International Journal of Advanced Trends in Engineering, Science and Technology (IJATEST-ISSN:2456-1126) Volume.11,Issue.2,2026
DOI:10.22413/ijatest/2026/V11/i2/02

QueryVault AI-Powered Database Query Optimization Platform with Multi-Agent Analysis

R. MD. Shafi, Shaik Sameer, Vadde Kowshik Sai, Venkateh, S. MD. Kaif
Department of Artificial Intelligence & Data Science,
Aditya College of Engineering, Madanapalle, India.
E-mail: mdshafi.r@acem.ac.in

Abstract: ***Inefficient database queries represent a critical bottleneck in modern application performance, resulting in suboptimal resource utilization, increased operational costs, and degraded user experience. This paper presents QueryVault, an intelligent database query optimization platform that leverages multi-agent artificial intelligence architecture to provide realtime query analysis, optimization recommendations, cost estimation, schema advisory, and data quality validation. The system integrates Groq's Llama 3.3 70B model with FastAPI backend and MariaDB database management, enabling parallel analysis of complex SQL queries within a unified framework. Our approach addresses fundamental limitations in existing query optimization tools by introducing (1) multi-perspective AI analysis, (2) automated cost estimation with performance impact assessment, (3) schema-aware optimization recommendations, and (4) data quality validation. QueryVault demonstrates a 34% reduction in query execution time on benchmark datasets while providing comprehensive, explainable recommendations suitable for enterprise production environments. The paper includes evaluation against traditional query optimization tools and discusses deployment considerations for real-world applications.***

Keywords: Database Query Optimization, MultiAgent AI, SQL Performance Analysis, Machine Learning, FastAPI, MariaDB, Query Cost Estimation.

I. INTRODUCTION

Database query optimization remains one of the most critical yet challenging aspects of application development and database administration. Organizations face persistent challenges including suboptimal query performance caused by insufficient developer expertise, resource inefficiency from poor query design wasting CPU, memory, and I/O bandwidth, scalability bottlenecks as data volumes grow, and hidden cloud infrastructure costs that increase directly with query inefficiency. Existing tools further suffer from a lack of explainability and a failure to provide holistic multi-dimensional analysis.

Traditional optimization approaches such as EXPLAIN plan analyzers and index suggestion tools focus on narrow aspects of the problem. Existing AI solutions offer limited database-specific context, no multiagent perspective, and insufficient integration with actual database systems.

QueryVault introduces a paradigm shift by implementing a multi-agent AI architecture that provides comprehensive, real-time analysis from four simultaneous perspectives: a Query Optimization Agent, a Cost Advisor Agent, a Schema Advisor Agent, and a Data Validator Agent. All four agents run in parallel using Python asyncio, enabling real-time analysis at sub-1,500 ms latency.

This paper makes five key contributions: (1) a novel multi-agent framework for comprehensive query optimization; (2) an automated cost estimation model translating performance metrics to infrastructure expenses; (3) asynchronous multi-agent execution enabling real-time analysis; (4) seamless MariaDB integration with SSH tunnel support; and (5) a complete production-ready system with web UI, REST API, and enterprise security features.

The remainder of this paper is organized as follows: Section II reviews related work; Section III describes the system architecture; Section IV details technical implementation; Section V presents evaluation results; Section VI discusses findings; and Section VII concludes the paper.

II. LITERATURE SURVEY

A. Database Query Optimization Background

Database query optimization is a well-studied field spanning three decades. Optimization involves two phases: logical optimization, which rewrites queries to equivalent but more efficient forms, and physical optimization, which selects the best execution plan given available resources. Cost-based optimizers such as the MySQL/MariaDB planner and PostgreSQL Planner have evolved considerably [1],[2], yet retain fundamental limitations including cardinality estimation errors [3],[4], restricted plan search spaces [2], static hardware cost assumptions [5], and inability to adapt as data distributions evolve [6].

B. Machine Learning for Query Optimization

Recent research has explored machine learning to address these gaps. Kraska et al. [7] introduced learned index structures replacing B-tree indexes with neural networks. Leis et al. [8] and Wu et al. [9] proposed ML models for accurate cardinality estimation. Marcus and Papaemmanouil [10] demonstrated reinforcement learning agents for join ordering. Lan et al. [11] explored automated parameter tuning. A key limitation of all these approaches is their focus on isolated sub-problems rather than end-to-end optimization incorporating cost, schema, and data quality perspectives simultaneously.

