

InferQ: A Database-Oriented Benchmark for Quantum Circuits Simulation

ANDREI ILINESCU*, Delft University of Technology, The Netherlands

AADI PATWARDHAN*, Delft University of Technology, The Netherlands

RIHAN HAI, Delft University of Technology, The Netherlands

Recent work suggests that relational database management systems (RDBMSs) can execute quantum circuit simulation by compiling the simulation into SQL workloads (primarily join-and-aggregate tensor contractions). While early results are promising, they largely focus on a narrow set of highly structured circuits and offer limited support for systematic database research, such as query optimization, physical design, and engine-level evaluation across a broad range of circuits. We present InferQ, a database-oriented benchmark for quantum circuit simulation. InferQ generates *general*, *compositional* circuits by assembling subcircuits from a set of circuit templates, emits each simulation task as an RDBMS-ready SQL workload, and extracts circuit and query features (static, graph, SQL, and dynamic) for workload characterization. InferQ also releases a large dataset of 202,975 circuits online, with a web-based viewer to support searching, filtering, and downloading circuits and feature records. In experiments across RDBMS engines (PostgreSQL, SQLite, DuckDB, and Umbra) and the widely used Qiskit Aer simulator, we find that RDBMSs achieve better peak memory usage than Qiskit Aer on more than 50% of the circuits generated by InferQ. Moreover, using InferQ features, lightweight machine learning models (linear and tree-based models) can accurately predict when SQL execution is preferable (with accuracy up to 95.6% for runtime and 97.4% for memory), enabling data-centric simulator selection and opening the door to principled optimization of SQL-based quantum circuit simulation.

CCS Concepts: • **Information systems** → **Data management systems**; • **Applied computing** → *Physics*.

Additional Key Words and Phrases: Quantum circuit simulation, benchmarking, relational database management systems, SQL workload generation, learned simulator selection

ACM Reference Format:

Andrei Ilinescu, Aadi Patwardhan, and Rihan Hai. 2026. InferQ: A Database-Oriented Benchmark for Quantum Circuits Simulation. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 278 (September 2026), 26 pages. <https://doi.org/10.1145/3837116>

1 Introduction

In the current noisy intermediate-scale quantum (NISQ) era, most quantum applications are developed and validated through classical simulation. Quantum programs are typically expressed in the circuit model as a sequence of gates, and the common practice is to execute the circuit on a simulator before running it on quantum hardware. Simulators produce the circuit's (predicted) output, e.g., measurement outcomes or their probabilities. Efficient circuit simulation, therefore, remains a cornerstone for near-term quantum algorithm development. For database researchers, this raises a data management question: can circuit simulation be expressed as a relational workload

*Both authors contributed equally to this research.

Authors' Contact Information: Andrei Ilinescu, a.ilinescu@student.tudelft.nl, Delft University of Technology, Delft, The Netherlands; Aadi Patwardhan, a.patwardhan@student.tudelft.nl, Delft University of Technology, Delft, The Netherlands; Rihan Hai, r.hai@tudelft.nl, Delft University of Technology, Delft, The Netherlands.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART278

<https://doi.org/10.1145/3837116>

whose execution can benefit from database management systems (DBMS) memory management, query optimization, and out-of-core processing?

Data management challenge. Recent work [5, 11] argues that relational databases can serve as a competitive simulation *engine* by compiling circuit simulation into relational workloads. A particularly promising result is that DuckDB can scale to *millions of qubits* for sparse circuits. Concretely, under a 2GB memory limit, DuckDB simulates up to 4,000,862 qubits for GHZ-state preparation and up to 53,017 qubits for W-state preparation [12], which are two standard state-preparation routines that initialize many qubits into widely used multi-qubit entangled states. These results raise an obvious question for the database community: *when does “an RDBMS as a simulator” actually work?*

The same study makes clear this scalability is highly workload-dependent: for a dense circuit such as Quantum Fourier Transform (QFT), the maximum supported qubit count is only 14 under the same setting [12]. Moreover, the largest improvements in those experiments arise from circuits with a specific structure. For instance, GHZ is a stabilizer circuit, which uses only Clifford gates (Hadamard, CNOT, Phase), and is known to be simulated efficiently on classical computers via the Gottesman–Knill theorem [1]. Moreover, a key systems gap is revealed in [5, 11]: existing approaches have not fully exploited RDBMS capabilities to handle simulation workloads through mechanisms such as query optimization, indexing, and materialization, emphasizing that closing this gap is important for an end-to-end database-driven simulation pipeline. Altogether, these results motivate a database-centric benchmark: to move beyond isolated case studies, the community needs a systematic way to generate *diverse* simulation workloads and study how SQL representations behave under query optimization and execution. Such a benchmark enables database researchers to study query planning, materialization, indexing, spill behavior, and backend engine selection for quantum circuit simulation. That is, these results point to a benchmark gap.

Why this differs from existing tensor workloads in RDBMSs. When a quantum circuit is compiled into SQL, the underlying computation *can* be expressed as tensor algebra: applying gates corresponds to multiplying and contracting tensors (e.g., updating a state vector by small operators). At a computational level, circuit simulation in RDBMS is a tensor workload: vector-matrix multiplication, matrix-matrix multiplication, and tensor contractions. One of the most frequent operations is matrix multiplication. Using SQL, matrix multiplication can be implemented as a join-and-aggregate operation: represent the matrices in long form as $A(i, k, a_{ik})$ and $B(k, j, b_{kj})$, compute $W = AB$ by joining rows of A and B on the shared inner index k , multiplying matching values, and summing over k :

$$w_{ij} = \sum_k a_{ik} b_{kj},$$

which corresponds to:

```
SELECT A.i, B.j, SUM(A.val * B.val) AS w_ij
FROM A JOIN B ON A.k = B.k
GROUP BY A.i, B.j;
```

This style of expressing tensor computation in SQL has been studied in the database community [6, 17, 22, 26, 34], including recent work on efficient Einstein summation in SQL [5] and relational abstractions for tensor and linear-algebra workloads in ML [8, 21, 28, 31, 33].

However, the simulation workload induced by quantum circuits differs from mainstream ML workloads in several fundamental ways. In transformer models, the core computation is dominated by dense linear algebra over matrices with dimensions in the thousands (e.g., attention uses dense QK^T and QV products) [30]. In contrast, a quantum circuit acts on a state vector of dimension 2^N : for qubit count $N = 50$, the state has $2^{50} \approx 1.13 \times 10^{15}$ amplitudes, which already requires about

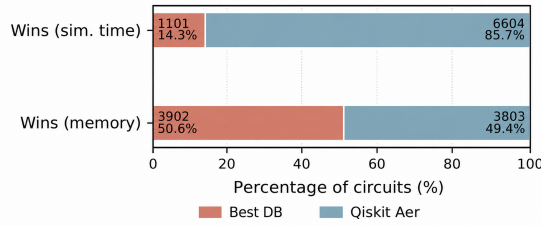


Fig. 1. Across 7,705 circuits, share of cases where RDBMSs (executing the emitted SQL workload) or Qiskit Aer achieves the lowest runtime and the lowest peak memory. Detailed experimental settings are in Section 6.

16 PB just to store a dense complex state vector. At the same time, each instruction applies only a tiny operator (typically 2×2 or 4×4 matrices), but it must be applied across this exponentially large index space. The structure of a quantum circuit determines a sequence of tensor contractions whose intermediate sparsity and entanglement can change abruptly across the gate sequence.

This mismatch creates database-specific difficulties: (i) intermediate results may be *enormous* unless the representation stays sparse or factored; (ii) whether sparsity survives depends on the *gate sequence*, and entangling operations can rapidly densify the state and cause abrupt blow-ups; and (iii) the computation is a long, dependency-constrained sequence of small updates rather than a few large tensor computations, so “optimize one big join” is not the right mental model. Consequently, optimizing SQL for quantum circuit simulation is not a direct reuse of existing in-database tensor/ML techniques: the bottlenecks, cardinality growth, and the importance of circuit structure (sparsity patterns, interaction graphs, composition rules) are qualitatively different. A benchmark that systematically varies these properties is therefore essential for a principled optimizer and execution-engine research in this space.

Gaps of existing benchmarks. Existing quantum benchmarks such as SupermarQ [29], MQT Bench [27], and QASM Bench [19] provide valuable circuit suites for the quantum community, but they do not directly target database execution. In particular, they generate circuits as quantum-circuit descriptions (e.g., QASM/Qiskit circuits) meant to be run by quantum toolchains, not as relational workloads. As a result, they are not designed to evaluate how an RDBMS behaves as a simulation engine, e.g., the impact of query optimization, physical design (indexes/partitioning), and execution strategies on circuit-simulation workloads.

Our approach. We present InferQ, a database-oriented benchmark for quantum circuit simulation. Given a small user configuration (Table 2), InferQ generates *compositional* circuits by assembling subcircuits from a set of quantum templates. Each circuit is accompanied by an *RDBMS-ready SQL workload* that expresses the simulation task as relational operators. This makes simulation directly accessible to database research: each circuit becomes both a quantum artifact and a relational workload whose query shape, intermediate behavior, and backend performance can be analyzed. Moreover, to support data-centric analysis, InferQ also extracts four groups of features: *static* and *graph* features from the circuit structure, *SQL* features from the emitted query workload, and *dynamic* features from the simulation. Our experimental results further clarify why exposing simulation as database workloads is useful. Running the generated SQL inside DBMSs makes simulation subject to database execution and optimization choices, enabling better memory management, runtime improvements for certain circuits, and out-of-core execution for large circuits under memory limits. Thus, RDBMS-based simulation enables systematic optimization of quantum circuit simulation through DBMS memory management, spill-aware execution, and backend selection.

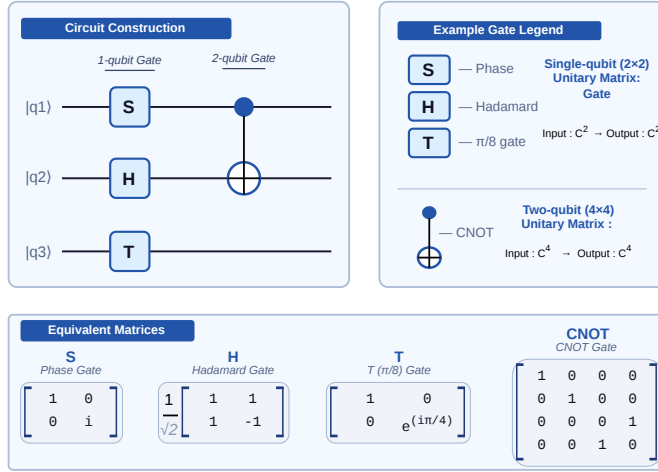


Fig. 2. Quantum gates represented as matrices.

