Simulation of quantum circuits with array-like DBMSs

An Empirical Case Study of SciDB versus Relational DBMSs

Bruno Faliszewski
Supervisor(s): Dr. Rihan Hai, Tim Littau
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Bruno Faliszewski
Final project course: CSE3000 Research Project
Thesis committee: Dr. Rihan Hai, Tim Littau, Prof. Dr. Stephanie Wehner

Abstract

Quantum circuit simulators running on classical hardware are essential for developing the field while quantum technology matures, but they struggle to scale to the large, memory-intensive states that many circuits produce. Recent work has shown that relational database management systems (RDBMSs) can simulate quantum circuits and that they have an advantage when the workload exceeds the available memory. However, quantum circuits are naturally expressed as tensor contractions, which suggests that array-based database systems - designed to store and operate on multi-dimensional arrays - might be a more natural fit. This possibility has not been systematically evaluated. This paper presents an empirical case study of SciDB, a representative array (tensor-based) DBMS, benchmarked against two relational engines (PostgreSQL and Umbra) on two contrasting circuits: GHZ state preparation, whose states are sparse, and the Quantum Fourier Transform, whose states are dense, across increasing qubit counts. We additionally apply autotuning via MLOS/SMAC to optimise SciDB's configuration. Across all tested cases, SciDB was the slowest engine. The sparse GHZ workload exposes a large fixed per-step overhead, while on the dense QFT, this overhead amortises as the gap to the relational engines narrows from over three orders of magnitude to roughly twofold at 24 qubits against PostgreSQL. Autotuning yielded no improvement over the default configuration, indicating that SciDB's bottleneck lies outside the tuned parameters. We conclude that SciDB offers no advantage over RDBMSs for in-core simulation at these scales, and identify out-of-core simulation - the regime in which database backends are expected to excel - as the central open direction.

1 Introduction

Relational database management systems (RDBMSs) have long been the backbone of data-intensive applications, with mature optimisation techniques, robust execution engines, and efficient storage models. They are especially efficient for the out-of-core calculations. At the same time, quantum computing has shown that, in theory, it can solve problems that are too computationally complex for current classical computers. However, this will be possible only when sufficient technology is developed, and until this time it is important to develop the field with use of simulators of quantum circuits on classical hardware. Current simulators are highly optimised, but still face scalability issues. This motivates the exploration of alternative approaches to simulation of quantum circuits.

Recent work has demonstrated that RDBMSs can be used to simulate quantum circuits by representing quantum states and operations as relational data and SQL workloads [4, 7, 1]. In particular, this approach shows promise for quantum circuits because, when quantum circuits have sparse intermediate states, the data representation can remain compact, and when data exceeds the available RAM in dense circuits, RDBMSs can spill out-of-core and continue the calculation [15]. It remains unclear whether RDBMSs are the most efficient database systems for simulating quantum circuits. Tensor-based database systems [9, 13] might be more optimised for this task, since they specialise in storing data as arrays.

This paper investigates whether SciDB, as a representative tensor-based DBMS, has an advantage over RDBMSs in simulating quantum circuits, since it was created to optimise storing tensors in databases. Furthermore, does autotuning with MLOS improve the performance of SciDB for simulating quantum circuits?

The main contribution of this work is an empirical evaluation of SciDB under a specific set of quantum circuits and tuning of the DBMS to optimise the performance. The paper is structured as follows: first, the methodology and experimental setup are presented, followed

by, the results and their analysis, and finally, conclusions and directions for future work are provided.

2 Problem Description

Classical computers store information in bits that can hold the value of either 0 or 1, however quantum computers operate on "qubits". Qubits can be 0, 1 or in a so-called superposition of the two, they have complex amplitudes of both 0 and 1 in that case. The hope for the advantage of quantum computers over classical machines is based on the fact that their input space grows exponentially with each additional qubit. For the early stages of the development of quantum technology it is necessary to use simulators implemented on classical computers, since current state-of-the-art quantum computers are far from perfect. They are noisy and can hold the values of the qubits for a very short time.

Current simulators, such as Qiskit [5], are highly optimised, however they struggle to scale for the large quantum states that require large amounts of memory. Recent work shows that RDBMSs (Relational Database Management Systems) can be more efficient when the workload of the quantum circuit exceeds the machine's RAM [15], since they are able to go for out-of-core and continue the computation.

However RDBMSs are designed to store tabular data, and quantum circuits are represented as an ordered series of quantum gates, which in turn are represented as matrices. Simulating such a circuit is a tensor contraction of all the gates and the input quantum state [1]. This design of quantum circuits seems more fitting for tensor-based DBMSs, which, in contrast to RDBMSs, store data in array-like data structures.

