D:Wave
The Quantum Computing Company™

D-Wave Solver Properties and Parameters Reference

USER MANUAL

2021-09-07

Overview

This document describes the solvers available in the D-Wave system, listing their properties and the parameters they accept with a problem submission.

CONTACT

Corporate Headquarters
3033 Beta Ave
Burnaby, BC V5G 4M9
Canada
Tel. 604-630-1428

US Office
2650 E Bayshore Rd
Palo Alto, CA 94303

Email: info@dwavesys.com

www.dwavesys.com

D-Wave User Manual 09-1169B-D

Notice and Disclaimer

D-Wave Systems Inc. (D-Wave), its subsidiaries and affiliates, makes commercially reasonable efforts to ensure that the information in this document is accurate and up to date, but errors may occur. NONE OF D-WAVE SYSTEMS INC., its subsidiaries and affiliates, OR ANY OF ITS RESPECTIVE DIRECTORS, EMPLOYEES, AGENTS, OR OTHER REPRESENTATIVES WILL BE LIABLE FOR DAMAGES, CLAIMS, EXPENSES OR OTHER COSTS (INCLUDING WITHOUT LIMITATION LEGAL FEES) ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS DOCUMENT OR ANY INFORMATION CONTAINED OR REFERRED TO IN IT. THIS IS A COMPREHENSIVE LIMITATION OF LIABILITY THAT APPLIES TO ALL DAMAGES OF ANY KIND, INCLUDING (WITHOUT LIMITATION) COMPENSATORY, DIRECT, INDIRECT, EXEMPLARY, PUNITIVE AND CONSEQUENTIAL DAMAGES, LOSS OF PROGRAMS OR DATA, INCOME OR PROFIT, LOSS OR DAMAGE TO PROPERTY, AND CLAIMS OF THIRD PARTIES.

D-Wave reserves the right to alter this document and other referenced documents without notice from time to time and at its sole discretion. D-Wave reserves its intellectual property rights in and to this document and its proprietary technology, including copyright, trademark rights, industrial design rights, and patent rights. D-Wave trademarks used herein include D-Wave®, Leap™ quantum cloud service, Ocean™, Advantage™ quantum system, D-Wave 2000Q™, D-Wave 2X™, and the D-Wave logos (the D-Wave Marks). Other marks used in this document are the property of their respective owners. D-Wave does not grant any license, assignment, or other grant of interest in or to the copyright of this document, the D-Wave Marks, any other marks used in this document, or any other intellectual property rights used or referred to herein, except as D-Wave may expressly provide in a written agreement. This document may refer to other documents, including documents subject to the rights of third parties. Nothing in this document constitutes a grant by D-Wave of any license, assignment, or any other interest in the copyright or other intellectual property rights of such other documents. Any use of such other documents is subject to the rights of D-Wave and/or any applicable third parties in those documents.

Contents

1 Introduction 1
- 1.1 About This Document . . . 1
- 1.2 SAPI Solvers . . . 1

2 Problem Parameters 5
- 2.1 bqm . . . 5
- 2.2 dqm . . . 6
- 2.3 h . . . 7
- 2.4 J . . . 8
- 2.5 label . . . 10
- 2.6 Q . . . 11
- 2.7 solver . . . 12

3 Solver Properties 13
- 3.1 anneal_offset_ranges . . . 14
- 3.2 anneal_offset_step . . . 15
- 3.3 anneal_offset_step_phi0 . . . 15
- 3.4 annealing_time_range . . . 15
- 3.5 beta_range . . . 16
- 3.6 category . . . 16
- 3.7 chip_id . . . 16
- 3.8 couplers . . . 17
- 3.9 default_annealing_time . . . 17
- 3.10 default_beta . . . 18
- 3.11 default_programming_thermalization . . . 18
- 3.12 default_readout_thermalization . . . 18
- 3.13 extended_j_range . . . 19
- 3.14 h_gain_schedule_range . . . 19
- 3.15 h_range . . . 19
- 3.16 j_range . . . 20
- 3.17 max_anneal_schedule_points . . . 20
- 3.18 max_h_gain_schedule_points . . . 21
- 3.19 maximum_number_of_biases . . . 21
- 3.20 maximum_number_of_variables . . . 22
- 3.21 maximum_time_limit_hrs . . . 22
- 3.22 minimum_time_limit . . . 22
- 3.23 num_qubits . . . 23
- 3.24 num_reads_range . . . 24
- 3.25 parameters . . . 24
- 3.26 per_qubit_coupling_range . . . 24
- 3.27 problem_run_duration_range . . . 25
- 3.28 programming_thermalization_range . . . 25
- 3.29 quota_conversion_rate . . . 26
- 3.30 qubits . . . 26
- 3.31 readout_thermalization_range . . . 27
- 3.32 supported_problem_types . . . 27
- 3.33 tags . . . 28

3.34 topology . . . 28
3.35 version . . . 29
3.36 vfyc . . . 29

4 Solver Parameters 30
4.1 anneal_offsets . . . 31
4.2 anneal_schedule . . . 32
4.3 annealing_time . . . 33
4.4 answer_mode . . . 34
4.5 auto_scale . . . 35
4.6 beta . . . 37
4.7 chains . . . 38
4.8 flux_biases . . . 38
4.9 flux_drift_compensation . . . 39
4.10 h_gain_schedule . . . 40
4.11 initial_state . . . 42
4.12 max_answers . . . 42
4.13 num_reads . . . 43
4.14 num_spin_reversal_transforms . . . 44
4.15 postprocess . . . 45
4.16 programming_thermalization . . . 46
4.17 readout_thermalization . . . 47
4.18 reduce_intersample_correlation . . . 48
4.19 reinitialize_state . . . 48
4.20 time_limit . . . 49

1 Introduction

SAPI Solvers—compute resources D-Wave makes available through SAPI that you can submit problems to—can be queried for their properties, which characterize behaviors and supported features; accept certain parameters that define the problem to be solved; and accept other parameters that control how the problem is run.

For example, Advantage™ quantum computers have a *num_reads_range* property that advertises the supported range for the number of requested anneals on a problem submitted to such a solver:

```
>>> from dwave.system import DWaveSampler
>>> DWaveSampler().properties["num_reads_range"]
[1, 10000]
```

Similarly, a Leap hybrid solver might accept a *bqm* and Assign Problem Labels as its *problem parameters* and *time_limit* as a *controlling parameter*, where the supported values of the latter are specified by this solver's *minimum_time_limit* property.

```
>>> from dwave.system import LeapHybridDQMSampler
>>> dqm = dimod.DiscreteQuadraticModel.from_file("my_dqm_problem")
>>> LeapHybridDQMSampler().sample_dqm(dqm, time_limit=10)
```

1.1 About This Document

This document describes the available *SAPI Solvers* and the supported

- *Problem parameters*
- *Solver properties*
- *Solver parameters*

1.2 SAPI Solvers

D-Wave provides quantum-classical *Hybrid Solvers*, *QPU Solvers*, and QPU-like solvers (*VFYC Solvers* and *emulators*) that are available through SAPI.

Note: Not all accounts have access to all these solvers. You can see the solvers your account can access on the Leap dashboard for your account or by using the dwave CLI (e.g., `dwave solvers -la`).

Additionally, Ocean software provides classical solvers such as dwave-greedy that you can run locally. Those are not described in this document. See the Ocean documentation for information on parameters of such solvers.

Hybrid Solvers

Leap's quantum-classical hybrid solvers are intended to solve arbitrary application problems formulated as binary quadratic models (BQM) or discrete[1] quadratic models.

These solvers, which implement state-of-the-art classical algorithms together with intelligent allocation of the quantum processing unit (QPU) to parts of the problem where it benefits most, are designed to accommodate even very large problems. Leap's solvers can relieve you of the burden of any current and future development and optimization of hybrid algorithms that best solve your problem.

These solvers have the following characteristics:

- There is no fixed problem structure. In particular, these solvers do not have properties *num_qubits*, *qubits*, and *couplers*.
- Solvers may return one solution or more, depending on the solver, the problem, and the time allowed for solution. Returned solutions are not guaranteed to be optimal.
- Solver properties and parameters are entirely disjoint from those of QPU and QPU-like solvers.
- Maximum problem size and solution times differ between solvers and might change with subsequent versions of the solver: always check your selected solver's relevant properties. For example, the solver selected below has limits on *maximum_number_of_biases* and *maximum_number_of_variables* that restrict problem size, and requirements on *minimum_time_limit* and *maximum_time_limit_hrs* that restrict solution time.

```
>>> sampler = LeapHybridDQMSampler()
>>> sampler.properties.keys()
dict_keys(['minimum_time_limit',
           'maximum_time_limit_hrs',
           'maximum_number_of_variables',
           'maximum_number_of_biases',
           'parameters',
           'supported_problem_types',
           'category',
           'version',
           'quota_conversion_rate'])
 >>> sampler.properties["maximum_time_limit_hrs"]
 24.0
```

[1] Where the standard problems submitted to quantum computers have binary valued variables (0, 1 for QUBO and $+1, -1$ for Ising formulation), discrete quadratic model (DQM) solvers solve problems with variables that have more than two values; for example, variables that represent colors or DNA bases.

