Hybrid Quantum-Classical Optimization for Bushy Join Trees

Hanwen Liu
hanwen_liu@usc.com
University of Southern California
Los Angeles, California, USA

Abhishek Kumar
akumar17@usc.edu
University of Southern California
Los Angeles, California, USA

Federico Spedalieri
fspedali@isi.edu
University of Southern California
Los Angeles, California, USA

Ibrahim Sabek
sabek@usc.edu
University of Southern California
Los Angeles, California, USA

Abstract

Optimal join order sequence significantly impacts query execution performance. Because the search space grows exponentially with the number of relations involved in the queries, DP-based approaches can compute optimal plans only for queries with a small number of relations. Many heuristic methods trade off solution quality for computational efficiency. Quantum computing excels at solving combinatorial optimization problems such as join order optimization. However, existing quantum-related methods either focus solely on left-deep join trees—thus narrowing the problem scope—or support bushy join trees but are constrained by quantum hardware and cannot scale to large queries. Importantly, neither branch has been integrated into an actual database system. In this paper, we present a hybrid quantum–classical optimization approach for bushy join trees, integrated within a real database system (PostgreSQL). This approach expands the search space for join order sequences and overcomes previous limitations. Evaluation on the Join Order Benchmark shows that our method can reduce query execution time by up to 92.7% and achieve a 1.42x improvement in end-to-end latency.

CCS Concepts

• **Computer systems organization** → **Quantum computing**; • **Information systems** → **Query optimization**.

Keywords

Join Order, Query Optimization, Quantum Computing

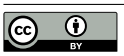
ACM Reference Format:
Hanwen Liu, Abhishek Kumar, Federico Spedalieri, and Ibrahim Sabek. 2025. Hybrid Quantum-Classical Optimization for Bushy Join Trees. In *Workshop on Quantum Computing and Quantum-Inspired Technology for Data-Intensive Systems and Applications (Q-Data '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3736393.3736695

1 Introduction

Join order (JO) optimization is a fundamental challenge in database systems. The objective is to determine the most efficient sequence for joining relations within a query, as the chosen sequence significantly impacts query execution performance. Since the search space grows exponentially with the number of relations, the JO problem is known to be NP-hard [3]. Dynamic programming (DP)–based approaches (e.g., [11]) can compute optimal plans by exhaustively enumerating all possible join sequences for queries with relatively few relations; however, they become impractical for larger queries. To overcome this limitation, various heuristic methods have been proposed that trade off solution quality for computational efficiency. Several works [5, 12, 16, 25] demonstrate that restricting the search space from bushy join trees to left-deep trees reduces the join order searching time, though at the cost of potentially missing more optimal join order sequences.

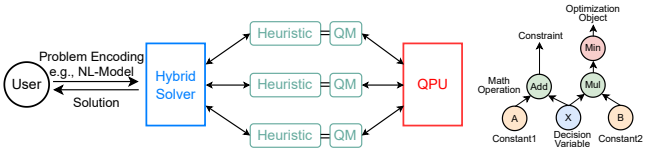
Quantum computing offers significant promise for solving combinatorial optimization problems such as join order optimization problem. A common approach formulates the JO problem as a Quadratic Unconstrained Binary Optimization (QUBO) problem [10] and solves it using quantum or quantum-inspired hardware (e.g., D-Wave [6]; Fujitsu Digital Annealer [2]). Recent quantum-related works [9, 21–24] formulate the JO problem by restricting the search to left-deep join trees. Another line of research [14, 15] uses a power-set based encoding for relations to support more general bushy join trees, thereby searching more possible join trees. However, their QUBO-based formulation will be directly sent to quantum annealers, which are constrained by the limited number of qubits and sparse connectivity (each qubit connects to only a few neighbors). As a result, their method supports join orders of at most five relations, limiting its applicability to larger-scale real-world queries [14]. Importantly, these existing quantum and quantum-inspired JO approaches remain at the simulation stage, as none have been integrated into real-world query optimizers, leaving their practical effectiveness as an open question.