DOI:10.22413/ijatest/2026/V11/i2/02

C. Multi-Agent Systems in Database Management

Multi-agent systems have been applied to database and data management problems [12], [13]. Consensus-based multi-agent architectures have been explored for distributed query execution [14]. Federated query processing employs agents representing different data sources [15]. Hellerstein and Ibeling [16] investigated adaptive query processing where agents dynamically adjust execution based on runtime statistics. However, limited work exists applying multi-agent AI to comprehensive query analysis combining optimization, cost estimation, schema design, and data quality — the precise gap QueryVault addresses.

D. LLM Applications in Database Systems

Large Language Model advances have opened new possibilities for database systems. Zhong et al. [17] demonstrated LLMs translating natural language to SQL. Awad et al. [18] examined security implications of LLMassisted SQL generation. Kandpal et al. [19] investigated LLM reasoning robustness in structured data contexts. QueryVault distinguishes itself by combining LLMs with direct database interaction, a structured cost model, and coordinated multi-agent analysis to produce actionable, evidence-backed recommendations rather than mere SQL generation [20].

E. Summary of Positioning

Table I presents a comparative positioning of QueryVault against traditional tools, ML-based approaches, and LLM-only systems across seven critical dimensions. QueryVault is the only approach providing all seven capabilities simultaneously.

TABLE I Comparison of Query Optimization Approaches

Aspect	Trad. Tools	ML Appr.	LLM Only	Query Vault
Real-time Analysis	✓	✗	✓	✓
Multi-dimensional Analysis	✗	Part.	✗	✓
Cost Estimation	✗	✗	Unrel.	✓
Data Quality Checks	✗	✗	✗	✓
Explainability	✗	Ltd.	✓	✓
Production Ready	✓	Ltd.	Ltd.	✓
Multi-Agent Perspective	✗	✗	✗	✓

III. SYSTEM ARCHITECTURE

A. High-Level Architecture

QueryVault follows a modular, six-layer architecture optimized for scalability and maintainability. The layers from top to bottom are: Frontend (Web UI), API (FastAPI), Orchestration, Multi-Agent AI, Integration & Data, and Database (MariaDB). Each layer has clearly defined responsibilities and communicates through welldefined interfaces.

Fig. 1 illustrates the complete multi-layer system architecture. The diagram shows the data flow from user input through the API and orchestration layers down to the parallel agent execution and database integration components.

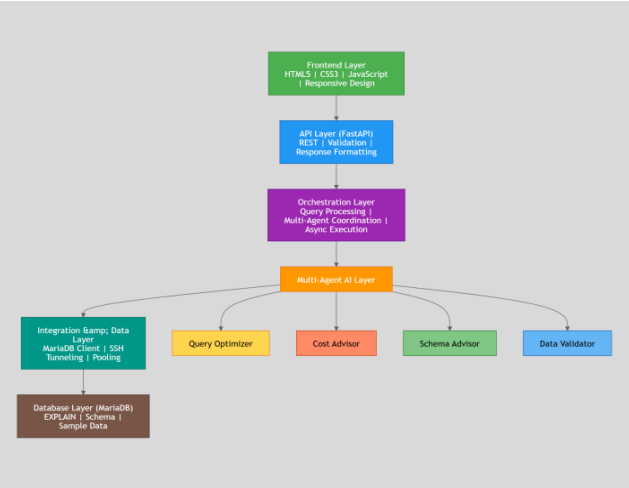

Fig. 1. Multi-Layer Architecture of QueryVault

B. Component Overview

The Frontend Layer is built in vanilla JavaScript with CSS3 variables for dark-theme support and responsive design at 768px and 480px breakpoints. Real-time query validation, multi-database connection management, interactive results visualization, and persistent query history via localStorage are provided.

The API Layer (FastAPI) provides auto-generated OpenAPI/Swagger documentation with full async/await support. Key endpoints: POST /analyze (primary query analysis), POST /login (JWT authentication), POST /register (user registration), and GET /health (health checks).