Contributions. We make the following contributions:

- We propose InferQ, a benchmark that produces quantum circuit simulation workloads in *relational form* (SQL), enabling direct evaluation inside RDBMS engines.
- We design a template-based generator that *scales* circuit size while preserving realistic structure via compositional circuit generation (Section 3).
- From the generated circuits, we extract *static*, *graph*, *SQL*, and *dynamic* features per circuit, enabling systematic workload characterization and correlation with DB performance (Section 4).
- Using InferQ workloads, we quantify when SQL-driven simulation is competitive with Qiskit Aer, including memory-efficient and out-of-core cases. InferQ features enable accurate, lightweight machine learning based selectors for deciding when to use an RDBMS engine (Section 6).

2 Background

2.1 Quantum Bits, Gates, and Circuits

A classical bit stores either 0 or 1. In contrast, a *quantum bit* (qubit) can be represented as a unit vector in a two-dimensional complex vector space, typically described using the computational basis of $|0\rangle$ and $|1\rangle$:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

An arbitrary single-qubit state can be written as a superposition of these basis states:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \quad |\alpha|^2 + |\beta|^2 = 1,$$

where α and β are complex amplitudes. If we perform a *measurement* on the quantum state $|\psi\rangle$, it collapses to a classical bit of 0 or 1. That is, a measurement of $|\psi\rangle$ against the computational basis yields outcome 0 with probability $|\alpha|^2$ and outcome 1 with probability $|\beta|^2$. An N -qubit state can be expressed as a superposition over computational basis states indexed by bitstrings:

$$|\Psi\rangle = \sum_{x \in \{0,1\}^N} c_x |x\rangle, \quad \sum_{x \in \{0,1\}^N} |c_x|^2 = 1.$$

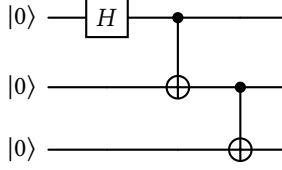


Fig. 3. GHZ state preparation circuit: it takes the all-zero state $|000\rangle$ as input and prepares a maximally entangled state.

Fix an ordering of the computational basis states (e.g., lexicographic order). The *state vector* representation of $|\Psi\rangle$ is the column vector of its amplitudes in that order: $\begin{bmatrix} c_{0\dots 0} \\ c_{0\dots 1} \\ \vdots \\ c_{1\dots 1} \end{bmatrix}$. For example, when $N = 2$, any two-qubit state $|\Psi\rangle$ can be represented as a linear combination of $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, and $|c_{00}|^2 + |c_{01}|^2 + |c_{10}|^2 + |c_{11}|^2 = 1$. The state vector representation of $|\Psi\rangle$ is $\begin{bmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{bmatrix}$.

Quantum computation follows the circuit model, which is analogous to classical logic circuits. Computation is performed by applying quantum gates (unitary operators) acting on one or more qubits to transform the quantum state. A matrix U is *unitary* if $U^\dagger U = U U^\dagger = I$. As shown in Figure 2, a quantum gate acting on N qubits is represented by a $2^N \times 2^N$ unitary matrix U . It maps an input state $|\psi\rangle \in (\mathbb{C}^2)^{\otimes N}$ to an output state $U|\psi\rangle \in (\mathbb{C}^2)^{\otimes N}$, where $\mathbb{C}$ denotes the complex numbers, e.g., one qubit corresponds to $\mathbb{C}^2$ and two qubits to $\mathbb{C}^4$. A gate set is called universal if it can approximate any unitary operator to arbitrary precision [25]. A widely used universal set is the Clifford gates (including the Pauli gates (X , Y , Z), the Hadamard gate H , the phase gate S , the controlled-NOT (CNOT) gate), and the T gate. Figure 2 illustrates several of these gates and their matrix representations. A *quantum circuit* is a sequence of quantum gates, and the overall computation corresponds to the composition of their unitary matrices.

Circuit depth is the number of sequential layers of quantum gates in a circuit, representing how many time steps are required to execute it when gates that act on different qubits can be applied in parallel. For example, in Figure 3, the circuit depth is 3. We can consider quantum circuits as higher-level operators composed of subcircuits. By abstracting away from hardware and physical implementation details, we can treat such structures modularly. Take the GHZ state preparation circuit in Figure 3 for instance. It can serve as a state preparation subcircuit for larger algorithms, i.e., it prepares an input state for other algorithms.

2.2 Limitations of Existing Benchmarks

Several quantum circuit benchmarks are widely used in the quantum computing community, including, for example, SupermarQ [29], MQT Bench [27], and QASM Bench [19]. While these suites are valuable for evaluating quantum circuits, they expose limited support for database-driven and data-centric studies of circuit simulation.

Gap 1: No broad DBMS-executable benchmark workloads. Existing benchmarks primarily distribute circuits as OpenQASM¹ and Qiskit QPY² files. These formats are well-suited for quantum toolchains, but they do not expose the induced simulation computation as optimizer-visible relational workloads that can be directly executed, inspected, or optimized by an RDBMS. As a result, studying circuit simulation inside an RDBMS typically requires substantial manual translation

¹<https://openqasm.com/>

²<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qpy>

and non-trivial quantum expertise. The two recent DB-oriented simulation works discussed above [5, 11] are important first steps within this gap: they show that SQL/RDBMS execution can support quantum circuit simulation. However, their workload coverage is limited. [5] targets general Einstein summation and evaluates SQL simulation on Google’s Sycamore quantum-supremacy circuit, while [11] studies W and GHZ state preparation and QFT circuits. Thus, these works motivate DBMS-backed simulation, but do not provide a benchmark-scale collection of diverse circuits with reusable SQL workloads and workload features.

We address this gap by having InferQ generate general, diverse circuits, each coming with an RDBMS-ready SQL workload (Section 3). This effort is important because it makes the simulation workload visible to database systems: tensor contractions become join-and-aggregate queries whose plans, indexes, materialization choices, and spill behavior can be studied and optimized. Consequently, InferQ enables evaluating whether DBMS execution can improve memory usage, runtime, and scalability under memory limits, rather than merely translating circuits into another format.

Gap 2: Lack of structured compositional circuits. Real quantum programs are typically assembled as pipelines of subroutines such as state preparation, algorithmic cores, estimation procedures, and measurement. However, existing benchmarks largely treat circuits as independent, fixed templates for standalone algorithms. They do not systematically compose building blocks into larger circuits, nor do they model dependencies between subcircuits. In practice, the choice of one subroutine often constrains what can follow (e.g., Hamiltonian simulation followed by phase estimation). We address this gap in Section 3.2 by modeling conditional, history-dependent template composition as a Markov transition process.

Gap 3: Limited scalability for data-centric analysis. Because many existing benchmarks focus on fixed algorithm templates, their scalability is inherently limited: the number of instances grows slowly, and structural diversity is constrained. This limits their usefulness for data-centric workflows that require large corpora, such as applying machine learning models, cost modeling, and feature-driven analysis. We address this gap with scalable compositional generation in Section 3, and release a large dataset of 202,975 circuits to support large-scale quantum circuit simulation studies. We also validate scalability experimentally in Appendix M of our online technical report [16].

Gap 4: Limited circuit-as-graph and SQL-level characterization. Quantum circuits are inherently graph-structured: qubits correspond to vertices and multi-qubit gates induce interactions as edges [23]. This structure is central to simulation cost and to the shape of the induced SQL workload. Nevertheless, existing benchmarks expose few graph-derived properties and provide limited support for analyzing structural diversity across circuits. We address this gap by explicitly extracting graph features (Section 4.2) and SQL features for the query representation (Section 4.3). **Summary.** Overall, existing benchmarks fall short of supporting database-driven and data-centric research on quantum circuit simulation. Addressing these gaps requires a benchmark that produces SQL workloads, supports conditional composition of subcircuits, scales to large and diverse circuits, and exposes graph- and query-level structure for systematic analysis.

3 InferQ Benchmark

InferQ has two components. First, it generates quantum circuits via templates. Second, it extracts a structured set of circuit features. This section presents the InferQ benchmark pipeline (Figure 4). We describe the features in Section 4.

Design goal. Quantum algorithms are typically built by composing reusable algorithmic blocks, often called *primitives*. Examples include state preparation, Hamiltonian simulation, quantum Fourier transform (QFT), quantum phase estimation (QPE), amplitude amplification and estimation, and

variational layers [2, 7, 9, 10, 24]. For example, the Harrow–Hassidim–Lloyd (HHL) algorithm [13] can be viewed, at a high level, as a composition of Hamiltonian simulation, QPE, and amplitude amplification. More broadly, many modern quantum algorithms are developed by composing primitive transformations in structured ways [10, 24]. Thus, a simulation benchmark should not be limited to a fixed set of textbook circuits, but should support diverse and extensible workloads that can cover recent and future quantum algorithms.

Database benchmarks such as TPC-H³, TPC-DS⁴, and the Join Order Benchmark (JOB) [18] use parameterized templates to obtain controlled scale, coverage, and reproducibility. Inspired by this design, InferQ generates circuits from a library of templates representing quantum primitives (Table 1) and composes them using feasibility constraints and history-dependent transition rules. In InferQ, randomness is used to vary parameters, scale, and primitive combinations, while preserving meaningful composition patterns. Compared with existing quantum benchmarks centered on a fixed set of textbook circuits [19, 27, 29], InferQ is more general and extensible: new primitives and transition rules can be added as quantum algorithms and simulation workloads evolve. This gives InferQ broad benchmark coverage while preserving the structure needed to study realistic quantum circuit simulation workloads.

3.1 Overview

InferQ targets the compositional structure found in existing quantum circuits, where a single circuit is assembled from multiple building blocks (Table 1).⁵ This differs from template-only benchmarks that generate isolated, standalone circuits. InferQ selects an ordered sequence of *circuit templates* and instantiates each template into a *subcircuit*. The resulting subcircuits are then composed into one benchmark circuit.

Concretely, InferQ generates one circuit instance as follows:

- (1) **Select templates.** InferQ sequentially samples a template sequence using the history-dependent distribution.
- (2) **Sample parameters.** InferQ samples user-configured local parameters (e.g., qubit count and depth).
- (3) **Generate subcircuits.** For each selected template, InferQ samples template parameters and generates the corresponding subcircuit.
- (4) **Output circuit.** The circuit generation process stops when a stopping rule triggers, after which InferQ composes all subcircuits into a single circuit.