In this paper, we present a hybrid quantum-classical optimization for bushy join trees *within a real database system* environment. Our approach is built on a state-of-the-art D-Wave hybrid solver, the Nonlinear-Program Solver (NL-Solver) [18]. The hybrid solver combines both classical heuristics and quantum capabilities, overcoming the limitations of using quantum annealers alone. We define a faithful bushy join tree representation with constraints within the NL-Model, enabling exploration of significantly larger queries than previous work. We then propose a corresponding join cost function as the optimization objective. We evaluate our approach on the Join Order Benchmark [8] in real database setup. Results show that query plans generated by our method reduce execution



Q-Data '25 , Berlin, Germany

ACM ISBN 979-8-4007-1944-8/2025/06
https://doi.org/10.1145/3736393.3736695



(a) The Workflow of D-Wave Hybrid Solvers. (b) NL-Model.

Figure 1: Hybrid Quantum-Classical Solvers and Model.

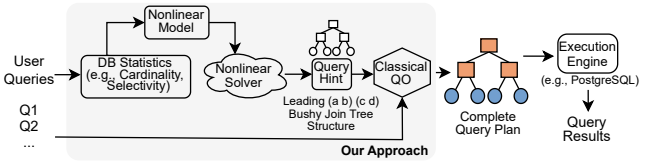


Figure 2: Overall Workflow of Our Approach.

time by up to 92.7% (for query q21) compared to PostgreSQL's default plans. Even when accounting for external quantum-service communication overhead, our approach achieves an end-to-end latency improvement of 1.42x for queries q62 and q63.

2 Preliminaries

Quantum annealing (section 2.1) is the theoretical foundation of the hybrid quantum-classical optimization (section 2.2). We apply them in hint-based query plan generation (section 2.3) to guide the join order optimization process in the database systems.

2.1 Quantum Annealing

Quantum Annealing [7] (QA) shares similarities with Simulated Annealing (SA) in that both methods explore the searching space by proposing random changes to the state configuration. In SA, a proposed change is accepted if it decreases the value of the cost function; if the cost increases, acceptance is decided by a certain probability. Moreover, QA leverages quantum tunneling to facilitate exploration, thereby reducing the likelihood of becoming trapped in local minima. QA is used to solve problems that can be naturally formulated as Quadratic Unconstrained Binary Optimization (QUBO) problems, which correspond to minimizing an energy function over binary variables representing qubit states [21]:

$$E(\vec{s}) = \sum_{ij} J_{ij}\, s_i\, s_j + \sum_i h_i\, s_i.$$

where s_i and s_j are binary variables representing the qubits' states, h_i denotes the local bias on qubit, and J_{ij} represents the interaction strengths between qubits. The Quantum Processing Unit (QPU) uses superconducting loops to represent individual qubits [1]. During quantum annealing, the initial superposition is transformed into a state that minimizes the system's energy, which, in turn, encodes the solution to the combinatorial optimization problem.

2.2 Hybrid Quantum-Classical Optimization

We utilize the hybrid quantum-classical optimization process provided by D-Wave's hybrid solver. Figure 1a provides an overview of its workflow. The solver accepts user-defined optimization problems. It then initiates multiple parallel threads, each incorporating both a classical heuristic module and a quantum module (QM). The classical heuristic module iteratively refines the problem's search space while the QM submits quantum queries from this refined space to the QPU. The feedback from the QPU iteratively guides the heuristic module toward more promising regions of the solution space. Finally, the best solutions found are returned to the user after a predefined time limit or a user-specified duration.

