Note: below in this document, “class” is used as a synonym to any Java type (class, interface, enum, annotation, etc.). “Method” is either Java method or constructor. “Variable” is used as a synonym to any Java fields, parameters and local variables.

PR Management

- Does the PR have all the appropriate tags?

- **Design Review** - for changes that will be hard to undo after they appear in some Druid release, and/or changes that will have lasting consequences in the codebase. Examples:
    - Major architectural changes or API changes
    - Adding new or changing behaviour of existing query types (e. g. changing and algorithm behind some query type or changing from floating point to double precision)
    - Adding new or changing existing HTTP requests and responses (e. g. a new HTTP endpoint)
    - Adding new or changing existing interfaces for extensions (@PublicApi and @ExtensionPoint)
    - Adding new or changing existing server configuration (e. g. altering the behavior of a config property)
    - Adding new or changing existing emitted metrics
    - Other major changes

The PR description should succinctly, but completely list all public API elements (@PublicApi or @ExtensionPoint), configuration options, emitted metric names, HTTP endpoint paths and parameters that are added or changed in the PR. See also items 1 and 2 in the “Naming” section in this checklist.

- **Incompatible** - for changes that may change the behaviour of Druid clusters if a new version of Druid with the subject change is rolled out, unless some server or query configurations are altered, or break Druid extensions on the source level because extensions API is changed. See examples above, in the section describing the “Design Review” tag. All “Incompatible” PRs should be tagged “Design Review” too, but not vice versa: some “Design Review” issues, proposals and PRs may not be “Incompatible”.

- **Release Notes** - for important changes that should reflected in the next Druid’s version release notes. Critically those are changes that require some server or query configuration changes made by Druid cluster operators to preserve the former cluster behaviour, i. e. the majority of PRs tagged “Incompatible”. However some “Incompatible” PRs may not need to be tagged “Release Notes”, e. g. PRs that only change some extension APIs, because when building extensions with the newer Druid version the incompatibility will be discovered anyway.

Secondarily, PRs that add new features, improve performance or improve Druid cluster operation experience could also be tagged “Release Notes”.

- **Bug** / **Security** / **Feature** / **Performance** / **Refactoring** / **Improvement** - could be used to distinguish between types of changes. **Compatibility** tag also falls into this category, it's specifically for PRs that restore or improve compatibility with previous Druid versions if it was inadvertently broken, or for changes that ensure forward compatibility with future Druid versions, forseening specific changes that would otherwise break the compatibility.

- **Development Blocker** - for changes that need to be merged before some other PRs could even be published. "Development Blocker" PRs should be prioritized by reviewers, so that they could be merged as soon as possible, thus not blocking author's work.

Discipline & Leadership

1. For PR authors: do you self-review your PRs before publishing?

2. For PR authors: do you make an effort to ease life of reviewers as much as possible: create a well-written PR description, explain the design and/or the structure of your changes, point to key added or changed classes (if there are a lot of changed classes in the PR), link to prior issues and PRs addressing related things, etc?

3. For reviewers: do you actually review all code in the PR, that is, read and ensure you understand each line of added code?

4. For reviewers: do you make an effort to ease life of PR authors as much as possible? If there is some problem with the PR because apparently the author is unaware of some rule, or design consideration, or historical context, do you explain it and provide a link to a relevant discussion, or link to the code review checklist?

5. For both PR authors and reviewers: when you see that someone else reviewed the PR and left comments pointing to something that you didn't notice during your (self-) review, do you question how did the other reviewer spot the problems that you overlooked? Do you try to internalize this new knowledge so that you will notice similar problems yourself in your own future PRs and in other people's code?

Accumulate and share knowledge

6. For both PR authors and reviewers: if some non-trivial coding or design consideration or a bug pops up during review, do you check the rest of the codebase for not having similar problems? At least, do you create a ticket (an action item) to perform such check?

7. If during development or code review you discover something or crystallize some knowledge that should be useful for most other developers on the project, but, apparently, little or none of them have this knowledge already, do you share that knowledge via the community forum? Examples:

- You discover an API in some library the project already depends on which would be useful across the codebase, yet there are no uses of this API in the codebase yet.