This design produces circuits that satisfy user constraints while supporting diverse compositions and realistic template transitions.

3.2 Select Templates

InferQ builds a circuit incrementally: at each step, it chooses one circuit template and later instantiates it as a subcircuit. The purpose of template selection is to keep this sequence valid while still producing diverse, realistic benchmark workloads. In InferQ, we maintain a set of circuit templates F . Table 1 lists representative templates, such as `StatePrep`; the full template set is documented online.⁶ Given a selected template $f \in F$, InferQ instantiates a *subcircuit* by sampling concrete,

³<https://www.tpc.org/tpch/>

⁴<https://www.tpc.org/tpcds/>

⁵Templates are compositional rather than disjoint: high-level primitives may include lower-level steps, but exposing common operations as reusable blocks supports flexible benchmarking. We use START/END and keep MEASURE explicit, so every generated sequence has the uniform form START $\rightarrow$ actual circuit templates $\rightarrow$ MEASURE $\rightarrow$ END.

⁶<https://github.com/InfiniData-Lab/InferQ/blob/main/README.md>

Table 1. Representative circuit templates and example compositions.

Current template	Full circuit template name	Next possible templates	Description	Example compositions
START	(start symbol)	StatePrep , Oracle/A , QFT , Variational	Common entry points (prepare input/eigenstate/ansatz), enabling realistic multi-block pipelines.	START → StatePrep START → Oracle/A START → QFT
StatePrep	State Preparation	HamSim , ModularExp , GroverIter , Variational	Prepares $ \psi\rangle/ b\rangle$ or initial superpositions before an algorithmic primitive.	StatePrep → HamSim StatePrep → Variational
HamSim	Hamiltonian Simulation	QPE , Variational , Random	Controlled time-evolution unitaries are typically read out via QPE (incl. IQFT decoding) in eigenvalue/spectral workflows.	HamSim → QPE
QFT	Quantum Fourier Transform	QPE , Measure , Random	Fourier-basis transform and dense benchmark block.	QFT → QPE QFT → Measure
Oracle/A	Oracle/Algorithm-A Wrapper	GroverIter , Variational , Measure	Encodes a problem-dependent subroutine A (e.g., oracle/feature map/encoding) that is naturally followed by Grover-style amplification.	A → GroverIter → AmpEst
ModularExp	Modular Exponentiation	QPE	Order-finding and Shor-style structure: modular arithmetic unitaries followed by phase estimation (incl. IQFT) to extract phases/periods.	ModularExp → QPE
GroverIter	Grover Iteration	GroverIter (repeat), AmpEst , Measure	Amplification layers typically repeat and then transition to an estimation wrapper or terminate with measurement.	GroverIter → GroverIter GroverIter → AmpEst GroverIter → Measure
AmpAmpl	Amplitude Amplification	Measure	Boosts the probability of a desired or postselected outcome.	AmpAmpl → Measure
QPE	Quantum Phase Estimation	AmpAmpl , Measure , Variational , Random	Phase readout/decoding stage (typically includes an IQFT); often followed by measurement or optional continuations.	QPE → AmpAmpl QPE → Measure QPE → Variational QPE → Random
Variational	Variational Circuit Layer/ Ansatz	Variational (repeat), Measure , Random	Layered ansatz patterns; measurement used for objective estimation; can be interleaved for hybrid workflows.	Variational → Variational Variational → Measure Variational → Random
Random	Random Circuit Block	Random , Measure	Diversity/stress-test tail; may extend circuit length or terminate by measurement.	Random → Random Random → Measure
AmpEst	Amplitude Estimation	Measure	Estimates a success amplitude or probability. It normally terminates the quantum subroutine by measurement and classical post-processing.	AmpEst → Measure
Measure	Measurement	END	Final step that yields classical outcomes and ends the whole circuit generation process.	Measure → END
END	(end symbol)	—	It marks the end of the template sequence.	END

circuit-specific parameter values. For example, from the **StatePrep** template we can generate a GHZ state preparation subcircuit with 10 qubits.

Addressing Gap 2. Database benchmarks typically instantiate each query template independently, yielding a workload that is a set (or multiset) of queries without sequential dependencies among template choices. In contrast, as discussed in Gap 2, the circuits targeted by InferQ are *ordered*

compositions of subcircuits, and the choice of the next subcircuit is often constrained by (and semantically coupled to) the current one. InferQ therefore samples templates sequentially: the already-chosen template affects which templates are allowed next and how probabilities over possible choices are updated.

Design element 1: Explicit synergy rules. InferQ provides a set of user-configurable *synergy rules* that encode statistical dependencies between templates and bias generation toward semantically coherent pipelines. For example, $\text{StatePrep} \rightarrow \text{HamSim} \rightarrow \text{QPE}$, $\text{ModularExp} \rightarrow \text{QPE}$, and $\text{A} \rightarrow \text{GroverIter} \rightarrow \text{AmpEst}$ (Table 1). Intuitively, these rules specify how earlier template choices should increase or decrease the probability of selecting specific templates later, making the sampling distribution explicitly history-dependent.

Definitions. Let $\mathcal{S}$ denote the space of subcircuits, and let T be the number of templates (equivalently, subcircuits) generated for one circuit instance. Let B denote the *context space*, i.e., the current circuit generation status (e.g., remaining allowed templates, and synergy rules in Table 1). A *circuit template* $f \in F$ is a parameterized generator

$$f : \Theta_f \times B \rightarrow \mathcal{S}. \quad (1)$$

Let $\text{feas}(b, f) \in \{0, 1\}$ be a hard feasibility predicate for choosing template $f \in F$ under context $b \in B$.

InferQ generates an ordered template sequence $f_1, f_2, \dots, f_T$. At step t , after selecting f_t and sampling parameters $\theta_t \in \Theta_{f_t}$, InferQ instantiates the corresponding subcircuit as

$$s_t = f_t(\theta_t, b_{t-1}), \quad (2)$$

where $b_{t-1} \in B$ is the current context given historical $t - 1$ steps.

At each step $t \in \{1, \dots, T\}$, InferQ maintains a categorical distribution p_t over the template set F . We write $p_t(f)$ for the probability assigned to template $f \in F$, so

$$\sum_{f \in F} p_t(f) = 1. \quad (3)$$

A larger value of $p_t(f)$ means that template f is more likely to be selected at step t .

Design element 2: Conditional composition as a Markov transition model. At step t , InferQ samples the next template from a categorical distribution over F . Given the current context b_{t-1} , it proceeds in two steps:

(1) Sample a template. First apply feasibility masking and renormalize:

$$\bar{p}_t(f) = \frac{p_t(f) \text{feas}(b_{t-1}, f)}{\sum_{g \in F} p_t(g) \text{feas}(b_{t-1}, g)}, \quad f \in F, \quad (4)$$

then sample $f_t \sim \bar{p}_t$. Next, InferQ samples local parameters for f_t , instantiates the corresponding subcircuit, appends it to the current circuit prefix, and updates the generation context.

(2) Reweight to obtain the next distribution. Let $\mathcal{R}$ be the set of synergy rules, where each rule is a triple (T, U, β) with $T \subseteq F$ (trigger), $U \subseteq F$ (target), and $\beta > 0$. After selecting f_t , define the reweighted (unnormalized) probabilities

$$\hat{p}_{t+1}(f) = \bar{p}_t(f) \cdot \prod_{(T, U, \beta) \in \mathcal{R}: f_t \in T, f \in U} \beta, \quad f \in F, \quad (5)$$

and then apply feasibility under the updated context b_t and renormalize:

$$p_{t+1}(f) = \frac{\hat{p}_{t+1}(f) \text{feas}(b_t, f)}{\sum_{g \in F} \hat{p}_{t+1}(g) \text{feas}(b_t, g)}, \quad f \in F. \quad (6)$$

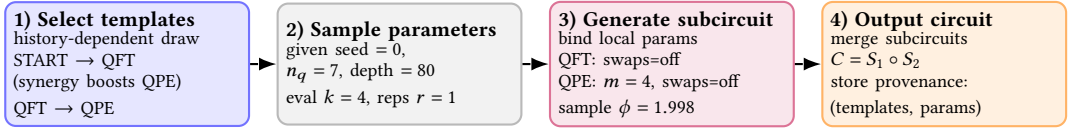


Fig. 4. InferQ benchmark's quantum circuit generation pipeline, and running example in Section 3.6.

Equations 4–6 keep the generation process simple: feasibility masking removes disallowed templates that do not fit the current circuit, while synergy rules encourage quantum subcircuits (primitives) that commonly appear together. This lets InferQ generate circuits that follow quantum algorithm design principles without enumerating every full circuit by hand. Allowed and disallowed compositions are captured by feasibility masking: a disallowed successor template receives probability zero in the current generation context and therefore cannot be sampled. The synergy rules are soft preferences over feasible successor circuit templates, guiding the generator toward meaningful algorithmic patterns, such as HamSim→QPE, without hard-coding a fixed enumeration of complete circuits. This separation is important because quantum algorithms, quantum hardware, and their simulation needs are evolving rapidly. The current design keeps InferQ configurable as new quantum primitives and workloads emerge.

Example 3.1 (Single-step reweighting under a synergy rule). Assume the previously selected template is StatePrep. By Table 1, the feasible successors are {HamSim, ModularExp, GroverIter, Variational}, with a uniform distribution $p_t(\cdot) = \frac{1}{4}$. Let $\mathcal{R}$ include $(T, U, \beta) = (\{\text{StatePrep}\}, \{\text{HamSim}\}, 2)$. Then

$$\begin{aligned}\widehat{p}_{t+1}(\text{HamSim}) &= \frac{1}{2}, & \widehat{p}_{t+1}(\text{ModularExp}) &= \frac{1}{4}, \\ \widehat{p}_{t+1}(\text{GroverIter}) &= \frac{1}{4}, & \widehat{p}_{t+1}(\text{Variational}) &= \frac{1}{4},\end{aligned}$$

and after normalization,

$$\begin{aligned}p_{t+1}(\text{HamSim}) &= \frac{2}{5}, & p_{t+1}(\text{ModularExp}) &= \frac{1}{5}, \\ p_{t+1}(\text{GroverIter}) &= \frac{1}{5}, & p_{t+1}(\text{Variational}) &= \frac{1}{5},\end{aligned}$$

3.3 Sample Parameters

Circuit generation is constrained by the global parameters in Table 2, which are configured by users. InferQ requires users to provide ranges for key parameters, such as qubit count, depth, repetitions count for repeatable subroutines, and number of qubits used during evaluation. A concrete value is drawn uniformly at random from each range. If a user prefers a fixed value, they set the range endpoints equal (e.g., min_qubits=max_qubits=10). This range-based interface provides more flexibility for users to construct workloads. For database evaluation over the generated simulation workload, these ranges play the role of scale factors: they control the size and shape of the generated SQL workloads.

