> *This documentation is up to date as of* `gatsby@2.23.0` .

Summary

Gatsby stores all data loaded during the source-nodes phase in Redux and it allows you to write GraphQL queries to query that data. This data, stored as individual "nodes", can be searched through using a query language that is inspired by MongoDB queries.

Filtering is used in GraphQL root fields of node types (e.g. for file type it would be `file` and `allFile`). The GraphQL `filter` argument is passed to the filtering system and will return all the nodes that match each of the given filters. The rest of the processing, such as pagination, is handled on the GraphQL resolver level.

History and Sift

For a long time Gatsby used the Sift library through which you can use MongoDB queries in JavaScript.

Unfortunately Sift did not align with how Gatsby used it and so a custom system was written to slowly replace it. This system was called "fast filters" and as of gatsby@2.23.0 (June 2020) the Sift library is no longer used.

Query Language

The syntax and API used by the filters is based on the MongoDB query syntax but keep in mind only a subset of comparators is supported in Gatsby.

In general a filter has a "filter path" component ending with a "comparator", and a "filter value". It's a way of programmatically asking "show me all nodes where `node[i].a.b.c` leads to a value less than `5` ".

- path: an exact property path when traversing an object (`{ a: { b: { c: { op: x } } } }` has path `a` , `a.b` , and `a.b.c`)
- value: the needle to compare against the value at the end of a path
- comparator: a way to tell the system how to compare the given filter value to the resulting value at the end of the path. Should it be equal, lower than, or match in a special way?

Filters are generally applied to a subset of all nodes, matching a specific `node.internal.type` value, like `` `PageData` `` or `` `markdownRemark` `` .

When applying a filter, the path will be checked for each node and the resulting value is compared against the requested filter value with the rules of the comparator used.

Example

Say we have a filter and a set of nodes:

```
filter = { post: { author: { name: { eq: "Alex" } } } }

nodes = [
  { id: 1, post: { title: "Hello, world!", author: { name: "Alex" } } },
  { id: 2, post: { title: "Debugging Gatsby", author: { name: "Clarissa" } } },
  { id: 3, post: { title: "Publishing on Gatsby", author: { name: "Ika" } } },
  { id: 4, post: { title: "Fixed a bug", author: { name: "Alex" } } },
]
```

The filter path is `post.author.name`, the comparator is `eq`, and the filter value is `'Alex'`.

This ought to return two nodes, the ones with id 1 and 4, because those are the only ones where the result value `node[i].post.author.name` equals `'Alex'`.

A filter path can combine multiple paths, for example:

- `{ id: { gt: 2 }, post: { author: { name: { eq: 'Alex' } } } }` would return only the node with id 4
- `{ post: { title: { regex: '/Gatsby/i' }, author: { eq: 'Ika' } } }` would return the node with id 3.

Supported comparators

Gatsby supports a subset of MongoDB comparators. Here is the list of supported comparators and how they match node(s);

- `eq`: is exactly the filter value.
- `ne`: is anything except for the filter value.
- `in`: matches any element in given array
- `nin`: matches none of the elements in given array
- `lt`: is lower than the filter value
- `lte`: is lower than or equal to the filter value
- `gt`: is greater than the filter value
- `gte`: is greater than or equal to the filter value
- `regex`: matches given JavaScript regex
- `glob`: matches given micromatch pattern

Internally the `glob` comparator is converted to a regular expression by the micromatch library and uses the same code path as the `regex` comparator.

`elemMatch`

One additional feature that Gatsby supports is the `elemMatch` query. This query has a path where one or more steps are named `elemMatch` and while traversing a node these steps should resolve to arrays. Gatsby will then attempt to apply the remainder of the path to each element in that array.

A contrived `elemMatch` example:

```
filter = { a: { elemMatch: { b: { eq: 5 } } } }

nodes = [
  {
    id: 1,
    a: [
      { a: 1, b: 8, c: 7 },
      { a: 3, b: 5, c: 6 },
    ],
  },
  {
    id: 2,
    a: [
```

```
      { a: 2, b: 4, c: 6 },
      { a: 6, b: 3, c: 3 },
    ],
  },
  {
    id: 3,
    a: [
      { a: 3, b: 5, c: 3 },
      { a: 5, b: 4, c: 1 },
    ],
  },
  {
    id: 4,
    a: [
      { a: 4, b: 7, c: 1 },
      { a: 9, b: 1, c: 6 },
    ],
  },
]
```

This will return the nodes with id 1 and 3 because `node[0].a[1].b` and `node[2].a[0].b` equal `5`.