- You notice a code or a design problem and realise that the problem follows some pattern and there may have already been other instances of this problem in the codebase, as well as similar problems may appear in the future. In addition to checking the codebase (see the previous item) and adding a corresponding item to the code review checklist (see the next item), it's important to announce the new checklist item in the community forum. An announcement is not needed only when the problem pattern could be reliably eradicated from the codebase using static analysis tools.

7. If during development or code review you notice a code or a design problem and realise that the problem follows some pattern, do you try to eliminate such problems in the whole codebase and ensure that similar problems don't appear in the future using static analysis tools? If it's impossible to create static analysis rules that would reliably catch all instances of the problem without a significant number of false positives, do you add an item to the code review checklist? Do you let other developers on the project know about this pattern (see the previous item)?

Note that if it's possible to create static analysis rule(s) that don't guarantee catching all instances of the problem pattern, but would catch a good portion of them (without producing many false positives) in practice, such static analysis rules should still be introduced in addition to an item in the code review checklist. For example, some problem patterns can be reliably spotted only using abstract syntax tree (AST) or data flow analysis of the code, but considering the project's code style and typical variable names some regular expressions (to be applied to the source code as plain text) might be created that would catch many instances of the problem pattern.

8. If any issue is recognized during the preparation of the PR of during the review, that is too big or unrelated to the PR so that it couldn't be fixed right in the PR, is an issue or a different PR created to address that issue?

- If the PR is somehow incomplete or some aspects of it should be improved and for some reason that couldn't be done in the PR itself, are the issues created and assigned to the PR author to address that?

- For reviewers: after the PR author claimed that your comment(s) are addressed and/or pushed a new commit to the branch, did you check that each comment is actually addressed in code? It happens that PR authors think that they address a comment but actually they forget, or they address the problem locally on their developer machines but forget to update the patch with corresponding changes.

- If while working on a patch you had to read through the code of some class or method to understand how does it work, did you take time to add documentation to it (or to improve existing documentation) so that future developers and readers don't have to spend as much time as you did?

- If while working on a patch you encountered some overly trickly expression or a part of logic, for example, a multi-line `if` condition with nested &&, || and negations, and have spent some time to understand how does it work, did you take time to simplify the code or and comments to it so that future developers and readers don't have to spend as much time as you did?

Complexity & Understandability

1. Is the PR designed well? Doesn't it add unneeded complexity, minus the unavoidable minimum, because some new functionality is added?

2. If the PR adds some feature or a complexity dimension to the system, is the structure/design of the system and the code changed so that it's the same as it would have been if the subject feature or the complexity dimension was there from the very beginning? Or the feature or the complexity dimension is merely added "on top" of the existing system design, possibly making it incoherent or overall suboptimal?

3. Are there alternative designs presented in the description of the PR? What are pros and cons of the alternative designs? How do they compare with the implemented design in terms of complexity, supportability, interference with other features, performance, etc?

4. If the PR adds some new functionality, that requires some code structures, even as small as a single method call, to be repeated in a lot of places, especially in implementations of some classes annotated @ExtensionPoint, is it possible to redesign the functionality so that the new code structures are centralized and new code shouldn't be repeated, especially if the absence of the new code is not caught on the compilation level and could be easily forgotten?

5. Instead of writing comments, is it possible to make it obvious in the code itself through method extraction (that allows to give a method a meaningful name), more elaborate variable names, or restructuring the code?

6. For reviewers: do you understand the code after the first reading? If not, ask the PR author to make the code more obvious (see above), restructure it, or (lastly) add more comments. Try to identify what exactly you was struggling to comprehend the code and ask the PR author to put that piece of information in the comments.

7. For every method: aren't there sizeable parts of the method that solve unrelated subproblems on a different level of abstraction? Could those parts be extracted into separate methods to make the subject methods cleaner, ease testing and facilitate code reuse? See [TAoRC Chapter 10] for more information.

Naming

1. Does the PR description include a complete list all public API elements (@PublicApi or @ExtensionPoint), configuration options, emitted metric names, HTTP endpoint paths and parameters, or any other named parts of the project API that are added or changed in the PR? For example, a PR description may have a section like the following: TODO example

2. For every configuration option, emitted metric name, HTTP endpoint path and parameter name, etc. listed in the PR desciption as per the previous item, is there additionally at least one alternative name presented in the PR description and explained why the name that appears in the PR is better than the alternative name?