In this paper, we employ D-Wave's Nonlinear-Program Solver (NL-Solver) [18], which is a hybrid solver designed for general optimization problems. In this solver, the nonlinear model (NL-Model) is used to encode the optimization objective and its constraints through several key components, such as *decision variables*, *constants*, and *mathematical operations* [18]. This encoding can be represented with a directed acyclic graph (DAG) [18], where symbols correspond to nodes and the computation flow over these symbols is represented by edges. Figure 1b illustrates a simple example. We consider A, B, and X as numeric values, where X is a decision variable, and A and B are constants. In this example, we set $\min(X \times B)$ as the optimization objective and $A + X \geq 5$ as a constraint. Notably, each mathematical operation creates a new intermediate variable node. After the NL-Model is constructed as the DAG representation, the NL-Solver fixes the model and performs multiple optimization iterations. In each iteration, the NL-Solver assigns a value to the decision variable X and executes the computation flow defined by the model's DAG to get the value of the optimization object. Note that the constants A and B remain fixed during computation.

2.3 Hint-based Query Plan Generation

Hint-based plan generation is a commonly used technique in many well-known query optimizers (e.g., PostgreSQL [20], MySQL [13], Oracle [17]). For example, *pg_hint_plan* [19] tweaks PostgreSQL execution plans by embedding "hints" in SQL comments. A hint can serve as join order directives, represented by a clause beginning with `Leading` to specify the join order of relations. Inner parentheses denote grouping precedence in the join order sequence. For example, a hint for bushy join tree `Leading ((a (b c)) (d e))` indicates that $b \bowtie c$ is performed first, followed by $a \bowtie (b \bowtie c)$, then $d \bowtie e$, and finally the two intermediate results are joined. Notably, this hint fixes the order of operands in each join (e.g., $d \bowtie e$ rather than $e \bowtie d$). PostgreSQL then selects join operators (e.g., hash join, nested loop join) for each join. Once the optimizer accepts a hint, it is converted into a complete query plan and used for execution.

3 Workflow Overview

Figure 2 illustrates the overall workflow of our approach. Upon receiving a user query, we retrieve cardinality and selectivity statistics from the DBMS engine, which serve as constant symbols in the NL-Model. The expected join order sequence is defined as the decision variable. Similar to the workflow in Figure 1b, we enforce the bushy join tree constraints and define a join order cost function to be minimized (see Section 4). Once the model is constructed, it is submitted to the NL-Solver via the D-Wave Leap platform [4]. The solver returns a join order sequence expected to minimize cost. This sequence is then translated into a bushy join tree plan hint,

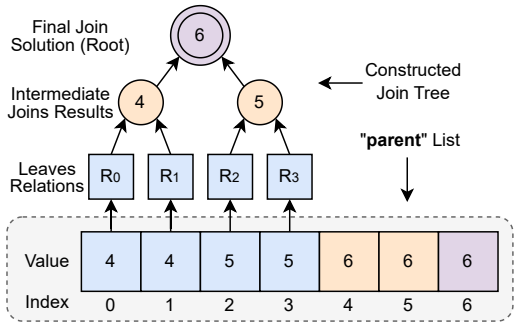


Figure 3: Bushy Join Tree Encoding and Construction.

guiding a classical query optimizer (e.g., PostgreSQL's optimizer) to generate a complete query plan. Finally, the execution engine executes this plan to produce and return the query results.

4 The Proposed Approach

We focus on identifying the optimal join orders among bushy join tree structures using a hybrid quantum-classical optimization approach. Our approach consists of two key components: (1) enforcing bushy join tree structure constraints (see Section 4.1), and (2) constructing the join cost optimization object within the NL-Model (see Section 4.2).

4.1 Bushy Join Tree

In this section, we propose an efficient representation for encoding arbitrary bushy join trees within the NL-Model, which allows us to define a minimal set of variables and constraints to enforce the structure. Our formulation starts by defining a directed connected graph $G = (N, E)$, where we model the join relationships as a directed graph whose vertices N comprise two types of nodes: *Leaf* and *Join node*. Suppose there are n relations in the given query. Leaf node N_i directly map to the input relation R_i, so that $N_0, \ldots, N_{n-1}$ denote the relations $\{R_0, R_1, \ldots, R_{n-1}\}$ in our query optimization formulation. Considering each join connects two nodes, n leaf nodes generate $n-1$ join nodes, resulting in a total $T = 2n-1$ of nodes. These join nodes are labeled N_n through N_{T-1}, with enumeration starting at n immediately after the leaf nodes. For each join node N_k, it represents the join of R_i and R_j, i.e. $R_k = R_i \bowtie R_j$. Note that for $n \le k < T$, R_k is not an original input relation but is introduced for notational consistency. The final join node N_{T-1} denotes the final solution of the query execution. Each pair of directed edge $(N_i \to N_k, N_j \to N_k) \in E$ composes the join $R_i \bowtie R_j$, where R_i appears on the left-hand side and R_j on the right-hand side.