3.4 Generate Subcircuits

Each selected circuit template f is instantiated by sampling parameters from its parameter domain Θ_f . In InferQ, we write one sampled parameter tuple as

$$\theta = (n_q, d, k, r, \xi) \in \Theta_f, \quad (7)$$

where n_q is the qubit count, d is the depth allocated to this subcircuit, k is the evaluation-qubit count, r is the repetition count for repeatable template sections, and ξ is a tuple of template-specific configuration variables. Moreover, n_q and d control the circuit size, while k and r control the evaluation and repetition settings. That is, these parameters determine the generated circuit

Table 2. Parameters.

Parameter	Type	Description
min_qubits	int	Minimum number of qubits per generated circuit
max_qubits	int	Maximum number of qubits per generated circuit
min_depth	int	Minimum target depth of a generated quantum circuit
max_depth	int	Maximum target depth of a generated quantum circuit
min_reps	int	Minimum number of repetitions for repeatable subroutines (e.g., Grover iterations or variational layers)
max_reps	int	Maximum number of repetitions for repeatable subroutines
min_eval_qubits	int	Minimum number of qubits used during evaluation
max_eval_qubits	int	Maximum number of qubits used during evaluation
measure	bool	Whether measurements are appended to the circuit
seed	int	Global random seed to ensure full reproducibility of circuit generation

structure and, therefore, the number and shape of relational operators in the generated SQL workload.

The domain Θ_f is derived from the parameters in Table 2. For example, the per-subcircuit depth d is sampled uniformly from the defined range between `min_depth` and `max_depth`. Quantum-specific choices about ξ can be found in Appendix K of our technical report [16].

3.5 Output Circuit

Generation stops when a maximum number of templates is reached or when a probabilistic stopping condition triggers. This prevents excessively long circuits while keeping circuit length variable across instances. After termination, the instantiated subcircuits $S_1, \dots, S_T$ are composed in order to produce the final benchmark circuit:

$$C = S_1 \circ S_2 \circ \dots \circ S_T. \quad (8)$$

Alongside C , InferQ outputs a provenance record that stores the template sequence $(f_1, \dots, f_T)$, sampled parameters (including repetitions and ξ), and global resource statistics (e.g., depth and gate counts). This record makes our generated circuit instances easy to inspect, query, and reproduce. This provenance is useful for database benchmarking because the same randomized workload can be replayed and compared across engines.

SQL query generation. Once a circuit is generated, we construct an equivalent SQL query using the implementation in [11]. We extended their package by making it a dependency to InferQ. The package uses the einstein summation (`einsum`) representation of a quantum circuit with tensor algebra. It then converts tensor contractions into relational joins as proposed by Blacher et al. [5].

Reproducibility guarantee. To make sure all sampling in InferQ is deterministic, we use seeded pseudo-random number generators⁷ and maintain a global random seed as part of the parameters in Table 2. Consequently, with the seed and the generated template sequence, both the subcircuit structure and the numerical gate parameters are reproducible across runs.

3.6 Running Example

Figure 4 illustrates one end-to-end circuit instance generated by InferQ. The workflow is intentionally analogous to a database benchmark that (i) selects a template sequence, (ii) samples

⁷See <https://docs.python.org/3/library/random#random.seed> and <https://numpy.org/doc/2.2/reference/random/generated/numpy.random.seed.html>.

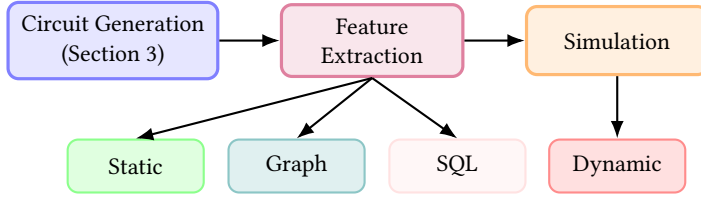


Fig. 5. InferQ feature extraction workflow. Static, graph-based, and SQL-derived features are extracted prior to simulation, whereas dynamic features can only be obtained after the simulation completes.

a scale/configuration, (iii) instantiates a small sequence of templates with parameters, and (iv) materializes both an artifact and its metadata.

Step 1 (Template selection). Starting from *START*, InferQ samples a short template sequence using the history-dependent distribution. Here the sequence is $QFT \rightarrow QPE$.

Step 2–3 (Parameters & subcircuits generation). Given a user seed, InferQ samples an instance configuration (n_q, d, k, r, ξ) . In this example, $n_q=7$ qubits, depth $d=80$, evaluation $k=4$, and repetition count $r=1$. Each template is then instantiated by binding template-local parameters (e.g., boolean flags and numeric values such as $\phi=1.998$). All choices are deterministic given the seed.

Step 4 (Merge & provenance). The instantiated subcircuits are composed in order to produce the final circuit artifact $C = S_1 \circ S_2$. Alongside C , InferQ stores a provenance record containing the template sequence and all sampled parameters, enabling exact replay and database-style inspection.

4 Circuit Feature Extraction

When we push quantum circuit simulation into an RDBMS, it induces a sequence of relational operators (e.g., joins and aggregations implementing tensor contractions) over large intermediate relations. In this setting, performance depends on cardinality growth, join selectivity, skew, and the size of intermediate states. However, such properties are not explicit in the quantum circuit representation. To make systematic progress on query optimization, physical design (e.g., indexing/partitioning), and *routing* (deciding when an RDBMS backend is beneficial), we need a set of circuit features that indicate what the induced relational query workload looks like. Ideally, a feature vector that approximates workload characteristics *before* executing the full simulation. Therefore, we include a feature extraction component in InferQ and aim to cover different sources of cost, including problem size (number of qubits, circuit depth), interaction structure, state evolution, and the generated SQL queries. As shown in Figure 5, InferQ models each circuit with four types of features: *static*, *graph*, *dynamic*, and *SQL* features, as listed in Table 4–7, respectively. Such a separation makes InferQ **extensible**: new features can be added easily.

Notably, as shown in Figure 5, feature extraction is an optional post-processing step: InferQ can generate circuits without extracting features, and computes them only when needed for analysis.

4.1 Static Features

Static features describe basic properties of a quantum circuit. These features are crucial for constructing heuristics for efficient classical simulation. For example, dense state-vector representations are only feasible for circuits with small width and a small number of qubits, as memory requirements grow exponentially with these parameters. For an RDBMS-backed simulator, gate counts and depth largely determine the number and types of relational operators generated.

As shown in Table 4, static features can be computed efficiently. We treat a circuit as an ordered list of instructions, where each instruction applies a gate to one or more qubits. This view allows us

Table 3. Notation used in circuit feature extraction.

Notation	Description
C	Quantum circuit (ordered list of gate instructions).
G	Number of circuit instructions (gate count).
N	Number of qubits in the circuit.
W	Circuit width (maximum simultaneously active qubits).
$G_I = (V, E)$	Qubit interaction graph induced by multi-qubit gates.
V	Vertex set of G_I (qubits; $ V = N$).
E	Edge set of G_I (qubit interactions; $ E $ is the edge count).
2^N	State-space dimension (length of a full statevector).
$\mathbf{p} = (p_1, \dots, p_{2^N})$	Probability vector derived from the saved statevector.
$H(\mathbf{p})$	Shannon entropy of $\mathbf{p}$ (superposition).
$\rho(\mathbf{p})$	Sparsity / support density: fraction of entries above threshold (superposition).
$S_{\text{vN}}^{(k)}$	von Neumann entropy across qubit line k (entanglement).
τ	Threshold for treating near-zero entries as zero in sparsity computation.
$\mathbb{I}(\cdot)$	Indicator function (1 if condition holds, else 0).
N_{AST}	Total number of nodes in the SQL abstract syntax tree (AST).
S	Number of SELECT statements in the query.
P	Number of predicates appearing in WHERE clauses.
J	Number of distinct relations participating in join conditions.

Table 4. Time and space complexities for static features.

Feature	Time	Space	Description
num_qubits	$O(1)$	$O(1)$	Total number of qubits
width	$O(G)$	$O(W)$	Maximum simultaneous active qubits
depth	$O(G)$	$O(W)$	Longest dependency chain
circuit_size	$O(1)$	$O(1)$	Total gate count
pauli_gate_count	$O(G)$	$O(1)$	Number of Pauli gates (X, Y, Z)
two_qubit_gate_count	$O(G)$	$O(1)$	Number of two-qubit gates
two_qubit_gate_percentage	$O(G)$	$O(1)$	Ratio of two-qubit gates
locality_ratio	$O(G)$	$O(1)$	Local vs. non-local gate ratio
idling_score	$O(G)$	$O(W)$	Idle qubit utilization
density_score	$O(G)$	$O(G)$	Gate density feature [4]
gate_counts	$O(G)$	$O(G)$	Per-gate-type histogram

to analyze the complexity of computing static features. Let G be the number of gates (instructions), N the number of qubits, and W the circuit width (maximum simultaneously active qubits). We assume $W \leq N$ and typically $G \gg N$. We summarize the notation in Table 3. All static features in Table 4 are computable in at most linear time in G with modest memory overhead, making them inexpensive yet informative metadata for the InferQ benchmark.

4.2 Graph Features

Graph features are computed from the *qubit interaction graph*, where vertices are qubits and edges represent interactions induced by multi-qubit gates. This abstraction enables the computation of classical graph properties, similar to existing works on quantum circuit compilation [3, 4] and graph-based circuit characterization [15].

Table 5. Time and space complexities for graph features.