3. Are the names of all classes, methods, variables, configuration options, etc. in this PR understandable, not abbreviated, and reflect the meaning or the purpose of their corresponding entities and concepts well? Are they specific enough? Couldn't they be understood wrong?

4. Aren't there clashes with terms already used in the codebase in different contexts? Aren't there clashes with terms that are commonly used outside of the project/codebase to point to something slightly different, and therefore readers may create false associations?

5. What terms are commonly used for the same things and contepts outside of the project/codebase? Doesn't the patch create new vocabulary needlessly?

6. If there are single letter variable names (including lambda parameters), are they understandable? Is it possible to make names a little longer to make code easier to read? This also applies to single letter part of names, like in "lastV" or "hMin".

7. Is it possible to give a short but more meaningful name to variables called "result", "res", "r", "ret", "retVal" or "retValue" that are later returned from a method?

8. Aren't there variables called just "tmp"? At very least, they should have additional suffix expressing the object type, for example, "tmpFile". The exception from this rule is code that swaps values between two variables or array/list elements, for example, in a sorting implementation.

9. If there nested loops, is it possible to give better names for loop variables than "i", "j" and "k"? Maybe call it "itemIndex" and "bucketIndex" ("item" and "bucket" are just examples here, assuming "items" and "buckets" are iterated in nested loops) so that mistakes in code like `buckets[itemIndex].someMethod(items[bucketIndex])` would be obvious?

10. Are all variables that contain values with units (of time, memory, etc) have units attached to their names? For example, "timeoutNs" rather than "timeout", "sizeBytes" rather than "size". See also item 12.3 in the concurrency checklist.

11. Aren't there variables called just "size"? This also applies to methods called like "size" and "getSize()". If the size is measured in memory units, they should be attached to the

member name: “sizeBytes”, “getSizeMegabytes”. If the size is the number of some entities, the variable (and methods) should better be called “numEntities” than “size”.

12. For variables and configuration options that have words “limit” or “threshold” in their names: is it possible to redefine them in terms of “min” or “max” of the limited things (e. g. “itemsLimit” -> “maxItems”), so that it’s clear whether the value of the variable or the configuration is within the limit or right outside of it?

13. “begin”/”end” from TAoRC

14. Aren’t there negated boolean variables, methods, configuration options? Often their names contain parts like “disable”, “dont”, “not”, and “no”, e. g. a configuration option “disableCache” (prefer “useCache” instead).

15. If the patch fixes some bugs that probably have happened due to bad API or naming (e. g. API misused, double negation of a boolean-returning method or a configuration option, etc), was it considered to change the API somehow or to rename things to make such mistakes less likely in the future?

16. Don’t methods that compute the return value rather than just read it from a field (i. e. plain getters) have “get” prefix? Such methods may give a false impression that they are plain getters. Users might call such methods repeatedly in loops (assuming they are cheap) instead of caching the returned value. See [TAoRC Chapter 3].

- Does each non-local use of lazy collections (Iterators.transform(), Lists.transform(), Collections2.transform(), and the related methods from Guava) and Streams has a purpose that is obvious from the code or explained in a comment? A non-local use is when a lazy collection or a Stream is returned from a method, or passed to a method or a constructor, or assigned to a field. A local use is when a lazy collection or a Stream doesn't escape a method body, preferably not even being set to a variable, such as in the idiomatic Java collection transformation pattern with streams: collection.stream().filter(...).map(...).collect(...)

- If some bugs have happened because of some methods with a lot of parameters of the same type so that some arguments are not passed in the right order, or a very large method body so that some local variables are confused, or generally complicated code, has some action been taken to reduce the complexity and therefore the probability of such bugs in the future?

- For PR authors: if during preparation of the PR you have stumbled upon something confusing, overly complex or non easily understandable, spent some time understanding that, have you added some Javadoc or simple comments in the right places to help future code readers?

- Are there explanations why specific constant values are chosen in the Javadoc comments for the constants? If there is no specific reason why some value is chosen, is that stated in the comment?

- If the code is deeply nested (this is called “arrow code”), is there a way to reduce it with “guard clauses” and extracting nested parts as methods and/or making extracting anonymous inner classes as proper inner classes? The latter has additional small advantage of being a subject of Error Prone’s ClassCanBeStatic check.

