponding arguments $(a_1, ..., a_n)$ and prediction y are updated by Tiresias via training.

4.1.1 Linear Regression The linear regression algorithm makes the prediction $F(a_1, ..., a_n) = y$ by learning weights $w_0, w_1, ..., w_n$ and predicting $F(a_1, ..., a_n) = w_0 + (a_1 \cdot w_1) + ... + (a_n \cdot w_n)$. Tiresias uses stochastic gradient descent to update the weights $(w_0, w_1, ..., w_n)$ with a mean squared error loss function $(o - y)^2$.

4.1.2 Non-Linear Regression Tiresias uses Dlib's [34] kernel recursive least squares algorithm (KRLS) [19] as its non-linear regressor. The KRLS algorithm works by learning a linear regressor (using recursive least squares) over a higher-dimensional feature space that is induced from the input arguments $(a_1, ..., a_n)$, observations o, and a kernel function. By default, we use the common and popular radial basis function kernel.

4.1.3 Neural Network Regressor To support a neural network regressor, Tiresias uses Dlib's [34] multilayer perceptron that uses backpropagation to update the internal weights of the network. Dlib uses a sigmoid activation function at each node so the network produces an output between 0 and 1, which Tiresias scales to between 0 and a predefined maximum latency (30 seconds). By default, Tiresias' neural networks have two hidden layers, allowing the model to learn arbitrary functions. For a cost function with n arguments, the input layer has $n + 1$ nodes, the output layer with 1 node, and two hidden layers each having $\frac{n+2}{2}$ nodes, which follows the principle of averaging the number of input and output nodes.

4.2 Recording Observations and Training

Systems use Tiresias' record API to collect the observed latencies of database operators for use in training of the cost functions. To

collect these observed latencies, Tiresias' clients record operator start and end times, and the input arguments associated with each invocation of a storage operator. In our update transaction example, the update operator records: the start time of its execution, the end time of its execution, the number of cells written, the size of the data columns read and written. This information is added to a per-thread observation data structure that stores a list of observed latencies and their arguments for each storage layout and operator pair. These observations are reported to Tiresias by swapping each thread's observations with an empty set and merging the observations together. Tiresias uses these merged and recorded observations as data for online training of its cost functions. Online training allows Tiresias to continually adjust its estimates of data access latencies based on the observations and to grow its models' understanding of the parameter space based on the workload. By contrast, offline model training requires training data consisting of observed costs for all combinations of storage layouts for each operator, which is difficult and prohibitively costly to collect.

5 ESTIMATING DATA ACCESS ARRIVALS

Recall that workloads follow patterns (Section 1), and predicting these patterns provides opportunities for adapting data storage and indexing so that they are optimized for the workload. This task involves predicting the number of data partition accesses by type (e.g., scan versus update) in a given upcoming time window.

Predicting when data will be accessed and the volume of accesses entails: (i) predicting trends at various temporal granularities, such as daily, weekly or yearly patterns, (ii) handling growth and spikes in requests, which can occur on specific dates such as the fiscal year-end, and (iii) adjusting to changes in workloads over time that arise if the submitted queries or transactions change due to user needs or preferences [37, 58]. Thus, Tiresias uses predictive models with both a periodic component to capture the long-term *periodic trend* and a local trend to capture *short term* transient effects such as spikes in the number of requests.

5.1 Predictors

Formally, Tiresias predicts $\delta(T, \tau)$ that represents the number of requests of type T that will arrive in time window τ. Tiresias supports two different predictors to estimate $\delta(T, \tau)$: SPAR and a hybrid-ensemble. Both predictors combine a *periodic trend* prediction, that is, how access patterns change over a recurring period (e.g., accesses follow an hourly cycle), and a *short term* effect component that considers the short term trend in accesses (e.g., accesses are becoming more common over an hour). The period is user-defined for SPAR while the hybrid-ensemble learns the period.

For both predictors, τ is set by default to five-minute intervals over a day, which trades-off storage space required to aggregate history with the granularity of predictions. For notational simplicity, if τ is less than the current time, i.e., in the past, we define $\delta(T, \tau)$ as the actual number of accesses to T in the time window represented by τ. We describe how Tiresias efficiently records this access history for training in Section 5.2.

5.1.1 SPAR Tiresias uses SPAR [13] to combine long-term *periodic trends* with *short term* effects. To learn the *periodic trend*, SPAR requires a user-defined period Ψ, and a number of periods to examine Ψ_n. SPAR considers the number of requests to T in previous time intervals (represented by $\delta(T, \tau - i\Psi)$ for $1 \leq i \leq \Psi_n$). Tiresias weights these previous numbers of requests by learned coeffecients b_i as shown in Equation 1.

$$\sum_{i=1}^{\Psi_n} (b_i \cdot \delta(T, \tau - i\Psi)) \tag{1}$$

Tiresias also uses SPAR to consider how *short term* access counts shift within the period Ψ compared to the access counts in previous periods. SPAR averages this access count, $\gamma(T, \tau)$, in Equation 2.

$$\gamma(T, \tau) = \delta(T, \tau) - \frac{1}{\Psi_n} \sum_{i=1}^{\Psi_n} \delta(T, \tau - i\Psi) \tag{2}$$

To compute the *short term* shift within the period, Tiresias uses SPAR to weight $\gamma(T, \tau - j)$ using learned coefficients c_j for each of the Ψ_k intervals within the period Ψ. Consequently, Tiresias computes the short term shift in access counts as shown in Equation 3.

$$\sum_{j=1}^{\Psi_k} (c_j \cdot \gamma(T, \tau - j)) \tag{3}$$

Tiresias combines the *periodic* prediction (Equation 1) with the *short term* trends (Equation 3) via a sum to predict $\delta(T, \tau)$.

