Quantum Optimisation of General Join Trees

Manuel Schönberger[1,*], Immanuel Trummer[2] and Wolfgang Mauerer[1,3]

[1]Technical University of Applied Sciences Regensburg, Regensburg, Germany
[2]Cornell University, Ithaca, NY, USA
[3]Siemens AG, Corporate Research, Munich, Germany

Abstract

Recent advances in the manufacture of quantum computers attract much attention over a wide range of fields, as early-stage quantum processing units (QPU) have become accessible. While contemporary quantum systems are very limited, mature QPUs are speculated to excel at optimisation problems. Their use is hence very attractive for the database domain, which features a broad range of complex optimisation problems with large solution spaces. Yet, the use of QPUs on database problems remains largely unexplored, prompting further evaluation of the aptitude of quantum systems in the database domain, starting with its most fundamental problems. In this paper, we address the long-standing join ordering problem, one of the most extensively researched database problems.

Rather than solving arbitrary problems, the use of QPUs requires specific mathematical problem encodings. Such a formulation was recently proposed for the join ordering problem, allowing first small-scale queries to be optimised on quantum hardware. However, the existing problem encoding is a faithful transformation of a mixed integer linear programming (MILP) formulation for JO, and hence inherits all limitations of the MILP method. Most strikingly, the existing encoding only considers a solution space with left-deep join trees, which tend to yield larger costs than general, bushy join trees.

In this paper, we hence propose a novel QUBO encoding for the join ordering problem. Rather than transforming existing formulations, we present a native encoding tailored to quantum systems, which allows quantum optimisation of general, bushy join trees. Thereby, we exhaust the full potential of QPUs for join order optimisation.

1. Introduction

Recent advances in the development of quantum computing hardware have sparked increased interest in this novel architecture from a plethora of research fields. While the prototypical nature of contemporary quantum hardware does not yet allow for practical utility, quantum processing units (QPU) are speculated to excel at optimisation problems, which govern a large portion of ongoing database research. However, despite the large potential of QPUs to accelerate computation-intense industrial processes, the potential of using quantum systems on database-related issues remains largely unexplored.

Still, recent database optimisation research has pioneered in exploring first means to harvest this potential: Trummer and Koch [1] investigated the use of quantum annealing on the multi query optimisation (MQO) problem. Groppe and Groppe [2], and Bittner and Groppe [3, 4], analysed QPU use for database (DB) transaction scheduling. Finally, Schönberger *et al.* [5] presented a method to solve join ordering (JO) problems on contemporary QPUs, by applying faithful transformation of the mixed integer linear programming (MILP) encoding for JO proposed by Trummer and Koch [6], into a quadratic unconstrained binary optimisation (QUBO) formulation. This encoding can be interpreted by QPUs, which enabled the JO optimisation of first small-scale queries on real QPUs.

However, by applying a transformation of the original MILP encoding into QUBO, their JO-QUBO inherits the limitations of the MILP encoding. Most strikingly, their formulation only accounts for left-deep join trees. While search space restriction to left-deep trees has historically been applied by JO methods, and offers the benefit of substantially lowering the exploration complexity, its downside quickly becomes apparent when considering empirical data comparing solution quality against approaches without this restriction. For instance, Neumann and Radke [7] conducted a comparison of various JO algorithms, including methods for general bushy trees and approaches restricted to left-deep ones. The former consistently produced solutions superior in quality to left-deep tree solutions. In extreme cases, the left-deep solutions were up to a factor of 58 worse than bushy solutions, illustrating the impact of restricting the search space, which limits the practical utility of the existing JO-QPU method.

Still, the prospect of using QPUs for JO optimisation remains attractive. Therefore, in this paper, we address the drawbacks of the existing JO-QUBO formulation: Rather than transforming an existing encoding into QUBO, thereby inheriting its limitations, we propose a novel QUBO encoding for optimising general bushy

Woodstock'22: Symposium on the irreproducible science, June 07–11, 2022, Woodstock, NY
*Corresponding author.
manuel.schoenberger@othr.de (M. Schönberger); itrummer@cornell.edu (I. Trummer); wolfgang.mauerer@othr.de (W. Mauerer)
0000-0002-6939-7582 (M. Schönberger); 0000-0002-9765-8313 (W. Mauerer)

trees, while moreover retaining the ability to identify cross product solutions. In contrast to many competing JO methods, which either do not consider cross products or restrict the join tree shape, our new encoding enables QPUs to explore the complete, unrestricted search space of the JO problem.

Contributions. In detail, our contributions are as follows:

1. Rather than transforming an existing formulation and inheriting its limitations, we propose a novel, native encoding of join ordering into QUBO.
2. We enable QPUs to explore the most extensive class of join ordering problems, allowing them to optimise general, bushy join trees while also enabling the use of cross products. Thereby, we exhaust the full potential of quantum hardware for join order optimisation.
3. We identify architectural bottlenecks of contemporary quantum systems, quantify their impact on join order optimisation, and discuss means to address them by tailoring quantum systems to problem requirements.

The remainder of this paper is structured as follows: We provide fundamentals on quantum computing, including the required QUBO formalism, in Sec. 2. We explain our considered join ordering model in Sec. 3. We discuss our novel encoding for bushy join trees in Sec. 4. Finally, we discuss related work in Sec. 5 and conclude in Sec. 6.

2. Quantum Fundamentals

Unlike conventional CPUs, QPUs cannot be used for executing arbitrary code to run any algorithm. Instead, distinct programming paradigms are required for quantum computation. The two prevailing paradigms required for contemporary QPUs consist of *gate-based quantum computation*, as implemented, *e.g.*, by IBM-Q systems [8], and *quantum annealing*, made accessible by D-Wave [9]. The latter exclusively solves optimisation problems, and thereby inherently meets our requirements of using QPUs for query optimisation, whereas gate-based QPUs allow the execution of gate-based quantum optimisation algorithms, most prominently the *quantum approximate optimisation algorithm* (QAOA) [10]. Fortunately, both approaches require the same problem encoding scheme.

2.1. QUBO Formalism

Specifically, they require problems formulated as *quadratic unconstrained binary optimisation* (QUBO) problems [11, 12], which (1) only allow *quadratic* interactions between variables, (2) allow *no explicit constraints*, (3) limit variables to a *binary domain* and (4) encode *optimisation* problems. Physically, we may consider a QUBO encoding an *energy formula*, where the minimum energy corresponds to an optimal solution to the problem. Mathematically, they are given by the multivariate polynomial

$$f(\vec{x}) = \sum_i c_{ii} x_i + \sum_{i \neq j} c_{ij} x_i x_j, \tag{1}$$

where $x_i \in \{0, 1\}$ are variables, and $c_{ij} \in \mathbb{R}$ coefficients (with $c_{ij} = c_{ji}$).