- Is it obvious what types lambda parameters have? If not, consider writing them explicitly:

```
myEntities.values().forEach(handle -> {
 …
});
=>
// If the lambda body is short `(MyHandleType h)` might also be acceptable in this case.
myEntities.values().forEach((MyHandleType handle) -> {
 …
});
```

Bottom line: **this is not a smartness contest. The dumber the code the better.**

Testing

1. Is the code unit tested?

2. If large parts of the logic are not unit-tested because it’s hard or impossible to do, is it possible to extract static utility methods or helper classes that are testable? See also item 7 in the “Complexity & Understandability” section.

3. An extension of the previous item: if it’s tedious to test some part of the system because a lot of scaffolding, setup code, dependency injection is required before anything can be tested, is it possible to simplify the design of the production code to make it more test-friendly? When it’s difficult to test some class or a subsystem, it may be an indicator that the class or the subsystem is too tightly coupled with other classes or subsystems. Is it possible to extract the logic that solves a more general problem as a class which is easily testable, and make the glue class with DI/coupling/configuration options reading to delegate to that more general class?

3. If applicable, e. g. a large new subsystem or a node type is added, is it checked through integration tests?

4. If some subsystem is refactored so that some benchmarks or tests become irrelevant or trivial, are they removed?

5. Aren’t there duplicated code in tests? Supporting methods should be extracted to eliminate duplication and boil down the code in the test methods to a handful of calls to

supporting methods with arguments that clearly represent the tested case. See [TAoRC Chapter 14] for more information.

Code

- Is String an appropriate type? Should enum or some domain-specific type be used instead of Strings?

- Does the code, added or changed in this PR, follow the Druid code style? Aren't there lines longer than 120 symbols? Aren't there method declarations, statements and expressions that are broken into multiple lines not in accordance with the Druid style?

- Aren't there method declarations, statements and expressions that could easily fit a single line, but are broken into multiple lines, for example

```
shortMethodName(
    a,
    b,
    c
);
```

?

- Are new fields, methods and inner classes, added to existing classes, placed well within them in accordance with the layout principle used in the corresponding class? For example, in some classes methods are arranged in the depth-first call order. In other classes, methods are grouped logically. In classes which are parallel primitive specializations (double/float/long), fields and methods should be in the same order.

- Are static fields and methods generally placed before instance fields, constructors and members?

- Are instance members placed in the following order: 1. fields 2. constructors 3. methods?

- Are parallel primitive specialization classes (double/float/long usually in Druid) kept absolutely identical except for the occurrences of the specialized type?

- Aren't there multiline ternary operators or multiline `if` conditions, that could be refactored into one or multiple simpler `if`s?

- Aren't there `Throwables.propagate()` calls? See Why we deprecated Throwables.propagate article from the Guava authors.

- Are static and instance fields that could be final made final?

- For PR authors: do you try to address warnings visible in your IDE (IntelliJ or Eclipse) so that there are as little as possible warnings in new code?

- Are empty method bodies have a comment like “intentionally left empty”, “nothing to do”, or “nothing to close” (in `close()` methods) so that readers could see that the empty body is not the result of forgetting filling IDE-generated method stub?

- Are non-capturing Comparator lambdas (possibly wrapped in Comprator.comparingInt(), comparingLong(), comparingDouble(), nullsFirst(), nullsLast(), reversed()) extracted as static final constants?

Concurrency

****Concurrency documentation****

- For every class, method and field that exhibits signs of being thread-safe, such as using `synchronized` keyword, `volatile` modifier on fields, using any classes from `java.util.concurrent.*` packages, or other concurrency primitives (such as `LifecycleLock`) or concurrent collections: does its Javadoc comment include ****justification for thread-safety**** and ****concurrent program flow documentation****?
  - Is it explained why a particular class, method or field has to be thread-safe?
  - Is it enumerated from what methods and in contexts of what threads (executors, thread pools) methods of a thread-safe class are called, and fields are accessed?

- For classes and methods annotated `@PublicApi` or `@ExtensionPoint`, is it mentioned in their Javadoc comments whether they are (or in case of `@ExtensionPoint`, should they be implemented as) ****immutable, thread-safe or not thread-safe****? For classes and methods that are or should be implemented as thread-safe, is it documented precisely with what other methods (or themselves) they could be called concurrently? See also “Item 82: Document thread safety” in “Effective Java”.