5.1.2 Hybrid-Ensemble Tiresias' hybrid-ensemble (HE) predictor combines a periodic predictor using a recurrent neural network (RNN) to capture the *periodic trend*, and a *short term* predictor using linear regression. Importantly, the HE predictor does not require used-defined periods as the RNN learns the periodic trend.

Tiresias' linear regression predictor $\delta_L(T, \tau)$ predicts the number of accesses to T at time τ based on past access counts over a sliding window. As the regression is linear, it captures only the *short term* access trend. That is, it captures whether accesses are increasing or decreasing over time, and if so, by how much.

The RNN predictor $\delta_R(T, \tau)$ captures *periodic trends* without requiring prior knowledge of the period. RNNs are a class of networks where nodes in the network have cycles [61], and Tiresias uses the long short-term memory (LSTM) architecture [26] for its RNNs. LSTMs allow the RNN to remember values over arbitrary time intervals, and learn the period. Tiresias uses LibTorch's [47] implementation of LSTMs with five internal layers.

Tiresias also accepts a user-defined custom holiday list that allows an administrator to define periods where additional access counts should be added or removed following a Gaussian function [58]. The holiday list allows Tiresias to account for events that repeat but not in a periodic cycle, such as Black Friday, which is not on the same date every year. For each holiday, the user defines the time over which the holiday occurs (range between h_s and h_e), and parameters for the Gaussian function (peak value α, width σ^2, and centre between h_s and h_e) that capture the access counts for that time as $\delta_H(T, \tau)$. If τ is within the defined time of the holiday (i.e., $h_s \leq \tau \leq h_e$) then Tiresias computes $\delta_H(T, \tau)$.

Tiresias combines the three components of the HE to produce a final prediction $\delta(T, \tau)$ from the average of the linear trend and periodic behaviour, and any holiday adjustments.

5.2 Recording Access History and Training

To predict the number of accesses to a data partition by type in a time window requires recording access history counts to train predictive models. The number of accesses to a partition by type within a one-minute window (by default) is recorded by each Tiresias client. As the access counts are kept on a per-partition basis, these counters have low levels of contention. Tiresias polls these clients to collect and record this information and aggregates the accesses within five-minute windows (by default). Using these newly collected access patterns, Tiresias updates its access estimators via training. In the case of SPAR, the collected observations are stored until they will no longer be needed for inference, that is, Ψ_n periods have passed. These access counts are stored in a circular buffer, and new access counts overwrite prior access histories. For the HE, the collected observations no longer need to be stored once the estimator has been updated; however, Tiresias keeps a sampled reservoir of access histories when updating the estimator so that the model has access to long-term history.

6 ADAPTIVE STORAGE AND INDEXING DECISIONS

Tiresias drives storage and indexing adaptation decisions by computing the expected benefit of a change. Given a specific choice of storage layouts or indexes S, the benefit of changing this layout is computed from (i) the upfront cost $U(S)$ to execute the storage change based on the operations needed to perform the change, and (ii) the expected cost effect $E(S)$ on requests predicted to arrive and on requests currently executing ($C(S)$). To compute $E(S)$ and $C(S)$, Tiresias computes the difference between predicted transaction latency under the current and proposed layouts, weighted by the likelihood of the transaction executing. That is, given a transaction T, Tiresias estimates the latency of the request under the current layout $L_{current}(S,T)$ and the latency of the request under the adapted layout $L_{adapt}(S,T)$. Tiresias estimates L_{adapt} by using the data access cost predictors (Section 4), adjusting the input arguments to the cost functions as necessary and using the predictor specific to the layout under consideration.

Tiresias weighs the estimated effect of the storage layout or indexing change on T by the likelihood of T's request arriving ($Pr(T)$), and the time to T's arrival ($\Delta(T)$). We derive $Pr(T)$ and $\Delta(T)$ from $\delta(T,\tau)$, which is the number of requests of type T that will arrive in time window τ (Section 5). Given $\delta(T,\tau)$, we compute $\Delta(T)$ at time t as $\tau - t$ for the minimum $\tau > t$ such that $\delta(T,\tau) > 1$, i.e., the first time in the future that we estimate at least one request will arrive. Given this τ, we estimate the probability $Pr(T)$ of such an arrival as the complement of a poisson estimate of the probability that no requests arrive at time τ. With these predictions, Tiresias combines the estimated effect of the storage layout or indexing change S on T as $E(S,T)$, defined in Equation 4.

$$E(S,T) = \left(L_{current}(S,T) - L_{adapt}(S,T)\right) \cdot \frac{Pr(T)}{\Delta(T)+1} \quad (4)$$

Observe that $E(S,T)$ is positive if $L_{current}(S,T) > L_{adapt}(S,T)$, which indicates that Tiresias expects the storage change to reduce the latency of executing the request. However, the magnitude of $E(S,T)$ is determined by: (i) the relative change in execution latency, (ii) how likely the request is to arrive, and (iii) the estimated time to request arrival.