To efficiently represent the graph G of any bushy join tree, we introduce a vector-based encoding, denoted by **parent** list. Figure 3 illustrates this encoding. Each node in the query plan, whether a leaf or a join node, has an entry in this list that indicates the index of its parent node. For example, the entries N_0 and N_1 both take the value 4, indicating that the parent of leaves R_0 and R_1 is the join node R_4. Similarly, the entries N_4 and N_5 both take the value 6, indicating that the parent of join nodes R_4 and R_5 is R_6. We impose the following constraints to enforce bushy join tree.

Constraint 1 - Acyclicity (Tree Structure): The query plan graph must not contain self-loops or cycles. In our directed representation, no node N_i may have an edge $(N_i \to N_i)$, whether N_i is a leaf or a join node, since a relation cannot join with itself. Moreover, there must be no reciprocal edges $(N_i \to N_j)$ and $(N_j \to N_i)$. As the join flow $R_i \to R_j$ and then $R_j \to R_i$ is meaningless in the query optimization context. We enforce this constraint by requiring:

$$\textbf{parent}[i] \ge i+1 \quad \text{for } 0 \le i < T-1, \quad \textbf{parent}[T-1] = T-1.$$

Since each node points to only one other node in our formulation, the above constraints also exclude cycles involving more intermediaries (e.g., a grandchild of a node cannot be its parent).

Constraint 2 - Binary-Tree Degrees: Each join node must have exactly two children. In other words, each join step combines exactly two relations R_i and R_j, and no node may have more than two inputs. Since the model construction phase does not involve actual values, we cannot directly identify which relations participate in each join. To address this, we define two auxiliary variables. The first is a list of constant symbols to facilitate comparisons: $\text{consts} = [i]_{i=0}^{T-1}$. The second is a two-dimensional **eq**ual-indicator matrix: $\text{eq}_{j,i} = \mathbb{I}(\text{parent}[i] = \text{consts}_j)$, where $\text{eq}_{j,i} \in \{0,1\}^T$. Here, $\text{eq}_{j,i} = 1$ if the ith entry of **parent** equals consts_j, indicating that node N_i is a child of N_j, and $\text{eq}_{j,i} = 0$ otherwise. We then enforce the following constraint on degree_j for each node N_j, which sums the indicator of its children across all other nodes except itself:

$$\text{degree}_j = \sum_{i=0}^{T-1} \text{eq}_{j,i} - \text{eq}_{j,j}, \quad \text{degree}_j = \begin{cases} 0, & j < n \quad \text{(leaves)}, \\ 2, & j \ge n \quad \text{(join nodes)}. \end{cases}$$

Constraint 3 - Bushy-Tree Structure: In strictly linear trees (e.g., left-deep trees), each join node $(R_i \bowtie R_j)$ is joined with a new relation R_m to form $(R_i \bowtie R_j) \bowtie R_m$. Instead, a bushy tree permits joining two join nodes. For instance, given $R_k = R_i \bowtie R_j$ and $R_l = R_m \bowtie R_n$, they may form $R_{k+l} = (R_i \bowtie R_j) \bowtie (R_m \bowtie R_n)$. To enable this flexibility, we constrain the range of possible values for each **parent** entry to $\{n, \ldots, T-1\}$. This ensures that each node, whether a leaf or a join node, can potentially be the child of join nodes $N_i, \ldots, N_{T-1}$. Combined with Constraint 1, this setup allows a node to join with any other valid node without forming loops.