Feature	Time	Space	Description
edge_count	$O(1)$	$O(1)$	Number of interaction edges
max_degree	$O(N)$	$O(1)$	Maximum node degree
min_cut	$O(NE)$	$O(V + E)$	Minimum cut
diameter	$O(N^3)$	$O(N^2)$	Graph diameter
radius	$O(N^3)$	$O(N^2)$	Minimum eccentricity
average_degree	$O(N)$	$O(1)$	Mean node degree
average_clustering_coefficient	$O(N^3)$	$O(1)$	Local clustering tendency
average_shortest_path_length	$O(N^3)$	$O(N^2)$	Mean shortest path length
central_point_of_dominance	$O(NE)$	$O(N)$	Graph centralization measure
std_dev_adjacency_matrix	$O(N^2)$	$O(1)$	Adjacency matrix variance

Table 6. Time and space complexities for extracting SQL features (excluding parsing).

Feature	Time	Space	Description
num_and_clauses	$O(N_{AST})$	$O(1)$	Counts logical AND operators during AST traversal
num_joins	$O(S \cdot P + J^2)$	$O(J^2)$	Counts explicit and implicit joins and stores join edges
num_select_columns	$O(N_{AST})$	$O(1)$	Counts column expressions in SELECT clauses
num_agg_funcs	$O(N_{AST})$	$O(1)$	Counts aggregate functions (e.g., COUNT, SUM, AVG)
num_where_clauses	$O(N_{AST})$	$O(1)$	Counts WHERE clause nodes in the AST
num_eq_predicates	$O(N_{AST})$	$O(1)$	Counts equality predicates used in filters and joins

Graph features make circuit structure directly accessible to database analysis: they indicate what the join structure and intermediate-result growth may look like in relational tensor computation. Intuitively, circuits whose interaction graphs are highly connected (e.g., with diameter-reducing shortcuts, high node degree, and strong centralization) tend to induce more global correlations and fewer separable subproblems, which often translates into harder contractions and less opportunity for decomposition. As summarized in Table 5, graph features can be computed in polynomial time and space with respect to qubit count N , with worst-case complexity bounded by $O(N^3)$. Let E be the number of edges in the interaction graph. Under a gate set dominated by one- and two-qubit gates, each gate introduces at most one interaction edge; hence $E = O(G)$, with worst-case space $O(N^2)$ for a weighted, undirected interaction graph. Building an adjacency-list representation takes $O(N + E)$ time and space. While some features require $O(N^3)$ time in the worst case, they remain practical for the qubit ranges where graph characterization is useful, and can be computed offline.

4.3 SQL Features

SQL features are extracted from the generated SQL queries, which are relevant to the performance of RDBMS engines for quantum circuit simulation. Table 6 summarizes the complexities to calculate these features from the circuit SQL query representation. To calculate the various joins and other clauses, we view the query tree as an abstract syntax tree (AST) and then do a traversal. Getting to this SQL representation of a quantum circuit is not expensive. Using existing methods [11, 20], we can generate these queries by translating the quantum circuit into Einstein summation (einsum) notation representing the tensor contraction sequence.

4.4 Dynamic Features

As shown in Table 7, dynamic features depend on the circuit's *output state* and cannot be derived from circuit structure alone. They are crucial for database-backed simulation because relational

Table 7. Time and space complexities for dynamic features.

Feature	Time	Space	Description
Shannon_entropy	$O(2^N)$	$O(2^N)$	Shannon Entropy of stored statevector
von_neumann_entropy	$O(N \cdot 2^N)$	$O(N \cdot 2^N)$	Von Neumann Entropy of stored statevector
sparsity	$O(2^N)$	$O(2^N)$	Sparsity of stored statevector

approaches typically rely on *sparse* or *factorized* state representations: if the state remains sparse, relational operators can operate over small relations and avoid exponential blow-up; if entanglement grows quickly, intermediate results densify and the advantage diminishes. We therefore extract three dynamic features that capture these effects: (i) Shannon entropy, (ii) von Neumann entropy, and (iii) a sparsity (non-zero support) measure of the saved statevector.

Using the vector representation for the quantum state defined in Section 2, we define sparsity and entropies as follows:

Let $\mathbf{p} = (p_1, \dots, p_{2^N})$ denote the probability distribution obtained from an N -qubit statevector in the computational basis. We compute the Shannon entropy, analogous to its use in classical and quantum information, as

$$H(\mathbf{p}) = - \sum_{i=1}^{2^N} p_i \log_2 p_i. \quad (9)$$

To quantify entanglement, we compute the von Neumann entropy across a partition defined by a cut along a qubit line k ,

$$S_{\text{vN}}^{(k)} = -\text{Tr}(\rho_{A_k} \log_2 \rho_{A_k}), \quad \rho_{A_k} = \text{Tr}_{B_k}(\rho), \quad (10)$$

where the bipartition corresponds to separating qubit line k from the remaining qubit lines. We define the sparsity

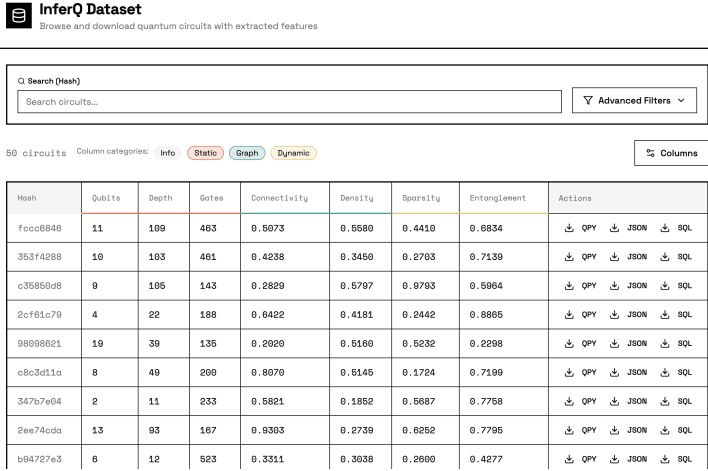
$$\rho(\mathbf{p}) = \frac{1}{2^N} \sum_{i=1}^{2^N} \mathbb{I}(p_i > \tau), \quad (11)$$

which we use synonymously with the support density, i.e., the fraction of basis states carrying non-negligible probability mass.

Here, the Shannon entropy $H(\mathbf{p})$ and sparsity $\rho(\mathbf{p})$ characterize the degree of superposition in the computational basis, with higher values indicating broader support over basis states, while the von Neumann entropy $S_{\text{vN}}^{(k)}$ measures quantum entanglement across each qubit line. We store the von Neumann entropy per qubit line to capture individual entanglement, rather than treating the full state as a combined vector, which being a pure state, has total entanglement entropy as zero. Characterizing both superposition and entanglement is essential for understanding the mechanisms underlying quantum advantage.

All dynamic features are expensive to compute and require exponential time and memory in N for exact computation due to the full pass over the length- 2^N vector. In particular, computing the von Neumann entropy across each of the N qubit lines incurs an additional factor of N on top of the full vector pass against the rest of the system, resulting in $O(N \cdot 2^N)$ time and memory. This motivates an additional use case for InferQ in Section 6.4: enabling accurate dynamic-feature estimators without requiring full simulation of the circuit.

In addition, we highlight InferQ's extensibility in supporting the addition of new features, which facilitates a wide range of use cases and experimental analyses.



InferQ Dataset
Browse and download quantum circuits with extracted features

Q Search (Hash)
Search circuits... Advanced Filters

50 circuits Column categories: Info Static Graph Dynamic Columns

Hash	Qubits	Depth	Gates	Connectivity	Density	Sparsity	Entanglement	Actions
fccc6846	11	109	463	0.5673	0.5580	0.4410	0.6834	QPY JSON SQL
353f4288	10	103	461	0.4238	0.3450	0.2703	0.7139	QPY JSON SQL
c35850d8	9	105	143	0.2829	0.5797	0.9703	0.5964	QPY JSON SQL
2cf61c70	4	22	188	0.0422	0.4181	0.2442	0.8805	QPY JSON SQL
98008021	10	39	135	0.2020	0.5100	0.5232	0.2298	QPY JSON SQL
c8c3d11a	8	49	200	0.8070	0.5145	0.1724	0.7199	QPY JSON SQL
347b7e04	2	11	233	0.5821	0.1852	0.5687	0.7758	QPY JSON SQL
2ee74cda	13	93	167	0.9303	0.2739	0.6252	0.7795	QPY JSON SQL
b04727e3	0	12	523	0.3311	0.3038	0.2600	0.4277	QPY JSON SQL

Fig. 6. InferQ’s web-based UI for exploring dataset and circuit features.

5 Implementation and Artifacts

InferQ ships as (i) a Python framework that turns a user configuration (Table 2) into *circuits*, *features*, and an *RDBMS-ready SQL workload* for simulation, and (ii) a web-based UI for interactive dataset exploration (Fig. 6). Concretely, a database researcher can provide a small configuration file (e.g., qubit count) and run the toolkit to obtain SQL queries (plus circuit artifacts in JSON and QPY formats) that are ready to execute in an RDBMS engine. Detailed descriptions of all circuit templates and the corresponding SQL code are provided in the online documentation.⁸ In addition to on-demand generation, InferQ releases a large dataset of 202,975 circuits online.⁹ Both database researchers and quantum scientists can search, filter, and download circuits and feature records through the web UI.

The framework implements the pipeline described in Section 3 and Section 4. Each circuit is assigned a *content-based hash* computed from its structure and sampled parameters. This hash serves as the primary key and links the circuit artifact, the SQL workload, and all extracted feature records. For each circuit instance, InferQ stores: (i) the generated SQL queries (as .sql scripts) that represent the simulation workload, (ii) the circuit in a portable Qiskit serialization format (e.g., .qpy), and (iii) a JSON record containing the template provenance and extracted features. InferQ includes a metadata analysis function that aggregates circuit statistics from JSON files, such as gate mix, depth, width, and interaction-graph properties. Details are provided in Appendix J of [16] and in the online repository.¹⁰

Reproducibility. The code of our benchmark can be found online.¹¹ InferQ is designed to be reproducible. Given the same code revision, dependency lockfile, and global seed (Table 2), the framework deterministically reproduces the same template sequence, sampled parameters, circuit artifact, emitted SQL, and extracted features. The content-based hash and provenance record make each benchmark instance easy to reference, regenerate, and audit. We include more implementation and reproducibility details in our technical report [16].

