Tests and Functionality

In this document we'll be listing the merging rules, keeping track of which test cases are implemented, and explaining the rationale behind the rules. If you need a playground to understand the status quo merging behavior, you can use the examples in the GraphQL schema docs. Valid tests do not throw errors and invalid tests do throw errors.

Overlapping Fields Tests

These rules and tests are largely derived from Young Kim's list.

Schema

Here is the schema being used for the following tests

```
interface SomeBox {
  deepBox: SomeBox
  unrelatedField: String
}

type StringBox implements SomeBox {
  scalar: String
  deepBox: StringBox
  unrelatedField: String
  listStringBox: [StringBox]
  stringBox: StringBox
  intBox: IntBox
}

type IntBox implements SomeBox {
  scalar: Int
  deepBox: IntBox
  unrelatedField: String
  listStringBox: [StringBox]
  stringBox: StringBox
  intBox: IntBox
}

interface NonNullStringBox1 {
  scalar: String!
}

type NonNullStringBox1Impl implements SomeBox & NonNullStringBox1 {
```

```
  scalar: String!
  unrelatedField: String
  deepBox: SomeBox
}

interface NonNullStringBox2 {
  scalar: String!
}

type NonNullStringBox2Impl implements SomeBox & NonNullStringBox2 {
  scalar: String!
  unrelatedField: String
  deepBox: SomeBox
}

type Connection {
  edges: [Edge]
}

type Edge {
  node: Node
}

type Node {
  id: ID
  name: String
}

type Query {
  someBox: SomeBox
  connection: Connection
}
```

1) Nullable field on an interface with aliases

In this case, SomeBox is an interface which IntBox implements. unrelatedField is a nullable String. If the someBox field returns an IntBox, then both inline fragments would be used. Giving each a different alias results in both fields in the response. If they were not aliased, GraphQL.js would attempt to merge them and they would clash on account of their different nullability. Since they are both aliased, both nullableField and nonNullableField can safely be included in the response.

However, if unrelatedField comes back null, then someBox would be null in the response. This is effectively the same result as if both fields were marked non-nullable.

```
{
  someBox {
    ...on SomeBox {
      nullableField: unrelatedField
    }
    ...on IntBox {
      nonNullableField: unrelatedField!
    }
  }
}
```

- [x] ~~? and ! is valid~~
- [x] ~~? and ? is valid~~
- [x] ~~! and ! is valid~~
- [x] ~~Unmarked and ! is valid~~
- [x] ~~unmarked and ? is valid~~
- [x] ~~Unmarked and unmarked is valid~~

2) Nullable field on an interface without aliases

Similar to the above, but no aliases in the query. Fields that would be merged with different nullabilities are rejected as invalid queries in this test set.

```
{
  someBox {
    ...on SomeBox {
      unrelatedField!
    }
    ...on IntBox {
      unrelatedField!
    }
  }
}
```

- [x] ~~? and ! is not valid~~
- [x] ~~? and ? is valid~~
- [x] ~~! and ! is valid~~

- ☑ ~~Unmarked and ! is not valid~~
- ☑ ~~unmarked and ? is valid~~
- ☑ ~~Unmarked and unmarked is valid~~

3) Non-nullable field on an interface with aliases

In this case, NonNullStringBox1 is an interface and NonNullStringBox1Impl implements NonNullStringBox1. scalar is a non-nullable String. The rules we're following are otherwise identical to test set 1.

```
{
  someBox {
    ...on NonNullStringBox1 {
      nonNullable: scalar!
    }
    ...on NonNullStringBox1Impl {
      nullable: scalar
    }
  }
}
```

- ☑ ~~? and ! is valid~~
- ☑ ~~? and ? is valid~~
- ☑ ~~! and ! is valid~~
- ☑ ~~Unmarked and ! is valid~~
- ☑ ~~Unmarked and ? is valid~~
- ☑ ~~Unmarked and unmarked is valid~~