Tiresias computes $E(S)$ and $C(S)$, by summing $E(S,T)$ for each request that is ongoing (i.e., $\Delta(T) = 0$ and $Pr(T) = 1$), or predicted to arrive. Observe that $E(S,T) \approx 0$ if $L_{current}(S,T) \approx L_{adapt}(S,T)$, $Pr(T) \approx 0$ or $\Delta(T)$ is sufficiently large. Thus, Tiresias restricts the set of requests that accesses data affected by the storage change or is likely to arrive in an upcoming window.

Finally, Tiresias combines $E(S)$, $C(S)$, and $U(S)$ into one equation that defines the net benefit of the storage change, as $N(S) = \lambda(E(S) + C(S)) - U(S)$. Observe that $\lambda > 0$ controls the importance of the expected benefit of the storage change compared to the upfront costs to perform the storage change. Therefore, if $N(S)$ is greater than 0, the storage change is considered beneficial.

7 EXPERIMENTAL EVALUATION

This section presents our experimental evaluation that demonstrates the effectiveness of Tiresias' predictive components and how prediction improves end-to-end system performance.

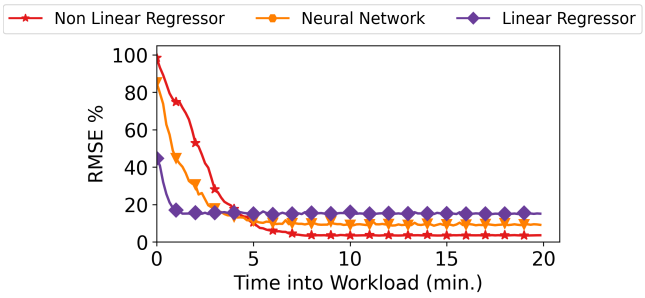
7.1 Methodology

Our experiments are conducted using machines with 12 cores, 32 GB of RAM and a 1 TB hard disk. A 10 Gbps network connects all machines. Unless mentioned otherwise, results are averages of at least 5 independent 20-minute OLTPBench [17] runs with 95% confidence intervals shown as error bars around the means.

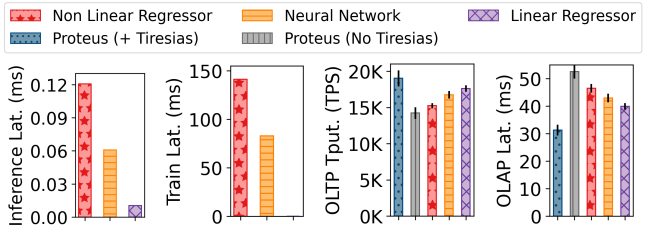
7.1.1 Workloads We conduct experiments using two HTAP workloads: the **CH-benCHmark** [14], and transactional **YCSB** [15] benchmarks. The CH-benCHmark consists of the TPC-C OLTP workload [1] and the TPC-H OLAP workload [2]. The transactional YCSB workload consists of 10 multi-key read-modify writes for OLTP transactions and an OLAP query that scans 500,000 rows with a predicate and aggregates the results. We use two real-world traces of access counts aggregated by minute: **Wikipedia** [59] and **Azure** BLOBs [51]. The Azure trace derives from a function-as-a-service workload and consists of many different client applications accessing data, which results in higher variability in access patterns compared to the more homogeneous Wikipedia trace. We also use the real-world Sloan Digital Sky Server (**SkyServer**) [3, 56] trace of OLAP selection predicates from queries that access areas of the sky to identify objects.

7.1.2 Enabling Proteus We compare Proteus' performance using 6 data sites with and without Tiresias to allow us to quantify the benefit of Tiresias' predictions. In Section 7.5, we study Tieresias' impact on overall system performance. We compare Proteus against two alternative distributed HTAP database system architectures **Janus** [9] and **TiDB** [29] as well as a **row**-oriented distributed database (row store or **RS**) targeted to optimize OLTP workload execution and a **column**-oriented distributed database (column store or **CS**) designed to optimize OLAP workload processing. TiDB and Janus fully replicate data between their OLTP-optimized and OLAP-optimized stores.

7.1.3 Enabling Predictive Cracking We use Tiresias to enable predictions in an OLAP DBMS [24] that cracks data and compare the performance of the system before and after it is enabled with Tiresias' predictive capabilities.



(a) RMSE over time



(b) Infer Lat. **(c) Train Lat.** **(d) OLTP TPut.** **(e) OLAP Lat.**

Figure 1: Performance metrics for cost predictors.

7.1.4 Enabling Automatic Indexing To evaluate Tiresias' ability to automatically and adaptively add or remove secondary indexes for a changing workload, we compare PostgreSQL with Tiresias to two static PostgreSQL configurations: with and without secondary indexes, neither of which can adapt to workload changes.