3 Methodology

3.1 Tensor-based database systems

3.1.1 TileDB

TileDB [9] is a storage manager for multi-dimensional arrays. It is optimised for both dense and sparse arrays by storing the data in ordered collections called fragments. Each fragment is supposed to represent a batch of updates to the array.

TileDB has an SQL endpoint to its data, which can be accessed via MariaDB, however the library that provided access to this endpoint has been deprecated [14], and currently it is only possible to do that through TileDB's cloud infrastructure. Therefore we determined that it is out of scope for this research project, because there is no possibility to collect meaningful and comparable data on the performance of the database.

3.1.2 SciDB

SciDB [13] is another tensor-based database system which specialises in storing scientific data. The creators of SciDB determined that arrays are the most natural data structures for data that is collected in science across a diverse range of fields.

It stores the data in a multi-dimensional array, where each dimension has a specified data type. Furthermore, each cell can also be a vector of values called attributes. SciDB stores rectilinear regions of the array in chunks [12]. The size of the chunks can be specified.

You can also choose if you want the data to be stored in a sparse array, in this case, only the non-zero values will be stored together with their coordinates in the array.

For the quantum circuits we store each unique gate as an array with 2 dimensions and 2 attributes. Gates are matrices with complex values, therefore we have 2 dimensions for the shape of the matrix, and the 2 attributes represent the complex values, one for the real part and the second for the imaginary part.

In SQL, tensor contractions are done by using $JOIN$, $GROUPBY$ and $SUM()$. SciDB, however, does not use SQL since it is not an RDBMS. Instead, its creators decided to make a new language designed specially for the array-like databases. AFL [12] (Array Functional Language) has a different syntax to meet the tensor-based database requirements. So in SciDB, we use $cross\_join$, $aggregate$, and $apply()$, which can be used to perform the exact same tensor contraction as in SQL.

SciDB also has a specific architecture. It uses an HTTP API to communicate with the client. The overview of the architecture is shown in Figure 1. The SciDB node consists of the SHIM [10] service, the SciDB instance, and the file system. If the client wants to execute a query on SciDB, they must send an HTTP request to SHIM, which handles all the logic. It communicates both with the SciDB instance and the file system.

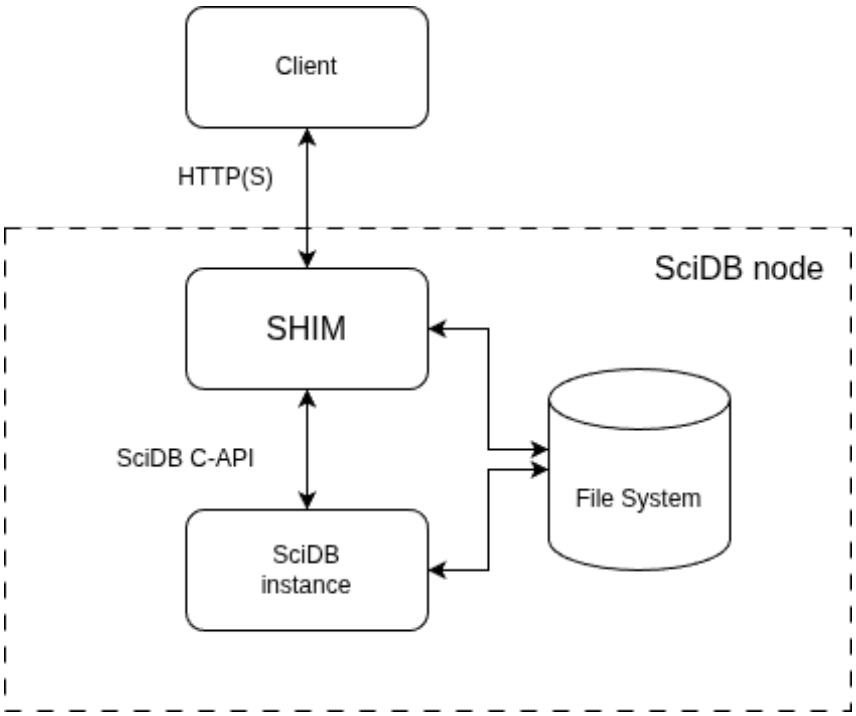

Figure 1: SciDB Architecture. Adapted from the SHIM documentation [10]

The first approach to simulating quantum circuits in SciDB was to fully copy the SQL logic. That is to create one big nested query using $cross\_join$, $aggregate$, and $apply()$, send it to the SciDB server and run it. However, it turned out that these queries were too large for the SHIM service to handle properly.

To overcome this issue, we decided to make use of SciDB's functionality. First, we create

and upload all of the unique tensors that will be used in the specific simulation to the SciDB file system one by one. After that, we make the contractions, again one after another. First, the input state with the first matrix, then the result of this contraction with the second matrix and so on. This way, the queries are smaller, and SHIM has a smaller workload that does not grow exponentially.

