PolygloDB: An Intelligent Polyglot Database Service with Automatic Data Partitioning

Dhanush M[1], Chirag[2], Rakshan R[3], Muhammad Sinan PK[4], Ganapathi Sharma K[5]
[1,2,3,4]*Department of AI & DS, SIT, Valachil, Mangalore – 574143, Karnataka, India*
[5]*Associate Professor, Department of AI & DS, SIT, Valachil, Mangalore – 574143, Karnataka, India*
doi.org/10.64643/IJIRTV12I12-206592-459

***Abstract*—Ultramodern software operations deal with a large spectrum of data types—from well-structured relational records to roughly organized JSON loads to free-form textbooks. Since no single database machine is equally effective in handling all these data formats, the engineering brigades are progressively abandoning the polyglot continuity design principle, which in- volves keeping several storehouse machines operational simultaneously. However, deciding and assigning the appropriate data- base for each data type is a homemade, moxie-dependent task. Misassignments in that respect are expensive and degrade query performance, erode transactional guarantees, and bloat functional outflow. This paper presents PolygloDB, a middleware sublayer that eliminates the guesswork in database selection by sketching incoming data and routing it automatically to the most appropriate backend: PostgreSQL for relational records, MongoDB for semi- structured documents, and Elasticsearch for textbook-heavy con- tent. Routing decisions are based on a binary-mode decision ma- chine that couple's deterministic rule-based heuristics with a trained machine-learning classifier, thus marrying speed and flexibility. A centralized Metadata roster records every placement decision so that a subsequent Query Router can retrieve data from whichever backends hold it, giving the visitor a single uniform answer. Standard results show that PolygloDB achieves superior query performance, selection delicacy, and lower inventor trouble than single-database birth and rule-only routing strategy.**

***Index Terms*—Polyglot Persistence, Automatic Data Partitioning, Hybrid Decision Engine, Federated Query Execution, Metadata Catalog, Database Management Systems**

I. INTRODUCTION

The digital geography of Moment produces data in shapes and sizes that would have been difficult to predict even a decade ago. A single business task can absorb simultaneously structured billing data, product hierarchies in JSON with nested structures, and streams of raw customer text reviews. Databases were traditionally designed for hard schemas and thick consistency. Now, ultramodern data conduits have a schema-free nature, and the drive towards Scripture continuity, a business strategy where diverse specialized stores run side-by-side enabling each server ma- chine to do its work very well. In a typical Scripture armature, a transactional machine-like PostgreSQL protects schema-bound records, a document store like MongoDB absorbs semi-structured or nested content, and a platform such as Elasticsearch powers Presto full-textbook reclamation. The benefit is a performance profile which a mono- lithic database can't hold a candle to. Every time a new data object arrives, the debit is dis-union, an inventor must assess its structure, decide which backend suits it, write the routing sense, and keep that sense aligned with a constantly evolving schema geography. It is time-consuming, requires deep familiarity with all three machines and is a rich ground for mis-calculations [1][2].

There are real implications for sending the right data to the wrong machine. Sending mostly relational records to a document store sacrifices ACID guarantees and referential integrity. Trying to push many textbook records into a relational table forces the query journal to do sequential lookups, instead of the reversed-indicator lookups that Elasticsearch performs out-of-the-box. In fact, even when the single machine selection decisions are correct, the operational overhead of managing different deployment pipe- lines and backup policies for three machines stresses the DevOps team Thinning out of the operations engineers. PolygloDB alleviates these pain points. Rather than requiring developers to make routing decisions by

hand, it intercept all incoming traffic, determines structure of the pay- load automatically, and directs it to the correct backend without any intervention. In the read pathway, it directs related requests through a dispatching sublayer, querying multiple backend machines for information simultaneously before aggregating the results. Effectively, the operations engineer deals with a single endpoint and is not aware of any complexities of a multi-engine, multi-model storage subsystem [3][4].

II. SYSTEM DESIGN

PolygloDB has been built up as a combination of six weakly coupled modules that handle each of the stages in the data life cycle and create a pipeline from input to the final result that all the modules have been fed to and in- formed about the location of the data by the participatory metadata subcaste, the six factors are: API Gateway, Data Analyzer, mongrel Decision Machine, Partition director, Metadata roster, Query Router

A. API Gateway

All of the data-load requests and all of the query requests are submitted to the API Gateway. The API Gateway is customer facing, and handles only one part of the communication with a customer operation: validation of the re- quest's format and schema, authentication and authorization routines, and the request's passage further into the channel.