Generally Available Hybrid Solvers

The generally available hybrid solvers depend on your Leap™ account. Check your Leap dashboard to see which hybrid solvers are available to you.

Generally-available hybrid solvers currently supported in Leap include:

- Hybrid BQM solver (e.g., `hybrid_binary_quadratic_model_version2`)

  These solvers solve arbitrary application problems formulated as binary quadratic models (BQM).
- Hybrid DQM solver (e.g., `hybrid_discrete_quadratic_model_version1`)

  These solvers solve arbitrary application problems formulated as **discrete** quadratic models (DQM).

QPU Solvers

The *Getting Started with D-Wave Solvers* guide is an introduction to QPU solvers. For a more advanced, technical description of QPUs, see the *QPU Solver Datasheet* guide.

Generally Available QPU Solvers

D-Wave makes available two QPU solvers: the Advantage and the DW-2000Q.

VFYC Solvers

A virtual full-yield chip (VFYC) solver emulates a fully connected DW-2000Q QPU working graph based on an idealized abstraction of the system. Through this solver, variables corresponding to a Chimera-structured graph that are not representable on a specific working graph are determined via hybrid use of the QPU and the integrated postprocessing system, which effectively fills in any missing qubits and couplers that may affect the QPU. For more information on the VFYC solver and how it is integrated with the postprocessing system, see the *QPU Solver Datasheet* guide.

Generally Available VFYC Solvers

VFYC solvers for D-Wave 2000Q systems are available server-side for some accounts. Check your Leap dashboard to see if such solvers are available to you.

Emulators

Software solvers are useful for prototyping optimization or sampling algorithms that make multiple calls to the hardware.

- **Optimizing emulator solvers** solve the same type of optimization problems as the QPU, but using a classical software algorithm.

  This class of solvers is entirely deterministic, so the semantics of some parameters are different from the QPU. The number of solutions returned is always the lesser of *num_reads* and *max_answers*. The solutions returned are the lowest-energy states, sorted in increasing-energy order.

- **Sampling emulator solvers** mimic the probabilistic nature of the QPU to draw samples that can be used to train a probabilistic model.

  This class of solvers mimic the probabilistic nature of the QPU, drawing samples from a Boltzmann distribution; that is, state s is sampled with probability proportional to:

  $$\exp(-\beta E(s))$$

  where β is some parameter and $E(s)$ is the energy of state s.

Generally Available Emulators

These solvers have names ending with the following strings:

- `-sw_optimize`
- `-sw_sample`

Emulator solvers are not available for all accounts. Check your Leap dashboard to see if such solvers are available to you.

2 Problem Parameters

The quantum machine instructions (QMI) you send to D-Wave QPU solvers comprise problem parameters (e.g., linear biases, `h`, of an Ising problem) and solver parameters that control how the problem is run (e.g., an annealing schedule, *anneal_schedule*). Ocean software tools and SAPI's REST API provide various abstractions for encoding the linear and quadratic biases that configure problems on D-Wave QPUs and enable expansion to problems such as the discrete binary model (DQM) solved on Leap's hybrid solvers.

For example, Ocean software's `DWaveSampler` accepts problems as a `BinaryQuadraticModel`, which contains both the *Q* of the QUBO format and *h* and *J* of the Ising format, as well as the latter three directly; e.g., `DWaveSampler().sample_qubo(Q)`,`DWaveSampler().sample_ising(h, J)` and `DWaveSampler().sample(bqm)`.

This chapter describes the problem parameters accepted by D-Wave solvers, in alphabetical order.

- *bqm*
- *dqm*
- *h*
- *J*
- *label*
- *Q*
- *solver*

2.1 bqm

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑ BQM	☑	☑ VFYC

Ocean software's `dimod.BinaryQuadraticModel` (BQM) contains linear and quadratic biases for problems formulated as binary quadratic models as well as additional information such as variable labels and offset.

For QPU solvers, Ocean software converts to Ising format and submits linear and quadratic biases.

Relevant Properties

- *maximum_number_of_variables* defines the maximum number of problem variables for hybrid solvers.
- *maximum_number_of_biases* defines the maximum number of problem biases for hybrid solvers.
- *minimum_time_limit* and *maximum_time_limit_hrs* define the runtime duration for hybrid solvers.

Example

This example creates a BQM from a QUBO and submits it to a QPU.

```
>>> import dimod
>>> from dwave.system import DWaveSampler, EmbeddingComposite
...
>>> Q = {(0, 0): -3, (1, 1): -1, (0, 1): 2, (2, 2): -1, (0, 2): 2}
>>> bqm = dimod.BQM.from_qubo(Q)
>>> sampleset = EmbeddingComposite(DWaveSampler()).sample(bqm, num_
↪reads=100)
```

2.2 dqm

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑ DQM	☒	☒

Ocean software's `dimod.DiscreteQuadraticModel` contains linear and quadratic biases for problems formulated as discrete binary models as well as additional information such as variable labels.

Relevant Properties

- *maximum_number_of_variables* defines the maximum number of problem variables.
- *maximum_number_of_biases* defines the maximum number of problem biases.
- *minimum_time_limit* and *maximum_time_limit_hrs* define the runtime duration for hybrid solvers.

Example

This example uses Ocean software's `LeapHybridDQMSampler`.

```
>>> from dwave.system import LeapHybridDQMSampler
>>> import dimod
>>> import random
...
>>> dqm = dimod.DiscreteQuadraticModel()
>>> for i in range(10):
...     dqm.add_variable(4)
>>> for i in range(9):
...     for j in range(i+1, 10):
...         dqm.set_quadratic_case(i, random.randrange(0, 4),
...                                j, random.randrange(0, 4), random.
 ↪random())
>>> sampleset = LeapHybridDQMSampler().sample_dqm(dqm, time_limit=10)
```

2.3 h

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑ BQM	☑	☑ VFYC, Emulators

For Ising problems, the h values are the linear coefficients (biases). A problem definition comprises both h and J values.

Because the quantum annealing process[1] minimizes the energy function of the Hamiltonian, and h_i is the coefficient of variable i, returned states of the problem's variables tend toward the opposite sign of their biases; for example, if you bias the qubits representing variable v_i with h_i values of -1, variable v_i is more likely to have a final state of $+1$ in the solution.

For QPU solvers, the programming cycle programs the solver to the specified h and J values for a given Ising problem (or derived from the specified Q values of a given QUBO problem). However, since QPU precision is limited, the h and J values realized on the solver may deviate slightly from the requested (or derived) values. For more information, see the *QPU Solver Datasheet* guide.

For QUBO problems, use Q instead of h and J; see the *Getting Started with D-Wave Solvers* guide for more information and converting between the formulations.

If you are submitting directly through SAPI's REST API, see Data Encoding and the *SAPI REST Developers Guide* for more information.

Default is zero linear biases.

[1] QPU-like solvers emulate this process.

Relevant Properties

- *h_range* defines the supported h range for QPU solvers.
- *maximum_number_of_variables* defines the maximum number of problem variables for hybrid solvers.
- *maximum_number_of_biases* defines the maximum number of problem biases for hybrid solvers.
- *qubits* and *couplers* define the working graph of QPU solvers.

Interacts with Parameters

- *auto_scale* enables you to submit problems to QPU solvers with values outside *h_range* and *j_range* and have the system automatically scale them to fit. You cannot use *auto_scale* with the *extended_j_range*.

Example

This example defines a 7-qubit problem in which q_1 and q_4 are biased with -1 and q_0, q_2, q_3, q_4, and q_6 are biased with $+1$. Qubits q_0 and q_6 have a ferromagnetic coupling between them with a strength of -1.

```
>>> h = [1, -1, 1, 1, -1, 1, 1]
>>> J = {(0, 6): -1}
```

2.4 J

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑ BQM	☑	☑ VFYC, Emulators

For Ising problems, the J values are the quadratic coefficients. The larger the absolute value of J, the stronger the coupling between pairs of variables (and qubits on QPU solvers). An Ising problem definition comprises both h and J values.

For QPU solvers:

- This parameter sets the strength of the couplers between qubits:
  - **J < 0**: ferromagnetic coupling; coupled qubits prefer to be in the same state, $(1, 1)$ or $(-1, -1)$.
  - **J > 0**: antiferromagnetic coupling; coupled qubits prefer to be in opposite states, $(-1, 1)$ or $(1, -1)$.
  - **J = 0**: No coupling; qubit states do not affect each other.
- The programming cycle programs the solver to the specified h and J values for a given Ising problem (or derived from the specified Q values of a given QUBO problem). However, since QPU precision is limited, the h and J values realized on the

solver may deviate slightly from the requested (or derived) values. For more information, see the *QPU Solver Datasheet* guide.