The biggest hurdle in solving optimisation problems on QPUs consists in determining problem formulations conforming to the QUBO requirements. Firstly, we have to identify validity constraints that must hold for every valid solution to our problem, and encode these constraints *implicitly*, by specifying terms akin to Eqn. 1 that evaluate to positive *energy penalties* for any constraint violation, thereby *penalising invalid variable configurations*. Secondly, further QUBO terms are set to add energy in accordance to the *quality*, or *costs*, of a solution. If built correctly, minimising the overall energy formula will produce a variable configuration corresponding to a solution that is both, valid and optimal.

2.2. Useful Patterns and Operators

To provide the reader with an understanding of the QUBO encoding process, we illustrate the encoding principles based on three recurring encoding patterns, or operators, that will prove to be very useful in our quest of formulating a JO encoding for bushy join trees.

2.2.1. N-Hot Encoding

The first recurring scheme concerns *groups of semantically matching variables*, out of which only a *limited amount n* may be active, or *hot*. As such, we refer to this pattern as *n-hot encoding*. Typically, this encoding is applied for $n = 1$. Hence, it is usually called *one-hot encoding*. Expressed as QUBO, its most basic form is given by

$$H_{1hot} = \left(n - \sum_{x \in X} x\right)^2,$$

where X denotes the set of binary variables. Expressing a quadratic term, H_{1hot} is clearly minimised iff the inner term evaluates to 0, which requires $n = \sum_{x \in X} x$. Therefore, minimising H_{1hot} produces a variable configuration where exactly n out of all variables within X are active. More complex versions of this encoding may substitute a linear term of variables and coefficients for the constant n.

This scheme is useful for ensuring a valid assignment of variables expressing mutually exclusive properties. For instance, we later apply it for enforcing that a relation r is initially joined by only one out of all possible joins.

2.2.2. Implication Operator

Further recurring encoding patterns involve logical operators. One of the most relevant operators is thereby given by the *implication operator*. Given a binary variable a and a set of binary variables B, we express $\forall b \in B : a \implies b$ in QUBO as

$$H_{impl} = a\left(|B| - \sum_{b \in B} b\right),$$

where $|B|$ denotes the size of B. The ground state energy of H_{impl} is given by 0, since clearly, neither a nor $|B| - \sum_{b \in B} b$ can assume negative values. Hence, to minimise H_{impl}, we require either $a = 0$ or $|B| - \sum_{b \in B} b = 0$. Since $|B| - \sum_{b \in B} b = 0$ requires $b = 1 \forall b \in B$, minimising H_{impl} activates all variables in B if $a = 1$.

In case of JO, the implication operator will prove useful to us, *e.g.*, for enforcing a relation r, initially joined by join j, to be considered an operand for all joins including and succeeding j.

2.2.3. And-Operator

Finally, given three binary variables a, b and c, we can moreover express the *logical and-operator* $a \wedge b = c$ in QUBO, as described in the D-Wave problem reformulation handbook [13]:

$$H_{and} = ab - 2ac - 2bc + 3c.$$

If $ab = 1$ and $c = 0$, only the term ab remains, inducing an energy penalty of 1, whereas $ab = 1$ in conjunction with $c = 1$ causes H_{and} to evaluate to 0. Likewise, $ab = 0$ and $c = 0$ lead to energy 0, whereas for $ab = 0$ and $c = 1$, only the term $3c$ remains, inducing a penalty of 3. Thus, minimising the energy for H_{and} indeed produces variable configurations in accordance to the and-operator.

We observe that we can use the and-operator to store the result $a \wedge b$ into a single variable c, which is useful to maintain a degree of 2 for our polynomial, as required by QUBO.

3. Join Ordering Model

Having laid out the fundamentals of encoding problems as QUBO, we next describe our encoding targets, *i.e.*, the various elements constituting a JO problem.

3.1. Query Graph

The input to a JO problem is given by a query graph $G = (V, E)$, where the nodes $v_1, ..., v_R \in V$ represent the R relations $r_1, ..., r_R$ with cardinalities $n_1, ..., n_R$ to be joined. Further, an edge $e_{ij} \in E$ corresponds to the join predicate p_{ij} with selectivity $0 < f_{ij} \leq 1$.

Some JO algorithms require a strict adherence to the query graph, only allowing joins between relations connected by a predicate. In contrast, other approaches consider joins between *any* pair of relations, including such with no predicates. Such operations are typically referred to as *cross products*. Alternatively, we may consider a cross product a join with selectivity 1, and add the missing edge in G accordingly.

Clearly, including cross products can drastically enhance the set of allowed operations (in particular for a sparse query graph), which motivates their exclusion by many JO approaches, to limit the size of the search space. In contrast, our method does not apply such restrictions, allowing it to benefit from the use of cross products, which can indeed be required by an optimal solution. We moreover consider *any query graph*, without any restrictions on its shape (whereas some other approaches require, *e.g.*, the graph to be acyclic [14]).

3.2. Join Tree

In contrast to a query graph, which corresponds to the input to the JO problem, a *join tree* represents a JO *solution*. Its leaf nodes thereby represent the base relations to be joined, whereas its intermediate nodes correspond to join operations. Edges are directed towards the root of the tree (*i.e.*, the final join). As each join requires two operands, each join node has two predecessors, corresponding to either a) a base relation, or b) another join, whose result is to be further joined. The join result further serves as an operand for another join, as expressed by an outgoing edge connecting to its successor. The only exception to this rule is given by the final join, which is no operand for any further join.

While these requirements must hold for any join tree, some JO methods further restrict the shape of join tree. Much like the exclusion of cross products, such restrictions are motivated by the greater efficiency of exploring a reduced search space. Most notably, some approaches only consider *left-deep* join trees, which require each join to take at least one base relation as an operand. Hence, individually joining two pairs of relations is not possible, since joining their results requires a join operating on the results of two preceding joins. Instead, valid left-deep join orders must correspond to a *permutation* of relations.

The restriction to left-deep trees was applied by the MILP method by Trummer and Koch [6], and hence moreover by the existing JO-QUBO proposed by Schönberger *et al.* [5], which faithfully transforms the MILP formulation into QUBO. However, the negative impact of this restriction on solution quality can be quite drastic, as shown, for instance, by the empirical analysis conducted by Neumann and Radke [7]. Therefore, our novel QUBO encoding considers general, or *bushy* join trees, which are not limited by any further restrictions.

3.3. Cost Function

Finally, a cost function evaluates each join tree, by assigning it a cost value. For our method, we consider the classic cost function C_{out}, which sums over the intermediate cardinalities of all joins of a join tree [15]. For a pair of relations, their cardinalities and predicate selectivity, it is given by $C_{out}(n_i, n_j) := n_i n_j f_{ij}$. Hence, we require product operations to express this cost function, which poses an issue for determining a JO-QUBO encoding limited to quadratic terms. In the next section, we show how to mitigate this issue, by applying the same strategy as used by the MILP [6] and existing QUBO encoding [5].