⁸<https://github.com/InfiniData-Lab/InferQ/blob/main/generators/sql-templates.md>

⁹https://github.com/InfiniData-Lab/InferQ/blob/main/analysis/training_data/estimator_training_data.parquet

¹⁰<https://github.com/InfiniData-Lab/InferQ/blob/main/analysis/distributions/distributions.py>

¹¹<https://github.com/InfiniData-Lab/InferQ/blob/main/README.md>

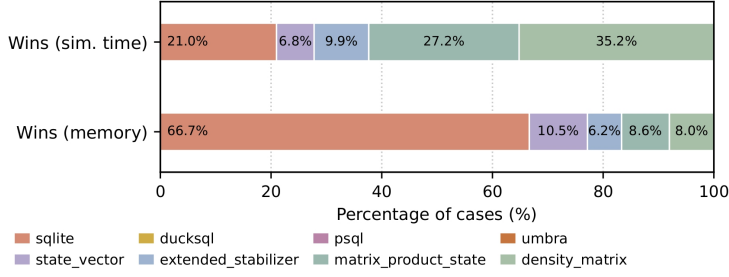


Fig. 7. Execution time and memory usage comparison between RDBMSs (SQLite, DuckDB, PostgreSQL, and Umbra) and Qiskit Aer (state vector, stabilizer, matrix product state, density matrix) over 162 circuits generated by InferQ.

Table 8. Routing performance on the 20% held-out test set. Models are evaluated using accuracy and normalized confusion-matrix entries. The columns represent true label $\rightarrow$ prediction label.

(a) Time-optimal routing						(b) Memory-optimal routing					
Model	Accuracy	rbms $\rightarrow$ rbms	qiskit $\rightarrow$ qiskit	rbms $\rightarrow$ qiskit	qiskit $\rightarrow$ rbms	Model	Accuracy	rbms $\rightarrow$ rbms	qiskit $\rightarrow$ qiskit	rbms $\rightarrow$ qiskit	qiskit $\rightarrow$ rbms
Logistic Regression	0.887	0.94	0.88	0.06	0.12	Logistic Regression	0.935	0.93	0.94	0.07	0.06
Linear SVM	0.908	0.93	0.90	0.07	0.10	Linear SVM	0.940	0.93	0.95	0.07	0.05
Decision Tree	0.940	0.78	0.97	0.22	0.03	Decision Tree	0.962	0.94	0.98	0.06	0.02
Random Forest	0.956	0.81	0.98	0.19	0.02	Random Forest	0.973	0.95	0.99	0.05	0.01
XGBoost	0.953	0.88	0.96	0.12	0.04	XGBoost	0.974	0.95	0.99	0.05	0.01

6 Evaluation

In this section, we use InferQ to quantify and explain when an RDBMS engine is a competitive simulation engine for *general*, *compositional* quantum circuits. First, in Section 6.2 we compare the InferQ-emitted SQL workloads on four RDBMS engines (PostgreSQL, SQLite, DuckDB, and Umbra) against Qiskit Aer for wall-clock runtime and peak memory. We have also trained lightweight selectors (linear and tree models) that predict when RDBMS should be chosen over Qiskit Aer. We validate the feature design via feature-importance analysis in Section 6.3 and demonstrate additional data-centric tasks enabled by InferQ in Section 6.4.

6.1 Experimental Settings

All experiments were run on a workstation equipped with an Intel Core i9–12900KF (24 threads), 64 GB RAM, and an NVIDIA RTX 3090 GPU, running Ubuntu 20.04.6 LTS with Linux kernel 5.15.0. To ensure a fair comparison between different RDBMS engines and other quantum circuit simulators, we disabled GPU and other hardware acceleration; all simulations were executed on CPU.

We implemented our prototype in Python 3.12, with all dependencies pinned in `pyproject.toml`. Core numerical routines use NumPy 2.2.5, SciPy 1.15.3, and scikit-learn (≥ 1.8). For in-memory quantum-state simulation, we use Qiskit 2.0.1 together with Qiskit Aer 0.17.0. For database-backed simulation, we generate SQL using the `sql-einsum` code generator¹² and build on the RDBMS simulator introduced by [11]. PostgreSQL is version 12.22 accessed via `psycopg2-binary` 2.9.10. SQLite is version 3.50.4, and DuckDB is version 1.1.3. Umbra (v0.2–1665–gfeeb4bb5d) is accessed through its PostgreSQL-compatible (18.1) server interface using `psycopg2-binary` 2.9.10. For the out-of-core experiments, memory limits are enforced with Docker Engine (≥ 24.0) using per-container cgroups. Unless otherwise stated, we report the mean values across five repetitions.

¹²https://github.com/ti2-group/sql-einsum/tree/main/generate_sql_code

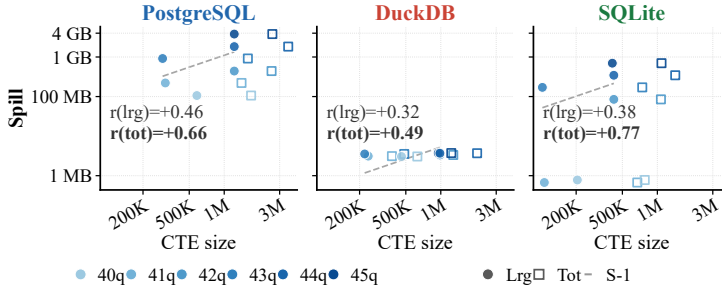


Fig. 8. Out-of-core spill versus intermediate relation sizes.

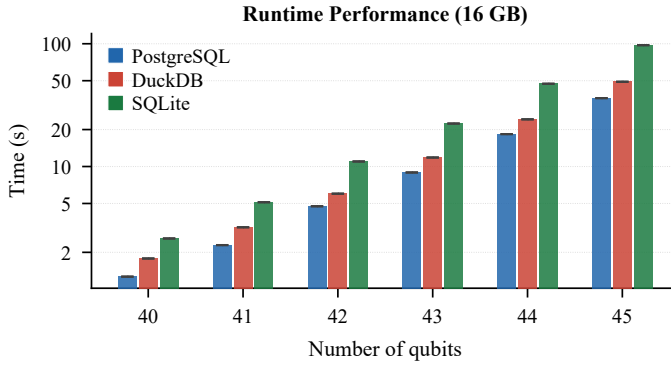


Fig. 9. Runtime versus #qubits.

6.2 Use case 1: RDBMS as Simulation Backend

Q1: Is an RDBMS an efficient simulation engine?

To answer this question, we generated 162 circuits¹³ using InferQ. For each circuit, we measured wall-clock runtime and peak memory when simulating it with (i) RDBMS engines (PostgreSQL, SQLite, DuckDB, and Umbra) executing the SQL workload emitted by InferQ, and (ii) Qiskit Aer using four built-in simulation methods¹⁴ (statevector, extended_stabilizer, matrix_product_state, and density_matrix).

Figure 7 reports, for each metric, the fraction of circuits for which a backend achieves the best result (minimum execution time or minimum peak memory) among all evaluated options. As expected, Qiskit Aer attains the best execution time on most circuits. SQLite achieves the lowest peak memory on 66.7% of the circuits (108/162). This suggests that RDBMS engines can be competitive simulation backends, particularly for *memory*. This result motivates further work on SQL workload optimization. Moreover, it is interesting to explore learning-based routing in simulation, i.e., predicting when an RDBMS engine is likely to outperform conventional simulators. This leads to our next question.

Q2: How to decide whether to choose an RDBMS as a simulation engine?

¹³https://github.com/InfiniData-Lab/InferQ/blob/main/analysis/training_data/rdbms_all_methods_training_data.parquet

¹⁴Qiskit Aer Simulation Methods: https://qiskit.github.io/qiskit-aer/tutorials/1_aersimulator.html#Simulation-Method-Option

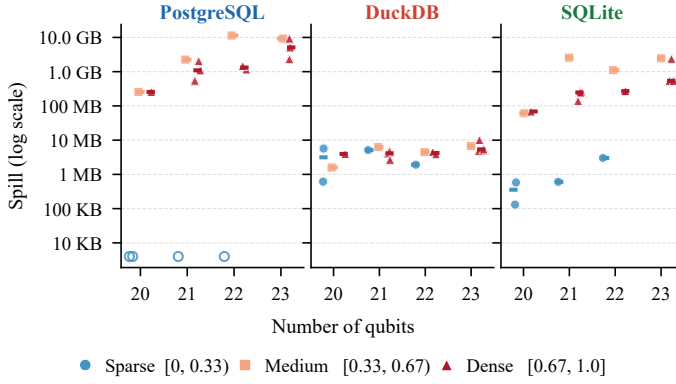


Fig. 10. Spill versus #qubits on sampled InferQ circuits.

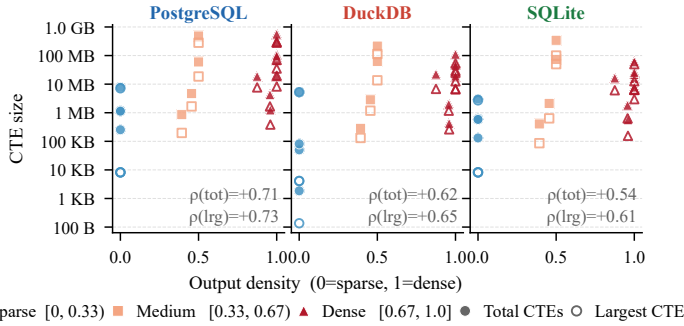


Fig. 11. Intermediate CTE sizes versus output density on sampled InferQ circuits.

We model this choice as a *binary selection* problem: given a circuit, predict whether an RDBMS engine achieves lower runtime and/or lower peak memory than a conventional simulator. Prior database-oriented studies of SQL-based simulation report strong results on a small set of highly structured circuits (e.g., sparse state-preparation workloads) [5, 11]. Using InferQ, we move beyond these constrained cases and quantify when an RDBMS is beneficial over a substantially broader set of general circuits. We have created a training dataset of 7,705 circuits¹⁵ and train five standard models as selectors: logistic regression, linear SVM, Decision Tree, random forest (RF), and XGBoost. For each circuit, we have logged runtime and peak memory for Qiskit Aer and the RDBMS backends. This yields ground-truth labels (*RDBMS wins* vs. *Qiskit wins*) for both objectives.

Table 8a shows that these models achieve high accuracy using the features extracted in Section 4, despite class imbalance (since SQLite is faster only for a subset of circuits). Random Forest achieves the highest runtime-routing accuracy (95.6%), while XGBoost follows closely (95.3%) and better identifies RDBMS-win cases. The best linear model, linear SVM, too achieves 90.8% accuracy with very low off diagonal confusion entries on the test set, suggesting that much of the time-selection boundary is close to linear in our feature space. Given the strong performance of these lightweight models, we did not evaluate more complex models that require substantially higher training and tuning cost.

