What are Schema Root Fields?

Schema Root Fields are the "entry point" of any GraphQL query. Gatsby generates two root fields for each `Node` type that gets created as a result of Schema Generation. Third-party schemas and the `createResolvers` API can add additional root fields on top of those.

The root fields that Gatsby generates are used to retrieve either one item of a certain type or many items of that type. For example, for type `BlogPost`, Gatsby will create `blogPost` and `allBlogPost` root fields.

While those fields can be used without any arguments, the additional power lies in the fact that they accept additional parameters to filter, sort or paginate the resulting data. As those parameters depend on a particular type that they are used for, Gatsby generates *Utility Types* to support those. Those types are used in the root fields to support filtering or sorting, as well as to return Paginated Data.

Plural root fields

Plural fields accept four arguments - `filter`, `sort`, `skip` and `limit`. `filter` allows filtering based on node field values (see Filter Types below), `sort` sorts the result (see Sorting types below). `skip` and `limit` offsets the result by `skip` nodes and limits it to `limit` items. Plural root fields return a Connection type (see Pagination Types below) for the returning type (for example, `BlogPostConnection`).

```
{
  allBlogPost(
    filter: { date: { lt: "2020-01-01" } }
    sort: { fields: [date], order: ASC }
  ) {
    nodes {
      id
    }
  }
}
```

Singular root fields

Singular root fields accept the same `filter` parameter as plural, but the `filter` is spread directly into arguments. Thus filter parameters need to be passed to the field directly. They return the resulting object directly.

If no parameters are passed, they return a random node of that type (if any exist). That random node type is explicitly undefined and there is no guarantee that it will be stable (or unstable) between builds or rebuilds. If no node was found according to the filter, null is returned.

```
{
  blogPost(id: { slug: "graphql-is-the-best" }) {
    id
  }
}
```

Pagination types

Gatsby uses a common pattern in GraphQL called Connections. This pattern has nothing to do with connecting anything, but rather is an abstraction over pagination. When you query a connection, you get a slice of the resulting data based on passed `skip` and `limit` parameters. It also allows doing additional operations on a list, like doing grouping or distinct operations.

- `edges` - edge is the actual Node object together with additional metadata regarding its location in the page. `edge` contains `node` - the actual object, and `next` / `prev` objects to get next or previous object from the current one.
- `nodes` - a flat list of Node objects
- `pageInfo` - additional pagination metadata
- `pageInfo.totalCount` - (also available as `totalCount` number of all nodes that match the filter, before pagination
- `pageInfo.currentPage` - index of the current page (based on 1)
- `pageInfo.hasNextPage`, `pageInfo.hasPreviousPage` - whether there is a next or previous page based on current pagination
- `pageInfo.itemCount` - number of items on current page
- `perPage` - requested number of items on each page
- `pageCount` - total number of pages
- `distinct(field)` - print distinct values for given field
- `group(field)` - return values grouped by a field

Filter types

For every Node type, a filter GraphQL input type is created. Gatsby has pre-created "operator types" for each scalar, like `StringQueryOperatorType`, that has keys as possible operators (for example, `eq`, `ne`) and values as appropriate values for them. Then Gatsby looks at every field in the type and executes roughly the following algorithm.

1. If field is a scalar 1.1 Get a corresponding operator type for that scalar type 1.2 Replace field with that type
2. If field is not a scalar - recursively go through the nested type's fields and then assign resulting input object type to the field.

Abridged resulting types:

```
input StringQueryOperatorInput {
  eq: String
  ne: String
  in: [String]
  nin: [String]
  regex: String
  glob: String
}

input BlogFilterInput {
  title: StringQueryOperatorInput
  comments: CommentFilterInput
  # and so forth
}
```

Sorting types

For sort, GraphQL creates an enum of all fields (including up to 3 levels of nesting) for a particular type.

```
enum BlogFieldsEnum {
  id
  title
  date
  parent___id
  # and so forth
}
```

This field is combined with enum that contains ordering (`ASC` or `DESC`) into a sort input type.

```
input BlogSortInput {
  fields: [BlogFieldsEnum]
  order: [SortOrderEnum] = [ASC]
}
```