Innovations in Modern Database Systems
Predictive Modeling, Generative AI, and Serverless Co-Design

CSCE – 5370 Distributive & Parallel Database Systems

Gayathri Baman

ID: 11697946

OUTLINE

- Introduction
- Descriptive Terminology for the Three Models
- Strengths and Weakness for the Models
- Comparative Analysis
- References

Introduction

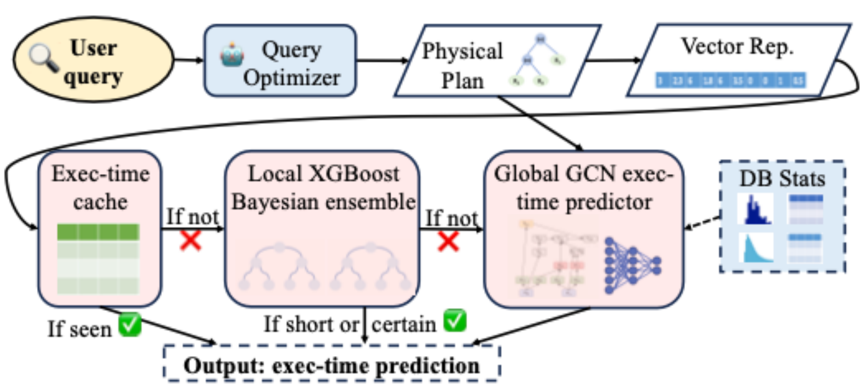
- The modern database workload environment is, therefore, not only extremely heterogeneous but also distributed and dynamic. The requirements from the applications include accurate performance predictions, low latency, highly-scalable multi-tenancy, and easy usability for even a non-expert user. All these have led to an evolving change in the architecture of databases from intelligent and adaptive to that of modular design.

Why Are New Database Concepts Needed?

- Old database systems were for simpler sets of requirements
- New requirements:
- Instant Answers-(Low latency)
- Everyone Must Use It- User Friendliness
- Many Users Simultaneously- Scalability
- New tools are required to hear these demands.

Stage: Stage — Predicting Query Time in Amazon Redshift

Stage: Query Execution Time Prediction in Amazon Redshift



- Purpose: Predict SQL query execution time in Amazon Redshift
- Multi-layer system:
  - Cache : If seen before → use previous time
  - Local XGBoost Ensemble : Fast machine learning guess
  - Global Graph Neural Network (GNN) :Deep learning for new queries

Pro's & Con's

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.

Verification of Relational Database Languages Codes Generated by ChatGPT

- Can write SQL, relational algebra (RA), and relational calculus (RC)
- Learns from natural language input
- Good for learning and quick coding

Example: “Find vendor names in Bangkok”

- SQL: Simple command -> SELECT VNAME FROM VENDOR WHERE LOCATION = ‘Bangkok’;
- RA: Uses selection and projection ->Π_*VNAME* (σ_*location* ='*Bangkok*'(*VENDOR*))
- RC: Uses logical expression -> $\upsilon.\ VNAME \mid \upsilon \in VENDOR \wedge \upsilon.\ Location$ = ‘Bangkok’.

In this example, relational calculus queries have reflected formal logic through them existential and universal quantifiers, demonstrating a great capability beyond the casual syntax of SQL.

Pro's & Con's

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.	Can get confused with tricky schemas	Doesn't optimize queries or aware of indexes.

Serverless Runtime / Database Co-Design With Asynchronous I/O

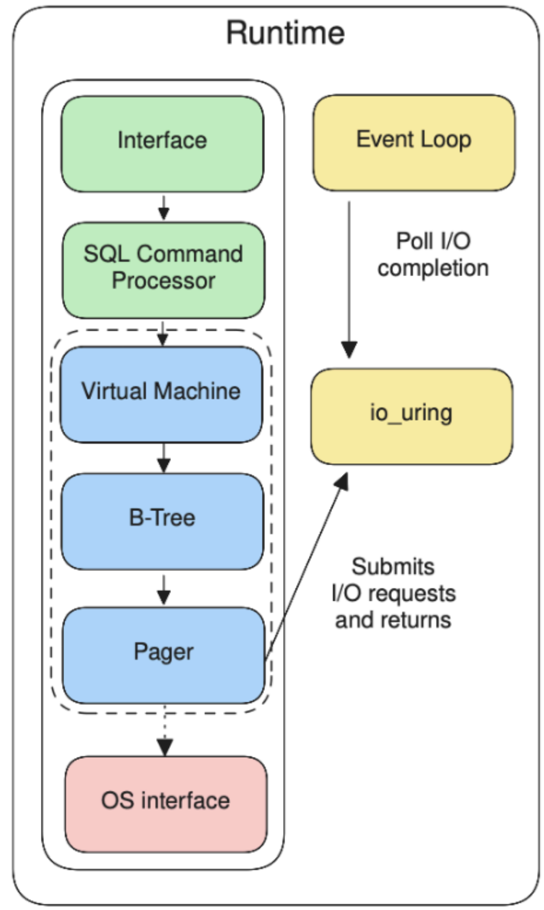
Limbo modifies SQLite for serverless runtimes by replacing blocking bytecode instructions (e.g., Next) with asynchronous versions.

How it works:

- Next → NextAsync → NextWait
- Separates query logic from disk access
- system handle many queries at once.

Results:

- Simulated 100 users querying SELECT * FROM users
- Limbo: stays fast and consistent



Pro’s & Con’s

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.

Comparative Analysis

Criteria	Stage	ChatGPT Code Verification	Limbo
Speed	✅ Fast (cache/local); ❌ Slower GNN (~100ms)	✅ Instant generation	✅ Runtime latency reduction up to 100x
Accuracy	✅ High for short/medium; ❌ Uncertain on long queries	✅ Correct across SQL/RA/RC	✅ Accurate with async logic
Memory Usage	❌ Higher (ensemble + GNN)	✅ Minimal (prompt-based)	✅ Efficient (coroutines + I/O buffer)
Scalability	⚠️ Limited by model cost	✅ Scales with demand	✅ Excellent (supports many tenants)
Robustness	✅ Uses uncertainty-aware switching	⚠️ Limited schema understanding	✅ Decoupled, modular architecture
Usability	⚠️ Backend-only; not user-facing	✅ Ideal for learners and non-programmers	⚠️ Developer-level usage only
Optimization Awareness	✅ Avoids unnecessary model calls	❌ No query performance optimization	✅ Asynchronous model avoids thread waste
Implementation Effort	❌ High (training, integration)	✅ None for users; ⚠️ Hard to verify semantics manually	❌ High (requires system-level engineering)

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] 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

[3] 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

THANK YOU