For QUBO problems, use *Q* instead of h and J; see the *Getting Started with D-Wave Solvers* guide for more information and converting between the formulations.

If you are submitting directly through SAPI's REST API, see Data Encoding and the *SAPI REST Developers Guide* for more information.

Default is zero quadratic biases.

Relevant Properties

- *j_range* defines the supported J range for QPU solvers.
- *extended_j_range* defines an extended range of values possible for the coupling strengths for some QPU solvers; however, be aware that you cannot use the extended J range with *auto_scale* or *num_spin_reversal_transforms* (see *extended_j_range* for more information).
- *per_qubit_coupling_range* defines the limits on coupling range permitted per qubit if you use *extended_j_range*.
- *maximum_number_of_variables* defines the maximum number of problem variables for hybrid solvers.
- *maximum_number_of_biases* defines the maximum number of problem biases for hybrid solvers.
- *qubits* and *couplers* define the working graph of QPU solvers.

Interacts with Parameters

- *auto_scale* enables you to submit problems to QPU solvers with values outside *h_range* and *j_range* and have the system automatically scale them to fit.
- *flux_biases* compensates for biases introduced by strong negative couplings if you use *extended_j_range*.

Example

This example defines a 7-qubit problem in which q_1 and q_4 are biased with -1 and q_0, q_2, q_3, q_4, and q_6 are biased with $+1$. Qubits q_0 and q_6 have a ferromagnetic coupling between them with a strength of -1.

```
>>> h = [1, -1, 1, 1, -1, 1, 1]
>>> J = {(0, 6): -1}
```

2.5 label

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC, Emulators

Problem label you can optionally tag submissions with. You can set as a label a non-empty string of up to 1024 Windows-1252 characters that has meaning to you or is generated by your application, which can help you identify your problem submission. You can see this label on the Leap dashboard and in solutions returned from SAPI.

Example

This example submits a simple Ising problem with a label and shows how such labels are displayed on the Leap dashboard.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
...
>>> sampleset = sampler.sample_ising({}, {('a', 'b'): -.05},
...                                  label="Test Ising Problem 1")
>>> print(sampleset.info["problem_label"])
Test Ising Problem 1
```

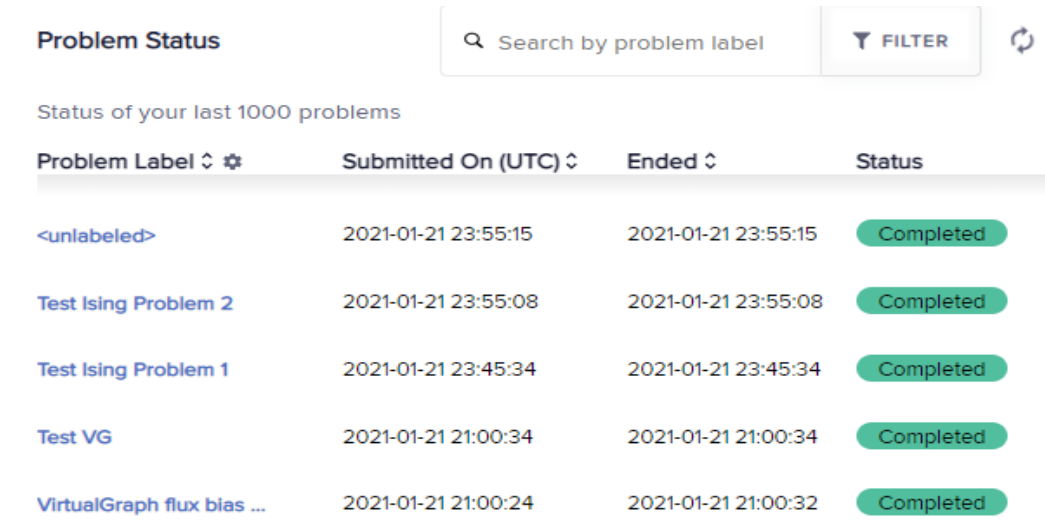

Figure 2.1: Problem labels on the Leap dashboard.

2.6 Q

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑ BQM	☑	☑ VFYC, Emulators

A quadratic unconstrained binary optimization (QUBO) problem is defined using an upper-triangular matrix, **Q**, which is an $N \times N$ matrix of real coefficients, and **x**, a vector of binary variables. The diagonal entries of **Q** are the linear coefficients (analogous to h, in Ising problems). The nonzero off-diagonal terms are the quadratic coefficients that define the strength of the coupling between variables (analogous to J, in Ising problems).

Input may be full or sparse. Both upper- and lower-triangular values can be used; (i, j) and (j, i) entries are added together.

For QPU solvers

- If a Q value is assigned to a coupler not present, an exception is raised. Only entries indexed by working couplers may be nonzero.
- QUBO problems are converted to Ising format before they run on the solver. Ising format uses h (qubit bias) and J (coupling strength) to represent the problem; see also *h* and *J*.
- The programming cycle programs the solver to the specified h and J values for a given Ising problem (or derived from the specified Q values of a given QUBO problem). However, since QPU precision is limited, the h and J values realized on the solver may deviate slightly from the requested (or derived) values. For more information, see the *QPU Solver Datasheet* guide.

If you are submitting directly through SAPI's REST API, see Data Encoding and the *SAPI REST Developers Guide* for more information.

Default is zero linear and quadratic biases.

Relevant Properties

- *h_range* and *j_range* define the supported ranges for QPU solvers. Be aware that problems with values outside the supported range will, by default, be scaled down to fit within the supported range; see the *auto_scale* parameter for more information.
- *maximum_number_of_variables* defines the maximum number of problem variables for hybrid solvers.
- *maximum_number_of_biases* defines the maximum number of problem biases for hybrid solvers.
- *qubits* and *couplers* define the working graph of QPU solvers.

Interacts with Parameters

- *extended_j_range* defines an extended range of values possible for the coupling strengths for some QPU solvers; you cannot use the extended J range with *auto_scale* or *num_spin_reversal_transforms* (see *extended_j_range* for more information).
- *per_qubit_coupling_range* defines the limits on coupling range permitted per qubit if you use *extended_j_range*.
- *auto_scale* enables you to submit problems to QPU solvers with values outside *h_range* and *j_range* and have the system automatically scale them to fit. You cannot use *auto_scale* with the *extended_j_range*.

Example

This example submits a QUBO to a QPU solver.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
...
>>> Q = {(0, 0): -3, (1, 1): -1, (0, 1): 2, (2, 2): -1, (0, 2): 2}
>>> sampleset = EmbeddingComposite(DWaveSampler()).sample_qubo(Q, num_
↪reads=100)
```

2.7 solver

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC, Emulators

The solver name, as a string in the JSON input data.

If you are submitting directly through SAPI's REST API, see the *SAPI REST Developers Guide* for more information; Ocean software allows you to explicitly select a solver and to automatically select a solver based on your required features.

Example

This example selects a Leap hybrid BQM solver with `binary_quadratic_model` in its name.

```
>>> from dwave.cloud import Client
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 
↪3}
>>> with Client.from_config(solver={'name__regex': '.*binary_quadratic_
↪model.*'}) as client:
...     solver = client.get_solver()
...     sampleset = solver.sample_qubo(Q, time_limit=5)
```

3 Solver Properties

The properties that characterize behaviors and supported features of SAPI solvers[1] can be queried through SAPI. This chapter defines these properties[2].

The examples below use the following setup:

```
>>> from dwave.system import DWaveSampler, LeapHybridSampler
...
>>> qpu_advantage = DWaveSampler(solver={'topology__type': 'pegasus'})
>>> qpu_2000q = DWaveSampler(solver={'topology__type': 'chimera'})
>>> hybrid_bqm_sampler = LeapHybridSampler()
```


- *anneal_offset_ranges*
- *anneal_offset_step*
- *anneal_offset_step_phi0*
- *annealing_time_range*
- *beta_range*
- *category*
- *chip_id*
- *couplers*
- *default_annealing_time*
- *default_beta*
- *default_programming_thermalization*
- *default_readout_thermalization*
- *extended_j_range*
- *h_gain_schedule_range*
- *h_range*
- *j_range*
- *max_anneal_schedule_points*
- *max_h_gain_schedule_points*
- *maximum_number_of_biases*
- *maximum_number_of_variables*
- *maximum_time_limit_hrs*
- *minimum_time_limit*

[1] Ocean software also provides classical solvers such as dwave-greedy that you can run locally. See the Ocean documentation for information on parameters of such solvers.

[2] Ocean software classes such as the `DWaveSampler` class may include various other properties that are derived and stored client-side, not read from SAPI in the `properties` dict; for example, the `dwave.cloud.solver.BaseSolver` might derive an `avg_load` property. See the Ocean documentation for those properties.