7.2 Cost Model Performance

Figure 1 depicts the performance of Tiresias' three different cost function estimators, the linear regressor, neural network, and non-linear regressor, in Proteus. Figure 1a shows the root mean square error (RMSE) of each predictor normalized to each observation and averaged over each of the different cost functions throughout the experiment. Observe that by the end of the experiment, the linear regressor has the largest RMSE and hence is the least accurate of the three models, while the non-linear regressor has the lowest RMSE and thus the highest accuracy of the models.

In Figure 1a, all predictors converge to their average RMSE within 10 minutes. The non-linear regressor takes the longest to converge, while the linear regressor converges within one minute. These differences in model convergence rates are an important reason why Tiresias uses multiple models when predicting costs. Tiresias makes prudent decisions by considering predictions from multiple models, even if not all of the models have converged.

Although all three models have different degrees of accuracy, all three models have RMSEs that allow Tiresias to distinguish between good and poor storage change decisions. If the cost predictions for two different decisions are similar then the actual cost is likely similar, but if one cost prediction is significantly higher than the other then it is likely to be a relatively poor decision.

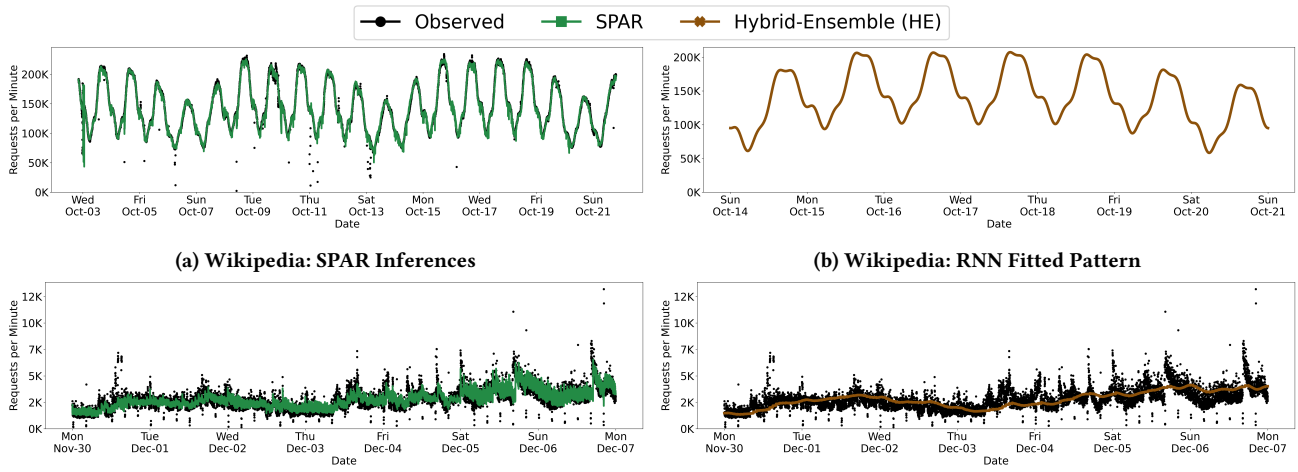
Compared to the linear regressor, increased model accuracy for the neural network and non-linear regressor comes with a trade-off: increased latency to perform inference and training (Figures 1b and 1c). The linear regressor performs inference more than 10× faster than the non-linear regressor as it combines coefficients in a sum instead of invoking the kernel function. Linear regressor inference is 5× faster than neural network inference, which incurs two layers of matrix multiplication and sigmoid activation functions. Moreover, the linear regressor performs a round of training more than 1500× and 2500× faster than the neural network and non-linear regressor, respectively. While incurring a slight decrease in model accuracy, these significant differences in latency justify using only the linear regressor when making predictions in a latency-sensitive situation such as selecting an execution plan. However, the more accurate neural networks and non-linear regressors are employed when generating storage layout and indexing change plans, which occur less frequently and thus outside most requests' critical path. The cost functions are also space efficient due to the small number of input parameters. The neural network has a larger model size (4 KB) from keeping a matrix of weights as opposed to the linear (64 bytes) and non-linear (112 bytes) regressors that keep a vector of weights.

To assess the effects of Tiresias' cost function estimators on Proteus performance, we measured the OLTP throughput and OLAP latency for the YCSB workload. In these experiments, each decision requires invoking only one type of cost function estimator when compared to Proteus that uses all three of Tiresias' cost function estimators (Figures 1d and 1e). The linear regression predictor has the best performance of the three models in terms of OLTP throughput and OLAP latency. This result demonstrates the importance of inference time on overall system performance as the inference latency directly contributes to end-to-end request latency. Moreover, both the neural network and non-linear regression predictors require significantly more training data than the linear regressor to produce accurate cost estimates. Although the linear regressor produces less accurate estimates by the end of the experiment than the neural network and non-linear regressor, the linear regressor is initially more accurate. This initial accuracy allows Proteus to quickly begin adapting storage to the workload to deliver high performance. Combining all three cost function estimators together in Tiresias yields the best performance, improving throughput by 1.33× and reducing latency by 40% over not using Tiresias' predictive capabilities.

