anatomy of a doobie query

`sql"SELECT a , b , c , d , e , f FROM … WHERE x = $g AND y < $h ".query[((A,B),C,(D,E,F))]`

A B C D E F
Get Get Get Get Get Get

G H
Put Put

`Get[A]` *defines the mapping between a JDBC/schema type and Scala type* `A`.

(A,B)
Read

C
Read

(D,E,F)
Read

G::H::HNil
Param

`Read[(A,B)]` *defines the mapping between a column vector and pairs* `(A,B)`, *derived from* `Get[A]` *and* `Get[B]`.

A `Read` *instance is required for the query's type argument, which means it must be a type that can be flattened into a column vector with* `Get` *instances for each data member.*

((A,B),C,(D,E,F))
Read

`Read` *instances compose implicitly, yielding mappings between underlying column vectors and increasingly structured data types.*

`Param[G::H::HNil]` *defines the mapping between a list of arguments and a underlying list of query parameters, derived from* `Put[G]` *and* `Put[H]`.
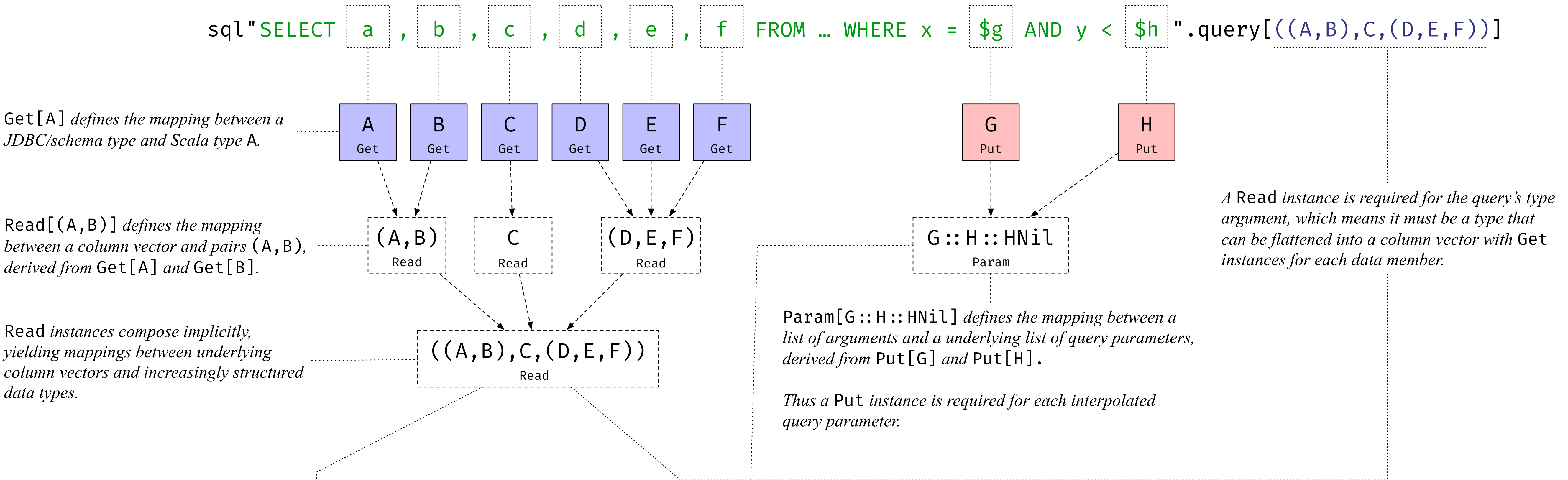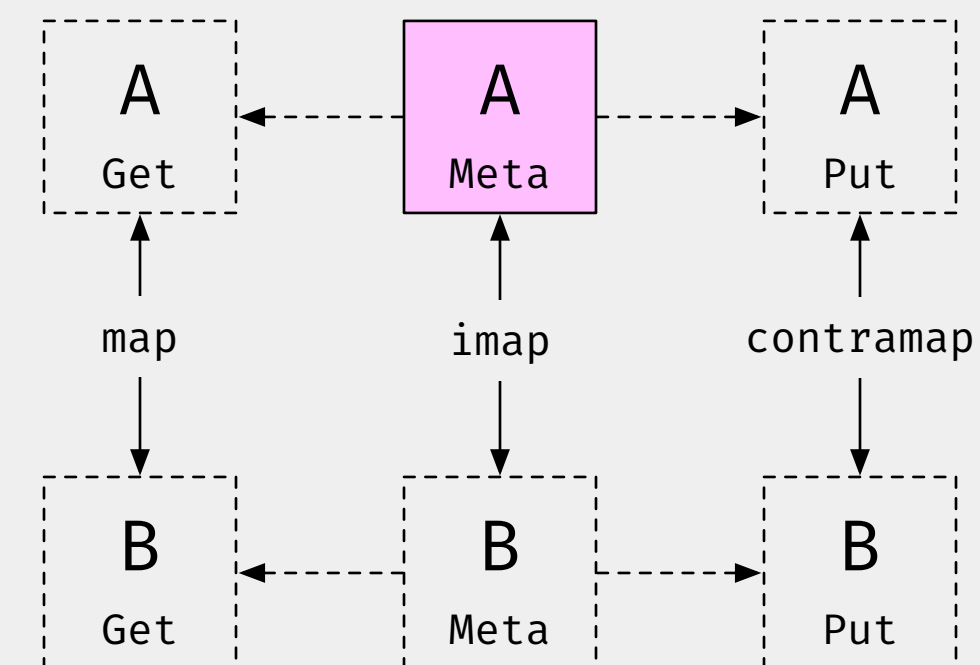
Thus a `Put` *instance is required for each interpolated query parameter.*

`Read` *instances are derived for all types with* `Get` *instances, as well as options thereof. Instances are then derived for tuples, HLists, and case classes whose members have* `Read` *instances.*

`Param` *instances are derived for any sequence of types (or options thereof) that have* `Put` *instances.*

A
Get

A
Meta

A
Put

map   imap   contramap

B
Get

B
Meta

B
Put

`Meta[A]` *is a convenience data type that gives rise to a* `Get`/`Put` *pair. Most base mappings are defined in terms of* `Meta`.

You can use `imap` *to derive a new* `Meta` *from an existing one;* `map` *to derive a new* `Get` *from an existing one; and* `contramap` *to derive a new* `Put` *from an existing one.*

`Get` *and* `Put` *instances are provided for all supported JDBC types, as well as some vendor-specific types. You can also define your own, either using low-level constructors or by mapping to and from an existing instance.*