For memory efficiency (Table 8b), performance is similarly strong: XGBoost achieves the highest accuracy (97.4%), while random forest follows closely (97.3%). Linear SVM also remains highly

¹⁵Training data (Parquet) available at https://github.com/InfiniData-Lab/InferQ/blob/main/analysis/training_data/rdbms_training_data.parquet. The results reported in Figure 1 also used this dataset.

competitive (94.0% accuracy). Overall, InferQ features in Section 4 make the RDBMS selection decision predictable with simple, practical models.

Routing overhead. The learned selector is trained offline on labeled workloads generated by InferQ and deployed only for inference. For XGBoost, which provides the best accuracy–efficiency trade-off among the models in Tables 8a and 8b, inference takes only 13.9 ms for runtime routing and 13.5 ms for memory routing per batch. Appendix C of our technical report [16] reports the full training and inference times for all models. Our goal is therefore not to add heavyweight ML components to simulation, but to study a lightweight learned routing policy for backend selection, aligned with recent database work on learned decision-making and cost models [14, 32].

The previous experiments compare runtime and peak memory only when all backends finish successfully. For larger circuits, conventional in-memory simulators such as Qiskit Aer can fail once the state representation exceeds available memory. RDBMS engines can instead execute queries out of core by spilling intermediate results to disk. This motivates the following question:

Q3: Can RDBMS-based simulation complete workloads that no longer fit in memory?

We evaluate Q3 using a family of large, sparse circuits with up to 45 qubits under a 16 GB memory limit and one CPU core. Full experimental details are in Appendix A of our technical report. Under this memory limit, Qiskit Aer’s exact statevector method fails from 30 qubits onward, while PostgreSQL, DuckDB, and SQLite complete all runs by spilling temporary data to disk. Figure 8 shows that intermediate relations remain MB-scale, but spill reaches GB-scale and correlates more with the total intermediate volume than with the single largest intermediate, indicating that cumulative joins and group-bys lead to externalization. Figure 9 reports runtime: PostgreSQL is fastest, DuckDB is second, and SQLite is slower but robust. We have also observed that runtime is strongly correlated with spill volume.

To understand out-of-core performance and the characteristics of the simulation workload, we have also sampled 17 InferQ circuits (20–23 qubits) with different output state sparsity. Figures 10 and 11 show that sparse outputs usually incur less spill, but output state sparsity alone does not determine spill: several medium-density circuits spill more than dense ones due to intermediate-relation growth and optimizer choices (e.g., join ordering and materialization). Full settings and additional results are reported in [16], Appendix A.

Take-away. (1) Across a wide range of *general, compositional* circuits, RDBMS engines can be competitive simulation engines, especially in peak memory, and in some cases also in runtime. *This enables follow-up work on optimizing the emitted SQL to further improve RDBMS performance.* (2) InferQ enables data-centric routing at scale: it generates diverse circuits, labels each circuit with observed runtime and peak memory, and exposes features that allow simple models to decide when an RDBMS engine should be used. *This provides a foundation for learning-based simulator selection and optimization, trained on InferQ-generated circuits.* (3) For large circuits that exceed memory, RDBMS backends can still complete simulation via out-of-core execution. *This highlights a distinct advantage of database-backed simulation and enables follow-up work on spill-aware planning and optimization.*

6.3 Feature Validation

Next, we want to go deeper and understand:

Q4: Which properties determine whether an RDBMS engine is an efficient simulation engine?

Table 9. Routing performance using only SQL-derived features on the 20% held-out test set. Models are evaluated using accuracy and normalized confusion-matrix entries. The columns represent true label $\rightarrow$ prediction label.

(a) Time-optimal routing						(b) Memory-optimal routing					
Model	Accuracy	rdbms $\rightarrow$ rdbms	qiskit $\rightarrow$ qiskit	rdbms $\rightarrow$ qiskit	qiskit $\rightarrow$ rdbms	Model	Accuracy	rdbms $\rightarrow$ rdbms	qiskit $\rightarrow$ qiskit	rdbms $\rightarrow$ qiskit	qiskit $\rightarrow$ rdbms
Logistic Regression	0.787	0.88	0.77	0.12	0.23	Logistic Regression	0.735	0.77	0.71	0.23	0.29
Linear SVM	0.803	0.88	0.79	0.12	0.21	Linear SVM	0.737	0.77	0.72	0.23	0.28
Decision Tree	0.915	0.76	0.94	0.24	0.06	Decision Tree	0.836	0.81	0.85	0.19	0.15
Random Forest	0.932	0.75	0.96	0.25	0.04	Random Forest	0.866	0.82	0.90	0.18	0.10
XGBoost	0.924	0.81	0.94	0.19	0.06	XGBoost	0.842	0.84	0.85	0.16	0.15

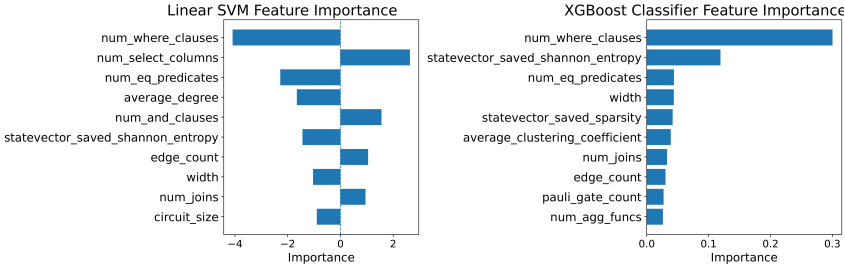


Fig. 12. Importance of top 10 InferQ numeric features for SVM and XGBoost trained for optimizing execution time.

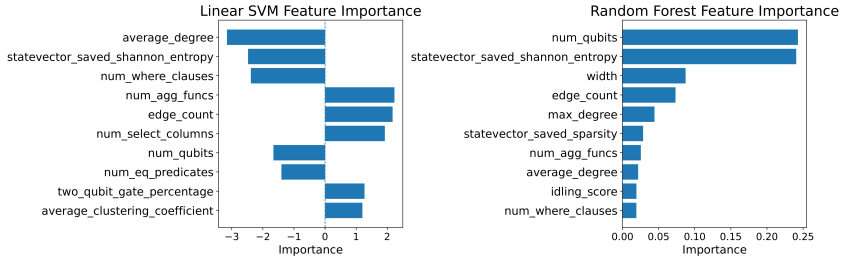


Fig. 13. Importance of top 10 InferQ numeric features for SVM and Random Forest trained for optimizing memory.

Our goal is to validate that the four feature groups introduced in Section 4 (*static*, *graph*, *dynamic*, *SQL*) capture the key signals behind the routing outcome in Section 6.2. We therefore analyze feature importance for two high-performing tree-based models: XGBoost for *runtime* and Random Forest for *peak memory*. We additionally report a Linear SVM to expose the *direction* of influence via signed coefficients. We also include an ablation study.

6.3.1 SQL-only features. We first restrict to SQL-derived features extracted from output SQL queries representing the generated circuits (Section 4.3, Table 6). Table 9a shows that these SQL-only features already yield strong routing accuracy, especially for time-optimal routing (XGBoost exceeds 92% accuracy on the held-out test set). This indicates that the engine choice is strongly correlated with *query shape*, without requiring circuit-specific features. We have included more results on dominant SQL signals in Appendix L of [16].

6.3.2 All InferQ features. We then include the full feature set from Section 4. For runtime (Figure 12), for both the Linear SVM and XGBoost, features associated with state growth and interaction

Table 10. Feature-group ablation for routing (XGBoost)

Feature set	Runtime		Memory	
	Acc.	F1	Acc.	F1
Static + Graph + SQL + Dynamic	0.95	0.84	0.98	0.98
Static + Graph + SQL	0.93	0.78	0.97	0.96
Static + Graph	0.92	0.74	0.96	0.95
Static	0.92	0.74	0.95	0.94

Table 11. Sparsity and Entropy (Dynamic metrics) estimators performance for Random Forest and Linear Regression models. Reported are coefficient of determination (R^2) and root mean squared error (RMSE) comparing predictions to actual values.

Model	Sparsity Model	Entropy Model
Random Forest R^2	0.7737	0.8936
Random Forest RMSE	0.1928	1.9931
Linear Regression R^2	0.3934	0.6956
Linear Regression RMSE	0.3156	3.3714

intensity are more important, including shannon_entropy, width, and edge_count, together with the SQL-derived features. With XGBoost, in addition, graph-related features are important, such as average_clustering_coefficient, average_shortest_path_length, and average_degree, indicating that irregular interaction structure matters for runtime differences.

For peak memory (Figure 13), the Linear SVM and Random Forest agree on the main features: num_qubits and shannon_entropy dominate, with additional contributions from width, edge_count, degree statistics, and sparsity measures. The Random Forest captures non-linear thresholds in these features, separating regimes where sparse relational execution substantially reduces peak memory.

6.3.3 Ablation study. To quantify the contribution of each feature group, we further run a feature-group ablation on the routing task. Table 10 reports the results for XGBoost on the 7705 dataset. The full results with all models are reported in Appendix B of [16]. In Section 6.3.1, we have observed that SQL-only features already provide a useful indication, especially for runtime routing, showing that the shape of the generated SQL query workload is informative. Here, we see that the full feature set gives the best overall accuracy and F1 score.

Take-away. (1) SQL-only features are already strong predictors: the SQL query workload shape (joins, predicates, aggregates) explains much of when an RDBMS should be chosen. *This enables follow-up work on SQL-level cost modeling and query optimization tailored to quantum circuit simulation workloads.*

(2) When using the full feature set, the top features are consistent with the underlying bottlenecks: state growth indicators such as entropies, circuit scale like qubit count, and interaction structure (graph features). *This supports future work on data-centric models that learn when to use an RDBMS engine and how to optimize execution from InferQ-generated training data.*

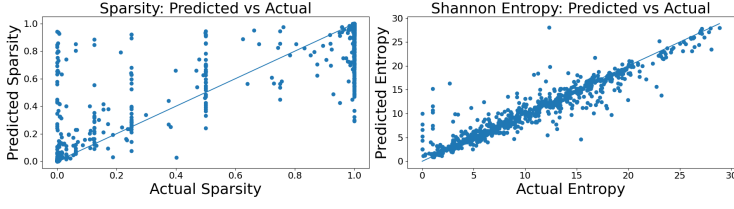


Fig. 14. Predicted vs Actual value for the dynamic feature RF regression estimators for subset (1000 entries) of test set.