7.3 Access Arrival Estimator Performance

To assess the performance of Tiresias' access arrival estimators, we used a real-world workload trace of Wikipedia accesses [59] and Azure [51]. For both SPAR and our HE technique, we train on the same amount of data (two weeks for Wikipedia, one week for Azure) before predicting the number of requests that will occur each minute for the next hour. We then provide the models with the actual observed number of requests that occurred over the hour before predicting the next hour. Figure 2a shows SPAR's predicted number of requests compared to the observed number of requests (shown as dots). HE's predictions are similar.

For both workloads, SPAR (Figures 2a and 2c) and HE's (Figure 2d) predictions match the observed number of requests. The Wikipedia workload features more regularity in the workload trend, and hence the RMSE (Figure 3a) is lower than for the more variable Azure workload (Figure 3b). In both workloads, SPAR is more accurate than HE but requires prior knowledge of the workload — the periodic pattern follows a daily cycle. By contrast, the HE method learns the period from the data using the RNN, which captures



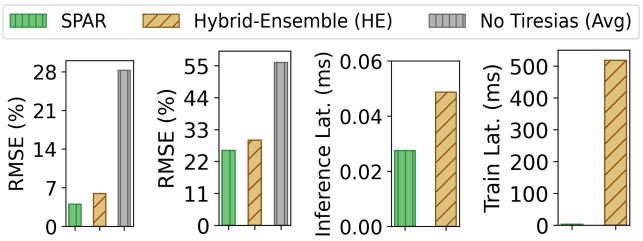
(a) Wikipedia: SPAR Inferences

(b) Wikipedia: RNN Fitted Pattern

(c) Azure: SPAR Inferences

(d) Azure: Hybrid-Ensemble Inferences

Figure 2: Access Arrival Predictions using Tiresias' access arrival estimators: SPAR and Hybrid-Ensemble.



(a) Wiki. Acc. (b) Azure Acc. (c) Infer Lat. (d) Train Lat.

Figure 3: Performance metrics for arrival estimators.

Wikipedia's periodic trend, as shown in Figure 2b over both the day and week without any user hints. If SPAR's period is misconfigured then it makes poor predictions resulting in RMSE 3× higher than for HE. Tiresias' estimators in both workloads provide significant improvements in accuracy compared to using the average of the arrival count in the training period (Figure 3: No Tiresias (Avg)).

A key difference between SPAR and HE in the Azure workload is that SPAR is more variable in its predictions (Figure 2c) compared to HE (Figure 2d). This arises due to SPAR averaging specific prior observations based on the period, resulting in inherently noisy and variable predictions compared to HE's smoother predictions.

SPAR has lower latency then HE for both inference (Figure 3c) and training (Figure 3d). However, HE is competitive in inference latency, which is critical for making predictions. By contrast, training happens periodically and asynchronously, and the latency of a round of training (sub-second) is orders of magnitude smaller than the time it takes to gather the observations (minutes). The model sizes for the arrival estimators are space efficient with respect to the stored data. The SPAR predictor (3 KB) has a smaller memory footprint than the HE predictor (90 KB) as SPAR keeps a linear number of weights for predictions compared to HE's RNN which has multiple layers and stores a matrix of weights for each layer.

7.4 End-to-End Performance: Adaptivity

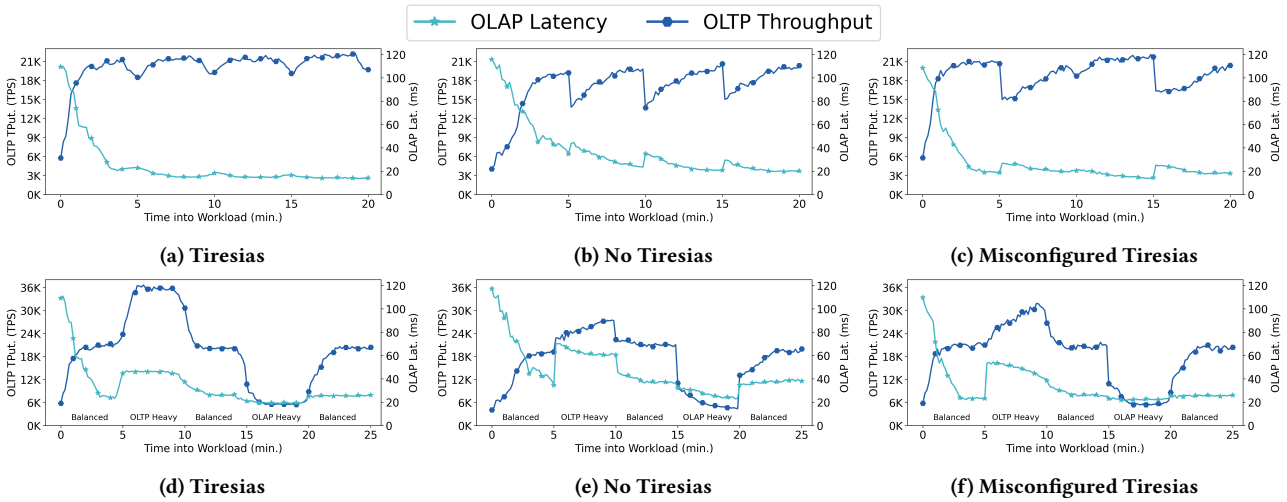
To understand the effect and contribution of Tiresias in enabling adaptive capabilities, we measure Proteus' OLTP throughput and OLAP latency over time to understand its behaviour as it learns both the workload access pattern and cost model (Figure 4). In the experiment in Figures 4a to 4c, the centre of skew in the OLTP workload shifts every 5 minutes following an hourly cycle. In Figures 4d to 4f, we shift the workload mix every 5 minutes among balanced, OLTP heavy and OLAP heavy workloads.

