# Survey Report On Innovations in Modern Database Systems — Predictive Modeling, Generative AI, and Serverless Co-Design

*Gayathri Bama*
*bgayathribaman@gmail.com*
*Student ID - 11697946*
*University of North Texas (Dept.of Computer Science)*

## ABSTRACT:

This survey presents a detailed analysis of three cutting-edge approaches transforming modern database systems: (1) *Stage*, a hierarchical query execution time prediction system for Amazon Redshift; (2) a study evaluating the relational completeness and correctness of SQL, relational algebra, and relational calculus queries generated by ChatGPT; and (3) *Limbo*, a re-architecture of SQLite for serverless environments that uses asynchronous I/O to improve latency and scalability. These approaches represent three vital trends in database system evolution: machine learning-enhanced query optimization, AI-assisted code generation, and runtime/database architectural co-design. This report discusses the techniques, compares their strengths and weaknesses, presents empirical results, and concludes with insights on their practical impact.

## KEYWORDS:

Query Performance Prediction, Relational Completeness, Asynchronous I/O, Serverless Computing, SQL Code Generation, Machine Learning in Databases

## 1. INTRODUCTION:

Modern database workloads have become increasingly diverse, distributed, and dynamic. Applications demand accurate performance predictions, minimal latency, scalable multi-tenant, and high usability—even for non-expert users.

As a result, database systems are undergoing a transformation toward more intelligent, adaptive, and modular architectures.

This survey focuses on three significant contributions to this transformation:

*Stage* introduces a hierarchical machine learning model to predict the execution time of SQL queries in Amazon Redshift. It balances accuracy and latency by layering a cache, a local model, and a global graph neural network.

*ChatGPT Relational Code Verification* evaluates whether large language models like ChatGPT can reliably generate semantically correct and relationally complete queries in SQL, relational algebra (RA), and relational calculus (RC).

*Limbo* re-architects SQLite for use in serverless environments by replacing blocking I/O operations with asynchronous equivalents using io_uring, enabling massive concurrency and latency reduction. These three systems showcase how machine learning, AI, and OS-level redesigns can address pressing challenges in database performance, usability, and scalability.

## 2. TECHNIQUES AND METHODS:

### *2.1 Stage:*

The Stage model proposed for Amazon Redshift consists of a three-layered prediction framework designed to optimize inference latency and the accuracy of the prediction. The design is motivated by the real-world challenges encountered in Redshift clusters: 60% of queries are repeated by day, and most queries finish within 100ms. Cold start handling, changing workloads, and very low inference latency become demands for the predictor.

#### *2.1.1 Execution Time Cache:*

Uses a hashed 33-dimensional feature vector derived from the query plan to retrieve predictions for repeated queries.

It is trained per-instance, using a vector of dimension 33 derived from physical plans.
Uses multiple XGBoost regressors to model the data and the model uncertainty.

With high uncertainty, the fallback to the global model is triggered.

- Stores recently observed queries and their corresponding execution times.
- Predicts latency using a heuristic: , where the historical mean,  is the most recent exec-time.
- Covers ~60% of Redshift queries.

### *2.1.2 Local Model:*

Uses an ensemble of XGBoost regressors trained per Redshift instance. Each model produces an estimate and its confidence score (uncertainty) and when the local model carries a low uncertainty, this estimate is selected.

- Uses multiple XGBoost regressors to model the data and the model uncertainty.
- With high uncertainty, the fallback to the global model is triggered.

### *2.1.3 Global GNN Model:*

When the local model is uncertain, or if an unknown query presents itself, this is used-a deep learning model trained across clusters that encode the SQL query plan as a graph (nodes=operations, edges=data flow).

- Each node such as join, scan and aggregate is featured with operator type, estimated cost/cardinality, storage format, and node-level statistics.
- GCN performs message-passing to encode operator interactions.
- In addition, it contains extra system context features such as node type, memory size, and level of concurrency.

### *Performance Summary:*

- Prediction median absolute error: 0.67 seconds (vs 2.03s in AutoWLM).
- Query latency reduced by 20% compared to prior predictors.
- Inference latency is kept low: cache (~µs), local model (~ms), global model (~100ms, used <3% cases).

### *Design Benefits:*

- Adaptable to repeated queries, dynamic workloads, and new clusters.
- Provides uncertainty-aware predictions.
- Successfully balances model complexity and real-time constraints in a production environment.