4) Non-nullable field without aliases

Similar to the above, but no aliases in the query. Fields that would be merged with different nullabilities are rejected as invalid queries in this test set.

```
{
  someBox {
    ...on NonNullStringBox1 {
      scalar!
    }
```

```
    ...on NonNullStringBox1Impl {
      scalar!
    }
  }
}
```

- [x] ~~? and ! is not valid~~
- [x] ~~? and ? is valid~~
- [x] ~~! and ! is valid~~
- [x] ~~Unmarked and ! is valid~~
- [x] ~~unmarked and ? is not valid~~
- [x] ~~Unmarked and unmarked~~

5) Exclusive nullable types without aliases

In this test set, we're working with fields with differing types. IntBox.scalar is a nullable Int, and StringBox.scalar is a nullable String. None of these test cases are expected to be valid because Int and String are different types. This behavior is already partially covered by other tests. Here's an existing test for differing types and here's an existing test for differing schema nullability.

```
{
  someBox {
    ... on IntBox {
      scalar!
    }
    ... on StringBox {
      scalar!
    }
  }
}
```

- [x] ~~? and ! is not valid~~
- [x] ~~? and ? is not valid~~
- [x] ~~! and ! is not valid~~
- [x] ~~Unmarked and ! is not valid~~
- [x] ~~Unmarked and ? is not valid~~
- [x] ~~Unmarked and unmarked is not valid~~

6) Exclusive non-nullable types without aliases

This would be the same test set as 5, but if scalar was non-nullable. I'm going to skip this test set for now since I think it covers essentially the same cases as test set 5. However, if someone disagrees, I'll happily implement it.

Executor Tests

As it stands, when a field is marked required, it's treated exactly as if it was marked required in the schema. Essentially clients have the option to override the nullability of a field from the client side. The behavior is otherwise identical to the status quo. If null is found for a required field, then it "bubbles up" to the first nullable parent. I've included a few tests illustrating this below, but this should largely be the behavior we're all used to.

Schema and Response

This test and response are used for all of the following test cases

Schema

```
type Query {
    food: Food
 }

type Food {
  name: String
  calories: Int
}
```

Response

```
food {
  name: null
  calories: 10
}
```

1) Null bubbles up when field that returns null is required

```
query {
  food {
    name!
    calories
  }
}
```

Response

```
data: { food: null },
errors: [
  {
    locations: [{ column: 13, line: 4 }],
    message: 'Cannot return null for non-nullable field Food.name.',
    path: ['food', 'name'],
  },
]
```

2) Null bubbles up when field that returns null and field that does not are both required

```
query {
  food {
    name!
    calories!
  }
}
```

Response

```
data: { food: null },
errors: [
  {
    locations: [{ column: 13, line: 4 }],
    message: 'Cannot return null for non-nullable field Food.name.',
    path: ['food', 'name'],
  },
]
```

3) Null bubbles up when field that returns null is required, but other aliased value is unaffected

Some reviewers of the initial RFC were concerned that if a field on a fragment was marked required, then you risk wiping out all fields of any query that uses that fragment in the case that null is returned for the required field. I'm including this test to illustrate one possible workaround

where all fragments that include a required field alias their nearest parent to protect other parts of the query.
One example of how this could be used is in the case of Relay where components have associated fragments. Normally those fragments would merge their fields in with a larger query, but if they wanted to protect the parent fragment from being blown out by a required field, they could adopt this strategy. This way, a null would be quarantined to the single subcomponent that used that fragment.

```
query {
  nonNullable: food {
    name!
    calories!
  }
  nullable: food {
    name
    calories!
  }
}
```

Response

```
data: { nonNullable: null, nullable: { calories: 10, name: null } },
errors: [
  {
    locations: [{ column: 13, line: 4 }],
    message: 'Cannot return null for non-nullable field Food.name.',
    path: ['nonNullable', 'name'],
  },
]
```

There’s no way to do this in lists though because you can’t alias a fragment.