First, shown in Figures 4a and 4d, we enable Proteus with Tiresias and an HE access arrival estimate model pre-trained using 12 hours of historical access patterns from the workload. Next, shown in Figures 4b and 4e, we disable Tiresias thereby removing Proteus' ability to predict access arrivals and merely leaving it to deal with the workload shifts. Finally, in Figures 4c and 4f, we configure Tiresias using SPAR but with a ten-minute period, rather than the 5 minute period that aligns with the workload shift.

In Figure 4a, slight shifts in performance are visible both before and after every 5 minute duration due to the workload shifts occurring at these time points. Tiresias induces Proteus to execute storage layout changes predictively in anticipation of the workload shift due to high confidence in the workload access pattern changes. Small performance shifts arise due to (i) storage layout changes consuming resources, and (ii) predictive storage layout changes that amortize costs to provide beneficial layouts for future accesses. Observe that Tiresias allows Proteus to rapidly improve OLTP throughput and OLAP latency. OLTP throughput increases by 5.4× over the workload execution while OLAP latency decreases by 7.9×. Moreover, it takes just a short amount of time (3 minutes) to reach within 15% of its peak OLTP throughput and roughly 10 minutes to reach within 15% of its minimum OLAP latency. These differences are due to the skew in OLTP accesses compared to the uniform OLAP accesses; the system executes more layout changes for data primarily accessed by OLAP transactions.

In contrast to the accurate access arrival estimator (Figure 4a), both Figures 4b and 4c show performance drops when the workload shifts. This drop results in a 35% decrease in OLTP throughput and a 1.47× increase in OLAP latency. By contrast, in Figure 4a, OLTP throughput degrades by just 15% and OLAP latency increases by just 1.21×.

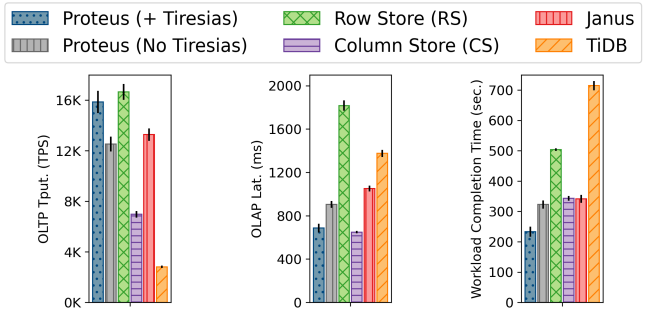
Observe that Proteus without Tiresias (Figure 4b) responds to the workload shifts at the 5 and 15-minute marks faster than Tiresias



(a) Tiresias (b) No Tiresias (c) Misconfigured Tiresias

(d) Tiresias (e) No Tiresias (f) Misconfigured Tiresias

Figure 4: Proteus' OLTP throughput and OLAP latency over time with shifting hotspots (4a-4c) and mixes (4d-4f).



(a) OLTP TPut. (b) OLAP Lat. (c) Exec. Time

Figure 5: CH-BenCHmark Results

with an intentionally misconfigured workload period (Figure 4c). With a misconfigured period, Tiresias delays responding to the workload shift with storage layout changes because of the disagreement in (intentionally) incorrect predicted access patterns and the actual access patterns. Over time, Tiresias adjusts the system's storage layout and performance improves. The adverse effects of misconfiguration of the workload period using SPAR highlight the benefit of Tiresias' HE predictor that learns the workload period.

The shifting workload mix experiments (Figures 4d– 4f) follow a similar trend: with Tiresias (Figure 4d), Proteus responds quickly to changes in the workload, increasing OLTP throughput and decreasing OLAP throughput compared to Proteus without Tiresias (Figure 4e). Tiresias enables Proteus to make predictive storage changes in anticipation of the mix shift, such as removing columnar replicas of data when the workload shifts from balanced to OLTP heavy. Overall, using Tiresias improves OLTP throughput by 16% and reduces OLAP latency by 30% compared to Proteus without Tiresias. These experiments show the benefit of predicting access arrival times and predictive storage changes on overall system performance while amortizing layout change costs.