3.2 Quantum Circuits

In this subsection, we will explain the quantum algorithms that were used in the experiments described later.

3.2.1 GHZ state preparation

GHZ preparation is a circuit that transforms the state vector $|0\dots0\rangle$ into GHZ state $\frac{|0\dots0\rangle+|1\dots1\rangle}{\sqrt{2}}$. This state is fully entangled, therefore it is commonly used in different quantum circuits or protocols. A thing to notice is that the resulting state always has only 2 non-zero coefficients, meaning that it is highly sparse. This is exactly the kind of circuit that RDBMSs perform well on. The circuit for GHZ preparation on 4 qubits can be seen in Figure 2.

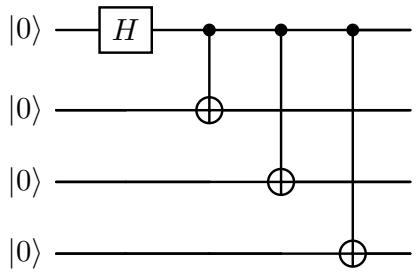

Figure 2: GHZ state preparation circuit on 4 qubits, producing $\frac{|0000\rangle+|1111\rangle}{\sqrt{2}}$.

3.2.2 Quantum Fourier Transform

QFT is an algorithm whose purpose is to change the basis of the input state, just like the classical Fourier transform changes a signal's basis from the time domain into the frequency domain. This circuit is used in many other quantum algorithms, such as Shor's algorithm for factoring [11]. Its intermediate states are very dense, thus it is an example of the worst-case scenario for the RDBMS simulations. The example of the QFT on 4 qubits can be seen on Figure 3.

3.3 Autotuning tool

To further optimise SciDB for the quantum circuit simulation we opted to use Microsoft's MLOS [2], which is an open-source tool that enables autotuning of systems against a specific workload. The workflow is presented in Figure 4 and works as follows [8]:

- Running a workload against a system

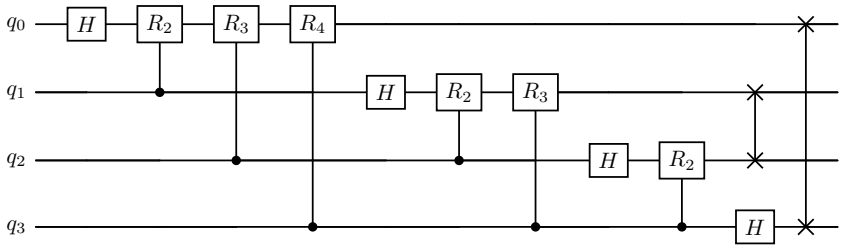

Figure 3: 4-qubit Quantum Fourier Transform (QFT). $R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}$.

- Retrieving results of the workload
- Feed that data into an optimiser
- Get a new suggested config from the optimiser
- Repeat the process with a new config applied to the system

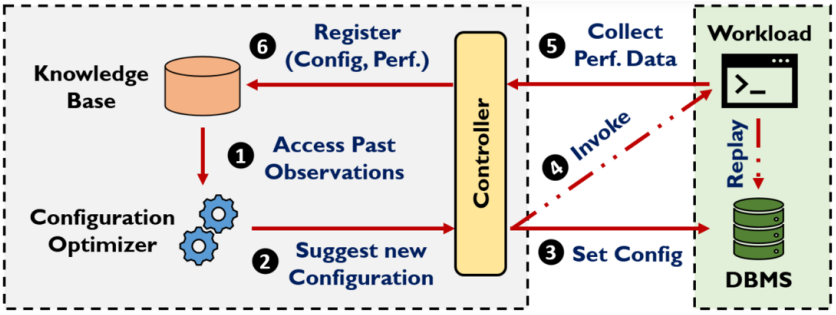

Figure 4: MLOS autotuning workflow. Reproduced from [8]

In this case, a workload is running 3 QFT circuits with 3, 4 and 5 qubits respectively against a SciDB system. As an optimiser, we used SMAC [6], which is a Bayesian Optimisation from the MLOS package. The score for each tuning iteration is defined as the mean execution time of all the provided circuits in seconds. Each circuit is run 10 times for 1 workload here. Therefore, the score is the mean execution time of all 30 runs in this case.

For the autotuning of SciDB, we used the mlos-bench library from Python. We specified 5 tunable knobs, which are the parameters that are passed to the configuration file and later tuned so that we can end up with the most optimised configuration of the database. The knobs were divided into two groups based on their application cost. The first group, `scidb_query_params`, can be applied per-query without restarting the database engine, making them cheap to evaluate during tuning. The second group, `scidb_server_params`, requires a full SciDB restart whenever their values change, and are therefore applied only when the optimiser selects a new value.