4. QUBO Encoding

To solve JO with optimisation methods such as MILP and QUBO, we have to encode variables and constraints in such a way that we receive valid join trees as solutions. However, unlike many other optimisation methods such integer linear programming, solution validity cannot be enforced via explicit constraints for QUBO encodings. Instead, as demonstrated on some common operators in Sec. 2, the encoding needs to ensure that a variable configuration minimising the QUBO formula inherently corresponds to a valid solution.

To guide the reader, we first provide an overview of our encoding methodology and preliminary considerations in Sec. 4.1. Next, in Sec. 4.2, we show, in detail, how to encode valid bushy trees in QUBO. Finally, we describe our cost function encoding that assigns a cost value to a join tree in Sec. 4.3.

4.1. Overview

In case of left-deep join trees, the structure of the tree is identical for all possible solutions. This allows the use of a priori knowledge about the tree structure for the QUBO formulation. For instance, we may establish that join 0 is a direct predecessor to join 1, and can use this information to efficiently encode constraints, as done by the existing MILP and QUBO formulations for left-deep join trees [6, 5]. However, in case of bushy trees, the relationship between joins is, in general, unclear: We cannot make any a priori assertions about whether a join i succeeds a join j, except for the final join, which is a successor to all other joins.

To circumvent this issue, it is possible to operate on a large tree structure that encompasses all possible bushy trees, providing us with a priori knowledge about the relationship between two joins, similarly to the left-deep scenario. However, this requires an exponentially growing number of tree nodes, making such an approach infeasible due to scaling limitations. Therefore, we apply a different formulation strategy for bushy trees. Instead of operating on a tree provided as an input, our encoding for bushy trees generates the join tree itself.

Thereby, we exploit global relationships that hold for every possible join tree. For instance, each join takes exactly two operands as inputs (which may either represent a base relation, or the result of a prior join), and produces precisely one result. Hence, each join has two incoming edges, and either one (in case of intermediary joins) or zero (in case of the final join) outgoing edges. As we later show in detail, we can translate these relationships into penalty terms, to enforce valid assignments for variables representing the edges in the join tree.

In stark contrast to left-deep trees, enforcing such relationships is not straightforward for bushy trees, since we lack a priori information about the relationship between a join pair. Hence, we require more sophisticated methods to enforce these conditions for bushy trees. These methods, in return, require the introduction of ancillary variables. For instance, by determining the depth all relations and joins in the join tree, and saving this information using ancillary variables, it is possible to prevent the occurrence of cycles in the join tree. As we will show, it is possible to enforce a valid configuration of variables by introducing a number of ancillary variables that is cubic in the number of relations.

Outline. Firstly, we show how to enforce an assignment of variables conforming to valid bushy trees. Hereby, in addition to the actual tree variables, we introduce variables to capture the depth of tree nodes. Thereby, we prevent the occurrence of cycles. Secondly, to evaluate the quality of a join tree, we describe variables and penalty terms to encode the logarithmic costs of a tree. Finally, we approximate the actual join order costs based on the logarithmic values.

Our encoding requires a plethora of binary variables with individual semantics. To guide the reader, Table 1 depicts an overview on the semantics of each variable type, alongside further information such as the required amount of variables.

4.2. Encoding Valid Bushy Trees

We begin by introducing variables representing the join tree. Let the binary variable rlj_{rj} (*Relation is Leaf for Join*), introduced for each relation r and join j, indicate whether j uses the input relation r as an operand. Further, let the binary variable jds_{ij} (*Join is Direct Successor*), added for every join pair (i, j), denote whether join j is a direct successor to join i.

Next, we enforce a valid assignment of the newly introduced variables, such that they represent one out of all possible bushy trees. Thereby, we exploit global constraints that must hold for every join tree (providing *semantics* in brackets), namely:

Table 1
Overview of all variables, their semantics and required amounts for queries joining R relations with J joins, using P predicates and T threshold values, at discretisation precision ω. Finally, $c_{j_{\max}}$ denotes the maximum logarithmic cardinality for join j.

Var^s	Variable Semantics	Variable Amount	Ancillary Amount
rlj_{rj}	Is *relation* r *leaf* for *join* j?	RJ	/
jds_{ij}	Is *join* j a *direct successor* to join i?	J^2	/
rd_{rd}	Does *relation* r have *depth* d?	RJ	RJ^2
jd_{jd}	Does *join* j have *depth* d?	J^2	J^3
roj_{rj}	Is *relation* r an *operand* for *join* j?	RJ	RJ^2
paj_{pj}	Is *predicate* p *applicable* for *join* j?	PJ	/
trj_{tj}	Is *threshold* t *reached* by *join* j?	$T(J-1)$	$T\sum_{j=1}^{J-1}\left(\left\lfloor \log_2\left(\frac{c_{j_{\max}}}{\omega}\right)\right\rfloor + 1\right)$

(a) Each node has two incoming edges, excluding leaf nodes, which have zero incoming edges (*each join has two operands*).

(b) Each node has one outgoing edge, excluding the root node, which lacks outgoing edges (*each intermediate join produces an intermediate result used by a subsequent join, and each relation is a leaf assigned to one join*).

(c) No cycles may appear within a tree (*each join is only applied once*).

In the following, we individually translate these constraints into QUBO terms, whose minimisation will hence ensure the construction of a valid join tree.

(a) Incoming Edges. Each join requires two operands, which may either be base relations or intermediate results produced by another join. As such, we enforce that each join node has exactly two incoming edges, using a two-hot encoding. In addition, the final join, which we indicate by index f, must receive the intermediate results of all preceding joins, and therefore have at least one join as a predecessor[1]. As such, we additionally enforce that at most one of the incoming edges connecting to the final join comes from a base relation:

$$H_a = \sum_{i=1}^{J}\left(2 - \sum_{j=1,j\neq i}^{J} jds_{ji} - \sum_{r=1}^{R} rlj_{ri}\right)^2 + \left(s - \sum_{r=1}^{R} rlj_{rf}\right)^2.$$

The latter inequality requires the introduction of an ancillary binary variable s, allowing a variable configuration such that the term evaluates to 0 if no more than one variable rlj_{rf} is active.

Example 4.1. *Let us consider a query joining three relations A, B and C with two joins i and j. Further, let $jds_{ij} = 1$ indicate that i is succeeded by j. To produce a valid tree, i must join A and B, or else any other pair of base relations, whereas j must join the emerging intermediate result and the remaining base relation C. Thus, successful minimisation may yield the join order $(A \bowtie B) \bowtie C$, expressed as $rlj_{Ai} = rlj_{Bi} = rlj_{Cj} = 1$, avoiding a penalty by H_a as $(2 - rlj_{Ai} - rlj_{Bi})^2 = 0$ for join i and $(2 - jds_{ij} - rlj_{Cj})^2 = 0$ for join j. The same applies mutatis mutandis for any other join order permutation.*

In contrast, consider the case where either join receives more or less than two operands. For instance, let $rlj_{Bj} = 1$ rather than $rlj_{Bi} = 1$, producing the energy penalty $(2 - rlj_{Ai})^2 = 1$ for join i and $(2 - jds_{ij} - rlj_{Bj} - rlj_{Cj})^2 = 1$ for join j.