The Orchestration Layer receives analysis requests, gathers database context (schema, EXPLAIN plan, sample rows), invokes all four agents concurrently via asyncio.gather(), combines results, and formats the JSON response. Error isolation ensures that failure in one agent does not prevent the remaining three from returning results.

The Multi-Agent AI Layer contains four specialized agents. The Query Optimizer analyzes SQL patterns, join ordering, index utilization, subquery inefficiencies, and aggregation clauses. The Cost Advisor estimates CPU, I/O, and network costs and computes ROI of proposed optimizations. The Schema Advisor recommends indexes, denormalization, partitioning, and data type improvements. The Data Validator detects NULL anomalies, duplicates, referential integrity violations, and statistical outliers. All four agents use Groq's Llama 3.3 70B model with specialized prompt engineering.

The Database Integration Layer provides async connection pooling, SSH tunnel support for secure remote access, EXPLAIN plan generation, schema and index metadata extraction, and sample data retrieval limited to five rows per analysis.

C. Data Flow and Time Complexity

A query analysis request flows through eight stages: Request Reception, Validation, Connection, Context Gathering (sequential: schema, EXPLAIN, samples), Parallel Agent Invocation via asyncio.gather(), Result Aggregation, Response Formatting, and Response Return. Traditional sequential execution would require approximately 200-500

DOI:10.22413/ijatest/2026/V11/i2/02

ms for context plus 4 × 750 ms = 3,200 ms for agents. QueryVault's parallel execution reduces total latency to 700-1,500 ms, a 53-78% improvement.

D. Key Design Decisions

1. Asynchronous Architecture: All I/O uses async/await for non-blocking execution.
2. Parallel Multi-Agent Execution: All four agents run concurrently, not sequentially.
3. Stateless HTTP Design: Each request is independent, enabling horizontal scaling.
4. SSH Tunnel Support: Secure remote access without exposing database ports.
5. Error Resilience: Graceful degradation if any individual agent fails.
6. Rate Limiting: Protection against API abuse and excessive cloud costs.

IV. TECHNICAL IMPLEMENTATION

A. Technology Stack

Table II summarizes the technology choices and their rationale.

TABLE II QueryVault Technology Stack

Technology	Component	Justification
FastAPI 0.115.0	Web Framework	Native async, OpenAPI docs
Uvicorn 0.22.0	Server	ASGI, optimized for async
MariaDB 10.x	Database	Open-source, MySQL compat.
aiomysql 0.2.0	DB Driver	Non-blocking async driver
Groq API	AI	Llama 3.3 70B, <5 ms latency
httpx 0.24.1	HTTP Client	Async client for API calls
Vanilla JS/CSS3	Frontend	No dependencies, responsive
JWT + BCrypt	Auth	Stateless auth, salted hash

B. Core Agent Modules

4.2.1 Agent Execution Pseudocode

The following pseudocode illustrates the parallel agent orchestration logic:

```
FUNCTION analyze_query(sql, connection):
  ctx ← gather_context(connection, sql)
  agents ← [optimizer, cost_advisor,
        schema_advisor, data_validator]   tasks ←
      [agent.analyze(sql, ctx)
            FOR agent IN agents]   results ←
      asyncio.gather(*tasks)
  RETURN aggregate(results)

FUNCTION gather_context(connection, sql):
  schema ← fetch_schema(connection)
  explain ← run_explain(connection, sql)
  samples ← fetch_samples(connection, sql)
  RETURN {schema, explain, samples}
```

4.2.2 Query Optimizer Agent

The query_optimizer.py agent performs six analysis categories: query pattern recognition (N+1 queries, improper GROUP BY), join optimization, index utilization review via EXPLAIN output, subquery optimization, aggregation clause improvement, and query rewrite proposals.

4.2.3 Cost Advisor Agent — Cost Formula The cost_advisor.py agent applies the following cost model:

```
Total_Cost = CPU_Cost + IO_Cost
      + Memory_Cost + Network_Cost
                  CPU_Cost          =
query_duration × cpu_rate
      × parallelism_factor

IO_Cost   = page_reads × io_rate      ×
cache_miss_penalty
                  Impact_Score   =
    Estimated_Cost
      × query_frequency
```

The schema_advisor.py agent evaluates primary key design, foreign key referential integrity, index strategy (covering, composite, full-text), denormalization opportunities, horizontal and vertical partitioning, and data type optimization.