Enforcement of the authorization routines allows the storehouse subspecies only access to authorized customers. The separation of the customers and the system logic is so important that the entire remainder of the store- house subsystem can be altered or modified without changing the way that customers will submit their requests [5][6].

B. Data Analyzer

When cargo passes through the gateway the Data Analyzer is put to work to profile it in an in-depth structural fashion. It profiles each field, discovering each field type (numeric, Boolean, string, date time, nested object), level of nesting of nested objects, and fields cardinality (i.e., Amount of null values), looking for join patterns (i.e. Foreign-crucial-style joins), quantifying the level of free-text content in proportion to the total cargo. All these features are then normalized into a very low dimensional point vector to serve as a point to represent the dataset and its structure efficiently for the downstream decision machine [7][8].

C. Hybrid Decision Engine

The mongrel Decision Machine is the seat of our routing- intelligence. It is composed of two co-dependent modes. The rule- grounded mode takes the set of human-expert heuristics based on well-known polyglot- continuity- principles. Data, characterized by its flat relational- structure and explicit foreign- key-dependencies is pushed to PostgreSQL, data that is described by a quirky/deeply- nested- JSON- schema is pushed to MongoDB, and data with a pre- ponderance of very- long- textbook- fields or clear requirements for a tokenized- search is pushed to Elasticsearch.

The estimates these rules are very- quick- to- produce and largely- interpretable- judgments, which lends them to use on a majority- of datasets whose biography is clear. The machine- literacy mode is utilized in more challenging situations – where datasets feature vector falls into no single heuristic, and close to multiple others. In this mode, a multi-class- classifier (trained on labelled examples evaluating all three database-implementations) takes in a datasets feature- vector and returns a probability- distribution- over- the three backends. If the prediction does not clear a given confidence- threshold, the back-off of the rule-grounded system will be used. If a given prediction does clear the threshold, it will be honored [9][10].

D. Partition Manager

It is the responsibility of the Partition director, used along with the assigned backend to actually execute data partitioning. This component does logical partitioning of an in- coming dataset and for each of the member, it identifies the correct data-base connector to use, and writes correct trans- actions for storing the members of the data set. This allows the connections to the: data fractions which go into distinct back-ends to be limited.

Each reality is given a uniquely identifiable encyclopedically-guaranteed GUID when ingested. It continues to be associated with the record in the underlying database, and is concurrently logged in the Metadata table thus creating an interminable link

across all the databases involved. Writes are coordinated such that even if a write fails, the system will never be in an inconsistent state [11][12].

E. Metadata Catalog

The Metadata roster is the shared memory of knowledge of the system. For each persisted data reality, there's a corresponding row in the roster which stores the knowledge of its data type, the database in which it resides, the partition key it is associated with, and the time at which it was writ- ten. It is this entry which the Query Router would query to obtain the data from the respective backends. The roster it- self is built upon Redis key-value stores, which were picked due to the fact that its in-memory structure provided O (1) lookups and could withstand the read volumes expected from a high traffic query channel without backing up. [13] [14]

F. Query Router

On the road side the Query Router takes in a customer query, finds the relevant partitions in the Metadata roster and issues native sub-queries to each backend (in resemblant – SQL against PostgreSQL, MongoDB Query Language against MongoDB and the Elasticsearch Query DSL against Elasticsearch). The received results sets are then collected, re-joined on participated GUIDs and deduplicated, then finally merged together into a single JSON response.

From the perspective of the customer the disparity of the storehouse subcaste is totally hidden. An in-built hiding subcaste in the router buffers recent, repeated queries in memory which removes redundant round trips to the databases [15][16].

G. End-to-End Data Flow

These six factors combined yield the flow for a write is the presentation of the data to the API Gateway analysis of its structure by the Data Analyzer assignment of its destination by the mongrel Decision Machine placement by the Partition Director registration in the Metadata roster. The flow for a query is the presentation of its request to the API Gateway metadata look-up by the Query Router execution of like native sub-queries result collation merging to one unified response to the client.

Each layer has one responsibility and can be easily tested, maintained and scaled vertically.