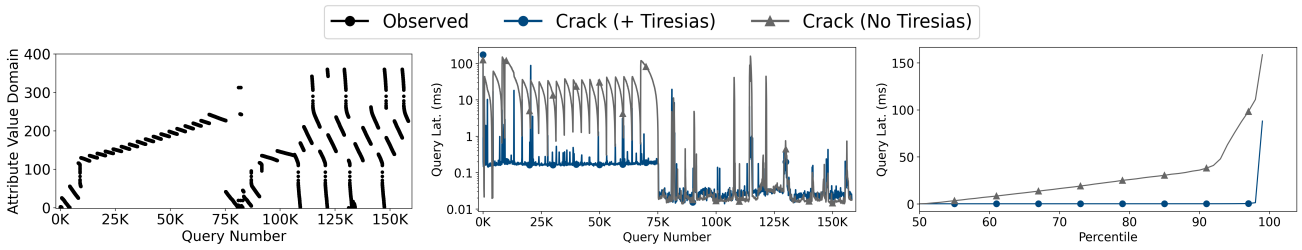
7.5 End-to-End Performance: CH-BenCHmark

We evaluate Proteus and Tiresias using the CH-benCHmark, an HTAP workload composed of 22 TPC-H OLAP queries and 5 TPC-C OLTP transactions. With Tiresias, Proteus' throughput is comparable to the top-performing OLTP system, within 5% of RS OLTP throughput (Figure 5a), *and* within 8% of the CS OLAP latency (Figure 5b). Neither RS nor CS can achieve this combined high performance for *both* OLTP throughput and OLAP latency. The static hybrid systems (Janus and TiDB) are inferior to Proteus with Tiresias as they fully replicate all data and place more data on disk, which increases data access costs, lowers OLTP throughput and raises OLAP query latency. Consequently, Proteus with Tiresias completes the CH workload faster than all competitors (Figure 5c), demonstrating that Tiresias enables adaptive storage that is well-suited for hybrid workloads. For the OLAP workload, Proteus with Tiresias has similar query latency to CS while taking slightly longer to execute some queries (e.g. Query 7) that feature joins across many tables, complex predicates and aggregations. Adaptive storage allows Proteus with Tiresias to remain competitive with CS on the overall OLAP workload while delivering superior performance of more than 2.2× OLTP throughput on the hybrid workload.

Examining Tiresias' decisions reveals that it learns and makes storage layout decisions without prior knowledge of the workload, enabling transformation of *new orders* from row to column layouts over time, thereby showing the benefit of predictive adaptive storage. By contrast, static systems keep this table in a single layout (RS and CS) or both (replicated) layouts (Janus and TiDB). Without Tiresias, Proteus takes nearly 1.4× as long to complete the workload, with performance comparable to Janus. Without Tiresias, Proteus can only react to the access pattern and thus is slower to transform data layouts from row to column, thereby increasing OLAP latency. Keeping data replicated for longer consumes memory, forcing more data to be evicted to disk, which decreases OLTP throughput.

7.6 Predictive Data Cracking

In Figure 6, we demonstrate the generalizability and applicability of Tiresias' techniques by leveraging them in an OLAP DBMS to predictively crack data. Figure 6b depicts the latency for each query in the SkyServer workload (Figure 6a), while Figure 6c shows tail latency. Tiresias greatly reduces average query latency by 38× and the 95th percentile tail latency by nearly 300×. Tiresias boosts performance because predictive cracking increasingly stores and



(a) SkyServer Data Access Pattern (b) SkyServer Latency over Time (c) SkyServer Tail Latency

Figure 6: SkyServer data access pattern and OLAP latency for SkyServer data cracking, with and without Tiresias.

(a) Workload Completion Time (b) OLTP Throughput Over Time (c) OLAP Latency Over Time

Figure 7: PostgreSQL performance with and without indexing, and Tiresias adapting indexes.

accesses data in sorted order, reducing the search range necessary to execute the queries. We found that 88% of queries benefited from Tiresias' predictions by executing over predictively cracked data. These results demonstrate that Tiresias provides a flexible and re-usable API that can be utilized generally.

7.7 Automatic Indexing in PostgreSQL

To measure the benefit of Tiresias in enabling automatic indexing in PostgreSQL, we modify the YCSB workload so that the OLAP query scans all rows using a predicate with 1% selectivity on a secondary column and aggregates the results. In our evaluation, we use an OLTP heavy mix and an OLAP heavy mix, and shift between these workloads every 5 minutes, or every 5000 transactions in the case of the workload completion time. As in Section 7.4, Tiresias has an access arrival estimate model pre-trained using 12 hours of historical access patterns from the workload.

Figure 7 shows the results of our experiment with Tiresias and indexing in PostgreSQL. In Figure 7a, we show the workload completion time for the overall workload (two shifts between OLTP heavy and OLAP heavy), as well as the workload completion time for one OLTP heavy and one OLAP heavy component of the workload. With Tiresias, the workload overall completes nearly 15% faster than with an index and 70% faster than without an index.

To understand why Tiresias completes the workload faster than its static competitors, Figures 7b and 7c display OLTP throughput and OLAP latency over time. Tiresias' results aside, observe that for both workload mixes, OLTP throughput is highest for PostgreSQL without an index while OLAP latency is lowest for PostgreSQL with an index. This result is expected: the secondary index allows an index-based scan that significantly reduces OLAP latency. However, maintenance of the secondary index in the presence of updates is costly, decreasing OLTP throughput. Consequently, PostgreSQL with an index completes the OLAP heavy part of the workload faster than PostgreSQL without an index but takes longer to complete the OLTP heavy part of the workload (Figure 7a). Remarkably, Tiresias completes the OLTP part of the workload nearly as quickly as PostgreSQL without an index and the OLAP part of the workload in a similar time to PostgreSQL with an index. Tiresias delivers this performance boost to complete the workload the fastest because it matches the static decision based on the ongoing workload: adding indexes during the OLAP heavy part of the workload and removing them in the OLTP heavy part of the workload.