- *num_qubits*
- *num_reads_range*
- *parameters*
- *per_qubit_coupling_range*
- *problem_run_duration_range*
- *programming_thermalization_range*
- *quota_conversion_rate*
- *qubits*
- *readout_thermalization_range*
- *supported_problem_types*
- *tags*
- *topology*
- *version*
- *vfyc*

3.1 anneal_offset_ranges

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Array of ranges of valid anneal offset values, in normalized offset units, for each qubit. The negative values represent the largest number of normalized offset units by which a qubit's anneal path may be delayed. The positive values represent the largest number of normalized offset units by which a qubit's anneal path may be advanced.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["anneal_offset_ranges"][50:55]
[[-0.6437569799443247, 0.5093937328282595],
 [-0.6409899716199288, 0.5697835920492209],
 [-0.6468663649068287, 0.5211452841036136],
 [-0.6435316809914349, 0.5011584701861175],
 [-0.6500326191157569, 0.5392206819650301]]
```

3.2 anneal_offset_step

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Quantization step size of anneal offset values in normalized units.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["anneal_offset_step"]
-0.00017565852000668507
```

3.3 anneal_offset_step_phi0

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Quantization step size in physical units (annealing flux-bias units): Φ_0.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["anneal_offset_step_phi0"]
1.486239425109832e-05
```

3.4 annealing_time_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of time, in microseconds with a resolution of 0.01 μs, possible for one anneal (read). The lower limit in this range is the fastest quench possible for this solver.

Default annealing time is specified by the *default_annealing_time* property.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["annealing_time_range"]
[1.0, 2000.0]
```

3.5 beta_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☒	☑ Sampling Emulator

Range of values for *beta*.

Default β is specified by the *default_beta* property.

3.6 category

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC, Emulators

Type of solver; for example,

- `qpu`—quantum computers such as the Advantage.
- `hybrid`—quantum-classical hybrid; typically one or more classical algorithms run on the problem while outsourcing to a quantum processing unit (QPU) parts of the problem where it benefits most.
- `software`—emulators such as a sampling emulator.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["category"]
'hybrid'
```

3.7 chip_id

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Name of the solver.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["chip_id"]
'Advantage_system1.1'
```

3.8 couplers

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Couplers in the working graph. A coupler contains two elements [*q1*, *q2*], where both *q1* and *q2* appear in the list of working qubits, in the range [0, *num_qubits* - 1] and in ascending order (i.e., *q1* < *q2*). These are the couplers that can be programmed with nonzero *J* values.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["couplers"][:5]
[[30, 31], [31, 32], [32, 33], [33, 34], [34, 35]]
```

3.9 default_annealing_time

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Default time, in microseconds with a resolution of 0.01 μs, for one anneal (read). You can change the annealing time for a given problem by using the *annealing_time* or *anneal_schedule* parameters, but do not exceed the upper limit given by the *annealing_time_range* property.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["default_annealing_time"]
20.0
```

3.10 default_beta

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☒	☑ Sampling Emulator

Default value for β.

3.11 default_programming_thermalization

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Default time, in microseconds with a resolution of 0.01 μs, that the system waits after programming the QPU for it to return to base temperature. This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

You can change this value using the *programming_thermalization* parameter, but be aware that values lower than the default accelerate solving at the expense of solution quality.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["default_programming_thermalization"]
1000.0
```

3.12 default_readout_thermalization

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Default time, in microseconds with a resolution of 0.01 μs, that the system waits after each state is read from the QPU for it to cool back to base temperature. This value contributes to the *qpu_delay_time_per_sample* field, which is returned by SAPI with the problem solutions.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["default_readout_thermalization"]
0.0
```

3.13 extended_j_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Extended range of values possible for the coupling strengths (quadratic coefficients), J, for this solver. Strong negative couplings may be necessary for some embeddings; however, such chains may require additional calibration through the *flux_biases* parameter to compensate for biases introduced by strong negative couplings. See also *per_qubit_coupling_range*.

extended J ranges and flux-bias offsets cannot be used with autoscaling (*auto_scale*) or spin-reversal transforms (*num_spin_reversal_transforms*).

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["extended_j_range"]
[-2.0, 1.0]
```

3.14 h_gain_schedule_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of the time-dependent gain applied to qubit biases for this solver.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["h_gain_schedule_range"]
[-4.0, 4.0]
```

3.15 h_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of values possible for the qubit biases (linear coefficients), h, for this solver.

The *auto_scale* parameter, which rescales h and J values in the problem to use as much of the range of h (*h_range*) and the range of J (*j_range*) as possible, enables you to submit problems with values outside these ranges and have the system automatically scale them to fit.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["h_range"]
[-2.0, 2.0]
```

3.16 j_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of values possible for the coupling strengths (quadratic coefficients), J, for this solver.

The *auto_scale* parameter, which rescales h and J values in the problem to use as much of the range of h (*h_range*) and the range of J (*j_range*) as possible, enables you to submit problems with values outside these ranges and have the system automatically scale them to fit.

See also *extended_j_range*.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["j_range"]
[-1.0, 1.0]
```

3.17 max_anneal_schedule_points

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Maximum number of points permitted in a PWL waveform submitted to change the default anneal schedule.

For reverse annealing, the maximum number of points allowed is one more than the number given in the *max_anneal_schedule_points* property.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["max_anneal_schedule_points"]
12
```

3.18 max_h_gain_schedule_points

Supported Solvers:	Leap's Hybrid	QPU	Other
	✗	✓	✓ VFYC

Maximum number of points permitted in a PWL waveform submitted to set a time-dependent gain on linear coefficients (qubit biases, see the *h* parameter) in the Hamiltonian.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["max_h_gain_schedule_points"]
20
```

3.19 maximum_number_of_biases

Supported Solvers:	Leap's Hybrid	QPU	Other
	✓	✗	✗

Maximum number of biases, both linear and quadratic in total, accepted by the solver.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["maximum_number_of_biases"]
200000000
```

3.20 maximum_number_of_variables

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☒	☒

Maximum number of problem variables accepted by the solver.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["maximum_number_of_variables"]
1000000
```

3.21 maximum_time_limit_hrs

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☒	☒

Maximum allowed run time, in hours, that can be specified for the solver.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["maximum_time_limit_hrs"]
24.0
```

3.22 minimum_time_limit

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☒	☒

Minimum required run time, in seconds, the solver must be allowed to work on the given problem. Specifies the minimum time required for the given problem, as a piecewise-linear curve defined by a set of floating-point pairs. The second element is the minimum

required time; the first element in each pair is some measure of the problem, dependent on the solver:

- For hybrid BQM solvers, this is the number of variables.
- For hybrid DQM solvers, this is a combination of the numbers of interactions, variables, and cases that reflects the "density" of connectivity between the problem's variables.

The minimum time for any particular problem is a linear interpolation calculated on two pairs that represent the relevant range for the given measure of the problem. For example, if `minimum_time_limit` for a hybrid BQM solver were `[[1, 0.1], [100, 10.0], [1000, 20.0]]`, then the minimum time for a 50-variable problem would be 5 seconds, the linear interpolation of the first two pairs that represent problems with between 1 to 100 variables.

For more details, see the Ocean samplers documentation for solver methods that calculate this parameter, and their descriptions.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["minimum_time_limit"]
[[1, 3.0],
 [1024, 3.0],
 [4096, 10.0],
 [10000, 40.0],
 [30000, 200.0],
 [100000, 600.0],
 [1000000, 600.0]]
```

3.23 num_qubits

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Total number of qubits, both working and nonworking, in the QPU.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["num_qubits"]
5760
```

3.24 num_reads_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Range of values possible for the number of reads that you can request for a problem.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["num_reads_range"]
[1, 10000]
```

3.25 parameters

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC, Emulators

List of the parameters supported for the solver.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["parameters"]
{'time_limit': 'Maximum requested runtime in seconds.'}
```

3.26 per_qubit_coupling_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Coupling range permitted per qubit for this solver.

Strong negative couplings may be necessary for some embeddings, and can be enabled by using the *extended_j_range* range of J values. However, the total couplings for a qubit must not exceed the range specified by this property. Also, chains may require additional calibration through the *flux_biases* parameter to compensate for biases introduced by strong negative couplings.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["per_qubit_coupling_range"]
[-18.0, 15.0]
```

3.27 problem_run_duration_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of time, in microseconds with a resolution of 0.01 μs, that a submitted problem is allowed to run.

The run-duration for a given problem, which is compared with this range, is calculated according to the following formula:

$$Duration = ((P_1 + P_2) * P_3) + P_4 \tag{3.1}$$

where P_1, P_2, P_3, and P_4 are the values specified for the *annealing_time* (or anneal_schedule), *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* parameters, respectively.