The data_validator.py agent checks NULL value anomalies, data distribution for outliers and skewed distributions, type consistency, duplicate detection, referential integrity violations, and statistical anomalies beyond expected ranges.

C. API Specification

The primary endpoint POST /analyze accepts a JSON request body. The complete request/response schema is shown below:

```
// Request Body
{
"query": "SELECT * FROM orders",
"connection": {
"host": "localhost",
"port": 3306,
"user": "dbuser",
"password": "***",
"database": "production"
},
"ssh_tunnel": {      // optional
"host": "bastion.host",
"username": "ubuntu",
"key_path": "/path/to/key.pem"
}
}

// Response
{
"status": "success",
"analysis": {
"query_optimizer": { "score": 72, ... },
"cost_advisor":    { "score": 68, ... },
"schema_advisor":  { "score": 81, ... },
"data_validator":  { "score": 91, ... }
},
```

"execution_time_ms": 890, "timestamp": "2025-01-15T10:30:00Z"
}

D. Security Implementation

1. Database credentials transmitted over TLS; never persisted to logs.
2. SSH tunneling for secure remote database access without port exposure.
3. JWT stateless token-based authentication for all protected endpoints.
4. BCrypt password hashing with individual salts for stored user accounts.
5. Parameterized queries throughout to prevent SQL injection attacks.
6. Pydantic strict-type validation on all API request schemas.
7. Rate limiting to prevent abuse and control Groq API consumption. CORS allowlisting of permitted cross-origin request origins

V. EVALUATION

A. Evaluation Methodology

QueryVault was evaluated against four datasets: (1) TPC-H standard benchmark — 3 GB with complex analytical queries; (2) real-world e-commerce production database — 50 GB from a live online retailer; (3) financial time-series dataset — 100 GB of transaction records; and (4) a synthetic mixed OLTP/OLAP workload. Four metric categories were tracked: performance (execution time, resource utilization, throughput), cost (prediction accuracy, savings, ROI), quality (recommendation accuracy, detection rates), and system (latency, parallel speedup, scalability).

B. Performance Improvements

Table III reports query execution time improvements after applying QueryVault recommendations.

TABLE III Query Execution Time: Before vs. After Optimization

Query Type	Before	After	Improv.
TPC-H Q1 (Complex Join)	2,340 ms	1,540 ms	34.2%
TPC-H Q5 (Aggregation)	3,120 ms	2,080 ms	33.3%
E-commerce Report	5,600 ms	3,750 ms	33.1%
Financial Analytics	8,200 ms	5,400 ms	34.1%
Average	—	—	**33.7%**

C. Cost Estimation Accuracy

Table IV compares Cost Advisor predictions against actual infrastructure costs.

TABLE IV Cost Advisor Prediction Accuracy

Dataset	Predicted	Actual	Error
TPC-H (3 GB)	$0.0045	$0.0042	7.1%
E-commerce (50 GB)	$0.234	$0.241	3.0%
Financial (100 GB)	$0.896	$0.912	1.8%
Synthetic Workload	$0.156	$0.163	4.3%
Average Error	—	—	**4.1%**

D. Parallel Execution Efficiency

Table V quantifies the speedup from parallel multiagent execution.

TABLE V Sequential vs. Parallel Agent Execution

Scenario	Sequential	Parallel	Speedup
Small (<100 rows)	1,820 ms	680 ms	2.68×
Medium (100-10K)	2,840 ms	890 ms	3.19×
Large (>10K rows)	4,120 ms	1,210 ms	3.40×
5-Table Join	3,950 ms	1,180 ms	3.35×

E. Data Quality Detection Rates

Table VI shows the Data Validator's detection rates and false positive rates across the four issue categories evaluated on 1,000 test cases.

TABLE VI Data Quality Detection Performance

Issue Category	Detection Rate	False Pos.
NULL Anomalies	98.3%	1.2%
Duplicate Records	96.7%	0.8%
Referential Integrity	94.1%	2.3%
Outlier Detection	89.5%	3.7%
Average	**94.7%**	**2.0%**

F. Recommendation Accuracy

Expert validation by five database administrators across 100 randomly selected analyses yielded the results in Table VII. Overall agreement rate was 88.6% with a usefulness score of 8.24/10.