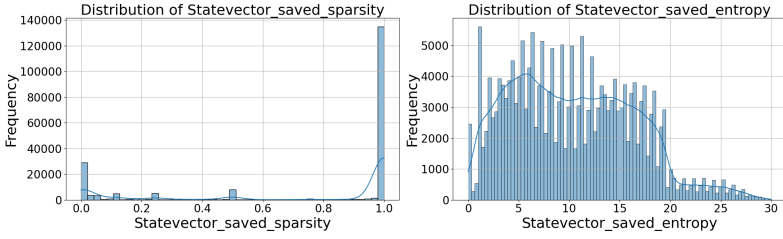


Fig. 15. Distribution of dynamic features over 202k entries in InferQ with Dynamic features.

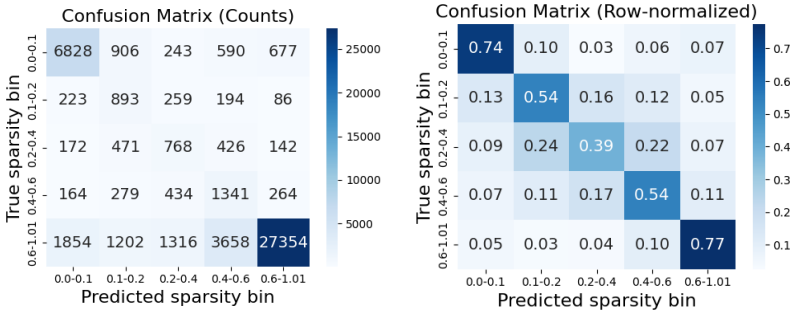


Fig. 16. Confusion matrices with counts and normalization for the random forest classifier used for sparsity-band prediction.

6.4 Use case 2: Predicting Entropies and Sparsity

Dynamic features from Section 4 (Shannon entropy, von Neumann entropy, and sparsity) are only available *after* simulation, since they are computed from the output state. This limits their use for larger circuits and motivates a second, data-centric use case: learning inexpensive estimators that predict these dynamic properties from static/graph/SQL features. Besides being useful for database researchers (Use case 1), these metrics are also fundamental to quantum analysis.

Q5: Can we estimate complex quantum properties using InferQ features?

We construct estimators for `sparsity` and `shannon_entropy` using 202,975 circuits for which dynamic features are available.¹⁶ We train (i) a Random Forest regressor and (ii) a linear regression baseline, and evaluate prediction quality using R^2 and RMSE as shown in Table 11.

As Table 11 shows, Random Forest substantially outperforms the linear model for both targets. Shannon entropy is modeled well by the feature set, while sparsity is harder to regress accurately. This is also visible in the prediction-versus-actual plots in Figure 14.

To understand why sparsity is challenging, we inspect its empirical distribution (Figure 15). Sparsity is not smoothly distributed. Instead, it concentrates in distinct bands. This suggests treating sparsity as a coarse-grained classification problem. Concretely, we bin sparsity into $[0.0, 0.1)$, $[0.1, 0.2)$, $[0.2, 0.4)$, $[0.4, 0.6)$, $[0.6, 1.01]$ and train a Random Forest classifier. This yields approximately 77% accuracy, and the normalized confusion matrix is concentrated near the diagonal (Figure 16), indicating that InferQ features capture sparsity even when exact regression is difficult.

Finally, feature-importance analysis for the sparsity-band classifier highlights `pauli_gate_count` as a stronger predictor than `num_qubits`. This suggests that (in our generated workload) sparsity is driven more by the gate mix than by state dimension alone, offering a concrete, data-derived signal that can guide both simulator design and feature engineering.

Take-away. (1) InferQ enables learning *cheap estimators* for expensive dynamic properties (e.g., entropy and sparsity) using only features that do not require simulation. *This enables follow-up work on fast, simulation-free predictors of state complexity for larger circuits.*

(2) The learned models reveal structure in the data (e.g., sparsity bands) and highlight which circuit characteristics are most predictive. *This lays a practical foundation for improving estimators (e.g., hybrid classification/regression) and for connecting circuit structure to quantum-state behavior in a data-centric manner.*

7 Conclusion

InferQ makes SQL-based quantum circuit simulation accessible as a database benchmarking problem: it generates compositional circuits as RDBMS-ready SQL workloads, attaches features that explain relational workload difficulty, and provides both an on-demand toolkit and a large dataset for reproducible analysis. Our evaluation shows that RDBMS engines can be competitive, especially in peak memory across a broad set of general circuits, and that lightweight models trained on InferQ features can reliably predict when an RDBMS engine will be preferable. We expect InferQ to enable follow-up work in (i) query optimization and physical design for emitted simulation SQL, and (ii) data-centric routing and cost modeling for simulator selection, trained at scale on InferQ-generated circuits and features.

Outlook. As future work, we plan to further optimize RDBMS-based simulation and develop array-native execution for tensor-aware DBMSs. InferQ can also be extended from circuit generation to end-to-end validation for quantum algorithm design, covering correctness checks, iterative test–debug cycles, and hardware-aware constraints such as native gate sets and device connectivity.

Acknowledgment

This publication was supported in part by the Dutch Research Council (VI.Veni.222.439). We thank Tim Littau and Wenbo Sun for programming assistance and sharing code from their prior work, and Arun Pati and Giancarlo Gatti for discussion and feedback on quantum primitive composition.

¹⁶Estimator training data (Parquet) available at https://github.com/InfiniData-Lab/InferQ/blob/main/analysis/training_data/estimator_training_data.parquet.

References

- [1] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Physical Review A* 70, 5 (Nov. 2004). doi:10.1103/physreva.70.052328
- [2] Daniel S Abrams and Seth Lloyd. 1999. Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors. *Physical Review Letters* 83, 24 (1999), 5162.
- [3] Medina Bandic, Carmen G. Almudever, and Sebastian Feld. 2023. Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques. *Quantum Machine Intelligence* 5, 2 (Oct. 2023). doi:10.1007/s42484-023-00124-1
- [4] Medina Bandic, Pablo le Henaff, Anabel Ovide, Pau Escofet, Sahar Ben Rached, Santiago Rodrigo, Hans van Someren, Sergi Abadal, Eduard Alarcon, Carmen G. Almudever, and Sebastian Feld. 2024. Profiling quantum circuits for their efficient execution on single- and multi-core architectures. arXiv:2407.12640 [quant-ph] <https://arxiv.org/abs/2407.12640>
- [5] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and Portable Einstein Summation in SQL. *Proceedings of the ACM on Management of Data (PACMOD)* 1, 2, Article 121 (jun 2023), 19 pages.
- [6] Matthias Boehm, Matteo Interlandi, and Chris Jermaine. 2023. Optimizing Tensor Computations: From Applications to Compilation and Runtime Techniques. In *Companion of the 2023 International Conference on Management of Data (SIGMOD)*. 53–59.
- [7] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. *Contemp. Math.* 305 (2002), 53–74.
- [8] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2017. Towards Linear Algebra over Normalized Data. *Proceedings of the VLDB Endowment* 10, 11 (2017).
- [9] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. <https://arxiv.org/abs/1411.4028>
- [10] András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. 2019. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st annual ACM SIGACT symposium on theory of computing*. 193–204.
- [11] Rihan Hai, Shih-Han Hung, Tim Coopmans, Tim Littau, and Floris Geerts. 2025. Quantum Data Management in the NISQ Era. *PVLDB* 18, 6 (2025), 1720–1729.
- [12] Rihan Hai, Shih-Han Hung, Tim Coopmans, Tim Littau, and Floris Geerts. 2025. Quantum Data Management in the NISQ Era: Extended Version. <https://arxiv.org/abs/2409.14111>
- [13] Aram W Harrow, Avinandan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters* 103, 15 (2009), 150502.
- [14] Roman Heinrich et al. 2025. How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks. *SIGMOD* 3, 3 (2025), 172:1–172:27. doi:10.1145/3725309
- [15] Javier Martín Hernández and Piet Van Mieghem. 2015. Classification of graph metrics. <https://api.semanticscholar.org/CorpusID:37136216>
- [16] Andrei Ilinescu, Aadi Patwardhan, and Rihan Hai. 2026. InferQ: A Database-Oriented Benchmark for Quantum Circuits Simulation. arXiv:2607.29134 [quant-ph] doi:10.48550/arXiv.2607.29134
- [17] Mahmoud Abo Khamis, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models Over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020).
- [18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
- [19] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2022. QASMBench: A Low-level QASM Benchmark Suite for NISQ Evaluation and Simulation. arXiv:2005.13018 [quant-ph] <https://arxiv.org/abs/2005.13018>
- [20] Tim Littau and Rihan Hai. 2025. Qymera: Simulating Quantum Circuits using RDBMS. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)*. ACM, 179–182. doi:10.1145/3722212.3725126
- [21] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic optimization of matrix implementations for distributed machine learning and linear algebra. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 1222–1234.
- [22] Nantia Makrynioti and Vasilis Vassalos. 2019. Declarative Data Analytics: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 33, 6 (2019), 2392–2411.
- [23] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (2008), 963–981.
- [24] John M Martyn, Zane M Rossi, Andrew K Tan, and Isaac L Chuang. 2021. Grand unification of quantum algorithms. *PRX quantum* 2, 4 (2021), 040203.

- [25] Michael A Nielsen and Isaac L Chuang. 2010. *Quantum Computation and Quantum Information*. Cambridge university press.
- [26] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions into DBMSs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 10 (2023), 10295–10308.
- [27] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* 7 (July 2023), 1062. doi:10.22331/q-2023-07-20-1062
- [28] Wenbo Sun, Qiming Guo, Wenlu Wang, and Rihan Hai. 2025. TranSQL+: Serving Large Language Models with SQL on Low-Resource Hardware. *SIGMOD* 3, 6, Article 371 (Dec. 2025), 27 pages.
- [29] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N. Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R. Martonosi, and Frederic T. Chong. 2022. SupermarQ: A Scalable Quantum Benchmark Suite. arXiv:2202.11045 [quant-ph] <https://arxiv.org/abs/2202.11045>
- [30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [31] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1919–1932.
- [32] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proc. ACM Manag. Data* 1, 4 (2023), 255:1–255:27. doi:10.1145/3626769
- [33] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor relational algebra for distributed machine learning system design. *Proceedings of the VLDB Endowment* 14, 8 (2021).
- [34] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database Meets Artificial Intelligence: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2020).

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026