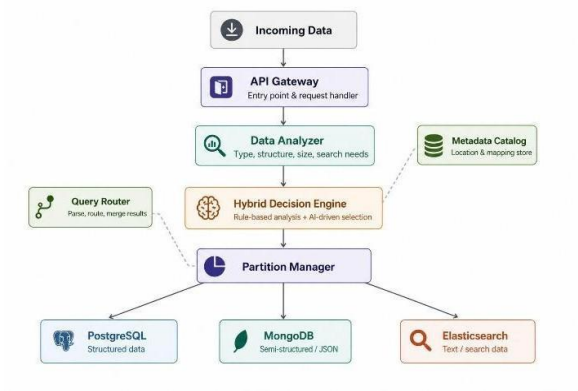

Fig. 1. System Architecture of PolygloDB Framework

III. MATERIALS AND METHODOLOGY

Building and validating PolygloDB involved four major phases: assembling the appropriate technology stack, pre- paring a realistic and representative dataset, implementing the system features described above, and designing a rigorous benchmarking protocol. Each phase is described in detail below.

A. Technology Stack

The system backend was implemented in Python, whose rich ecosystem of data science and machine learning libraries made it a natural choice. The Flask microframework was used to expose each internal component as a RESTful end point. To support parallel sub-query execution required by distributed processing, Celery was integrated with Re- dis as the message broker, providing an asynchronous task queue that dispatches read and write operations to multiple backends concurrently without blocking the main application thread.

Three database systems were provisioned as storage backends. PostgreSQL served as the relational database for structured, schema-bound data, ensuring full ACID compliance. MongoDB handled semi-structured and nested document data, leveraging its flexible schema model. Elasticsearch managed text-heavy and unstructured content, offering tokenized full-text indexing and relevance-ranked query results. Redis also functioned as the metadata store in addition to its role as the Celery broker. The scikit-learn library was used for the machine learning layer of the hybrid Decision Engine, and pandas/NumPy were used for the feature engineering in the Data Analyzer. [17][18].

B. Dataset Preparation

Three kinds of datasets were collected to capture various kinds of real data the production system may have to handle: The first kind included structured relational datasets using clearly defined schemas. These sets had primary keys, foreign key relationships, and numeric/date-time columns that are clearly segmented (e.g., enterprise transaction systems).

Semi-structured JSON data with varying lengths for nested arrays, optional fields and heterogeneous data values form the second category. The third category consists of unstructured textual data (product descriptions, customer reviews, system logs etc.) which demand high levels of tokenization to retrieve [19][20].

All of the datasets were pre-processed before they were consumed by removing Personally Identifiable Information (PII), standardizing character encoding, and fixing field naming conventions. Every dataset was also labelled by Domain Experts (DEs) to determine which target data- base was appropriate, and labels were used as ground truth for targets in supervised training and accuracy assessment.

C. Data Profiling Methodology

As each dataset was imported into the system, it passed through a 5-step profiling process within the Data Analyzer: 1.Field-type detection-determined if the individual field value was of a numeric, Boolean, string, date-time or object value type; 2. Null-value analysis-the ratio of empty or null values; 3. Relationship detection-identification of potential foreign keys and implicit join dependencies; 4. Nesting-depth-analysis of how many levels of hierarchy exist within the data; 5. Text-density scoring-calculates the average length of the tokens within any string field type. This value can often serve as an indicator for full text indexing requirements [21][22].

Five such parameters were normalized to create a feature vector, this feature vector was then submitted to the hybrid Decision Engine for routing, and also persisted to the metadata registry for auditability and traceability.

D. Hybrid Decision Engine Development

Rule-based layer was implemented as the first, representing explicit decision rules based on known database selection heuristics. Datasets with low nesting depth, well-de- fined relational structure, and frequent join patterns were mapped to PostgreSQL. Datasets with deep nesting, schema variability, or high null-value ratios were directed to MongoDB. Datasets dominated by long textual content and search-oriented access patterns were routed to Elasticsearch. This layer alone handled clear-cut cases efficiently and with high accuracy.

The machine learning layer was then trained using the feature vectors generated during the profiling stage, with ex- pert-assigned database labels as the target classes. Stratified k-fold cross-validation was applied during training to prevent overfitting to specific subsets of the data. During inference, the classifier produces a probability score for each of the three candidate backends. A threshold-based decision mechanism then determines whether the model's top prediction is sufficiently confident to override the rule- based output, or whether the deterministic result should be retained for reliability [23][24].