[1]Of course, this only holds for queries joining at least three relations, using at least two joins, which we consider the minimum size for meaningful JO optimisation.

(b) Outgoing Edges. Similar considerations as for incoming edges apply to outgoing ones: Each join produces one result, which may be the final result in case of the final join f, or else serve as an input to another join. Therefore, each intermediate join node has one outgoing edge, which we enforce by a one-hot encoding. In contrast, the final join node has no outgoing edge. Hence, each jds_{fi} for any join i adds a penalty of 1. In addition, we enforce that a relation may only be a leaf for a single join, using a one-hot encoding:

$$H_b = \sum_{i=1,i\neq f}^{J}\left(1 - \sum_{j=1,j\neq i}^{J} jds_{ij}\right)^2 + \sum_{i=1,i\neq f}^{J} jds_{fi} + \sum_{r=1}^{R}\left(1 - \sum_{i=1}^{J} rlj_{ri}\right)^2.$$

Example 4.2. *(cont'd) We continue our example for a query joining three relations, where $rlj_{Ai} = rlj_{Bi} = rlj_{Cj} = jds_{ij} = 1$, expressing the join order $(A \bowtie B) \bowtie C$. This configuration satisfies our constraint: Relation A is initially joined only by join i, hence $(1 - rlj_{Ai} - rlj_{Aj})^2 = 0$. The same applies mutatis mutandis for relations B and C. Further, join i is succeeded only by join j. Thus, $(1 - jds_{ij})^2 = 0$, whereas join j, as the final join, does not have any outgoing edges, hence avoiding energy penalties.*

In contrast, let us consider an invalid configuration with missing leaf assignments, where $rlj_{Ai} = 0$ rather than $rlj_{Ai} = 1$. Hence, $(1 - rlj_{Ai} - rlj_{Aj})^2 = 1$ penalises the lack of

join assignment for relation A. The same energy penalty is induced for $rlj_{Ai} = rlj_{Aj} = 1$, illustrating that H_b penalises both, insufficient and excessive amounts of outgoing edges.

(c) Preventing Cycles. So far, our constraints ensure that each join node has the correct amount of incoming and outgoing edges. To complete our set of constraints for valid trees, we have to enforce one final property that needs to hold for every tree: Each tree must be an acyclic graph. Enforcing this property is significantly more complex, as it concerns connections between an arbitrary number of tree nodes, while QUBO restricts variable interactions to contain at most two variables. To solve this issue, we may introduce a set of ancilla variables to store and later retrieve node properties useful to enforce this constraint. Specifically, we will label each join node with a depth value, such that join j must have depth d if preceded by a join i of depth $d+1$. In case of at least one cycle, a well-defined and unambiguous depth assignment becomes impossible, as shown by Theorem 4.1.

Theorem 4.1. *Enforcing an unambiguous, valid depth assignment for each join tree node prevents cycles.*

Proof. We consider a depth assignment valid if each join is labeled by an unambiguous depth $d \geq 0$, and $d_p = d_s + 1$ holds for any join p with depth d_p that precedes join s with depth d_s. Let us assume an acyclic join tree with any set of joins $(j_0, j_1, ..., j_n)$, where j_n denotes the final join (*i.e.*, the root of the tree), which we assign depth $d_n = 0$. Then, each join p directly preceding the root is assigned depth $d_p = d_n + 1 = 1$. The same applies, mutatis mutandis, for the remaining joins.

Let us now consider a configuration with two joins i and j, where i precedes j by a joins (*i.e.*, for $a = 1$, i directly precedes j, whereas for $a = 2$, i precedes an intermediate join k, which finally precedes j), where $a \geq 1$, and j precedes i by b joins, where $b \geq 1$, hence creating a cycle. Assuming a depth labeling with depths d_i and d_j is possible, $d_i = d_j + a$ must hold, since i precedes j by a joins. However, since j also precedes i by b joins, $d_j = d_i + b$ must moreover hold. Hence, $d_i = d_j + a = d_i + b + a$, contradicting our assumption, since $a \geq 1$ and $b \geq 1$. Therefore, enforcing an unambiguous depth assignment prevents the emergence of cycles. □

A depth constraint enforcing an unambiguous depth assignment for joins penalises any variable configuration that contains cycles, since such a configuration cannot conform to the constraint. Hence, we next introduce the required variables and QUBO terms. We moreover introduce variables and terms assigning depth values to relations, which we require for subsequent steps.

Let the binary variables rd_{rd} (*Relation has Depth*), introduced for each relation r and possible depth $d_r, 1 \leq d_r \leq J$ (the largest possible depth equals the number of joins), and jd_{jd} (*Join has Depth*), added for each join j and possible depth $d_j, 0 \leq d_j \leq J-1$, indicate whether r or j have depth d_r or d_j respectively.

Using one-hot encodings, we first enforce that each join and relation has an unambiguous depth:

$$H_c = \sum_{j=1}^{J} \left(1 - \sum_{d=0}^{J-1} jd_{jd}\right)^2 + \sum_{r=1}^{R} \left(1 - \sum_{d=1}^{J} rd_{rd}\right)^2.$$

Next, we ensure that only the final join f may have depth 0, by inducing a penalty if $jd_{f0} = 0$ and $jd_{j0} = 1$ for any $j \neq f$:

$$H_d = (1 - jd_{f0}) + \sum_{j=1, j\neq f}^{J} jd_{j0}.$$

We further ensure that no join of the maximum join depth $d_{max} = J - 1$ has a predecessor:

$$H_e = \sum_{i=1}^{J} \sum_{j=1, j\neq i}^{J} jds_{ij} jd_{jd_{max}}.$$

Finally, we need to assign the correct depths in accordance to the join tree expressed by the variables jds and rlj. Specifically, $jd_{jd} = 1$ has to respectively imply $jd_{i(d+1)} = 1$ if $jds_{ij} = 1$ or $rd_{r(d+1)} = 1$ if $rlj_{rj} = 1$. However, directly expressing $jd_{jd} jds_{ij} \implies jd_{i(d+1)}$ is not possible for QUBO, as it requires a degree 3 polynomial. To circumvent this issue, we can first apply the and-operator to store the result of $jd_{jd} \wedge jds_{ij}$ in an ancillary variable sj_{dij} as

$$H_f = \sum_{d=0}^{J-2} \sum_{i=1}^{J} \sum_{j=1}^{J} jd_{jd} jds_{ij} - 2jd_{jd} sj_{dij} - 2jds_{ij} sj_{dij} + 3sj_{dij},$$

allowing us to further implement the desired implication operation:

$$H_g = \sum_{d=0}^{J-2} \sum_{i=1}^{J} \sum_{j=1}^{J} sj_{dij}(1 - jd_{i(d+1)}).$$

Unfortunately, we require one ancillary variable for each join depth $0 \leq d \leq J-2$ and join pair (i, j). Therefore, the number of needed ancillary variables is cubic in the number of joins, which results in a significant variable overhead in comparison to left-deep join trees.