### *Formula:*

$$T(x) = \left\{T_{cache}(x),\ if\ x \in Cache\right\}$$
$$T(x) = \left\{T_{local}(x),\ if\ \sigma_{local}(x) < \epsilon\right\}$$
$$T(x) = \left\{T_{global}(x),\ otherwise\right\}$$

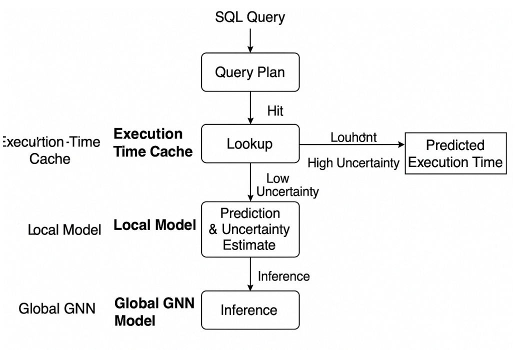


**Figure:1: Stage hierarchical flowchart**

With this architecture, repeated or similar queries receive fast cached or low-latency predictions whereas new or complex queries are passed on to a powerful GNN.

## *2.2. ChatGPT Relational Code Verification:*

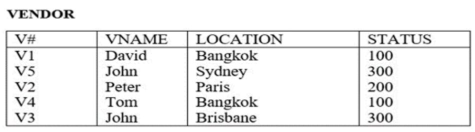
This technique evaluates the power of ChatGPT-a giant language model to create syntactically valid and relationally complete database queries using SQL, relational algebra, and relational calculus. The correctness is verified against essential relational operators such as selection, projection, join, union, intersection, difference and division.

This study tests ChatGPT on relational query generation tasks across SQL, RA, and RC. The queries cover:

- Selection ()
- Projection ()
- Join ()
- Union ()
- Intersection ()
- Difference ()
- Division ()

*Schema:* VENDOR(V#, VNAME, LOCATION, STATUS

*Example:*
Retrieve vendor names from Bangkok.



VENDOR

| V# | VNAME | LOCATION | STATUS |
|---|---|---|---|
| V1 | David | Bangkok | 100 |
| V5 | John | Sydney | 300 |
| V2 | Peter | Paris | 200 |
| V4 | Tom | Bangkok | 100 |
| V3 | John | Brisbane | 300 |

**Figure:2: Vendor table for the example query**

SQL: SELECT VNAME FROM VENDOR WHERE LOCATION = 'Bangkok';
RA :
$\Pi_{VNAME}(\sigma_{location='Bangkok'}(VENDOR))$
RC: $v.VNAME \mid v \in VENDOR \wedge v.Location =$ 'Bangkok'.

*Process:*

1. Natural language query input
2. Prompt ChatGPT to generate SQL, RA, and RC.
3. Validate outputs for RC
4. Compare logical equivalence

The generated queries were manually verified for relational completeness and semantic correctness.

*Relations Completeness:*

- ChatGPT successfully generated correct and complete representations for the six operations.
- As the most intricate division was approached accurately by applying nested SQL subqueries and quantified RC logic.
- In this example, relational calculus queries have reflected formal logic through their existential and universal quantifiers, demonstrating a great capability beyond the casual syntax of SQL.

This method demonstrates that ChatGPT can be a helpful tool for education, rapid prototyping, and accessible interaction with relational systems.

*Significance:*

- ChatGPT shows the ability to transform natural language queries into many formal relational languages.
- It can be used to help with education, query prototyping, and accessible user interfaces.
- The limitations of not having any deep schema understanding and no execution optimization have been dealt with.

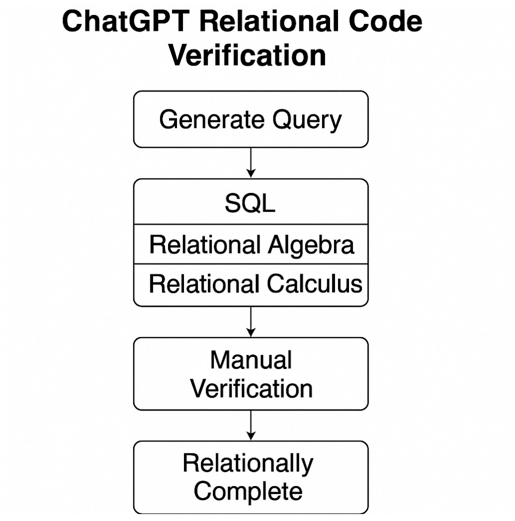


**Figure: 3: ChatGPT t flow chart for query**

## *2.3 Limbo:*

Serverless-Compatible SQLite via Asynchronous I/O. Limbo modifies SQLite for serverless runtimes by: Replacing blocking bytecode instructions (e.g., Next) with asynchronous versions (NextAsync, NextWait).