Note: This is not the actual QPU access time of a given problem as that includes programming time and readout time. For more information, see the *QPU Solver Datasheet* guide.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["problem_run_duration_range"]
[0.0, 1000000.0]
```

3.28 programming_thermalization_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of time, in microseconds with a resolution of 0.01 μs, possible for the system to wait after programming the QPU for it to cool back to base temperature. This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

You can change this value using the *programming_thermalization* parameter, but be aware that values lower than the default accelerate solving at the expense of solution quality.

Default value for a solver is given in the *default_programming_thermalization* property.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["programming_thermalization_range"]
[0.0, 10000.0]
```

3.29 quota_conversion_rate

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC

Rate at which user or project quota is consumed for the solver as a ratio to QPU solver usage. Different solver types may consume quota at different rates.

Time is deducted from your quota according to:

$$\frac{num\_seconds}{quota\_conversion\_rate}$$

See the *QPU Solver Datasheet* guide for more information.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["quota_conversion_rate"]
1
```

3.30 qubits

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Indices of the working qubits in the working graph.

Note: In a D-Wave QPU, the set of qubits and couplers that are available for computation is known as the *working graph*. The yield of a working graph is typically less than the total number of qubits and couplers that are fabricated and physically present in the QPU.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["qubits"][:5]
[30, 31, 32, 33, 34]
```

3.31 readout_thermalization_range

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Range of time, in microseconds with a resolution of 0.01 μs, possible for the system to wait after each state is read from the QPU for it to cool back to base temperature. This value contributes to the *qpu_delay_time_per_sample* field, which is returned by SAPI with the problem solutions.

Default readout thermalization time is specified in the *default_readout_thermalization* property.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["readout_thermalization_range"]
[0.0, 10000.0]
```

3.32 supported_problem_types

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☑	☑ VFYC, Emulators

Indicates what problem types are supported for the solver.

QPU and QPU-like solvers support the following energy-minimization problem types:

- `qubo`—Quadratic unconstrained binary optimization (QUBO) problems; use 0/1-valued variables.
- `ising`—Ising model problems; use −1/1-valued variables.

Hybrid solvers support the following energy-minimization problem types:

- `bqm`—binary quadratic model (BQM) problems; use 0/1-valued variables and −1/1-valued variables.
- `dqm`—discrete quadratic model (DQM) problems; use variables that can represent a set of values such as `{red, green, blue, yellow}` or `{3.2, 67}`.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["supported_problem_types"]
['ising', 'qubo']
```

3.33 tags

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

May hold attributes about a solver that you can use to have a client program choose one solver over another.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_2000q.properties["tags"]
['lower_noise']
```

3.34 topology

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Indicates the topology `type` (`chimera` or `pegasus`) and `shape` of the QPU graph.

Example

The topology seen in this example is a C16 Chimera graph, meaning that the D-Wave 2000Q QPU has 16 x 16 blocks of Chimera unit cells, and each unit cell has K4,4 connectivity.

For the solver configured in the *example setup* above,

```
>>> qpu_2000q.properties["topology"]
{'type': 'chimera', 'shape': [16, 16, 4]}
```

3.35 version

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☒	☒

Version number of the solver.

Example

For the solver configured in the *example setup* above,

```
>>> hybrid_bqm_sampler.properties["version"]
'2.0'
```

3.36 vfyc

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Flag indicating whether this solver is a VFYC solver.

Example

For the solver configured in the *example setup* above,

```
>>> qpu_advantage.properties["vfyc"]
False
```

4 Solver Parameters

The quantum machine instructions (QMI) you send to D-Wave QPU solvers comprise *problem parameters* (e.g., linear biases, `h`, of an Ising problem) and solver parameters that control how the problem is run (e.g., an annealing schedule, *anneal_schedule*). Likewise, Leap hybrid solvers accept parameters that control how the problem is run.

This chapter describes, in alphabetical order, the solver parameters[1] accepted by SAPI solvers[2].

- *anneal_offsets*
- *anneal_schedule*
- *annealing_time*
- *answer_mode*
- *auto_scale*
- *beta*
- *chains*
- *flux_biases*
- *flux_drift_compensation*
- *h_gain_schedule*
- *initial_state*
- *max_answers*
- *num_reads*
- *num_spin_reversal_transforms*
- *postprocess*
- *programming_thermalization*
- *readout_thermalization*
- *reduce_intersample_correlation*
- *reinitialize_state*
- *time_limit*

[1] Ocean software tools include various other parameters that are processed client-side, not sent on to SAPI; for example, the `DWaveSampler` class might accept a `retry_interval` to configure the time its failover routine waits for a solver if no solver is found. See the Ocean documentation for those parameters.

[2] Ocean software also provides classical solvers such as dwave-greedy that you can run locally. See the Ocean documentation for information on parameters of such solvers.

4.1 anneal_offsets

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Provides offsets to annealing paths, per qubit.

Provide an array of anneal offset values, in normalized offset units, for all qubits, working or not. Use 0 for no offset. Negative values produce a negative offset (qubits are annealed *after* the standard annealing trajectory); positive values produce a positive offset (qubits are annealed *before* the standard trajectory). Before using this parameter, query the solver properties to see whether the anneal_offset_ranges property exists and, if so, to obtain the permitted offset values per qubit.

Default is no offsets.

Relevant Properties

- *anneal_offset_ranges* defines the ranges of valid anneal offset values.
- *anneal_offset_step* and *anneal_offset_step_phi0* define the quantization steps.

Example

This example offsets the anneal of a qubit in a two-qubit illustrative Ising problem.

```
>>> from dwave.system import FixedEmbeddingComposite, DWaveSampler
>>> qpu = DWaveSampler(solver={'topology__type': 'pegasus'})
>>> J = {(1, 2): -1}
>>> embedding = {1: [30], 2: [2940]}
>>> print(qpu.properties['anneal_offset_ranges'][2940])
[-0.7012257815714587, 0.6717794151250857]
>>> sampler = FixedEmbeddingComposite(qpu, embedding)
>>> offset = [0]*qpu.properties['num_qubits']
>>> offset[2940]=0.2
>>> sampleset = FixedEmbeddingComposite(qpu, embedding).sample_ising({},␣
 ↪J,
...                                         num_reads=1000, anneal_
 ↪offsets=offset)
```

The D-Wave system used for this example is an Advantage that has couplers between active qubits 30 and 2940. Select a suitable embedding for the QPU you run examples on.

4.2 anneal_schedule

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Introduces variations to the global anneal schedule. For a reverse anneal, use the *anneal_schedule* parameter with the *initial_state* and *reinitialize_state* parameters.

An anneal schedule variation is defined by a series of pairs of floating-point numbers identifying points in the schedule at which to change slope. The first element in the pair is time t in microseconds with a granularity of 0.01 μs for Advantage and 0.02 μs for D-Wave 2000Q systems; the second, normalized persistent current s in the range [0,1]. The resulting schedule is the piecewise-linear curve that connects the provided points.

The following rules apply to the set of anneal schedule points provided:

- Time t must increase for all points in the schedule.
- For forward annealing, the first point must be $(0, 0)$.
- For reverse annealing, the anneal fraction s must start and end at $s = 1$.
- In the final point, anneal fraction s must equal 1 and time t must not exceed the maximum value in the annealing_time_range property.
- The number of points must be ≥ 2.
- The upper bound is system-dependent; check the max_anneal_schedule_points property. For reverse annealing, the maximum number of points allowed is one *more* than the number given by this property.
- Additional rules that govern maximum slope vary by product; check the QPU properties document for your system.

Default anneal schedules are described in the QPU-specific anneal schedules documents.

Relevant Properties

- *max_anneal_schedule_points* shows the maximum number of points permitted in an anneal schedule.
- *default_annealing_time* shows the default annealing time for the solver.
- *annealing_time_range* defines the limits of the allowable range for the anneal schedule.

Interacts with Parameters

- *annealing_time* and *anneal_schedule* parameters are mutually exclusive.
- *num_spin_reversal_transforms*: spin-reversal transforms are incompatible with reverse annealing.
- *anneal_schedule* (or *annealing_time*), *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.

Example

This illustrative example configures a reverse-anneal schedule on a random native problem.