Similarly, for each relation r, depth $0 \leq d \leq J-1$ and join j, we add the constraints

$$H_h = \sum_{d=0}^{J-1} \sum_{r=1}^{R} \sum_{j=1}^{J} jd_{jd} rlj_{rj} - 2jd_{jd} sr_{rdj} - 2rlj_{rj} sr_{rdj} + 3sr_{rdj},$$

$$H_i = \sum_{d=0}^{J-1} \sum_{r=1}^{R} \sum_{j=1}^{J} sr_{rdj}(1 - rd_{r(d+1)}),$$

where sr_{rdj} denotes the required ancillary variable.

Example 4.3. *(cont'd) To illustrate the prevention of cycles, we continue our example with variable configuration* $rlj_{Ai} = rlj_{Bi} = rlj_{Cj} = jds_{ij} = 1$*, where we further consider* $jds_{ji} = 1$*. The configuration now contains a cycle, since join i precedes join j and vice-versa. In accordance to Theorem 4.1, we can penalise such invalid configurations by encoding an unambiguous assignment of depth labels for each join tree node. Hence, we introduce depth variables jd and rd for all joins and relations, with maximum join depth* $d_{max} = 1$*, and add the newly discussed penalty terms.*

Minimising the term H_d *assigns our final join j depth* $d_j = 0$*, by setting* $jd_{j0} = 1$*. Then, minimising* H_f *and* H_g *requires* $jd_{i1} = 1$*, since* $jds_{ij}jd_{j0} = 1$*. However, as join j now precedes a join of the maximum depth* $d_{max} = 1$*, penalty* $jds_{ji}jd_{id_{max}} = 1$ *is added in accordance to* H_e*.*

For the sake of illustrating the remaining terms, let us assume the maximum depth $d_{max} = 3$*, such that* H_e *won't penalise the configuration, and further depth variables beyond the maximum depth. Since* $jd_{i1}jds_{ji} = 1$*, we further require* $jd_{j2} = 1$ *to minimise* H_f *and* H_g*. However, this begets the energy penalty* $(1 - jd_{j0} - jd_{j2})^2 = (-1)^2 = 1$ *in accordance to* H_c*, since depth assignment for join j is no longer unambiguous. Due to the cyclic relationship between joins i and j, minimisation of* H_f *and* H_g *further activates their remaining depth variables, which begets increasingly higher energy penalties that can only be avoided by acyclic variable configurations. Minimising the discussed terms hence ensures acyclic join trees.*

For the remainder of our running example, we set $jds_{ji} = 0$*, which begets the valid depth assignment* $jd_{i1} = jd_{j0} = rd_{A2} = rd_{B2} = rd_{C1} = 1$*.*

Finally, we combine all terms discussed in this section into an overarching Hamiltonian for bushy join trees:

$$H_{bushy} = H_a + H_b + H_c + H_d + H_e + H_f + H_g + H_h + H_i.$$

4.3. Encoding Join Order Costs

Next, we assign each bushy tree a cost value, by encoding a cost function. Similarly to Schönberger *et al.* [5], we consider the cost function C_{out} [16], which sums over the sizes of intermediate join results. However, neither MILP nor QUBO support the product operations required for C_{out}. For their MILP approach, Trummer and Koch [6] therefore propose to substitute sums of logarithmic cardinalities and selectivities for these product operations, as the logarithm of a product equals the sum of logarithms of its factors. Based on the logarithmic intermediate result sizes, Trummer and Koch approximate the actual cardinalities using an arbitrary number of threshold values. Similarly to Schönberger *et al.* [5], who faithfully transformed the MILP cost approximation into QUBO, we show, in the following, how to apply this approximation to our native QUBO formulation for bushy trees.

4.3.1. Cost Variables

Based on the variables expressing a valid bushy tree, we derive a corresponding assignment of cost variables needed for calculating the cost of the tree. Let the binary variable roj_{rj} (*Relation is Operand for Join*), introduced for each relation *r* and join *j*, indicate whether *r* is an operand for *j*. Further, let the binary variables paj_{pj} (*Predicate is Applicable for Join*), added for each predicate *p* and join *j*, denote whether predicate *p* can be applied for join *j*.

First, we need to derive a valid assignment of roj_{rj} variables based on the bushy join tree expressed by the variables introduced in Sec. 4.2. Thereby, our ultimate goal is to enforce that, once joined by a join *j*, a relation *r* moreover serves as an operand for every join succeeding *j*. Further, *r* must not appear as an operand for a join *j* unless *r* is initially joined by *j*, or is an operand for any join preceding *j*.

We begin by enforcing that an activated leaf node variable rlj_{rj} implies the corresponding variable roj_{rj} to be activated, using the implication operator:

$$H_j = \sum_{r=1}^{R} \sum_{j=1}^{J} rlj_{rj}(1 - roj_{rj}).$$

The next step is more complex, as our goal is to enforce that, once joined for join *j*, a relation is also present for all joins succeeding *j*. In terms of variables, $roj_{rj} = 1$ must imply $roj_{ri} = 1$ if $jds_{ji} = 1$ (*i.e.*, join *i* is a direct successor to *j*).

Much like before, to avoid degree 3 polynomials, we first apply the and-operator to store the result of $roj_{rj} \wedge jds_{ji}$ into an ancillary variable s_{tij}:

$$H_k = \sum_{r=1}^{R} \sum_{i=1}^{J} \sum_{j=1}^{J} roj_{rj}jds_{ji} - 2roj_{rj}s_{tij} - 2jds_{ji}s_{tij} + 3s_{tij},$$

allowing us to implement the required implication operation:

$$H_l = \sum_{r=1}^{R} \sum_{i=1}^{J} \sum_{j=1}^{J} s_{tij}(1 - roj_{ri}).$$

Once again, this construction requires an amount of ancillary variables that is cubic in the number of relations to be joined.

However, the constraints enforced so far are not yet sufficient to produce a valid configuration of *roj* variables: While we do ensure that all *required roj* variables are active, we must moreover enforce that *no variable* roj_{rj} is active *unless required* by the join tree. Specifically, a relation must only serve as an operand for a join *j* if it is also an operand for either join preceding *j*, or else if it is a base relation initially joined by *j*. Enforcing this constraint in the same manner as H_k and H_l is, however, very expensive, as we require constraints for each relation *t* and join triplet (i, j, k) (since each constraint involves an

intermediate join i and two potential predecessors j and k), engendering an amount of ancillary variables that is quartic in the number of relations.