Decoupling query execution from storage via an external I/O module using io_uring.

*Architecture Overview*

- Limbo eliminates any I/O interface interaction with SQLite library by the operating system.
- An external I/O module-an application provides-will handle the primary handling of all I/O requests, such as reading or writing B-tree pages.
- I/O Interface is plugable, libio-uring and future release of async I/O features.

### *2.3.1. Asynchronous Bytecode Instruction:*

Limbo transforms the bytecode instruction set of SQLite's virtual machine into a set that supports the addition of various asynchronous counterparts for the blocking instructions.

NextAsync submits the I/O request but does not block; NextWait resumes execution once I/O is completed.

sqlite3_step() returns a SQLITE_IO code after an async instruction. From then on, it can do other things until the operation is done with its I/O.

### *2.3.2. Decoupled Query-and-Storage:*

Conventional database systems tightly couple query execution with I/O to introduce blocking and hence limit concurrency.

Limbo separates these:

- The query engine stays in the database library.
- The storage engine will now be provided externally by the I/O module.
- The runtime is responsible for maintaining io_uring submission and completion queues.

This paradigm of runtime enables many concurrent tenants to use it without blocking, thus increasing multi-tenancy and throughput.

*Problem:* SQLite is being troubled by the problem: The original design is based on blocking I/O (e.g., read(), write()), leading to thread blocking and latency under concurrent situations.

*Solution :* Make asynchronous alternatives for synchronous bytecode
Next → NextAsync + NextWait
OpenRead → OpenReadAsync + OpenWaitAsync
Rewind → RewindAsync + RewindWait

*Abstraction for I/O:* I/O gets externalized for dispatching through io_uring for Linux. This separates computation from I/O; the CPU keeps executing while waiting for disk operation.

*Significance:*

- Limbo is a non-blocking thread that enables incredibly effective multitenancy in handling usage.
- Execution is isolated from storage access under the asynchronous model.
- Initial results indicate limbo will be beneficial in runtime/database systems co-designed for applications, notably in latency-sensitive and multi-tenant situations.

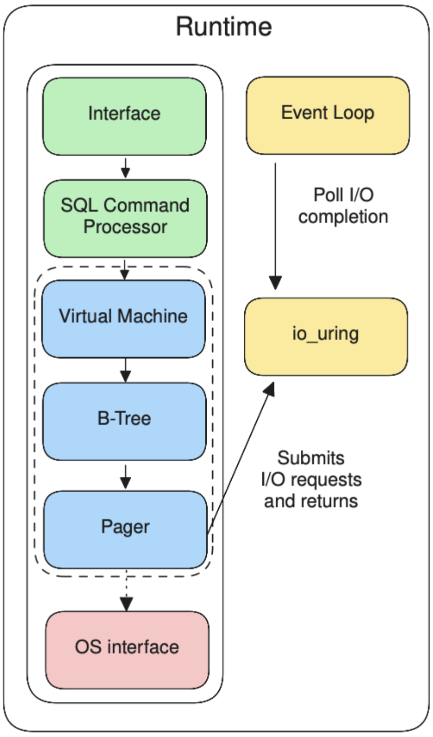


**Figure: 4: Limbo mapping and workflow**

## 3. STRENGTHS & WEAKNESSES:

### 3.1. Stage:

*Strengths:*
Good performance in short and medium queries, Fast inference for repeated queries (cache/local).
Generally, falls back with GNN while failing generalization provisions.

*Weaknesses:*
GNN inference latency (~100ms).
Failed for new unseen long queries with no generalization.
Requires lots of memory resources for operations and training.

### 3.2.ChatGPT Code Verification:

*Strengths:*
All relational operators are supported.
Formulates correct queries in SQL, RA, and RC.
Very lightweight, User friendly.