Constraint 4 - Arbitrary Permutation of Leaf Assignments: Our formulation can already represent any bushy tree structure $\text{BT} = \{BT_1, \ldots, BT_p\}$. Furthermore, it requires permuting all leaf nodes $N_0, \ldots, N_{n-1}$ in each structure BT_i to cover all possible join order sequences of the input relations. We achieve this by allowing each entry inside **parent** to take values independently. In this way, all permutations are first allowed, and then the above-defined constraints are applied to construct a valid bushy join tree.

4.2 Join Cost Construction

Applying the previously defined constraints yields a valid **parent** list representing a bushy join tree. We then construct the join order cost function as the optimization objective. Since we remain in the model construction phase, the specific relation pair (R_i, R_j) for each join step is unknown. In this case, we introduce a matrix-based formulation to handle all possibilities within the model. Recall that the NL-Solver uses this join order cost to select the most promising value of decision variable (join order sequence) that yields the

Algorithm 1 Cost Construction for Bushy Join Tree in NL-Model

Input: Number of relations n, list of integer variables enforcing bushy-tree structure **parent** $\in \mathbb{R}^T$, number of nodes in **parent**: $T = 2n - 1$, penalty constant M, initial cardinalities $\mathbf{c} \in \mathbb{R}^n$, initial selectivity matrix $S \in \mathbb{R}^{n \times n}$, previously defined **consts** $\in \mathbb{R}^T$ and **eq** $\in \{0,1\}^{T \times T}$

Output: Join cost for given bushy join tree including penalty

1: **Initialization and preparation:**
2: $\tilde{S} \leftarrow S$, $\tilde{S}_{ii} \leftarrow 0 \; \forall i = 0, \ldots, n-1$ ▷ Zero-diagonal selectivities
3: $\mathcal{D} \leftarrow$ FetchNotConnectedRelationPairs$\{(R_i, R_j)\}$
4: **zero** $\leftarrow [0 \mid j = 0, \ldots, T-1]$
5: $\ell \leftarrow [1 \ll i \mid i = 0, \ldots, n-1] \cup [0 \mid i = n, \ldots, T-1] \in \mathbb{R}^T$ ▷ List of the compositions of the original relations for each node
6: $Cost \leftarrow []$ ▷ List of join costs
7: **for** $v = n, \ldots, T-1$ **do** ▷ For all the new generated nodes
8: **Select children nodes for v in matrix representation:**
9: $z \leftarrow 0_{T-|\mathbf{c}|}$; $\tilde{\mathbf{c}} \leftarrow$ hstack($\mathbf{c}$, z)
10: $y \leftarrow$ where(**eq**$[v]$, $\tilde{\mathbf{c}}$, **zero**) ▷ Children selection
11: **Compute cost and cardinality for new join node:**
12: $cost_v \leftarrow \sum_{0 \le i < j < v} y_i * \tilde{S}_{ij} * y_j$ ▷ Join cost
13: $penalty \leftarrow \sum_{\substack{(i,j) \in \mathcal{D} \\ i<v,\, j<v}} \mathbf{eq}[v][i] * \mathbf{eq}[v][j] * M$ ▷ Penalty
14: $cost_v \leftarrow cost_v + penalty$
15: $\mathbf{c} \leftarrow$ hstack($\mathbf{c}$, $cost_v$) ▷ Append new cardinality
16: $Cost$.append($cost_v$); ▷ Accumulate cost
17: **Update contained original relations for new join node:**
18: $mask_v \leftarrow (\ell \odot eq[v])$.sum() ▷ Merge compositions
19: $\ell \leftarrow$ put(ℓ, $[v]$, $mask_v$)
20: **Compute selectivities related to the new join node:**
21: $idxs, vals \leftarrow [\,], [\,]$
22: **for** $k = 0, \ldots, v-1$ **do**
23: $\sigma_{v,k} \leftarrow 1$ ▷ Selectivity between v and k
24: **for** $i = 0, \ldots, n-1$ **do**
25: **for** $j = 0, \ldots, n-1,\ i \neq j$ **do**
26: $\sigma_{v,k} \leftarrow \sigma_{v,k} \times$where$(eq[v][i] \wedge eq[k][j],\ \tilde{S}_{ij},\ 1)$
27: **end for**
28: **end for**
29: $overlap \leftarrow$ logical$(\sum_{i=0}^{T-1} eq[v][i] * eq[k][i])$
30: $\sigma_{v,k} \leftarrow$ where$(overleap,\ M,\ \sigma_{v,k})$ ▷ Penalty
31: $idxs$.extend$([v * T + k, k * T + v])$ ▷ Positions
32: $vals$.extend$([\sigma_{v,k}, \sigma_{v,k}])$ ▷ Update Values
33: **end for**
34: $\tilde{S} \leftarrow$ put$(\tilde{S},$ constant$(idxs),$ stack$(tuple(vals)))$
35: **end for**
36: **Return:** min $\sum_{v=n}^{T-1} Cost[v-n]$