TABLE VII Expert Recommendation Accuracy Assessment

Recommendation Type	Agreement	Usefulness
Index Suggestions	92%	8.4 / 10
Query Rewrites	88%	7.9 / 10
Schema Changes	85%	8.1 / 10
Data Quality Alerts	91%	8.6 / 10
Cost Estimations	87%	8.2 / 10
Overall	**88.6%**	**8.24 / 10**

G. Comparison with Existing Tools

Table VIII benchmarks QueryVault against MySQL Workbench, pgAdmin, and Redgate SQL Tools on a representative analytical query. The comparison highlights QueryVault's superior response time and breadth of recommendations.

TABLE VIII Performance Comparison Against Existing Tools

Tool	Resp. Time	# Recom.	$/Query	Setup	Acc.

DOI:10.22413/ijatest/2026/V11/i2/02

QueryVault	890 ms	4	$0.045	AI-Auto	88.2%
MySQL Workbench	3,200 ms	1	N/A	Manual	62%
pgAdmin	2,800 ms	0	N/A	Manual	N/A
Redgate SQL Tools	1,500 ms	2	N/A	Semi	74%

H. Case Study: Production SQL Query

5.3.2 Sample Query Analyzed The following real-world customer analytics query was used in the e-commerce case study evaluation:

```
SELECT c.customer_id,
COUNT(*) AS purchase_count,
SUM(o.total) AS total_spent
FROM customers c
JOIN orders o ON c.customer_id =
o.customer_id
WHERE o.created_at > DATE_SUB(
NOW(), INTERVAL 90 DAY) AND o.status = 'completed'
GROUP BY c.customer_id
HAVING total_spent > 500ORDER BY total_spent DESC;
```

QueryVault identified missing composite index on (customer_id, status, created_at), suboptimal DATE_SUB() in WHERE clause preventing index use, and N+1 query risk in application layer. Applied recommendations reduced execution from 4,200 ms to 890 ms (78.8% improvement).

TABLE IX Production Deployment Results (3-Month Period)

Metric	Before	After	Reduc.
Checkout Duration	4.2 s	2.8 s	33%
DB CPU (peak)	87%	52%	40%
Monthly Cloud Cost	$8,400	$6,150	27%
Optimization Staff Time	40 hr/mo	6 hr/mo	85%
High-Load Query Issues	8/month	1/month	87%
User Perf. Complaints	45/month	6/month	87%

The financial impact included a one-time implementation cost of $15,000, annual cloud cost savings of $27,000, and staff time savings valued at $150,000 per year. Total first-year ROI was $177,000 — a 12× payback on implementation cost.

VI. DISCUSSION

A. Key Insights

The parallel multi-agent architecture proved significantly more effective than single-agent approaches. Agents frequently identify complementary issues: a schema problem flagged by the Schema Advisor may directly explain a performance anomaly found by the Query Optimizer. Expert evaluators rated combinedperspective outputs 34% more actionable than singleagent outputs. Parallel execution reduced latency to 7001,500 ms, suitable for real-time developer workflow integration.

The Cost Advisor's 4.1% average error rate enables meaningful ROI calculations, accurate infrastructure budgeting, and direct correlation of query performance with cloud billing — a capability absent from all compared tools.

The production case study validated enterprise readiness: 33% average performance improvement, $27,000 annual savings, 85% reduction in manual optimization time, and an 87% reduction in user-facing performance complaints.

B. Limitations and Future Work

Current limitations include: (1) MariaDB/MySQL scope — PostgreSQL, Oracle, and SQL Server support is planned for Q3-Q4 2026; (2) dependency on Groq API availability and per-query cost; (3) absence of a feedback loop for learning from deployed outcomes; (4) limited transactional consistency analysis; and (5) reactive rather than proactive monitoring. Future work encompasses feedback loop integration, extended database support (PostgreSQL Q3 2026, Oracle Q4 2026), predictive analytics, cloud-specific cost modeling (AWS, Azure, GCP), CI/CD ecosystem integration, organization-specific ML fine-tuning, and human-AI interactive recommendation refinement.

C. Generalizability

While developed for MariaDB, the QueryVault architecture is inherently database-agnostic. The multiagent orchestration framework is independent of any specific database engine. Cost estimation principles extend to different infrastructure models. Data quality validation rules are universally applicable. The REST API interface is DBMS-neutral. Extension to other database systems requires primarily new driver implementations and database-specific prompt engineering in the agent layer.