- **dim_chunk_factor** (float, range [0.1, 1.0], default 1.0) - A scaling factor applied to the chunk size along dimensional axes of SciDB arrays. SciDB stores multi-dimensional

arrays in fixed-size chunks. This factor allows the optimiser to reduce the effective chunk size proportionally, trading off I/O granularity against memory pressure and parallelism.

- **flat_chunk_factor** (float, range [0.1, 1.0], default 1.0) - Analogous to `dim_chunk_factor` but applied to flat (one-dimensional) array storage. Tuning this factor affects how data is partitioned on disk and in memory for flat array representations, influencing scan and aggregation performance.
- **mem_array_threshold_mb** (int, range [5000, 10000] MB, default 7000 MB) - The maximum amount of memory, in megabytes, that SciDB is permitted to use for in-memory array storage before spilling intermediate results to disk. Higher values reduce spill overhead at the cost of increased memory pressure; lower values constrain memory usage but may increase I/O.
- **merge_sort_nstreams** (int, range [2, 20], default 4) - The number of sorted streams that SciDB merges simultaneously during external sort operations. A higher value increases merge fan-in, potentially reducing the number of merge passes needed for large datasets, but also increases the memory required per sort operation.
- **execution_threads** (int, range [2, 15], default 4) - The number of worker threads allocated to query execution within each SciDB instance. Increasing this value allows greater intra-query parallelism, which can benefit CPU-bound operations, while excessively high values may introduce scheduling overhead or contend with system-level concurrency.

4 Experimental Setup

To evaluate the performance of SciDB in simulating quantum circuits, we set up the following two experiments.

First, we simulated the GHZ state preparation for SciDB, PostgreSQL, and Umbra. This was done over a range of qubits, from 3 to 40. Each engine ran the GHZ circuit 5 times. Median from these runs was used to present the execution time and the peak memory usage. The line in the plot is the median time, and it has a shaded min-max band around it.

The second experiment was a QFT (Quantum Fourier Transform) circuit. We also ran it across SciDB, PostgreSQL, and Umbra. With qubit counts from 3 to 30, and each engine also ran 5 times, so that the 2 experiments are comparable. The same plot was used to visualise the execution time and peak memory.

All the engines SciDB, PostgreSQL, and Umbra were run in their respective Docker containers. SciDB was run on version 19.11 of the rvernica/scidb Docker image, and scidb-py python library version 19.3.1 was used for the simulation. PostgreSQL was run on version 16 of its official Docker image, with psycopg2-binary version 2.9.10 for Python. Umbra was run on version 26.02 of its official Docker image umbradb/umbra, alongside the same version of psycopg2.

The code to implement the AFL tensor contraction was developed in an InfiniData-Lab/Quantum repository on a scidb branch [3].

All experiments were run on a single machine equipped with AMD Ryzen 7 4800H (8 cores / 16 threads) running at 2.9 GHz, 16 GB of RAM, and NVMe SSD, under Ubuntu

26.04 LTS. SciDB, PostgreSQL, and Umbra were executed in Docker containers on this same host, so all engines were measured under identical hardware conditions.

For each circuit, we measured the end-to-end execution time of the simulation and its peak memory usage. Execution time was recorded as wall-clock time using `default_timer` from Python's `timeit` module, measured around the full sequence of tensor contractions, excluding circuit generation, for SciDB the execution time also includes the tensors upload, arrays creation, and loading gates time, other engines do not require these steps since they only execute a single query, these are necessary steps for SciDB implementation and therefore they are included in the execution time.

We measure the peak memory usage based on Docker's cgroup statistics. A background thread polls these statistics every 2 ms, and they are reported as the current overall memory usage of the container, subtracted by the memory allocated for the inactive files. These are the files that the container keeps in the memory, but it does not need them and can drop them with zero cost at any moment.

To verify that all of the engines correctly calculate the quantum states after the circuits, we validated the output states of the vectors returned from all engines in the QFT circuit up until 8 qubits. We compared these results to a vector generated with the same circuits by the `opt_einsum` Python library, which was also used to generate the optimal order of the tensor contractions.

For the autotuning of SciDB, we used the mlos-bench library from Python. We ran it over 60 iterations and used the tunable knobs that were explained in the Methodology. The mlos-bench folder in the repository contains all the necessary configs, scripts, and results.

5 Results

5.1 GHZ Preparation Circuit Simulation

We first simulated GHZ state preparation, whose output state has only two non-zero amplitudes and is therefore highly sparse across SciDB, PostgreSQL, and Umbra for qubit counts from 3 to 40. Figure 5 reports the median execution time and Figure 6 the median peak memory usage.