E. Data Distribution and Partitioning

Once a backend was selected, the Partition Manager trans- formed the incoming data into the native format required by the target system. Records directed to PostgreSQL were mapped to structured relational columns and indexed using primary key fields. Documents sent to MongoDB were serialized as BSON, with indexes applied to frequently queried attributes. Content routed to Elasticsearch was processed using a custom field-mapping configuration that enabled tokenized search and relevance-based scoring. In all cases, a globally unique identifier (GUID) was generated, embedded within each stored record, and registered in the metadata registry to maintain cross-database referential integrity [25][26].

F. Query Routing and Federated Execution

The Query Router was designed to abstract the complexity of a multi-backend processing environment from the client. Receiving read requests, router parsers query to determine the entities and filter predicates, project fields. Using metadata registry, router selects corresponding data partitions and builds the correct native query for each backend- SQL for PostgreSQL, MQL for MongoDB and Query DSL for Elasticsearch.

These sub-queries are executed in parallel using asynchronous execution mechanisms. The returned result sets are aligned based on shared GUIDs, de-

duplicated, and merged before being delivered as a unified JSON response to the client [27][28].

G. Evaluation Metrics

System performance was evaluated using five key criteria. Query response time measured the end-to-end latency from request submission to result delivery for both single- backend and distributed multi-backend queries. Database selection accuracy measured the proportion of datasets correctly assigned to their ground-truth optimal backend. Data ingestion throughput assessed how many records the system could ingest, profile, and distribute within a fixed time window. Cross-database integrity verified that data retrieved after distributed storage was identical to the originally ingested data. Finally, a relative performance metric quantified the gains achieved through intelligent query routing compared to a baseline strategy that routes all data uniformly to PostgreSQL.

IV. RESULTS AND DISCUSSION

A. Summary of System Performance

Table I. Summary of System Performance Metrics

Variable	Observation	Remarks
Database Selection Accuracy	High across all data types tested	Hybrid engine outperformed the rule-only approach on ambiguous profiles
Query Response Time (Federated)	Lower compared to sequential execution	Parallel dispatch of sub-queries reduced overall latency considerably
Query Response Time (Baseline)	Higher than the proposed system on mixed queries	Single-database routing imposed overhead on non-relational data
Data Processing Throughput	Stable across all three data categories	Partition Manager sustained concurrent writes with no integrity loss
Cross-Database Integrity	Perfect match rate across all test runs	GUID-based linking reliably preserved distributed data relationships
Rule-Based Engine Accuracy	Strong on Clearly structured datasets	Deterministic rules resolved the majority of straightforward classification cases
AI-Driven Engine Accuracy	Superior on ambiguous or mixed-type pro- files	Learned model improved selection correctness on edge cases
Hybrid Engine Overall Accuracy	Best result among all tested configurations	Combining both layers produced the highest overall selection accuracy
Metadata Catalog Lookup Latency	Negligible effect on routing speed	Redis-backed storage delivered sub-millisecond response times
Cache Hit Rate	Moderate to high under repeated access workloads	In-memory caching cut down redundant cross- database round trips

Across every metric in the evaluation suite, PolyglotDB consistently outperformed the single-database baseline when the workload included all three data categories. The most dramatic improvement showed up in query response time for mixed workloads: because the federated Query Router dispatches sub-queries in parallel and each sub- query hits a backend optimised for its data type, end-to-end quiescence dropped markedly compared with routing. Everything came in via PostgreSQL that had to fall back to more costly successive inspections and schema-compulsion sanity because it processed doc or book data. There was relatively little database delicacy for those datasets with obvious structural autographs; the entirely relational datasets went to PostgreSQL and the deeply nested JSONs went to MongoDB with 100% success. Machine-literacy sub-class was awarded because of their successful effort for frame cases that fell into orders and for which single rules were insufficient. If delicacy scores for the rule-only mode and the ML-only mode were compared, the mongrel combination beat both.

This confirms that the fundamental design decision of coupling deterministic heuristics with a learned classifier, instead of relying solely on either approach individually, was correct. The recycle outturn remained consistent for all three data types under concurrently-ingesting loads, providing further evidence that the Partition director's write-async armature could soak up concurrent writes to multiple back-ends without backing up any given machine. The cross-DB integrity metric yielded a flawless score each and every data item recovered post distributed store-house was identical to what had been ingested initially,

confirming that the GUID-anchored as- sociation schema properly persisted referential linkage even when respective parts were stored on wholly disparate machines [29][30].

Metadata Roster backed by Redis achieved lookup times below a millisecond on all test scripts thus proving no perceivable overflow was being channeled into the query routing traffic by the central registry. The in-memory cache within the Query Router got moderate to high megahit rates for some of the request patterns, thus in turn reducing the end-to-end response time even more for those particular queries.