minimum cost across iterations; any candidate value of the decision variable incurring a large cost or penalty will be less preferred.

Algorithm 1 presents the detailed procedure. It begins with initialization and preparation (lines 1–6). We design a list variable ℓ, where each value encodes which original relations are selected at the current node. For example, a four-bit binary vector represents $\{R_0, R_1, R_2, R_3\}$: `0100` indicates the leaf node R_2, and `1011` indicates the join node has already combined leaf nodes R_0, R_1 and R_3. The original selectivity matrix S is transformed by setting its diagonal entries to zero, yielding $\tilde{S}$. This transformation enables and simplifies the cost computation for a given join order in the nonlinear model, as the current nonlinear model supports only a limited programming grammar and set of operations.

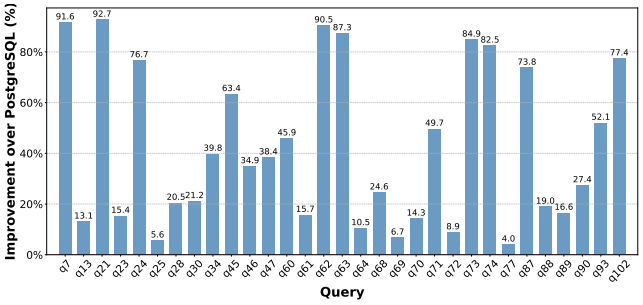


Figure 4: A Selection of Query Plans Generated by Our Approach Outperforms PostgreSQL's Default Plans.

For each new join node v, the algorithm first selects its children using the matrix representation (lines 8–10), and then computes the join cost and cardinality for v (lines 11–16). If v joins invalid relations (i.e., two original relations that are not connected), the cost incurs a penalty. Next, ℓ is updated for node v (lines 17–19). Afterward, the selectivities between all previous nodes k and the new join node v are computed (lines 20–34) following the bushy-tree scenario: the selectivity $\sigma(v,k) = \prod_{i \in \text{leaf}(v)} \prod_{j \in \text{leaf}(k)} s(i,j)$. Another penalty is applied when nodes v and k share the same original relations (denoted as `overlap`). Since the selectivity matrix is symmetric, we set $\tilde{S}_{vk} = \tilde{S}_{kv} = \sigma(v,k)$. The computed values are then updated in place. Finally, the sum of all generated join costs is set as the optimization objective to be minimized.

After enforcing the bushy join tree structure and constructing the join cost optimization target, we get a complete NL-Model. This model is then submitted to the NL-Solver to find solutions.

5 Experimental Evaluation

We first introduce our experimental setup in Section 5.1. We then evaluate our approach by addressing the following questions: (1) How effective is our approach in solving the hint-based query optimization problem, as reflected in query execution latency? (2) What is the end-to-end latency of our approach, and does it maintain an advantage when communicating with an external quantum service? (3) What is the distribution of query plan shapes generated by our approach, i.e., what is the ratio of left-deep trees and bushy trees?