To circumvent this issue, we can apply a different approach to ensure a variable roj_{rj} remains inactive unless required: We may apply a constraint that bounds the number of allowed active *roj* variables by the correct amount, such that any additional active variables induce energy penalties. This requires us to determine the correct amount of joins considering a relation r as an operand, and therefore the number of *roj* variables allowed to be active for r. Fortunately, we have already determined the correct amount for each join in a previous step, since this number corresponds to the depth of r in the join tree, as shown by Theorem 4.2.

Theorem 4.2. *The number of joins that consider r as an operand is given by the depth d_r of r in the join tree.*

Proof. Assume that a relation l is initially joined by the final join f. Since f is the root of the tree, it has depth 0. Consequently, any tree node connecting to f has depth 1, including the leaf relation l and join i, which we assume precedes j. It follows that any leaf relation and further join connecting to i has depth 2. The same applies mutatis mutandis for all further joins and leaf relations in the join tree.

It is clear that the depth d_k of a join k then corresponds to the number of joins succeeding k. If k initially joins a relation r, r has depth $d_r = d_k + 1$. Consequently, r must serve as an operand for exactly d_r joins (*i.e.*, join k and all joins succeeding k). □

Making use of the existing depth variables, we apply an n-hot encoding, where n is given by each respective depth, to achieve our goal of enforcing the correct number of active *roj* variables:

$$H_m = \sum_{r=1}^{R} \left(\sum_{d=1}^{R} (d \cdot rd_{rd}) - \sum_{j=1}^{J} roj_{rj} \right)^2 .$$

Example 4.4. *(cont'd) To demonstrate the assignment of cost variables, we continue our example for a query joining three relations, where $rlj_{Ai} = rlj_{Bi} = rlj_{Cj} = jds_{ij} = jd_{i1} = jd_{j0} = rd_{A2} = rd_{B2} = rd_{C1} = 1$. We begin by adding cost variables roj for all relations and joins, and all terms discussed above, where H_j enforces $roj_{Ai} = roj_{Bi} = roj_{Cj} = 1$, following from rlj variable assignments. Since $jds_{ij} = 1$, relations A and B are moreover required as operands for join j, which is enforced by minimising H_k and H_l, setting $roj_{Aj} = roj_{Bj} = 1$. Thereby, our encoding has ensured all required variables roj are active.*

In addition, it must moreover prevent the activation of roj variables unless required. For instance, cost minimisation may activate $roj_{Ci} = 1$ even though relation C is not an operand for join i. However, since $rd_{C1} = 1$, labeling relation C with depth 1, C must be an operand for exactly one join, in accordance to Theorem 4.2. This is ensured by term H_m, adding energy penalty $(1 \cdot rd_{C1} - roj_{Ci} - roj_{Cj})^2 = (-1)^2 = 1$ if both, $roj_{Ci} = 1$ and $roj_{Cj} = 1$. Hence, minimising H_m yields $roj_{Ci} = 0$.

Finally, based on the correctly assigned *roj* variables, we derive valid assignments for predicate variables *paj*. Specifically, we need to enforce that $paj_{pj} = 1$ only holds if both relations associated with predicate p are operands for join j, which we implement using the implication operator:

$$H_n = \sum_{p=1}^{P} \sum_{j=1}^{J} paj_{pj}(2 - roj_{Rel_1(p)j} - roj_{Rel_2(p)j}),$$

where $Rel_i(p), 1 \leq i \leq 2$, corresponds to the first or second relation associated with p.

Example 4.5. *(cont'd) To illustrate the inclusion of predicates, we continue our example for a query joining three relations, where $roj_{Ai} = roj_{Aj} = roj_{Bi} = roj_{Bj} = roj_{Cj} = 1$. We further consider two join predicates p_1, associated with relations A and B, and p_2, for relations B and C. Accordingly, we add variables paj_{1i}, paj_{1j}, paj_{2i} and paj_{2j}. To minimise the cost terms discussed below, any minimisation method seeks to apply as many predicates as possible. Ideally, $paj_{1i} = paj_{1j} = paj_{2i} = paj_{2j} = 1$. However, predicate p_2 may not be applied for join i, since relation C is not an operand. Accordingly, energy penalty $paj_{2i}(2 - roj_{Bi} - roj_{Ci}) = 1(2 - 1 - 0) = 1$ is added to the overall costs, in accordance to H_n. The configuration $paj_{2i} = 0$ is hence enforced by minimising H_n, assuming a proper balance between validity and cost terms, as discussed in Sec. 4.4, to ensure the energy penalty for activating paj_{2i} is larger than any potential cost savings.*

4.3.2. Logarithmic Cost Calculation

Based on the cost variables introduced in the previous section, we can now gradually encode the join tree costs. Following the existing MILP and QUBO formulations for left-deep trees [6, 5], we rely on an approximation of logarithmic costs, which allows us to encode the classic cost function C_{out} in QUBO. We express the logarithmic intermediate cardinality for join j as:

$$LogIntCard(j) = \sum_{r=1}^{R} LogCard(r) roj_{rj} + \sum_{p=1}^{P} LogSel(p) paj_{pj},$$

where $LogCard(r)$ and $LogSel(p)$ provide the logarithmic cardinality for relation r and the logarithmic selectivity for predicate p respectively.

Example 4.6. *(cont'd) To illustrate the logarithmic cost calculation, we continue our example for a query*

joining three relations, where $roj_{Ai} = roj_{Aj} = roj_{Bi} = roj_{Bj} = roj_{Cj} = paj_{1i} = paj_{1j} = paj_{2j} = 1$. *Thereby, we consider logarithmic input cardinalities* $n_A = n_B = n_C = 2$, *and logarithmic predicate selectivities* $s_1 = s_2 = -1$. *Then, for join i, we obtain the logarithmic result size* $LogIntCard(i) = n_A roj_{Ai} + n_B roj_{Bi} + s_1 paj_{1i} = 2+2-1 = 3$, *and* $LogIntCard(j) = n_A roj_{Aj} + n_B roj_{Bj} + +n_C roj_{Cj} + s_1 paj_{1j} + s_2 paj_{2j} = 2+2+2-1-1 = 4$ *for join j.*

Based on the logarithmic costs, we next approximate the actual intermediate join cardinalities.