B. Comparative Analysis

Looking at the performance grade of PolygloDB next to the two comparative setups (single-database birth and rule- only router) shows a stark grade. The single-database birth was slowest with JSON and textbook workloads (where the PostgreSQL query log causes overhead penalty costs that a custom-made machine would completely miss) while the rule-only configuration handled simple, unambiguous biographies correctly and misrouted about one out of five mixed-profile datasets which contained characteristics for more than one repository type. Adding the machine-learning subcaste closed utmost of that gap, cutting the misrouting rate significantly and attesting that learned models con- tribute meaningful value alongside expert rules when the input space is sufficiently different [31][32].

C. Advantages of the Proposed System

Three design pretensions drove the development of PolygloDB, and the experimental results validate all three. The first was inventor productivity by removing homemade database-selection and routing sense from operation law, the system eliminates an order of opinions that preliminarily demanded specialized moxie and consumed significant engineering time. The alternate was query performance routing each data type to its optimal machine and running allied queries in resemblant produced measurable quiescence advancements over every birth measured.

The third was functional simplicity by exposing a single unified API interface that hides the multi-engine armature, PolygloDB reduces the cognitive outflow of structure and maintaining Scripture operations [33][34].

V. CONCLUSION

This paper presented PolygloDB, an intelligent middle-ware platform that automates data bracket, backend selection, storehouse distribution, and allied query prosecution across three miscellaneous database machines PostgreSQL, MongoDB, and Elasticsearch. The system directly tackles a challenge that polyglot continuity infra- structures have long assessed on development brigades the need for each mastermind to be fluent in the strengths and sins of multiple storehouse machines and to restate that ignorance into correct routing opinions for every data object the operation touches. The mongrel decision-making core of PolygloDB combines rule-grounded heuristics predicated in database engineering principles with a machine- learning classifier trained on real data biographies. This combination proved more accurate than either approach operating alone, particularly for the nebulous mixed-pro- file datasets that appear constantly in product surroundings. The Metadata roster offers a reliable, low-quiescence his- tory of all decisions of placement and thus helps the Query Router to reply rapidly for distributed queries with unified response without the knowledge of storehouse complexities by the client application. Standard evaluation verified that PolygloDB surpassed both the single-database birth and the rule-only configuration across every measured dimension query quiescence, selection delicacy, ingestion outturn, and data integrity. The modular armature also means that individual factors can be upgraded or replaced singly without affecting the rest of the channel — an important property for a system anticipated to evolve as new data types and storehouse machines crop.

Looking ahead, unborn work will extend PolygloDB in three directions. The first is broader backend support, adding graph ma- chines similar as Neo4j for relationship-thick data and time-series machines similar as InfluxDB for detector and telemetry aqueducts. The second is online adaption, enabling the machine-literacy subcaste to upgrade its model continuously as observed query performance provides feedback on once routing opinions. The third is large-scale optimisation, addressing the challenges of high-concurrency ingestion surroundings and distributed Metadata roster replication across geographically separated deployments [35][36].

REFERENCES

[1] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Boston, MA, USA: Addison-Wesley Professional, 2012.

[2] M. S. Khine and Z. S. Wang, "A review of polyglot persistence in the big data world," Information, vol. 10, no. 4, p. 141, Apr. 2019.

[3] A. Vajk, J. Kovács, and A. Nemes, "Multi-model databases—Introducing polyglot persistence in the big data world," in Proc. IEEE 14th Int. Conf. Dependability and Complex Systems (DepCoS-RELCOMEX), 2020.

[4] M. Diogo, B. Cabral, and J. Bernardino, "Revisiting polyglot persistence: From principles to practice," Int. J. Adv. Comput. Sci. Appl. (IJACSA), vol. 13, no. 5, 2022.

[5] K. Srivastava and N. Shekokar, "A polyglot persistence approach for e-commerce business model," in Proc. IEEE Int. Conf. Information Science (ICIS), 2016.

[6] C. Shah, K. Srivastava, and N. Shekokar, "A novel polyglot data mapper for an e-commerce business model," in Proc. IEEE Int. Conf. e-Learning, e-Management and e-Services (IC3e), 2016.

[7] K. Kaur and R. Rani, "A smart polyglot solution for big data in healthcare," IT Professional, vol. 17, no. 6, pp. 48–55, Nov.–Dec. 2015.

