> *This documentation isn't up to date with the latest version of Gatsby.*
>
> *Outdated areas are:*
>
> - *implementation details are out of date*
>
> *You can help by making a PR to update this documentation.*

Query execution

Query execution is kicked off by bootstrap by calling `createQueryRunningActivity()`. The main files involved in this step are:

- index.js
- queue.ts
- query-runner.ts

Here's an overview of how it all relates:

```
digraph {
  compound = true;

  subgraph cluster_other {
    style = invis;
    extractQueries [ label = "query-watcher.js", shape = box ];
    componentsDD [ label = "componentDataDependencies\l(redux)", shape = cylinder ];
    components [ label = "components\l (redux)", shape = cylinder ];
    createNode [ label = "CREATE_NODE action", shape = box ];
  }

  subgraph cluster_pageQueryRunner {
    label = "page-query-runner.js"

    dirtyActions [ label = "dirtyActions", shape = cylinder ];
    extractedQueryQ [ label = "queueQueryForPathname()", shape = box ];
    findIdsWithoutDD [ label = "findIdsWithoutDataDependencies()", shape = box ];
    findDirtyActions [ label = "findDirtyActions()", shape = box ];
    queryJobs [ label = "runQueriesForPathnames()", shape = box ];

    extractedQueryQ -> queryJobs;
    findIdsWithoutDD -> queryJobs;
    dirtyActions -> findDirtyActions [ weight = 100 ];
    findDirtyActions -> queryJobs;
  }

  subgraph cluster_queryQueue {
    label = "query-queue.js";
    queryQ [ label = "better-queue", shape = box ];
  }

  subgraph cluster_queryRunner {
    label = "query-runner.js"
    graphqlJs [ label = "graphqlJs(schema, query, context, ...)" ];
```

```
    result [ label = "Query Result" ];

    graphqlJs -> result;
  }

  diskResult [ label = "/public/static/d/${dataPath}", shape = cylinder ];
  jsonDataPaths [ label = "jsonDataPaths\l(redux)", shape = cylinder ];

  result -> diskResult;
  result -> jsonDataPaths;

  extractQueries -> extractedQueryQ;
  componentsDD -> findIdsWithoutDD;
  components -> findIdsWithoutDD;
  createNode -> dirtyActions;

  queryJobs -> queryQ [ lhead = cluster_queryQueue ];

  queryQ -> graphqlJs [ lhead = cluster_queryRunner ];
}
```

Figuring out which queries need to be executed

The first thing this query does is figure out what queries need to be run. You might think this would be a matter of running the Queries that were enqueued in Extract Queries, but matters are complicated by support for `develop`. Below is the logic for figuring out which queries need to be executed (code is in runQueries()).

Already queued queries

All queries queued after being extracted (from `query-watcher.js`).

Queries without node dependencies

All queries whose component path isn't listed in `componentDataDependencies`. In Schema Generation, all Type resolvers record a dependency between the page whose query is running and any nodes that were successfully resolved. So, If a component is declared in the `components` Redux namespace (occurs during Page Creation), but is *not* contained in `componentDataDependencies`, then by definition, the query has not been run. Therefore it needs to be run. Checkout Page -> Node Dependencies for more info. The code for this step is in findIdsWithoutDataDependencies.

Pages that depend on dirty nodes

In `develop` mode, every time a node is created, or is updated (e.g. via editing a markdown file), that node needs to be dynamically added to the enqueuedDirtyActions collection. When your queries are executed, the code will look up all nodes in this collection and map them to pages that depend on them (as described above). These pages' queries must also be executed. In addition, this step also handles dirty `connections` (see Schema Connections). Connections depend on a node's type. So if a node is dirty, the code marks all connection nodes of that type dirty as well. The code for this step is in popNodeQueries. *Note: dirty ids is really talking about dirty paths.*

Queue queries for execution

There is now a list of all pages that need to be executed (linked to their Query information). Gatsby will queue them for execution (for real this time). A call to runQueriesForPathnames kicks off this step. For each page or static query, Gatsby creates a Query Job that looks something like:

```
{
  id: // page path, or static query hash
  hash: // only for static queries
  jsonName: // jsonName of static query or page
  query: // raw query text
  componentPath: // path to file where query is declared
  isPage: // true if not static query
  context: {
    path: // if staticQuery, is jsonName of component
    ...page // page object. Not for static queries
    ...page.context // not for static queries
  }
}
```

This Query Job contains everything it needs to execute the query (and do things like recording dependencies between pages and nodes). It gets pushed onto the queue in query-queue.js and then waits for the queue to empty. Next, this doc will cover how `query-queue` works.

Query queue execution

query-queue.js creates a better-queue queue that offers advanced features like parallel execution, which is handy since queries do not depend on each other so Gatsby can take advantage of this. Every time an item is consumed from the queue, it calls query-runner.ts where it can finally execute the query!

Query execution involves calling the graphql-js library with 3 pieces of information:

1. The Gatsby schema that was inferred during Schema Generation.
2. The raw query text. Obtained from the Query Job.
3. The Context, also from the Query Job. Has the page's `path` amongst other things so that Gatsby can record Page -> Node Dependencies.

Graphql-js will parse the query, and executes the top level query. E.g. `allMarkdownRemark( limit: 10 )` or `file( relativePath: { eq: "blog/my-blog.md" } )`. These will invoke the resolvers defined in Schema Connections or GQL Type, which both use sift to query over all nodes of the type in Redux. The result will be passed through the inner part of the GraphQL query where each type's resolver will be invoked. The vast majority of these will be `identity` functions that just return the field value. Some however could call a custom plugin field resolver. These in turn might perform side effects such as generating images. This is why the query execution phase of bootstrap often takes the longest.

Finally, a result is returned.

Save query results to Redux and disk

As queries are consumed from the queue and executed, their results are saved to Redux and disk for consumption later on. This involves converting the result to pure JSON, and then saving it to its dataPath. Which is relative to `public/static/d`. The data path includes the jsonName and hash. E.g: for the page `/blog/2018-07-17-announcing-gatsby-preview/`, the queries results would be saved to disk as something like:

```
/public/static/d/621/path---blog-2018-07-17-announcing-gatsby-preview-995-a74-
dwfQIanOJGe2gi27a9CLKHjamc.json
```

For static queries, instead of using the page's jsonName, Gatsby uses a hash of the query.

Now Gatsby needs to store the association of the page -> the query result in Redux so it can be recalled later. This is accomplished via the `json-data-paths` reducer which is invoked by creating a `SET_JSON_DATA_PATH` action with the page's jsonName and the saved dataPath.