4.3.3. Cost Approximation

Following the MILP and QUBO approaches for left-deep join trees [6, 5], a set of threshold values is added to the model to approximate the actual intermediate join cardinalities based on the logarithmic ones: Let the binary variables trj_{tj} (*Threshold is Reached by Join*), added for every threshold value $1 \leq t \leq T$ and intermediate join $1 \leq j \leq J-1$ (we exclude the final join, whose cost is invariant w.r.t. the join tree), indicate whether the logarithmic threshold $\log(\theta_t)$ is exceeded by the cardinality of the intermediate result produced by join j. In this case, the threshold θ_t is be added to the overall costs, by setting the following cost term:

$$H_{cost} = \sum_{t=1}^{T} \sum_{j=1}^{J-1} trj_{tj} \theta_t.$$

In the original MILP encoding, the following inequality constraint activates trj_{tj} if $\log(\theta_t)$ is exceeded by the result size produced by join j:

$$LogIntCard(j) - trj_{tj} \cdot \infty_{tj} \leq \log(\theta_t).$$

Specifically, if $LogIntCard(j) > \log(\theta_t)$, the constraint can only be satisfied by activating trj_{tj}, which subtracts the sufficiently large constant ∞_{tj}.

However, such inequality operations are not inherently supported by QUBO. Following Schönberger *et al.* [5], we hence convert the inequalities to equality constraints, using a continuous variable s_{tj}:

$$LogIntCard(j) - trj_{tj} \cdot \infty_{tj} + s_{tj} = \log(\theta_t). \tag{2}$$

However, QUBO only allows binary variables, rather than the required continuous variable. Therefore, we next approximate s_{tj} as $s_{tj} \approx \omega \sum_{i=1}^{n} 2^{i-1} b_i$, substituting the continuous variable for multiple binary variables b_i. We may tune the accuracy of this discretisation by choosing the number of allowed decimal positions d. Then, $\omega = (0.1)^d$ denotes a discretisation precision, and we require $n = \lfloor \log_2(c_{j_{max}}/\omega) \rfloor + 1$ binary variables for the discretisation, where $c_{j_{max}}$ is the maximum logarithmic cardinality possible for join j. For further details on this approximation, we refer the reader to Schönberger *et al.* [5].

Encoding the resulting equality described by Eqn. 2 as a Hamiltonian, we get

$$H_o = \sum_{t=1}^{T} \sum_{j=1}^{J-1} \left(LogIntCard(j) - trj_{tj} \cdot \infty_{tj} + s_{tj} - \log(\theta_t) \right)^2.$$

Example 4.7. *(cont'd) We complete our running example by illustrating the cost approximation for our query joining three relations. Recall the logarithmic intermediate result sizes* $LogIntCard(i) = 3$ *and* $LogIntCard(j) = 4$ *for joins i and j respectively, as calculated in Example 4.6. We consider two thresholds* $\theta_1 = 100$ *and* $\theta_2 = 1000$ *for the cost approximation, and hence introduce variables* trj_{1i}, trj_{1j}, trj_{2i} *and* trj_{2j}. *To minimise cost, a minimisation method will seek to leave as many threshold variables inactive as possible. Ideally,* $trj_{1i} = trj_{1j} = trj_{2i} = trj_{2j} = 0$. *However, since* $LogIntCard(i) = 3 > 2 = \log(\theta_1)$, trj_{1i} *must be active to avoid a penalty induced by* H_o. *The same holds for* trj_{1j}, *since* $LogIntCard(j) = 4 > 2 = \log(\theta_1)$, *and* trj_{2j}, *since* $LogIntCard(j) = 4 > 3 = \log(\theta_2)$. *Hence,* $H_{cost} = \theta_1 trj_{1i} + \theta_1 trj_{1j} + \theta_2 trj_{2j} = 100+100+1000 = 1200$ *adds cost accordingly. We observe that the accuracy of the approximation strongly depends on the selected thresholds, and approximation quality increases as more thresholds are added.*

Finally, we combine all validity terms into an overarching validity Hamiltonian

$$H_{val} = H_{bushy} + H_j + H_k + H_l + H_m + H_n + H_o.$$

4.4. The Complete Encoding

Based on the validity and cost Hamiltonians, we can now construct the complete Hamiltonian H:

$$H = AH_{val} + H_{cost}.$$

The penalty weight A thereby amplifies the penalty arising from violating validity terms, such that the added energy always outweighs any potential cost savings resulting from the violation. However, setting A arbitrarily large begets issues such as slowdowns [17]. Hence, we next derive a lower bound for A.

This lower bound depends on the smallest possible penalty that may be engendered by any constraint violation due to H_{val}. Most of our penalty terms induce an energy value of 1 when violated, with the exception of H_p which enforces an activation of threshold variables if the corresponding thresholds were exceeded. For determining A, we hence focus on the term H_p, as it can engender penalties less than 1: Following Schönberger *et al.* [5], the smallest violation in H_p depends on the discretisation precision ω. Specifically, $A = C/\omega^2 + \epsilon$, where $C = \sum_{t=1}^{T} \sum_{j=1}^{J-1} \theta_t$ expresses the maximum value that may be assumed by H_{cost}, and ϵ is some small constant.

5. Related Work

Quantum computing remains largely unexplored within database research. However, the aptness of QPUs was recently analysed for a small selection of problems in the database domain. These problems include database transaction scheduling [2, 3, 4] and multiple query optimisation [1]. However, their methods and encodings are problem-specific, and hence cannot be re-used for JO optimisation, originally solved on QPUs by Schönberger *et al.* [5]. Improving their method is our primary goal in this paper.

The JO problem is one of the most fundamental and well studied problems in query optimisation [18, 15, 19, 7, 6, 20]. Methods to handle this problem largely follow one of two fundamental paradigms: Firstly, dynamic programming (DP) approaches [21, 22, 14, 23, 24], yielding optimal solutions, yet unable to scale to large queries, given the super-polynomial search space growth of common JO classifications. Hence, heuristic methods [25, 26, 27, 28, 29, 30, 31] seek to optimise larger queries, where DP methods fail, at the cost of no guarantees on solution quality. This category includes both, the baseline JO method for QPUs by Schönberger *et al.* [5], and our novel method.

Our work heavily improves on the baseline encoding by Schönberger *et al.* [5] in several ways: (1) While the baseline encoding is a faithful transformation of the JO-MILP method by Trummer and Koch [6], and hence inherits MILP limitations, our novel encoding natively encodes JO as QUBO, and is hence tailored to contemporary QPUs. Further, (2) the baseline approach restricts its solution space to left-deep join trees, which commonly yield significantly higher cost than bushy join trees [7]. In contrast, our novel encoding applies no restrictions on the JO solution space, enabling quantum optimisation of general, unrestricted trees, in conjunction with the ability to include beneficial cross products. Our encoding can hence derive cheap plans unobtainable within more restricted JO solution spaces.

6. Discussion and Conclusion

Our novel JO-QUBO encoding exhausts the full potential of quantum hardware for join ordering, allowing QPUs to optimise general, unrestricted join trees. However, even for solution spaces restricted to left-deep trees, current early-stage QPUs can only optimise small-scale queries [5]. Therefore, we cannot expect contemporary systems to yield meaningful results when optimising yet more complex JO classifications.

