Pagination

> Different pagination models enable different client capabilities

A common use case in GraphQL is traversing the relationship between sets of objects. There are a number of different ways that these relationships can be exposed in GraphQL, giving a varying set of capabilities to the client developer.

Plurals

The simplest way to expose a connection between objects is with a field that returns a plural type. For example, if we wanted to get a list of R2-D2's friends, we could just ask for all of them:

```
{
  hero {
    name
    friends {
      name
    }
  }
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker
        },
        {
          "name": "Han Solo"
        },
```

Slicing

Quickly, though, we realize that there are additional behaviors a client might want. A client might want to be able to specify how many friends they want to fetch; maybe they only want the first two. So we'd want to expose something like:

```
{
  hero {
    name
    friends(first:2) {
      name
    }
  }
}
```

But if we just fetched the first two, we might want to paginate through the list as well; once the client fetches the first two friends, they might want to send a second request to ask for the next two friends. How can we enable that behavior?

Pagination and Edges

There are a number of ways we could do pagination:

- We could do something like `friends(first:2 offset:2)` to ask for the next two in the list.
- We could do something like `friends(first:2 after:$friendId)`, to ask for the next two after the last friend we fetched.
- We could do something like `friends(first:2 after:$friendCursor)`, where we get a cursor from the last item and use that to paginate.

In general, we've found that **cursor-based pagination** is the most powerful of those designed. Especially if the cursors are opaque, either offset or ID-based pagination can be implemented using cursor-based pagination (by making the cursor the offset or the ID), and using cursors gives additional flexibility if the pagination model changes in the future. As a reminder that the cursors are opaque and that their format should not be relied upon, we suggest base64 encoding them.

That leads us to a problem; though; how do we get the cursor from the object? We wouldn't want cursor to live on the `User` type; it's a property of the connection, not of the object. So we might want to introduce a new layer of indirection; our `friends` field should give us a list of edges, and an edge has both a cursor and the underlying node:

```
{
  hero {
    name
    friends(first:2) {
      edges {
        node {
          name
        }
        cursor
      }
    }
  }
}
```

The concept of an edge also proves useful if there is information that is specific to the edge, rather than to one of the objects. For example, if we wanted to expose "friendship time" in the API, having it live on the edge is a natural place to put it.

End-of-list, counts, and Connections

Now we have the ability to paginate through the connection using cursors, but how do we know when we reach the end of the connection? We have to keep querying until we get an empty list back, but we'd really like for the connection to tell us when we've reached the end so we don't need that additional request. Similarly, what if we want to know additional information about the connection itself; for example, how many total friends does R2-D2 have?

To solve both of these problems, our `friends` field can return a connection object. The connection object will then have a field for the edges, as well as other information (like total count and information about whether a next page exists). So our final query might look more like:

```
{
  hero {
    name
    friends(first:2) {
      totalCount
      edges {
        node {
          name
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

Note that we also might include `endCursor` and `startCursor` in this `PageInfo` object. This way, if we don't need any of the additional information that the edge contains, we don't need to query for the edges at all, since we got the cursors needed for pagination from `pageInfo`. This leads to a potential usability improvement for connections; instead of just exposing the `edges` list, we could also expose a dedicated list of just the nodes, to avoid a layer of indirection.

Complete Connection Model

Clearly, this is more complex than our original design of just having a plural! But by adopting this design, we've unlocked a number of capabilities for the client:

- The ability to paginate through the list.

- The ability to ask for information about the connection itself, like `totalCount` or `pageInfo`.
- The ability to ask for information about the edge itself, like `cursor` or `friendshipTime`.
- The ability to change how our backend does pagination, since the user just uses opaque cursors.

To see this in action, there's an additional field in the example schema, called `friendsConnection`, that exposes all of these concepts. You can check it out in the example query. Try removing the `after` parameter to `friendsConnection` to see how the pagination will be affected. Also, try replacing the `edges` field with the helper `friends` field on the connection, which lets you get directly to the list of friends without the additional edge layer of indirection, when that's appropriate for clients.

```
{
  hero {
    name
    friendsConnection(first:2 aft
      totalCount
      edges {
        node {
          name
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friendsConnection": {
        "totalCount": 3,
        "edges": [
          {
            "node": {
              "name": "Han Solo"
            },
            "cursor": "Y3Vyc29yMg
          },
          {
            "node": {
              "name": "Leia Organ
            },
            "cursor": "Y3Vyc29yMw
          }
        ],
        "pageInfo": {
          "endCursor": "Y3Vyc29yM
          "hasNextPage": false
```

Connection Specification

To ensure a consistent implementation of this pattern, the Relay project has a formal specification you can follow for building GraphQL APIs which use a cursor based connection pattern.