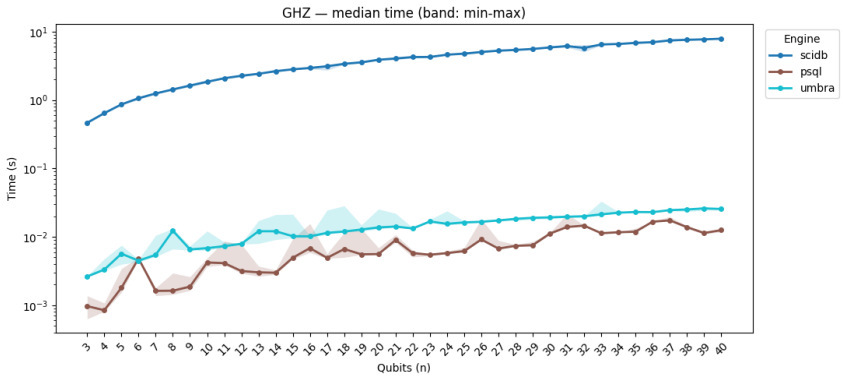

Figure 5: Execution time of SciDB alongside other RDBMSs over 5 runs of the GHZ preparation with increasing number of qubits

On this sparse workload, the two relational systems remained inexpensive across the entire range. Their execution times stayed in the millisecond regime even at 40 qubits, growing only mildly: PostgreSQL rose from about 1 ms at 3 qubits to about 12 ms at 40 qubits, and Umbra following the same near-linear trend, roughly 2–25 ms. SciDB, by contrast, was several orders of magnitude slower and grew approximately linearly with the number of qubits, from about 0.46 s at 3 qubits to about 7.9 s at 40 qubits. Across the whole range SciDB was roughly two to three orders of magnitude slower than the relational engines, and the ordering (PostgreSQL fastest, then Umbra, then SciDB) did not change.

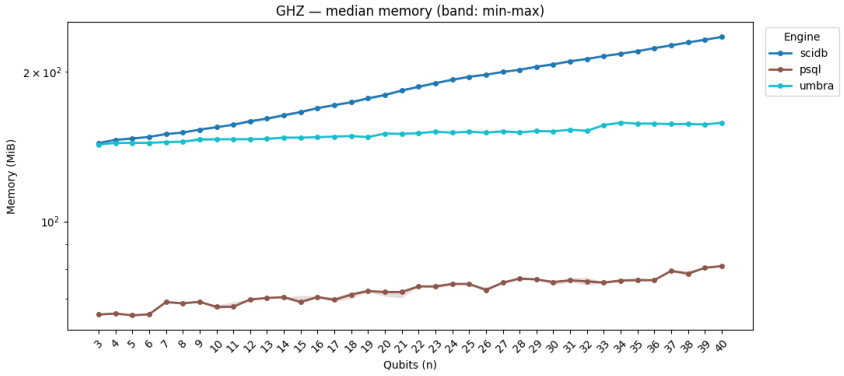

Figure 6: Memory usage of SciDB alongside other RDBMSs over 5 runs of the GHZ preparation with increasing number of qubits

Memory usage was likewise stable across the range, as expected for a state that does not grow with the qubit count. SciDB held the largest and nearly constant footprint (around 143–234 MiB), Umbra grew modestly from about 142 MiB to about 157 MiB, and PostgreSQL remained the lowest at roughly 64–81 MiB. For all engines the memory footprint was dominated by fixed server overhead rather than by the (constant-size) sparse state.

5.2 QFT Circuit Simulation

The second experiment simulated the Quantum Fourier Transform, whose intermediate states are dense, across the same three engines for qubit counts from 3 to 30. Figure 7 reports the median execution time and Figure 8 the median peak memory usage.

SciDB was the slowest system at every qubit count at which it ran. It already required about 0.75 s at 3 qubits, where the relational systems finished in well under a millisecond to a few milliseconds, then was stable holding the times of 6–9 s from 9 to 18 qubits before climbing steeply to 155 s at 24 qubits and 277 s at 25. In contrast to the sparse case, the relational systems no longer scaled gently: both eventually grew approximately exponentially, roughly doubling per added qubit at the upper end, consistent with the dense state vector doubling in size with each qubit. PostgreSQL grew while SciDB remained consistent over the 9 to 18 qubit range. Therefore, the gap narrowed from more than three orders of magnitude at 3 qubits to only about two to three times at 24. Umbra moved the other way, pulling one to two orders of magnitude clear of every other engine once past the low qubit counts.