```
>>> from dwave.system import DWaveSampler
>>> import random
>>> qpu = DWaveSampler()
>>> J = {coupler: random.choice([-1, 1]) for coupler in qpu.edgelist}
>>> initial = {qubit: random.randint(0, 1) for qubit in qpu.nodelist}
>>> reverse_schedule = [[0.0, 1.0], [5, 0.45], [99, 0.45], [100, 1.0]]
>>> reverse_anneal_params = dict(anneal_schedule=reverse_schedule,
...                              initial_state=initial,
...                              reinitialize_state=True)
>>> sampleset = qpu.sample_ising({}, J, num_reads=1000, **reverse_anneal_
↪params)
```

4.3 annealing_time

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Sets the duration, in microseconds with a resolution of 0.01 μs for Advantage and 0.02 μs for D-Wave 2000Q systems, of quantum annealing time, per read. This value populates the *qpu_anneal_time_per_sample* field returned in the *timing* structure. Supported values are positive floating-point numbers.

Default value is shown by *default_annealing_time*.

Relevant Properties

- *annealing_time_range* defines the supported range of valid times.

Interacts with Parameters

- *annealing_time* and *anneal_schedule* parameters are mutually exclusive. Configuring a value of `T` for *annealing_time* is equivalent to configuring `anneal_schedule=[[0, 0], [T, 1]]`.
- *annealing_time* (or *anneal_schedule*), *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.

Example

This illustrative example configures the anneal time on a random native problem.

```
>>> from dwave.system import DWaveSampler
>>> import random
>>> qpu = DWaveSampler()
>>> J = {coupler: random.choice([-1, 1]) for coupler in qpu.edgelist}
>>> long_time = qpu.properties["annealing_time_range"][1]
>>> sampleset = qpu.sample_ising({}, J, num_reads=10, annealing_
↪time=long_time)
```

4.4 answer_mode

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Indicates how answers are returned from the solver[3]. Supported values are,

- `raw`—Answers returned individually in the order they were read from the solver. Use this setting if the returned time sequences are an important part of the solution data.

  The answer contains two fields, *solutions* and *energies*. The *solutions* field is a list of lists; the inner lists all have length *num_qubits* and entries from $-1, +1$ for Ising problems or $0, 1$ for QUBO problems. Values of 3 denote unused or inactive qubits. The *energies* field contains the energy of each corresponding solution.
- `histogram`—Answers returned as a histogram sorted in order of increasing energies. Answers contain the *solutions* and *energies* fields, but solutions are unique and sorted in increasing-energy order. Duplicate answers are merged and include a *num_occurrences* field, which indicates how many times each solution appeared.

For optimizing emulator solvers, when *answer_mode* is `histogram`, the *num_occurrences* field contains all ones, except possibly for the lowest-energy solution. That first entry is set so that the sum of all entries is *num_reads*.

Default value is `histogram`.

[3] Ocean tools receive these answers from SAPI and process them. For example, if you submit a problem using Ocean's `EmbeddingComposite`, the answer is mapped from qubits to the logical variables of your problem.

Interacts with Parameters

- *num_reads* defines the number of reads.
- *max_answers* specifies the maximum number of answers.

Example

This illustrative example sets the answer format to `raw`.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> sampleset = EmbeddingComposite(DWaveSampler()).sample_ising({}, J,
...                                                 num_reads=100, answer_
 ↪mode='raw')
```

4.5 auto_scale

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Indicates whether h and J values are rescaled:

- `true`—h and J values in the problem are rescaled to use as much of the range of h (*h_range*) and the range of J (*j_range*) as possible. When enabled, h and J values need not lie within the solver's range of h and J, but must still be finite.
- `false`—h and J values in the problem are used as is. If the h and J values are outside the range of the solver, problem submission fails.

For problems that use the regular J range of a solver, this parameter is enabled by default. For problems that use the extended J range, it is disabled (and cannot be enabled while the extended range is in use). See also *j_range* and *extended_j_range*.

Auto-scaling works as follows. Each QPU has an allowed range of values for the biases and strengths of qubits and couplers. Unless you explicitly disable auto-scaling, the values defined in your problem are adjusted to fit the entire available range, by dividing them by a positive (non-zero) factor defined as:

$$\max\Big\{ \max\left(\frac{\max(h)}{\max(h\_range)}, 0\right), \max\left(\frac{\min(h)}{\min(h\_range)}, 0\right), \max\left(\frac{\max(J)}{\max(J\_range)}, 0\right), \max\left(\frac{\min(J)}{\min(J\_range)}, 0\right)\Big\}$$

Ocean software's samplers often have a *chain strength* parameter: because the QPU's qubits are sparsely connected, problem variables might be represented by more than one physical

qubit (a "chain" of qubits), strongly coupled so as to return the same value. Typically, chains are generated by minor-embedding tools such as Ocean's minorminer. Setting a value for chain strength determines the values set for the couplers used in forming these chains. When using auto-scaling, the J values of chain couplers are scaled together with the given or converted J values. Similarly, if you disable auto-scaling, any chain strength you specify must result in coupling values within the allowed range for the QPU.

Problems specified in QUBO form are always converted to Ising for the submitted QMIs. When using auto-scaling, the converted problem's h and J values are rescaled as described above. Note that bias values in the converted form, which have a dependency on the number of quadratic interactions in the QUBO, can be larger than the maximum bias of the original form. For example, the four-variable QUBO below, which has a maximum bias value of 2,

$$\begin{bmatrix} 2 & 2 & 1.5 & 2 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -1.0 \end{bmatrix}$$

when converted to an Ising model, has a bias with a value greater than 2.0, $h_1 = 2.375$, as shown below:

```
>>> import dimod
>>> Q = {(1, 1): 2, (2, 2): 1.5, (3, 3): -0.5, (4, 4): -1.0,
...      (1, 2): 2, (1, 3): 1.5, (1, 4): 2}
>>> dimod.qubo_to_ising(Q)
({1: 2.375, 2: 1.25, 3: 0.125, 4: 0.0}, {(1, 2): 0.5, (1, 3): 0.375, (1,
 ↪4): 0.5}, 2.375)
```

Default is to auto-scale problems.

Interacts with Parameters

- *auto_scale* cannot be used with *extended_j_range* and *flux_biases*.

Example

The example checks a QPU's range of h and J before submitting a two-variable Ising problem to a QPU. *auto_scale* is implicitly *True* for the `DWaveSampler` class, so the h and J values are automatically rescaled by $\frac{-3.6}{-2} = 1.8$.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler())
...
>>> sampler.child.properties['j_range']
[-1.0, 1.0]
>>> sampler.child.properties['h_range']
[-2.0, 2.0]
>>> h = {'a': -3.6, 'b': 2.3}
>>> J = {('a', 'b'): 1.5}
>>> sampleset = sampler.sample_ising(h, J)
```

4.6 beta

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q	☑ VFYC, Sampling Emulators

Provides a value for the Boltzmann distribution parameter. Used when sampling postprocessing is enabled on D-Wave 2000Q and earlier systems. Can be any finite float.

As in statistical mechanics, β represents inverse temperature: $1/(k_B T)$, where T is the thermodynamic temperature in kelvin and k_B is Boltzmann's constant. In the D-Wave software, postprocessing refines the returned solutions to target a Boltzmann distribution characterized by β, which is represented by a floating point number without units. When choosing a value for β, be aware that lower values result in samples less constrained to the lowest energy states. For more information on β and how it is used in the sampling postprocessing algorithm, see the *QPU Solver Datasheet* guide.

Default value is 3.0 for sampling emulators and 10.0 for the D-Wave 2000Q VFYC .

Relevant Properties

- *annealing_time_range* defines the supported range of valid times.
- ***default_beta*** **specifies the default value for sampling** emulators.
- *beta_range* specifies the supported range for sampling emulators.

Interacts with Parameters

- *postprocess* defines what type of postprocessing the system runs.

Example

This illustrative example sets `beta` on a DW-2000Q system.

```
>>> from dwave.system import DWaveSampler
>>> import random
>>> qpu = DWaveSampler(solver={'topology__type': 'chimera'})
>>> J = {coupler: random.choice([-1, 1]) for coupler in qpu.edgelist}
>>> sampleset = qpu.sample_ising({}, J, num_reads=10, postprocess=
 ↪'sampling',
...                               beta=100)
```

4.7 chains

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q	☑ VFYC

Defines which qubits represent the same logical variable. Used only when postprocessing is enabled on D-Wave 2000Q and earlier systems. Ensures that all qubits in the same chain have the same value within each sample.

Provide the indices of the qubits that are in a chain. Qubits in a chain must be connected, and no qubit may appear more than once.

Default is no chains.

Interacts with Parameters

- *postprocess* defines what type of postprocessing the system runs.

Example

This illustrative example provides the chains found by Ocean software's minorminer.

```
>>> from dwave.system import DWaveSampler, FixedEmbeddingComposite
>>> import minorminer
...
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> qpu = DWaveSampler(solver={'topology__type': 'chimera'})
>>> embedding = minorminer.find_embedding(J.keys(), qpu.edgelist)
>>> sampleset = FixedEmbeddingComposite(qpu, embedding).sample_ising({},␣
↪J,
...                                                 num_reads=10,
...                                                 postprocess='sampling',
...                                                 chains=list(embedding.
↪values()))
```

4.8 flux_biases

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

List of flux-bias offset values with which to calibrate a chain. Often required when using the extended J range to create a strongly coupled chain for certain embeddings, as described in the *QPU Solver Datasheet* guide.

Provide an array of flux-bias offset values, in normalized offset units[4], for all qubits, working or not. Use 0 for no offset.

Default is no flux-bias offsets.

[4] Flux-biases applied to the qubit body are in units of Φ_0.

Relevant Properties

- *extended_j_range* defines the extended range of values possible for the coupling strengths.
- *per_qubit_coupling_range* defines the coupling range permitted per qubit for this solver.
- *flux_drift_compensation* indicates whether the D-Wave system compensates for flux drift.

Interacts with Parameters

- Cannot be used with *auto_scale* or *num_spin_reversal_transforms*.

Example

This example sets a flux-bias value of a qubit in a two-qubit illustrative Ising problem.