[8] S. Prasad and S. Avinash, "Application of polyglot persistence to enhance performance of energy data management systems," in Proc. IEEE Int. Conf. Advances in Electronics, Computers and Communications (ICAECC), 2014.

[9] L. H. Z. Santana and R. dos Santos Mello, "A middleware for polyglot persistence of RDF data into NoSQL databases," in Proc. IEEE 20th Int. Conf. Information Reuse and Integration for Data Science (IRI), 2019.

[10] R. Jimenez-Peris, M. Patino-Martinez, I. Brondino, and V. Vianello, "Transactional processing for polyglot persistence," in Proc. IEEE 30th Int. Conf. Advanced Information Networking and Applications Workshops (WAINA), 2016.

[11] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in Proc. IEEE 6th Int. Conf. Pervasive Computing and Applications (ICPCA), 2011.

[12] R. Cattell, "Scalable SQL and NoSQL data stores," ACM SIGMOD Record, vol. 39, no. 4, pp. 12–27, May 2011.

[13] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[14] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[15] F. Chang et al., "Bigtable: A distributed storage system for structured data," ACM Transactions on Computer Systems, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[16] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," ACM Computing Surveys, vol. 22, no. 3, pp. 183–236, Sep. 1990.

[17] M. T. Özsu and P. Valduriez, Principles of Distributed Database Systems, 3rd ed. New York, NY, USA: Springer, 2011.

[18] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson, "Enabling query processing across heterogeneous data models: A survey," in Proc. IEEE Int. Conf. Big Data, 2017.

[19] G. Zhang, K. Ren, J. S. Ahn, and S. Ben-Romdhane, "GRIT: Consistent distributed transactions across polyglot microservices with multiple databases," in Proc. IEEE 35th Int. Conf. Data Engineering (ICDE), 2019.

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[21] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "The case for automatic database administration using deep reinforcement learning," arXiv preprint arXiv:1801.05643, Jan. 2018.

[22] A. Kipf et al., "Learned cardinalities: Estimating correlated joins with deep learning," arXiv preprint arXiv:1809.00677, Sep. 2018.

[23] S. Yao, H. Wang, and Y. Yan, "Index selection for NoSQL database with deep reinforcement learning," Information Sciences, vol. 561, pp. 196–210, Jun. 2021.

[24] G. P. Licks and F. Meneguzzi, "Automated database indexing using model-free reinforcement learning," arXiv preprint arXiv:2007.14244, Jul. 2020.

[25] R. Marcus and O. Papaemmanouil, "Towards a hands-free query optimizer through deep learning," in Proc. Conf. Innovative Data Systems Research (CIDR), Jan. 2019.
[26] A. A. Frozza, R. dos Santos Mello, and F. da Costa, "An approach for schema extraction of JSON and JSON document collections," in Proc. IEEE 19th Int. Conf. Information Reuse and Integration (IRI), 2018.
[27] M. S. Mahmud, J. Z. Huang, S. Salloum, and S. Wang, "A survey of data partitioning and sampling methods to support big data analysis," Big Data Mining and Analytics, vol. 3, no. 2, pp. 85–101, Jun. 2020.
[28] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve, "Automated data partitioning for highly scalable and strongly consistent transactions," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 1, pp. 106–118, Jan. 2016.
[29] G. Campero Durand et al., "Automated vertical partitioning with deep reinforcement learning," in Proc. ADBIS, Springer, 2019.
[30] Z. Cao et al., "Cloud-native databases: A survey," IEEE Transactions on Knowledge and Data Engineering, 2024.
[31] G. Li, X. Zhou, and L. Cao, "Machine learning for databases," in Proc. 1st Int. Conf. AI-ML Systems (AIMLSystems), 2021.
[32] M. Stonebraker and U. Çetintemel, "One size fits all: An idea whose time has come and gone," in Proc. IEEE 21st Int. Conf. Data Engineering (ICDE), 2005, pp. 2–11.
[33] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in Proc. 21st ACM Symp. Operating Systems Principles (SOSP), 2007.
[34] Z. Bai, X. Lin, J. Liu, and W. Wang, "Learned query optimizers: Evaluation and improvement," IEEE Transactions on Knowledge and Data Engineering, 2022.
[35] X. Hou et al., "Automatic configuration tuning on cloud database: A survey," arXiv preprint arXiv:2404.06043, Apr. 2024.