The ordering among the relational engines reversed as the circuits grew. PostgreSQL was the fastest engine on small circuits but scaled the worst, while Umbra, the slowest on small

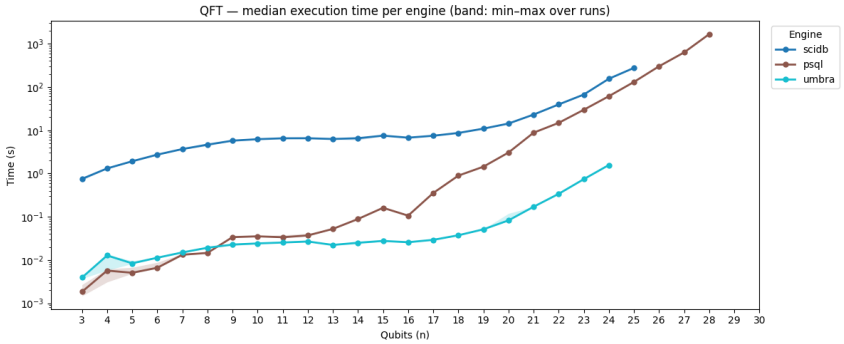

Figure 7: Execution time of SciDB alongside other RDBMSs over 5 runs of the QFT with increasing number of qubits

circuits, scaled the best, becoming the fastest relational engine at around 12–13 qubits. At 24 qubits, the largest count all three completed, the relational ordering was Umbra (1.6 s), then PostgreSQL (61 s), with SciDB at 155 s. SciDB stopped at 25 qubits. Beyond this, only PostgreSQL continued, reaching 28 qubits at about 1669 s.

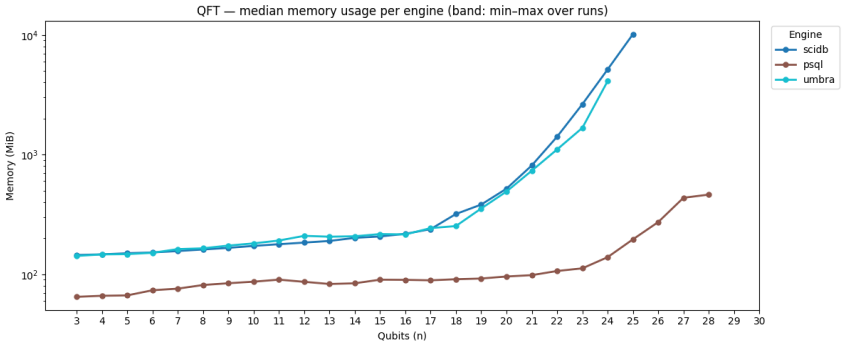

Figure 8: Memory usage of SciDB alongside other RDBMSs over 5 runs of the QFT with increasing number of qubits

The memory measurements, in Figure 8, reveal a sharper contrast and largely track why each engine stopped where it did. PostgreSQL kept its footprint bounded across the whole range, rising from about 65 MiB at 3 qubits to only about 112 MiB at 23 and 463 MiB at 28, far below the exponential growth of the dense state. SciDB and Umbra instead grew steeply, roughly doubling per qubit near the top: SciDB from about 145 MiB to 2635 MiB at 23 qubits, 5150 MiB at 24, and 10138 MiB at 25. Umbra from about 142 MiB to 1672 MiB at 23 and 4111 MiB at 24. This footprint sets the ceiling for Umbra, whereas both SciDB and PostgreSQL stopped because of the growing runtime.

To validate that all the engines correctly calculate the final state vector, we did a fidelity check against a reference output vector. In Table 1, we can see that all the engines calculated correctly the states up to 8 qubits.

Engine	Qubits					
	3	4	5	6	7	8
SciDB	1.00	1.00	1.00	1.00	1.00	1.00
PostgreSQL	1.00	1.00	1.00	1.00	1.00	1.00
Umbra	1.00	1.00	1.00	1.00	1.00	1.00

Table 1: State fidelity of each database backend against the `opt_einsum` reference for QFT circuits.

5.3 MLOS Autotuning

We run the MLOS tool over 60 iterations. The initial configuration for SciDB had dim_chunk_factor set to 1.0, flat_chunk_factor to 1.0. For the server-side parameters, we specified mem_array_threshold_mb equal to 7000, 4 merge_sort_nstreams, and 4 execution_threads.

First iteration with the default configuration obtained the score of 4.14. Then, through the next 35 iterations, the tool tried to explore as many configurations as possible. But after this phase, MLOS started to use the parameters that were similar to the initial config. The biggest difference in score was when the configuration did not contain 1.0 for both dim_chunk_factor and flat_chunk_factor. This is presented in the Figure 9