Roughly 1 min. into the workload (Figure 7b), Tiresias' throughput drops, matching the performance of PostgreSQL with an index. This drop occurs because Tiresias is exploring its decision space and collecting observations of PostgreSQL's performance without an index. As Tiresias builds its models online, it has no prior knowledge of the performance effect that indexing has on OLTP throughput or OLAP latency (Figure 7c). Similarly, when the workload shifts at the 5-min. mark, Tiresias does not add an index before it has collected performance observations on this mix. By the second and third workload shifts (at 10 and 15 minutes respectively), Tiresias predictively removes and adds indexes based on its cost model and access arrival estimates, demonstrating its efficacy.

8 CONCLUSION

We presented Tiresias that uses prediction capabilities to enable storage layout and index adaptations to deliver high performance for mixed workloads. Tiresias' predictions of data access costs and patterns enable cost-driven storage adaptation decisions. We study the trade-offs among three different cost predictors and two access arrival estimators and their effect on performance. We demonstrate the benefits of predictive adaptation on end-to-end system performance within the Proteus distributed HTAP system, OLAP database cracking and indexing. We conclude that data systems should leverage prediction to autonomously improve performance.

ACKNOWLEDGMENTS

This project was supported by funding from NSERC, WHJIL, CFI, and ORF.

REFERENCES

[1] 2010. The Transaction Processing Council. TPC-C Benchmark (Revision 5.11).

[2] 2018. The Transaction Processing Council. TPC-H Benchmark (Revision 2.18).

[3] 2022. The Sloan Digital Sky Survey (SkyServer).

[4] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.

[5] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 967–980.

[6] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. 2007. Materialization strategies in a column-oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 466–475.

[7] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *SIGMOD*.

[8] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.

[9] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2017. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017), 689–702.

[10] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, New York, NY, USA, 583–598.

[11] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[12] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.

[13] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services.. In *NSDI*, Vol. 8. 337–350.

[14] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM symposium on Cloud Computing (SoCC)*. ACM, 143–154.

[16] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[17] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB* 7, 4 (2013), 277–288.

[18] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[19] Yaakov Engel, Shie Mannor, and Ron Meir. 2004. The kernel recursive least-squares algorithm. *IEEE Transactions on signal processing* 52, 8 (2004), 2275–2285.

[20] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 592–603.

[21] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. 2020. Chronocache: Predictive and adaptive mid-tier query result caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2391–2406.

[22] Runsheng Benson Guo and Khuzaima Daudjee. 2020. Research challenges in deep reinforcement learning-based join query optimization. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–6.

[23] Himanshu Gupta. 1997. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory*. Springer, 98–112.

[24] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland HC Yap. 2012. Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment* 5, 6 (2012), 502–513.

[25] Benjamin Hilprecht and Carsten Binnig. 2021. One model to rule them all: towards zero-shot learning for databases. *arXiv preprint arXiv:2105.00642* (2021).

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

[27] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards workload-aware self-management: Predicting significant workload shifts. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 111–116.

[28] Marc Holze and Norbert Ritter. 2007. Towards workload shift detection and prediction for autonomic databases. In *Proceedings of the ACM first Ph. D. workshop in CIKM*. 109–116.

[29] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[30] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 297–308.

[31] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[32] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-memory Column-stores. *Proc. VLDB Endow.* 4, 9 (June 2011), 586–597. https://doi.org/10.14778/2002938.2002944

[33] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.

[34] Davis E. King. 2009. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research* 10 (2009), 1755–1758.

[35] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[36] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active learning for ML enhanced database systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 175–191.

[37] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, New York, NY, USA, 631–645.

[38] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1248–1261.

[39] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. 2017. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 28–40.

[40] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[41] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[42] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212* (2018).

[43] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132* (2019).

[44] 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.

[45] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[46] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, New York, NY, USA, 1771–1775.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[48] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner* (2014).

[49] M Pezzini, D Feinberg, N Rayner, and R Edjlali. 2016. Real-time Insights and Decision Making using Hybrid Streaming, In-Memory Computing Analytics and Transaction Processing. *Gartner* (2016).

[50] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, et al. 2020. Seagull: an infrastructure for load prediction and optimized resource allocation. *Proceedings of the VLDB Endowment* 14, 2 (2020), 154–162.

[51] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the 2021 ACM symposium on Cloud Computing (SoCC)*. ACM, 122âĂŞ137.

[52] 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*.

[53] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.

[54] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. 2013. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1184–1185.

[55] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[56] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. 2002. The SDSS skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 570–581.

[57] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 205–219.

[58] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[59] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845. http://www.globule.org/publi/WWADH_comnet2009.html.

[60] Dana Van Aken et al. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*.

[61] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989), 270–280.

[62] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[63] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.