*Weaknesses:*

Semantic superficiality or schema-specific optimization may also be missing.
Verbose for relational calculus.
It is not performance tuning or aware of indexes.

### 3.3.Limbo :

*Strengths:*
Up to 100x tail latency reduced.
Cohorts can handle massive concurrent workloads.
Very efficient CPU and I/O resource usage.

*Weaknesses:*
With runtime integration it's a real hassle.
Clean-slate design isn't compatible with SQLite.
Requires runtime support for io_uring and event loops.

## 4. COMPARATIVE ANALYSIS ACROSS THE MODELS:

### 4.1.Speed:

*Best:* Limbo — Latency reduced by asynchronous execution.
*Satisfactory:* Stage (only above-cached/local layers).
*Acceptable:* ChatGPT — Instant code generation; not relevant to runtime.

### 4.2.Accuracy:

*Best:* Stage (with ensemble and fallback to GNN).
*Good*: ChatGPT (complete and correct but surface-level).
*Good:* Limbo (accurate I/O if implemented properly).

### 4.3.Memory Efficiency:

*Best:* ChatGPT (no state; stateless generation).
*Good*: Limbo (very lightweight coroutine-based threading).
*Fair:* Stage (ensemble+GNN=greater use of memory).

### 4.4.Scalability:

*Best:* Limbo — Designed for many tenants.
*Good:* ChatGPT — Stateless so scale easily.
*Moderate:* Stage Scaling through fallback logic but the cost of the model is a bottleneck.

### 4.5.Usability:

*Best:* ChatGPT — Very open for beginners.
*Moderate:* Stage — Usable once deployed but got no face for users.
*Lowest:* Limbo — Needs low-level system integration.

### 4.6.Robustness / Generalization:

*Best:* Stage — Explicitly using GNN cross-instance learning.
*Good:* Limbo — Facilitate I/O modularity.
*Fair:* ChatGPT — May lack semantics but very correct syntax.

Stage most beneficial in post performance tuning and workload management in production-grade data warehouses. Essentially, it does latency versus generalization smart trade-offs.

ChatGPT excels in code generation and education without performance awareness and makes SQL and RA/RC accessible to learners and nonprogrammers.

Limbo is an innovation necessary in system-level things that make legacy DBs viable in serverless, multi-tenant environments where high concurrency meets low latency, thanks to asynchronous execution.

Together, they cover three significant domains:

- Performance Prediction (Stage)
- Natural Language Query Interfaces (ChatGPT)
- Runtime/Database Integration for Edge Computing (Limbo)

## 5. CONCLUSION:

These three papers illustrate the future of intelligent, adaptive, and modular database systems:
*Stage* demonstrates how multi-tiered machine learning models can enable more accurate and context-aware performance prediction in cloud-scale databases like Redshift.

The *ChatGPT verification study* confirms that AI models can generate correct, complete queries across multiple formal languages—enhancing database accessibility and reducing the barrier to entry.

*Limbo* shows how asynchronous I/O and modular design can significantly enhance the performance and scalability of embedded databases for serverless applications, especially in latency-critical edge environments.

Together, they represent complementary innovations addressing speed, usability, and architectural flexibility in modern data systems.

## 6. References:

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (SIGMOD '03). Association for Computing Machinery, New York, NY, USA, 666. https://doi.org/10.1145/872757.872858

[2] J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*. USENIX Association, 137–150.

[3] Ziniu Wu, Alex Ratner, Cong Yu, and Matei Zaharia. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Proceedings of the 2024 International Conference on Management of Data (SIGMOD-Companion '24)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3626246.3653391

[4] Putsadee Pornphol and Suphamit Chittayasothorn. 2023. Verification of Relational Database Languages Codes Generated by ChatGPT. In *Proceedings of the 2023 Asia Symposium on Software Engineering (ASSE 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3634814.3634817

[5] Pekka Enberg, Sasu Tarkoma, Jon Crowcroft, and Ashwin Rao. 2024. Serverless Runtime / Database Co-Design With Asynchronous I/O. In *Proceedings of the 7th Workshop on Edge Systems, Analytics and Networking (EdgeSys '24)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3642968.3654821

[6] T. Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550.

[7] M. Stonebraker and J. M. Hellerstein. 2005. What goes around comes around. In *Readings in Database Systems*, 4th ed., MIT Press.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster computing with working sets. In the 2nd *USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*.