5.1 Experimental Setup

We run all our experiments on a single server equipped with an Intel Core i9-9980XE CPU and 128 GB of RAM. We employ the hybrid_nonlinear_program_version1p solver as our NL-Solver on the D-Wave Leap quantum service platform. We set up PostgreSQL 16.4 with the corresponding *pg_hint_plan* version. We evaluate our approach using the Join Order Benchmark (JOB) [8], which comprises 113 analytical queries over a real-world dataset from the Internet Movie Database (IMDB). These queries involve complex joins and predicates, ranging from 3 to 16 joins, with an average of

8 joins per query. This complexity makes JOB a suitable and widely adopted benchmark for testing join-order optimization.

5.2 Evaluation Results

Query Plan Quality. We evaluate query plan quality by measuring the actual execution time (Exec Time) of JOB queries on PostgreSQL. Figure 4 compares the latencies of the plans generated by our method and by PostgreSQL. Among 113 queries, our approach yields speedups on 31 queries, with a maximum latency improvement of 92.7% and an average improvement of 42.09%.

For the other queries whose generated hints did not yield execution time improvements, our approach exhibited limited performance degradation, with execution times averaging 11.77% slower than PostgreSQL. Notably, 11 queries achieved near parity with PostgreSQL (within a 2% execution-time difference). These results complete the performance analysis of our approach.

End-to-End (E2E) Latency. We measure E2E latency to compare the complete query execution pipeline. For PostgreSQL, E2E latency comprises planning time and execution time. In our approach, it includes PostgreSQL planning time (with hints), PostgreSQL execution time, and NL-Solver time. Table 1 presents a detailed breakdown of these components for four JOB queries. Although the NL-Solver contains additional overhead, primarily due to cloud-service communication, our approach still achieves overall E2E latency improvements. These gains result from our method generating better join order sequences compared to PostgreSQL's default ones, yielding execution time speedups ranging up to 13.15x.

Join Tree Shape. We analyzed the distribution of join-tree shapes produced by our approach across the benchmark queries. For queries with fewer joins or a chain-type join structure, left-deep trees predominate (80%). This trend is exemplified by queries q21, q62, and q63, which yielded execution latency improvements of 92.70%, 90.46%, and 87.35%, respectively. Simultaneously, our method employs bushy trees for the remaining 20% of queries, particularly those with complex join relations that benefit from parallel join execution. Queries such as q102 (17 relations) and q93 (12 relations) achieved improvements of 76.88% and 52.06%, respectively.

6 Conclusion and Future Work

In this paper, we present a hybrid quantum–classical optimization approach for bushy join trees. Our evaluation, based on a real-world workload and an actual database setup, demonstrates its effectiveness from improving query execution latency and end-to-end latency. For future work, we plan to extend this approach to handle even larger queries and to evaluate it on more complex workloads. Moreover, we intend to progressively expand the decision space of our hybrid quantum–classical approach within the query optimization context, to support finer-grained hint generation, including join-operator selection (e.g., hash join, nested-loop join) and table-scan strategies (e.g., index scan, sequential scan).

Table 1: End-To-End Latency Breakdown and Comparison Between Our Approach and PostgreSQL.

	Time (ms)	q21	q60	q62	q63
PG	Planning	1.00	3.14	3.17	1.16
	Execution	3581.40	10951.31	4680.26	4846.42
Ours	Planning with hints	2.24	8.31	9.03	9.02
	Execution	272.35	6010.00	429.58	585.60
	NL-Solver	2530.22	2748.32	2854.31	2816.42
Gain	**Exec Time ↑**	**13.15x**	**1.82x**	**10.89x**	**8.28x**
	E2E Latency ↑	**1.28x**	**1.25x**	**1.42x**	**1.42x**

References

[1] MHS Amin. 2005. Flux qubit in charge-phase regime. *Physical Review B—Condensed Matter and Materials Physics* 71, 2 (2005), 024504.