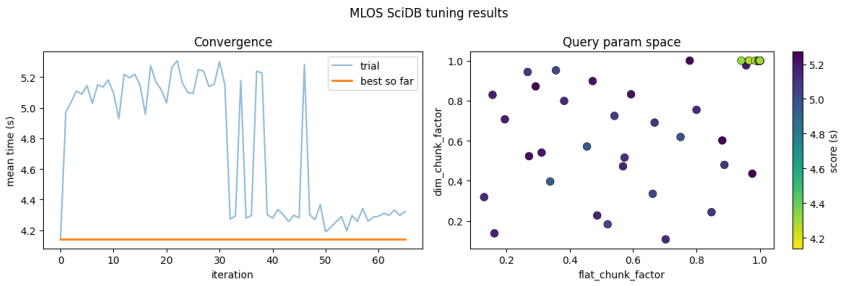

Figure 9: Autotuning scores over 60 iterations

The best configuration turned out to be the initial one, and all 10 of the best scores had configurations similar to the initial one. The best 10 results are shown in the Table 2

id	Tunable Knobs					score
	dim_chunk_factor	execution_threads	flat_chunk_factor	mem_array_threshold_mb	merge_sort_nstreams	
0	1.0	4	1.0	7000	4	4.1404
50	1.0	3	0.99	7003	4	4.1905
54	1.0	4	1.0	6094	4	4.1966
51	1.0	3	0.99	7001	4	4.2193
43	1.0	15	1.0	7047	5	4.2559
52	1.0	4	0.99	6858	4	4.2568
56	1.0	3	1.0	6767	4	4.2576
58	1.0	15	1.0	6998	4	4.2586
48	1.0	4	1.0	8243	4	4.2694
32	1.0	8	0.96	5930	12	4.2715

Table 2: Top 10 SciDB configurations found by MLOS/SMAC, ranked by score.

6 Responsible Research

This study did not raise any ethical concerns regarding humans, animals, or personal data, as it is just an empirical study comparing different database engines. All of the code is shared on a branch of a public repository `InfiniData-Lab/Quantum` [3], which contains all the scripts that were used for the experiments and their analysis, as well as the configuration files for the MLOS study.

7 Discussion

7.1 SciDB Overhead

The most general takeaway from the results of our experiments is that SciDB is the worst-performing database system compared to PostgreSQL and Umbra when we consider the execution time of the 2 used quantum circuits. But there are some interesting points for discussion. We deliberately chose to run the GHZ State Preparation circuit and the QFT algorithm.

GHZ is a circuit that has very sparse intermediate and final states. There are always 2 non-zero coefficients, no matter how many qubits we use. Therefore, it is only the number of contractions itself that has an impact on the execution time. And we can clearly see exactly that in Figure 5. All the curves for the execution time appear to grow linearly. This is also seen in the exact results, SciDB takes about 0.46s for 3 qubits and 7.9s for 40 qubits, the execution time grows by about 0.2s for each new gate. Already at the beginning, with 3 qubits, SciDB is heavily outperformed, and the offset persists for all the DBMSs throughout the whole range of qubit counts. This can only be explained by some fixed cost because at 3 qubits, there are practically no heavy computations. This is most likely caused by the mix of SHIM HTTP queries' round-trip time, query parsing and planning, and the tensors upload time to the file system, and the fact that the offset to other databases stays the same only supports this claim.

If we look at the peak memory usage plot in Figure 6, it is also consistent with the fact that the sparse arrays are implemented correctly in SciDB, because the memory usage does not grow significantly throughout all qubit counts. So the array always only saves the non-zero coefficients, and the growth of memory usage can be assigned to the increased number of gates, because if it were otherwise, the memory usage would be exponential to represent the exponentially growing output state.

The QFT algorithm is the exact opposite of the GHZ state preparation circuit. The intermediate states in this circuit are very dense, so the array size will grow exponentially even if we use the sparse representation for the tensors. Therefore, this can be looked at as the worst-case scenario for the DBMS quantum simulations.

When we look at the results for the execution time in Figure 7, we can once again see the fixed cost of the SHIM HTTP overhead at the lower qubit counts. SciDB takes about a second for the 3-qubit QFT, while other DBMSs do it in milliseconds. But the interesting part happens in the higher qubit counts, because, as mentioned before, the execution time should grow exponentially, so the fixed overhead gets irrelevant as most of the execution time is taken by the computation itself. That is exactly what happens in the case of SciDB. After about 18 qubits, all the DBMSs execution times are stabilised and grow exponentially. SciDB, even though at the beginning was orders of magnitude worse compared with other engines at the 24 qubits, is around 2 times slower than the second worst, PostgreSQL, and

the gap is slowly closing. None of the engines managed to compute the results for all of the QFT circuits up until 30 qubits. Umbra stopped at 24, and SciDB run until 25, while PostgreSQL managed to run all circuits until 28 qubits. This aligns with the peak memory usage data, since both SciDB and Umbra appear to use significantly more memory than PostgreSQL.

PostgreSQL stopped at 28 qubits simply because the timeout was set to 1800 seconds, and the median for 28 qubits was already at 1668 seconds. SciDB used around 10 GiB for 25 qubits, but the reason why it stopped was a timeout as well, and Umbra used around 4 GiB for 24 qubits, and it went out of memory while calculating the QFT for 25 qubits.