- Are `ConcurrentHashMap` and `ConcurrentSkipListMap` objects (if the collections actually need to be concurrent) stored in variables of `ConcurrentHashMap` or `ConcurrentSkipListMap` or ****`ConcurrentMap` type****, but not generic `Map`?

- Is ****`@GuardedBy`**** used? If accesses to some fields should be protected with some lock, are those fields annotated with `@GuardedBy`? Are private methods that are called from critical sections in other methods (or those other methods are `synchronized`) annotated with `@GuardedBy`?

****Thread-safety “duplication”****

- Isn’t there *thread-safety “duplication”*, such as each modifiable variable of a class is `volatile` or an atomic, and each collection field is a concurrent collection (e. g. `ConcurrentHashMap`), although all accesses to those fields are protected with a lock? There shouldn’t be any “extra” thread-safety, *there should be just enough of it*. Thread-safety duplication confuses readers because they may think that those extra thread-safety precautions are (or used to be) needed for something but will fail to find the purpose.

- The exception from this rule is the `volatile` modifier on the lazily initialized field in the recommended double-checked locking pattern. This `volatile` modifier is unnecessary when the lazily initialized object has at least one `final` field. But without a `volatile` modifier thread-safety could easily be broken by a change (removal of the `final` field) in the class of lazily initialized objects, however that class should not be aware of concurrency implications.

- Regarding *each* field with a **`volatile` modifier**: does it really need to be `volatile`? Is it possible to redesign the code so that the field doesn't need to be `volatile`, without compromising the performance? If it's not possible and the field has to be `volatile`, is it justified in the Javadoc comment of the field? Is it made clear in the comment why Java Memory Model semantics of `volatile` field reads and writes are required for the field?

Typical race conditions
- Aren't **`ConcurrentMap`s updated with multiple separate calls to `get()`, `put()` and `remove()`** instead of single calls to `compute()`/`computeIfAbsent()`/`computeIfPresent()`/`replace()`?

- Aren't non thread-safe collections such as `HashMap` and `ArrayList` **iterated outside of critical sections**, while they could be modified concurrently? It could happen when an `Iterable`, `Iterator` or `Stream` over the collection is returned from a method of a thread-safe class, even though this iterator or stream is created within a critical section. Note that this still applies to `ArrayList` even when elements could only be added to the end of the list (details).

- More generally, aren't non-trivial objects that could be mutated concurrently returned from getters on a thread-safe class? This is a frequent problem in **`Aggregator.get()` and `BufferAggregator.get()` on complex object aggregators**: see this issue for details.

- If there are multiple variables in a thread-safe class that are **updated at once, but have individual getters**, isn't there a race condition in the code that calls those getters? If there is, those variables should be made `final` fields in a separate POJO class, that is stored in a field or in an `AtomicReference` in the thread-safe class. Rather than exposing parts of the state via individual getters, a POJO is returned from a getter as a whole. See `FileSessionCredentialsProvider` for an example of this approach.

Avoiding intrinsic locks and increasing granularity
- Is it possible to use concurrent collections and/or utilities from `java.util.concurrent.*` and **avoid using `Object.wait()`/`notify()`/`notifyAll()`**? Almost always it's possible to redesign the code around concurrent collections and utilities so that it's *clearer* and *less error-prone* than equivalent logic implemented with `Object.wait()`/`notify()`/`notifyAll()`. See "Item 81: Prefer concurrency utilities to `wait` and `notify`" in "Effective Java" for more info.

- Instead of using locking and a `boolean` field, isn't it better to use **`LifecycleLock`**?

- Is it possible to **increase locking granularity and/or use `ReadWriteLock`** instead of simple locks to increase concurrency? If a class primarily updates data in some map, is it

possible to **make critical sections lambdas provided to `ConcurrentHashMap.compute()`**/`computeIfAbsent()`/`computeIfPresent()` method, so that they could enjoy effective per-key locking granularity? Otherwise, is it possible to use Guava's **`Striped`**?