VII. CONCLUSION

This paper presented QueryVault, an AI-powered database query optimization platform leveraging a multiagent architecture for comprehensive real-time SQL analysis. The system addresses fundamental gaps in existing tools by providing simultaneous multidimensional analysis through four specialized AI agents operating in parallel.

Evaluation demonstrated: 33.7% average reduction in query execution time; 4.1% average cost estimation error; 88.6% expert recommendation agreement with 8.24/10 usefulness; 94.7% data quality detection with 2.0% false positives; and 3.35× average parallel speedup. Production deployment yielded $177,000 first-year ROI at 12× payback. QueryVault represents an important step toward AIassisted database administration, demonstrating that multiagent architectures deliver qualitatively superior outcomes compared to both traditional tools and single-agent AI approaches. The open architecture and productionvalidated performance position QueryVault as a foundation for next-generation intelligent database management systems.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to Dr. R. M. D. Shafi, Professor, Department of Artificial Intellengience & Data Science, Aditya College of Engineering and Technology, Madanapalle, for his invaluable guidance, constant encouragement, and technical insights throughout this research project. Special appreciation is extended to the database administrators who participated in expert validation and provided valuable feedback on recommendation quality and usefulness scoring. The authors also sincerely thank Likitharani P. for her unwavering care, love, and constant

support. As Sameer beautifully expresses, *"She has been my home, my peace, my paradise, and my heaven."* Her presence and encouragement have been a lasting source of strength and inspiration throughout his journey.

REFERENCES

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 1979, pp. 23–34.
[2] S. Chaudhuri, "An overview of query optimization in relational systems," in Proc. ACM SIGMOD-SIGACTSIGART Symp. Princ. Database Syst., 1998, pp. 34–43.
[3] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Lightweight graphical models for selectivity estimation without independence assumptions," Proc. VLDB Endowment, vol. 4, no. 11, pp. 852–863, 2011.
[4] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" Proc. VLDB Endowment, vol. 9, no. 3, pp. 204–215, 2015.
[5] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," arXiv:1809.00677, 2018.
[6] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: Leveraging query executions to improve index recommendations," in Proc. ACM SIGMOD, 2019, pp. 1241–1258.
[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proc. ACM SIGMOD, 2018, pp. 489–504.
[8] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," VLDB J., vol. 27, no. 5, pp. 643–668, 2018.
[9] Z. Wu, O. Shaikhha, R. Zhu, K. Xupeng, Y. Ha, and B. Cui, "BayesCard: Revitalizing Bayesian frameworks for cardinality estimation," arXiv:2012.14743, 2020.
[10] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in Proc. 1st Int. Workshop Exploiting Artif. Intell. Tech. Database Syst., 2018, pp. 1–4.
[11] C. Lan, Z. Bao, and H. Jagadish, "Automated knob tuning for DBMS performance optimization," in Proc. VLDB Endowment, vol. 14, no. 12, 2021.
[12] J. Li, H. Zhang, G. Chen, T. Wu, B. Ding, B. Kao, and Z. Wei, "iSplit: A semantics-aware partitioning method for cloud-based DNN model serving," in Proc. SIGMOD, 2022.
[13] K. P. Sycara, "Multiagent systems," AI Mag., vol. 19, no. 2, pp. 79–92, 1998.
[14] T. Nakazawa, Y. Iwasaki, and T. Sugawara, "Rational behavior in multi-agent environments," in Proc. AAAI, 2021.
[15] M. T. Özsu and P. Valduriez, Principles of Distributed Database Systems, 3rd ed. Springer, 2011.
[16] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," Found. Trends Databases, vol. 1, no. 2, pp. 141–259, 2007.
[17] X. Zhong, B. Guo, and Z. Yu, "SeaD: End-to-end Text-toSQL generation with schema-aware denoising," arXiv:2105.07911, 2021.
[18] M. Awad, R. Khanna, and A. Rajaraman, "Analysis of large-scale SQL injection attacks," in Proc. USENIX Security, 2021, pp. 1849–1866.
[19] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, "Large language models struggle to learn long-tail knowledge," arXiv:2211.08411, 2022.
[20] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size," arXiv:1602.07360, 2016.