Note that `elemMatch` will contribute each node at most once, even if multiple elements in the array would match.

You can have nested `elemMatch` occurrences and they work the same every step. Every comparator can be used in an `elemMatch` query.

Comparator details

Certain comparators restrict the kind of filter value they will accept. In general, and due to the GraphQL layer, only "scalar" or "primitive" values can be used. That is to say; numbers, strings, and booleans. It is possible to match against `null` but every operator has specific edge cases in how these are treated.

In the next guide, a "node with partial path" is any node that has none, or only a part, of the filter path. Internally that results in `undefined`, as any non-existing property would be. For brevity there's no distinction between a partial and non-existing path.

Specific rules:

- `eq`
  - This is like `filterValue === resultValue` in JavaScript
  - Can only be used with numbers, strings, booleans, and null
  - Strict comparison except for `null`
  - When matching `null`, and only then, it also returns all nodes with partial paths
- `ne`
  - This is like `filterValue !== resultValue` in JavaScript
  - Can only be used with numbers, strings, booleans, and null
  - Strict comparison except for `null`
  - Will always return any nodes with partial paths
- `in`
  - This is like `filterValue.includes(resultValue)` in JavaScript

    - Can only be used with an array of numbers, strings, booleans, and null
    - Strict comparison
    - When matching `null`, and only then, it also returns all nodes with partial paths
- `nin`
    - This is like `!filterValue.includes(resultValue)` in JavaScript
    - Can only be used with an array of numbers, strings, booleans, and null
    - Strict comparison
    - Will always return any nodes with partial paths
- `lt`, `lte`, `gt`, `gte`
    - This is like `<`, `<=`, `>`, `>=` respectively
    - Can only be used with numbers, strings, booleans, and null
    - If filter value is `null`;
        - `lt` and `gt` will never match anything
        - `lte` and `gte` will only match nodes with a result value of exactly `null`
    - Will never return nodes with partial paths
    - Weak comparison due to the nature of the JavaScript operators
- `regex`, `glob`
    - This is like `new RegExp(filterValue).test(resultValue)` (with caveats for the `filterValue` syntax)
    - Glob pattern is converted to a JavaScript RegExp with micromatch
    - The `regex` `filterValue` must be a stringified regular expression, including leading and trailing forward slash and optional flags; Like `"/foo/g"`
    - Never returns nodes with partial paths
    - While testing, result values are explicitly cast to a string through `String(resultValue)` before passing it to `regex.test()`

Nulls and partial paths

Due to legacy support for MongoDB compatibility, there are edge cases for each comparator when it comes to `null` values and for partial or non-existing paths. It's best to try and avoid these cases altogether.

Performance

The key metric that impacts the performance of your queries is the node count for your type of nodes. You can see a dump of these counts when using the `--verbose` flag while building (`gatsby build --verbose`). It tells you the node counts per type during the bootstrap sequence. There will be a separate message of the number of page nodes, since they are generated later.

While the number of pages is definitely a factor for the number of nodes, there can be other factors at play that cause many internal nodes even when the actual number of pages seems to be low.

The actual performance of filtering is a combination of the number of nodes you have to search through, the kind of comparator being used, and how many unique filters a query consists of.

For a low node count none of this matters. Roughly speaking, a site with fewer than 1000 - 10000 nodes should not have to worry too much about this.

When scaling up these are some guidelines to keep in mind when filtering:

- the `eq` comparator is by far the fastest comparator

    - the `regex` and `glob` comparators are the slowest and do not scale
    - the `gt` and `gte` comparators are slower than their `lt` and `lte` counterparts
    - all range comparators (`lt` , `lte` , `gt` , and `gte`) must copy a subset of the array they match (slow at scale)
    - the `in` and `nin` comparators create a `Set` of all nodes before applying their exclusions
- a single unique path will always outperform multiple paths
- the `elemMatch` feature does have a fixed one-time cost but should not impact overall performance at scale
- while not an absolute guarantee, the output array of a filter is normally ordered by insertion order

While you can't always avoid going for slower comparators or filters with multiple paths (like for ranges), you should keep in mind that filters with equal logic do not have equal performance. So `{ eq: 5 }` will perform much better than `{ in: [5] }` . No internal effort is done to detect these cases.

Custom resolvers

Before applying filters, all fields of any node that a filter wants to match should be completely resolved. As such, before applying a filter all nodes will go through a final resolve step which populates the `node.__gatsby_resolved` field.

This means your custom resolver may be invoked multiple times even if fewer (or zero) nodes are actually returned by a filter step. But at most once per build, unless the node's state somehow changes afterwards.