```
>>> from dwave.system import FixedEmbeddingComposite, DWaveSampler
>>> qpu = DWaveSampler(solver={'topology__type': 'chimera'})
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> embedding = {'s1': [1444], 's2': [1441], 's3': [1445, 1443]}
>>> fb = [0]*qpu.properties['num_qubits']
>>> fb[1445] = 5*qpu.properties["anneal_offset_step_phi0"]
>>> fb[1443] = -2*qpu.properties["anneal_offset_step_phi0"]
>>> sampleset = FixedEmbeddingComposite(qpu, embedding).sample_ising({},␣
↪J,
...                num_reads=100, flux_drift_compensation=False, flux_
↪biases=fb)
```

The D-Wave system used for this example is a DW-2000Q that has particular couplers. Select a suitable embedding for the QPU you run examples on.

4.9 flux_drift_compensation

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Boolean flag indicating whether the D-Wave system compensates for flux drift. The procedure it follows to do so is described in detail in Appendix A of the *QPU Solver Datasheet* guide.

- `flux_drift_compensation=true`—Compensate for flux drift.
- `flux_drift_compensation=false`—Do not compensate for flux drift.

Default is to compensate for flux drift.

Interacts with Parameters

- *flux_biases* enables you to apply flux-bias offsets manually, which you may want to do If you disable this parameter.

Example

This example disables flux-drift compensation.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> sampleset = sampler.sample_ising({}, J, num_reads=100,
...                                  flux_drift_compensation=False)
```

4.10 h_gain_schedule

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Sets a time-dependent gain for linear coefficients (qubit biases, see the *h* parameter) in the Hamiltonian. This parameter enables you to specify the $g(t)$ function in,

$$\mathcal{H}_{ising} = -\frac{A(s)}{2}\left(\sum_i \hat{\sigma}_x^{(i)}\right) + \frac{B(s)}{2}\left(g(t)\sum_i h_i\hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j}\hat{\sigma}_z^{(i)}\hat{\sigma}_z^{(j)}\right) \tag{4.1}$$

where $\hat{\sigma}_{x,z}^{(i)}$ are Pauli matrices operating on a qubit q_i and h_i and $J_{i,j}$ are the qubit biases and coupling strengths.

This time-dependent gain, $g(t)$, is specified, similarly to the *anneal_schedule* parameter, by a series of pairs of floating-point numbers identifying points in the schedule at which to change the gain applied to *h*. The first element in the pair is time, t in microseconds with a resolution of 0.01 μs for Advantage and 0.02 μs for D-Wave 2000Q systems; the second, the unitless g in the range *h_gain_schedule_range*. The resulting time-dependent gain is the piecewise-linear curve that connects the provided points over the same range of times as the *anneal_schedule*.

The following rules apply to the set of gain points provided:

- Time t, in microseconds, must increase for all points in the schedule.
- The first point of time must be zero, $t = 0.0$.
- The last point of time must match the last time in the *anneal_schedule* or the *annealing_time*.
- The number of points must be ≥ 2.
- Additional rules that govern maximum slope (i.e., how quickly $g(t)$ can change) vary by product; check the QPU properties document for your system.

Default $g(t)$, when left unspecified, is 1, which can be explicitly coded as

```
h_gain_schedule=[[0,1],[t_final,1]]
```

where *t_final* is the requested annealing time.

Relevant Properties

- *h_gain_schedule_range* defines the range of the time-dependent gain values permitted for the solver.

Note: In conjunction with the *auto_scale* parameter, the *h_gain_schedule* parameter enables you to extend the range of your submitted problem's linear coefficients (*h*) beyond the advertised *h_range*. Such use is not recommended for standard problem solving: the QPU is calibrated for linearity only within the advertised *h_range* and *j_range*. Increased integrated control errors (ICE) are expected outside that range.

If you configure `auto_scale=False` when using this parameter, ensure that $\max_i(h\_gain * h_i)$ and $\min_i(h\_gain * h_i)$ are within *h_range*.

- *max_anneal_schedule_points* defines the maximum number of anneal-schedule points permitted.
- *max_h_gain_schedule_points* defines the maximum number of gain changes allowed.

Interacts with Parameters

- *h* defines the linear biases for the problem.
- *anneal_schedule* defines the anneal schedule.

Example

This illustrative example sets an h-gain schedule.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> h = {'s1': 1, 's2': 1.5, 's3': -0.75}
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> anneal_schedule = [[0.0, 0.0], [40.0, 0.4], [140.0, 0.4], [150, 1.0]]
>>> h_schedule = [[0.0, 1], [40.0, 1], [140.0, 2], [143.0, 0], [150, 0]]
>>> sampleset = sampler.sample_ising(h, J, num_reads=500,
...                                  anneal_schedule=anneal_schedule,
...                                  h_gain_schedule=h_schedule)
```

4.11 initial_state

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Initial state to which the system is set for reverse annealing. Specifies the initial classical state of all qubits.

Provide (*qubit*, *state*) pairs, where *qubit* is the qubit index, *i*, and *state* is:

- -1 or 1—Ising problems, active qubits
- 0 or 1—QUBO problems, active qubits
- 3—Unused or inactive qubits

Interacts with Parameters

- *anneal_schedule* defines the anneal schedule. When *initial_state* is provided, indicates that the requested anneal schedule change is a reverse anneal.
- *reinitialize_state* reinitializes for each anneal. Note that this impacts timing.
- *num_spin_reversal_transforms* is incompatible with reverse annealing.

Example

This illustrative example configures a reverse-anneal schedule on a random native problem.

```
>>> from dwave.system import DWaveSampler
>>> import random
>>> qpu = DWaveSampler()
>>> J = {coupler: random.choice([-1, 1]) for coupler in qpu.edgelist}
>>> initial = {qubit: random.randint(0, 1) for qubit in qpu.nodelist}
>>> reverse_schedule = [[0.0, 1.0], [5, 0.45], [99, 0.45], [100, 1.0]]
>>> reverse_anneal_params = dict(anneal_schedule=reverse_schedule,
...                              initial_state=initial,
...                              reinitialize_state=True)
>>> sampleset = qpu.sample_ising({}, J, num_reads=1000, **reverse_anneal_
 ↪params)
```

4.12 max_answers

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Specifies the maximum number of answers returned from the solver. Must be an integer > 0.

- If *answer_mode* is `histogram`—Total number of distinct answers. Because answers in this mode are sorted by energy, these are the best `max_answers` answers.
- If *answer_mode* is `raw`—Limits the returned values to the first *max_answers* of *num_reads* samples. In this mode, *max_answers* should never be more than *num_reads*.

Default value is *num_reads*.

Interacts with Parameters

- *num_reads* defines the maximum number of requested answers.
- *answer_mode* defines the answer mode.

Example

This illustrative example resulted in fewer samples than the configured *num_reads*.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> J = {('s1', 's2'): 0.5, ('s1', 's3'): 0.5, ('s2', 's3'): 0.5}
>>> sampleset = sampler.sample_ising({}, J, num_reads=1000,
...                                  max_answers=5)
>>> print(sampleset)
  s1 s2 s3 energy num_oc. chain_.
0 +1 -1 -1   -0.5     202     0.0
1 -1 -1 +1   -0.5     132     0.0
2 +1 -1 +1   -0.5     112     0.0
3 -1 +1 -1   -0.5     248     0.0
4 -1 +1 +1   -0.5      95     0.0
['SPIN', 5 rows, 789 samples, 3 variables]
```

4.13 num_reads

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC, Emulators

Indicates the number of states (output solutions) to read[5] from the solver. Must be a positive integer.

Default value is `1`.

[5] Terms synonymous to *reads* are *anneals* and *samples*.

Interacts with Parameters

- *max_answers* defines the maximum number of answers supported for the solver.
- *anneal_schedule* or *annealing_time*, *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.