- If at a `ConcurrentHashMap.computeIfPresent()` call site it's expected that a key is almost always absent in the map, was it considered to call `containsKey()` in front of `computeIfPresent()`? It helps to avoid locking for 31-53% of keys, depending on the load factor of a `ConcurrentHashMap`. Similarly, if `computeIfAbsent()` is preluded by `get()`, locking could be avoided for 10-30% of keys.

Lazy initialization & double-checked locking
- If something is initialized lazily, couldn't something bad happen if two concurrent threads do the initialization at the same time? If yes, double-checked locking should be added.

- If something is initialized lazily under a simple lock, shouldn't it use double-checked locking pattern instead to improve performance?

- If something is initialized lazily under a simple lock or using double-checked locking, does it really need to? If nothing bad could happen if two threads do the initialization at the same time, a benign race could be allowed (but the field should be `volatile` then to ensure there is a happens-before edge between initializing and reading threads).

- If there is double-checked locking, does it follow the specific pattern described in this comment?

- Doesn't net impact of double-checked locking and lazy field initialization on performance and complexity overweight the benefits of lazy initialization? Isn't it ultimately better to go with eager initialization in a particular case?

See also "Item 83: Use lazy initialization judiciously" in "Effective Java".

- When `Thread`s are created, are they given names and is `setDaemon(true)` called for them? The same about to `Executor`s and `ExecutorService`s: they should be created with thread factories that name threads and make them daemons. It's convenient to use static factory methods from `Execs` class.

Annotations

- Are the newly added classes and methods annotated as `@PublicApi` or `@ExtensionPoint`, if they need to? See the javadocs of those annotations.

- Are the newly added classes annotated with `@HotLoopCallee`, if they need to, and some of their methods with `@CalledFromHotLoop`? See the javadocs of those annotations.

- If the code of a method checks the parameters for nullability, except in order to throw an exception right away, i. e. something different from `Objects`/`Preconditions.requireNonNull(param)`, is it needed? Could nulls actually be passed as arguments? If they could, should the method actually treat them “normally”, or it’s better off to throw an NPE, by calling `Objects.requireNonNull()`? If nulls need to be treated “normally”, are the nullable parameters annotated `@Nullable`?

- If some method returns null in some cases, is it actually allowed to do so? Is the implemented method in the interface or a superclass annotated `@Nullable`? Are all clients of the method (or the implemented method in the supertype) prepared for null returned from this method? Is the method annotated `@Nullable` itself? Is it possible to design the method so that instead of null it returns something else, such as an empty collection?

Javadocs and Comments

- Does every top-level class has a Javadoc comment, unless that's a simple collection of properties with getters and setters, i. e. a config class or a class existing for RPC?
    - If the class is not a config class nor an RPC thing, but it's hard to write a Javadoc comment that's not obvious and doesn't just repeat the class name with spaces between the words, it might be a sign that the class shouldn't be a top-level class (if it's used in one place only). Maybe it should be a private inner or an anonymous class.

- Does the Javadoc comment for a class reference related classes and methods using @see tags?

- Do all public API methods (see @PublicApi and @ExtensionPoint annotations) which are not simple getters and setters, or inherited from conventional interfaces such as Closeable and Runnable, have a Javadoc comment?

- Do all comments to Java members, either public or private classes, methods and fields appear to be Javadoc comments, not ordinary Java's multiline or single-line (//) comments?

- Do all references to classes and methods in Javadoc comments appear as {@link} tags?

- Are existing Javadoc and ordinary comments updated to reflect changes happened in the code? Don't comments become out-of-date after the PR?

Documentation

- If the PR adds some new functionality, feature, config parameter, etc., is it described in the documentation?

- Is existing documentation updated to reflect changed happened in the code? Doesn’t documentation become out-of-date after this PR?

Automation

- If you have spotted a style violation, API misuse, bad pattern or something similar in the PR, is it possible to create an automatic rule to exclude such defects in the future using either checkstyle (codestyle/checkstyle.xml), or error-prone (pom.xml, "strict" profile definition), or error-level IntelliJ inspections (TODO: instruction about how to add an inspection), or forbidden-apis (codestyle/druid-forbidden-apis.txt)? If yes, ask the PR author to add that, or create a separate PR yourself, or at least create an issue describing what rule should be added.
- IntelliJ code style? Eclipse code style?