7.2 MLOS Autotuning

As we could see in the Table 2, the MLOS system autotuning tool did not improve the performance of SciDB on small circuits that remain in-core. The biggest takeaway is that if the values for dim_chunk_factor and flat_chunk_factor are below 1.0, the performance significantly degrades. That is because the chunk factor below 1.0 means that the array has to be split into more than 1 memory chunk. So more chunks mean more per-chunk handling costs, and since the whole arrays fit into 1 chunk anyway, there is no reason to split them. Overall, autotuning showed that the bottleneck of SciDB is not exposed in any of the 5 used knobs.

8 Conclusions and Future Work

As for simulating quantum circuits on the DBMSs, SciDB appears to perform the worst out of all the studied engines here. So the answer to the research question: Does SciDB, as a representative tensor-based DBMS, have an advantage over RDBMSs in simulating quantum circuits? is simply no. All of the other engines outperform SciDB in all studied cases, even though we can see that the gap in the execution time gets smaller for dense circuits as the number of qubits grows. We can also see the approximate fixed cost of SciDB in the execution time of the GHZ preparation circuit, since it is an example of a very sparse circuit, so the computation time is minimal. It takes around 0.5s for the 3-qubit case, which is almost entirely the overhead of SciDB.

The second research question: Does autotuning with MLOS improve the performance of SciDB for simulating quantum circuits? can also be concluded simply as no. MLOS did not manage to find a better configuration for SciDB than the default one. Therefore, the bottleneck of simulating quantum circuits on SciDB is not exposed by the knobs used to autotune the database configuration. However, this experiment revealed that using chunk sizes smaller than the arrays containing the quantum states significantly degrades SciDB's performance.

In the future, it would be beneficial to also test the out-of-core capabilities of SciDB, because none of the experiments conducted here required more memory than the available RAM. RDBMSs are known to have an advantage exactly in such cases, therefore SciDB could potentially behave similarly.

References

[1] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. Efficient and portable einstein summation in SQL. *Proc. ACM Manag. Data*, 1(2):121:1–121:19, 2023.

[2] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, Markus Weimer, and Yiwen Zhu. MLOS: an infrastructure for automated software performance engineering. In Sebastian Schelter, Steven Whang, and Julia Stoyanovich, editors, *Proceedings of the Fourth Workshop on Data Management for End-To-End Machine Learning, In conjunction with the 2020 ACM SIGMOD/PODS Conference, DEEM@SIGMOD 2020, Portland, OR, USA, June 14, 2020*, pages 3:1–3:5. ACM, 2020.

[3] Bruno Faliszewski. Simulation of Quantum Circuits with SciDB: Benchmarking and Autotuning. `https://github.com/InfiniData-Lab/Quantum/tree/scidb`, 2026. `scidb` branch of the InfiniData Lab Quantum repository, extending the simulator of Hai et al. (2025).

[4] Rihan Hai, Shih-Han Hung, Tim Coopmans, Tim Littau, and Floris Geerts. Quantum data management in the NISQ era. *Proc. VLDB Endow.*, 18(6):1720–1729, 2025.

[5] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit. *CoRR*, abs/2405.08810, 2024.

[6] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022.

[7] Tim Littau and Rihan Hai. Qymera: Simulating quantum circuits using RDBMS. In Volker Markl, Joseph M. Hellerstein, and Azza Abouzied, editors, *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, pages 179–182. ACM, 2025.

[8] Microsoft. MLOS: A Project to Enable Autotuning for Systems. `https://github.com/microsoft/MLOS`.

[9] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy G. Mattson. The tiledb array data storage manager. *Proc. VLDB Endow.*, 10(4):349–360, 2016.

[10] Paradigm4, Inc. shim: A Simple HTTP Service for SciDB. `https://github.com/Paradigm4/shim`. Documentation: `http://paradigm4.github.io/shim/help.html`.

[11] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[12] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In Judith Bayard Cushing, James C. French, and Shawn Bowers, editors, *Scientific and Statistical Database Management - 23rd International Conference, SSDBM*

2011, Portland, OR, USA, July 20-22, 2011. Proceedings, volume 6809 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2011.

[13] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. Scidb: A database management system for applications with complex analytics. *Comput. Sci. Eng.*, 15(3):54–62, 2013.

[14] TileDB, Inc. TileDB-MariaDB (MyTile): A MariaDB Storage Engine for TileDB Arrays. `https://github.com/TileDB-Inc/TileDB-MariaDB`. Deprecated; SQL support superseded by TileDB Tables.

[15] Immanuel Trummer. Towards out-of-core simulators for quantum computing. In Ibrahim Sabek, Immanuel Trummer, and Stefan Prestel, editors, *Workshop on Quantum Computing and Quantum-Inspired Technology for Data-Intensive Systems and Applications, Q-Data 2024, Santiago, Chile, June 9-15, 2024*. ACM, 2024.