Still, our novel encoding provides valuable architectural insights: For instance, the reduction of degree 3 terms, required to conform to contemporary quantum hardware, yields a variable overhead that is cubic in the number of relations, which (1) substantially increases the number of required qubits, and (2) severely enhances the search space, to the detriment of scalability. We hence identify the requirement for quadratic terms as a clear limitation of current quantum systems, which can be addressed by augmenting QPUs to allow degree 3 polynomials. Thereby, we can significantly enhance the conformance of QPUs for JO optimisation. Hence, rather than merely waiting for the arrival of mature quantum systems, it is necessary to specify the *requirements for such systems*, by identifying, analysing and optimising problem encodings, such that QPUs tailored to those requirements can be crafted.

References

[1] I. Trummer, C. Koch, Multiple query optimization on the D-Wave 2X adiabatic quantum computer, Proceedings of the VLDB Endowment 9 (2016) 648–659.

[2] S. Groppe, J. Groppe, Optimizing transaction schedules on universal quantum computers via code generation for grover's search algorithm, in: 25th International Database Engineering & Applications Symposium, IDEAS 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 149–156. URL: https://doi.org/10.1145/3472163.3472164. doi:10.1145/3472163.3472164.

[3] T. Bittner, S. Groppe, Hardware accelerating the optimization of transaction schedules via quantum annealing by avoiding blocking, Open Journal of Cloud Computing (OJCC) 7 (2020) 1–21.

[4] T. Bittner, S. Groppe, Avoiding blocking by scheduling transactions using quantum annealing, in: Proceedings of the 24th Symposium on International Database Engineering & Applications, IDEAS '20, Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3410566.3410593. doi:10.1145/3410566.3410593.

[5] M. Schönberger, S. Scherzinger, W. Mauerer, Ready to leap (by co-design)? join order optimisation on quantum hardware, in: Proceedings of the 2023 ACM International Conference on Management of Data, ACM, Seattle, WA, USA, 2023.

[6] I. Trummer, C. Koch, Solving the join ordering problem via mixed integer linear programming, in: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, New York, NY, USA, 2017, pp. 1025–1040.

[7] T. Neumann, B. Radke, Adaptive optimization of very large join queries, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Com-

puting Machinery, New York, NY, USA, 2018, p. 677–692. URL: https://doi.org/10.1145/3183713.3183733. doi:10.1145/3183713.3183733.

[8] IBM Quantum, Cloud access to quantum computers provided by IBM, 2022. URL: https://quantum-computing.ibm.com.

[9] C. McGeoch, P. Farré, The D-Wave Advantage system: An overview, Technical Report 14-1049A-A, D-Wave Systems Inc, 2020.

[10] E. Farhi, J. Goldstone, S. Gutmann, A quantum approximate optimization algorithm (2014). arXiv:1411.4028.

[11] Z. Bian, F. Chudak, W. Macready, G. Rose, The Ising model: Teaching an old problem new tricks, Technical Report, D-Wave Systems Inc., 2010.

[12] M. Lewis, F. Glover, Quadratic unconstrained binary optimization problem preprocessing: Theory and empirical analysis (2017). arXiv:1705.09844.

[13] D.-W. S. Inc., Problem-solving handbook, 2023. URL: https://docs.dwavesys.com/docs/latest/handbook_reformulating.html.

[14] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, in: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, SIGMOD '79, Association for Computing Machinery, New York, NY, USA, 1979, p. 23–34. URL: https://doi.org/10.1145/582095.582099. doi:10.1145/582095.582099.

[15] G. Moerkotte, Building query compilers, 2020. URL: https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf.

[16] S. Cluet, G. Moerkotte, On the complexity of generating optimal left-deep processing trees with cross products, in: Database Theory — ICDT '95, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 54–67.

[17] B. O'Gorman, R. Babbush, A. Perdomo-Ortiz, A. Aspuru-Guzik, V. Smelyanskiy, Bayesian network structure learning using quantum annealing, The European Physical Journal Special Topics 224 (2015) 163–188.

[18] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, Query optimization through the looking glass, and what we found running the join order benchmark, The VLDB Journal 27 (2018) 643–668.

[19] T. Neumann, Query simplification: Graceful degradation for join-order optimization, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 403–414. URL: https://doi.org/10.1145/1559845.1559889. doi:10.1145/1559845.1559889.

[20] X. Yu, G. Li, C. Chai, N. Tang, Reinforcement learning with tree-lstm for join order selection, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), 2020, pp. 1297–1308. doi:10.1109/ICDE48307.2020.00116.

[21] A. Meister, G. Saake, GPU-accelerated dynamic programming for join-order optimization, Technical Report, 2020. URL: https://www.inf.ovgu.de/inf_media/downloads/forschung/technical_reports_und_preprints/2020/TechnicalReport+02_2020-p-8268.pdf.

[22] G. Moerkotte, T. Neumann, 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, VLDB '06, VLDB Endowment, 2006, p. 930–941.

[23] B. Vance, D. Maier, Rapid bushy join-order optimization with cartesian products, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96, Association for Computing Machinery, New York, NY, USA, 1996, p. 35–46. URL: https://doi.org/10.1145/233269.233317. doi:10.1145/233269.233317.

[24] G. Moerkotte, T. Neumann, Dynamic programming strikes back, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 539–552. URL: https://doi.org/10.1145/1376616.1376672. doi:10.1145/1376616.1376672.

[25] M. Steinbrunn, G. Moerkotte, A. Kemper, Heuristic and randomized optimization for the join ordering problem, The VLDB journal 6 (1997) 191–208.

[26] J.-T. Horng, C.-Y. Kao, B.-J. Liu, A genetic algorithm for database query optimization, in: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, 1994, pp. 350–355 vol.1. doi:10.1109/ICEC.1994.349926.

[27] N. Bruno, C. Galindo-Legaria, M. Joshi, Polynomial heuristics for query optimization, in: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), 2010, pp. 589–600. doi:10.1109/ICDE.2010.5447916.

[28] Y. E. Ioannidis, Y. Kang, Randomized algorithms for optimizing large join queries, volume 19, Association for Computing Machinery, New York, NY, USA, 1990, p. 312–321. URL: https://doi.org/10.1145/93605.98740. doi:10.1145/93605.98740.

[29] A. Swami, A. Gupta, Optimization of large join queries, volume 17, Association for Computing Machinery, New York, NY, USA, 1988, p. 8–17. URL: https://doi.org/10.1145/971701.50203. doi:10.1145/971701.50203.

[30] A. Swami, Optimization of large join queries: Com-

bining heuristics and combinatorial techniques, in: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD '89, Association for Computing Machinery, New York, NY, USA, 1989, p. 367–376. URL: https://doi.org/10.1145/67544.66961. doi:10.1145/67544.66961.

[31] I. Trummer, C. Koch, Parallelizing query optimization on shared-nothing architectures, Proc. VLDB Endow. 9 (2016) 660–671. URL: https://doi.org/10.14778/2947618.2947622. doi:10.14778/2947618.2947622.