Example

This illustrative example requests 1250 samples.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 3}
>>> sampleset = sampler.sample_qubo(Q, answer_mode='raw', num_reads=1250)
>>> len(sampleset)
1250
```

4.14 num_spin_reversal_transforms

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Specifies the number of spin-reversal transforms[6] to perform.

Applying a spin-reversal transform can improve results by reducing the impact of analog errors that may exist on the QPU. This technique works as follows: Given an n-variable Ising problem, we can select a random $g \in \{\pm 1\}^n$ and transform the problem via $h_i \mapsto h_i g_i$ and $J_{ij} \mapsto J_{ij} g_i g_j$. A spin-reversal transform does not alter the mathematical nature of the Ising problem. Solutions s of the original problem and s' of the transformed problem are related by $s'_i = s_i g_i$ and have identical energies. However, the sample statistics can be affected by the spin-reversal transform because the QPU is a physical object with asymmetries.

Spin-reversal transforms work correctly with postprocessing and chains. Majority voting happens on the original problem state, not on the transformed state.

Use this parameter to specify how many spin-reversal transforms to perform on the problem. Valid values range from 0 (do not transform the problem; the default value) to a value equal to but no larger than the num_reads specified. If you specify a nonzero value, the system divides the number of reads by the number of spin-reversal transforms to determine how many reads to take for each transform. For example, if the number of reads is 10 and the number of transforms is 2, then 5 reads use the first transform and 5 use the second.

Be aware that each transform reprograms the QPU; therefore, using more than 1 transform will increase the amount of time required to solve the problem. For more information about timing, see the *QPU Solver Datasheet* guide.

[6] Also known as *gauge transformations*.

Default is no spin-reversal transforms.

Interacts with Parameters

- *num_reads* defines the maximum number of requested answers, and this is the maximum allowed value of spin-reversal transforms.
- *flux_biases* and *extended_j_range* ranges are incompatible with spin-reversal transforms.

Example

This illustrative example executes 5 spin-reversal transforms.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'): 3}
>>> sampleset = sampler.sample_qubo(Q, num_reads=100,
...                                 num_spin_reversal_transforms=5)
```

4.15 postprocess

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q	☑ VFYC

Postprocessing optimization and sampling algorithms provide local improvements with minimal overhead to solutions obtained from the quantum processing unit (QPU).

Ocean software provides postprocessing tools, and you can optionally run postprocessing online on D-Wave 2000Q and earlier systems.

Defines what type of postprocessing the system runs online on raw solutions:

- "" (empty string)—No postprocessing (default). Note that if this option is selected for the VFYC solver, sampling postprocessing runs.
- `sampling`—Runs a short Markov-chain Monte Carlo (MCMC) algorithm with single bit-flips starting from each sample. The target probability distribution is a Boltzmann distribution at inverse temperature β.
- `optimization`—Performs a local search on each sample, stopping at a local minimum.

When postprocessing is enabled, qubits in the same chain, defined by the *chains* parameter, are first set to the same value by majority vote. Postprocessing is performed on the logical problem but qubit-level answers are returned. For more information about postprocessing, see the *QPU Solver Datasheet* guide.

For problems that use the VFYC solver, postprocessing always runs. If you do not choose a postprocessing option, sampling postprocessing runs. When sampling postprocessing runs, the default β value is 10.

Example

This illustrative example uses postprocessing on a DW-2000Q system.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler(solver={'topology__type':
 ↪'chimera'}))
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'):␣
 ↪3}
>>> sampleset = sampler.sample_qubo(Q, num_reads=10, postprocess=
 ↪'sampling')
```

4.16 programming_thermalization

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Sets the time, in microseconds with a resolution of 0.01 μs for Advantage and 0.02 μs for D-Wave 2000Q systems, to wait after programming the QPU for it to cool back to base temperature (i.e., post-programming thermalization time). Lower values accelerate solving at the expense of solution quality. Supported values are positive floating-point numbers. This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

Default value for a solver is given in the *default_programming_thermalization* property.

Relevant Properties

- *programming_thermalization_range* defines the range of allowed values.

Interacts with Parameters

- *anneal_schedule* or *annealing_time*, *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.

Example

This illustrative example sets a value of half the supported maximum.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'):␣
 ↪3}
>>> pt = int(sampler.child.properties["programming_thermalization_range
 ↪"][1]/2)
```

(continues on next page)

(continued from previous page)

```
>>> sampleset = sampler.sample_qubo(Q, num_reads=10,
...                                 programming_thermalization=pt)
```

4.17 readout_thermalization

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑	☑ VFYC

Sets the time, in microseconds with a resolution of 0.01 μs for Advantage and 0.02 μs for D-Wave 2000Q systems, to wait after each state is read from the QPU for it to cool back to base temperature (i.e., post-readout thermalization time). This value contributes to the *qpu_delay_time_per_sample* field, which is returned with the problem solutions. Supported values are positive floating-point numbers.

Default value for a solver is given in the *default_readout_thermalization* property.

Relevant Properties

- *readout_thermalization_range* defines the range of allowed values.

Interacts with Parameters

- *anneal_schedule* or *annealing_time*, *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.

Example

This illustrative example sets a value of half the supported maximum.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'):␣
 ↪3}
>>> rt = int(sampler.child.properties["readout_thermalization_range"][1]/
 ↪2)
>>> sampleset = sampler.sample_qubo(Q, num_reads=10,
...                                 readout_thermalization=rt)
```

4.18 reduce_intersample_correlation

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

Reduces sample-to-sample correlations caused by the spin-bath polarization effect[7] by adding a delay between reads.

Boolean flag indicating whether the system adds a delay.

- `reduce_intersample_correlation=true`—Adds delay.
- `reduce_intersample_correlation=false` (default)—Does not add delay.

Important: Enabling this parameter drastically increases problem run times. To avoid exceeding the maximum problem run time configured for your system, limit the number of reads when using this feature. For more information on timing, see the *QPU Solver Datasheet* guide.

Default is to not add delay between reads.

Example

This illustrative example configures a delay between reads.

```
>>> from dwave.system import EmbeddingComposite, DWaveSampler
>>> sampler = EmbeddingComposite(DWaveSampler())
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'):␣
 ↪3}
>>> sampleset = sampler.sample_qubo(Q, num_reads=10,
...                                 reduce_intersample_correlation=True)
```

4.19 reinitialize_state

Supported Solvers:	Leap's Hybrid	QPU	Other
	☒	☑ 2000Q, Advantage	☑ VFYC

When using the reverse annealing feature, you must supply the initial state to which the system is set; see the *initial_state* parameter. If multiple reads are requested in a single call to the Solver API, you have two options for the starting state of the system. These are controlled by the *reinitialize_state* Boolean parameter:

- `reinitialize_state=true`—Reinitialize to the specified initial state for every anneal-readout cycle. Each anneal begins from the state given in the *initial_state* parameter. The amount of time required to reinitialize varies by system; typical D-Wave 2000Q systems require between 100 and 600 microseconds for this operation.

[7] See the *QPU Solver Datasheet* for more information on this effect.

- `reinitialize_state=false`—Initialize only at the beginning, before the first anneal cycle. Each anneal (after the first) is initialized from the final state of the qubits after the previous cycle. Be aware that even if this parameter is disabled, reverse annealing adds a small amount of time ($\approx 10\ \mu s$) for each read.

See also *anneal_schedule*.

Default is to reinitialize to the specified initial state for every anneal in reverse-anneal submissions.

Interacts with Parameters

- *anneal_schedule* defines the anneal schedule.
- *anneal_schedule* or *annealing_time*, *readout_thermalization*, *num_reads* (samples), and *programming_thermalization* values taken together must meet the limitations specified in *problem_run_duration_range*.
- *num_spin_reversal_transforms* is incompatible with reverse annealing.

Example

This illustrative example configures a reverse-anneal schedule on a random native problem with each anneal initialized from the final state of the previous cycle.

```
>>> from dwave.system import DWaveSampler
>>> import random
>>> qpu = DWaveSampler()
>>> J = {coupler: random.choice([-1, 1]) for coupler in qpu.edgelist}
>>> initial = {qubit: random.randint(0, 1) for qubit in qpu.nodelist}
>>> reverse_schedule = [[0.0, 1.0], [5, 0.45], [99, 0.45], [100, 1.0]]
>>> reverse_anneal_params = dict(anneal_schedule=reverse_schedule,
...                              initial_state=initial,
...                              reinitialize_state=False)
>>> sampleset = qpu.sample_ising({}, J, num_reads=1000, **reverse_anneal_
↪params)
```

4.20 time_limit

Supported Solvers:	Leap's Hybrid	QPU	Other
	☑	☒	☒

Specifies the maximum run time, in seconds, the solver is allowed to work on the given problem. Can be a float or integer.

Default value is problem dependent.

Relevant Properties

- *minimum_time_limit* defines the range of supported values for the given problem.

Example

This illustrative example configures a time limit of 6 seconds.

```
>>> from dwave.system import LeapHybridSampler
>>> Q = {('x1', 'x2'): 1, ('x1', 'z'): -2, ('x2', 'z'): -2, ('z', 'z'):␣
 ↪3}
>>> sampleset = LeapHybridSampler().sample_qubo(Q, time_limit=6)
```