[2] Maliheh Aramon, Gili Rosenberg, et al. 2019. Physics-inspired optimization for quadratic unconstrained problems using a digital annealer. *Frontiers in Physics* 7 (2019), 48.

[3] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products. In *International Conference on Database Theory*. Springer, 54–67.

[4] D-Wave Leap's Hybrid Solvers. 2024. https://docs.dwavesys.com/docs/latest/doc_leap_hybrid.html.

[5] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing N-relational joins. *ACM Trans. Database Syst.* 9, 3 (Sept. 1984), 482–502. doi:10.1145/1270.1498

[6] M. W. Johnson, M. H. S. Amin, et al. 2011. Quantum annealing with manufactured spins. *Nature* 473, 7346 (2011), 194–198. doi:10.1038/nature10012

[7] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum annealing in the transverse Ising model. *Phys. Rev. E* 58 (Nov 1998), 5355–5363. Issue 5. doi:10.1103/PhysRevE.58.5355

[8] Viktor Leis, Andrey Gubichev, et al. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594

[9] Hanwen Liu, Pranshi Saxena, Federico Spedalieri, and Ibrahim Sabek. 2025. Optimizing Join Orders via Constrained Quadratic Models (Abstract). In *Proc. 1st Workshop Quantum Data and Machine Learning* (HongKong, China) *(QDML '25)*.

[10] Andrew Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics* 2 (2014). doi:10.3389/fphy.2014.00005

[11] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*. 930–941.

[12] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 539–552.

[13] MySQL Documentation Team. 2025. *MySQL 8.4 Reference Manual* (8.4 ed.). https://dev.mysql.com/doc/en/

[14] Nitin Nayak, Jan Rehfeld, et al. 2023. Constructing Optimal Bushy Join Trees by Solving QUBO Problems on Quantum Hardware and Simulators. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments (BiDEDE '23)*. Article 7. doi:10.1145/3579142.3594298

[15] Nitin Nayak, Tobias Winker, et al. 2024. Quantum Join Ordering by Splitting the Search Space of QUBO Problems. *Datenbank-Spektrum* 24, 1 (2024), 21–32.

[16] Thomas Neumann and Bernhard Radke. 2018. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*. 677–692.

[17] Oracle Corporation. 2025. *Oracle Database Reference, 19c* (19c ed.). https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/index.html

[18] Eneko Osaba and Pablo Miranda-Rodriguez. 2024. D-Wave's Nonlinear-Program Hybrid Solver: Description and Performance Analysis. arXiv:2410.07980 [cs.ET]

[19] OSSC-DB. [n. d.]. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.

[20] PostgreSQL Global Development Group. [n. d.]. https://www.postgresql.org/.

[21] Pranshi Saxena, Ibrahim Sabek, and Federico Spedalieri. 2024. Constrained Quadratic Model for Optimizing Join Orders. In *Proc. 1st Workshop Quantum Comput. Quantum-Inspired Technol. Data-Intensive Syst. Appl.* (Santiago, AA, Chile) *(Q-Data '24)*. 38–44. doi:10.1145/3665225.3665447

[22] Manuel Schönberger, Stefanie Scherzinger, et al. 2023. Ready to Leap (by Co-Design)? Join Order Optimisation on Quantum Hardware. *Proc. ACM Manag. Data* 1, 1, Article 92 (may 2023), 27 pages. doi:10.1145/3588946

[23] Manuel Schönberger, Immanuel Trummer, and Wolfgang Mauerer. 2023. Quantum-inspired digital annealing for join ordering. *Proceedings of the VLDB Endowment* 17, 3 (2023), 511–524.

[24] Manuel Schönberger, Immanuel Trummer, and Wolfgang Mauerer. 2023. Quantum Optimisation of General Join Trees.. In *VLDB Workshops*.

[25] P. Griffiths Selinger, M. M. Astrahan, et al. 1979. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD '79)*. 23–34. doi:10.1145/582095.582099