Secondary indexes with LMDB

> ***Note:*** *this doc only covers secondary index implementation when using LMDB as a data store. Gatsby in-memory store uses slightly different approach using native JS structures.*

Gatsby uses composite indexes to execute queries efficiently (Compound and MultiKey indexes in MongoDB terms). The implementation and limitations are pretty similar to that of MongoDB.

Indexes for all object types are stored in the same LMDB database (aka collection). Keys in this database are represented as arrays of scalar JS values (via).

Index structure

Conceptually an index is a huge flat map where keys are *sorted* tuples of node attributes and values contain unique id of the indexed objects (and maybe some additional attributes).

For example, imagine a node like this:

```
const a1 = {
  id: "a1",
  a: "foo",
  b: ["bar", "baz"],
  internal: { type: "A" },
}
```

A secondary index for fields `{ a: 1, b: 1 }` will include **2** keys for this node (sorted by key):

```
["A/a:1/b:1", "foo", "bar", "a1"]: "a1"
["A/a:1/b:1", "foo", "baz", "a1"]: "a1"
```

The **first** column is index id: `"A/a:1/b:1` (the actual implementation uses numeric index id for more compact keys)

The **second** column `"foo"` corresponds to the value of the indexed field `A.a`.

The **third** column (`"bar"` and `"baz"` for the 1st and 2nd rows respectively) corresponds to the field `A.b`. But since it is an array we add a separate entry for each element with the same prefix.

In other words, index includes a **cartesian product** of all indexed node attributes.

> *This is the same concept as MultiKey index in MongoDB (with similar limitations).*

The **last** column in the index key is node id `"a1"`. This column helps to differentiate entries of different nodes. For instance if we add another node `a2` with the same set of attributes `a: "foo", b: "bar"` but different `id: "a2"` - our index will have 3 entries:

```
["A/a:1/b:1", "foo", "bar", "a1"]: "a1"
["A/a:1/b:1", "foo", "bar", "a2"]: "a2"
["A/a:1/b:1", "foo", "baz", "a1"]: "a1"
```

> *Alternatively we could have used `dupSort` feature of lmdb-store. But it doesn't fully solve the problem of duplicates for composite multikey indexes and was way less performant with `counts`.*

> *We should switch when this is fully supported: https://github.com/DoctorEvidence/lmdb-store/issues/66*

Creating an index

`createIndex` expects node type name and index config as input (similar to MongoDB format):

```
await createIndex(nodeTypeName, { foo: 1, "nested.bar": 1 })
```

Each key in this config is a field to be indexed. Supports dot-notation to express indexes on nested fields.

> *Note: in MongoDB config values express sort order of the field in index, but we do not support mixed sort order with indexing yet, so this field is redundant (for now).*

To actually build an index we traverse all nodes for a given type and get values for all fields defined in the index config as described in index structure chapter.

While creating an index we collect various stats, but the most important is `multiKeyFields`. If it contains fields, then the index is MultiKey index and has several limitations.

This is later used for various optimizations when actually scanning the index.

Queries that can use index

> ***tldr;*** *basically only `eq` filters can always use index with `sort`. All range filters (`in`, `gt`, etc) can only use index with overlapping sort fields. Other filters (`ne`, `regex`, `glob`) cannot use indexes.*

Not all filters can use indexes. The following Gatsby filters **can**:

```
eq, in, gt, gte, lt, lte, ne, nin
```

> *Note: `ne` and `nin` **can not** use MultiKey indexes because they contain duplicates. E.g. `{ foo: { ne: "bar" } }` will still match `{ foo: ["foo", "bar"] }` because of "foo" duplicate.*

Those filters **can not** use index:

```
regex, glob
```

Furthermore, only *some* combinations of `filters` + `sort` can use index.

Let's say we have an index `{ a: 1, b: 1 }`.

Only the following combinations can be fully resolved using this index:

1. Queries using `eq` filter:

```
{ filter: { a: { eq: "foo" } }, sort: { fields: ["a"] } }
{ filter: { a: { eq: "foo" } }, sort: { fields: ["a", "b"] } }
{ filter: { a: { eq: "foo" } }, sort: { fields: ["b"] } }
```

Plus queries with any additional filter on field `b` (i.e., `eq`, `in`, `gt`, `gte`, `lt`, `lte`):

```
{ filter: { a: { eq: "foo" }, b: { gt: "bar" } }, sort: { fields: ["a"] } }
{ filter: { a: { eq: "foo" }, b: { in: ["bar"] } }, sort: { fields: ["a", "b"] } }
{ filter: { a: { eq: "foo" }, b: { lte: "bar" } }, sort: { fields: ["b"] } }
```

etc

2. Queries using `in`, `gt`, `gte`, `lt`, `lte` filters, e.g.:

```
{ filter: { a: { gt: "foo" } }, sort: { fields: ["a"] } }
{ filter: { a: { gt: "foo" } }, sort: { fields: ["a", "b"] } }
```

Plus any other filter on field `b` (`eq, in, gt, gte, lt, lte`).

That's it.

Any other combinations **can not** use index for both `filter` and `sort` (only for one of those).

For example, the following queries can only use index `{ a: 1, b: 1 }` for `filter` (but not `sort`):

```
{ filter: { a: { gt: "foo" } }, sort: { fields: ["b"] } }
{ filter: { a: { gt: "foo" }, b: { eq: "bar" } }, sort: { fields: ["b"] } }
```

While those queries can only use index for `sort`:

```
{ filter: { b: { eq: "foo" } }, sort: { fields: ["a"] } }
{ filter: { b: { eq: "foo" } }, sort: { fields: ["a", "b"] } }
{ filter: { b: { gt: "foo" } }, sort: { fields: ["a"] } }
```

etc.

Suggest index for a query

Unlike databases that must select one of the existing indexes created by users, we actually decide which index to **create** for a given query (and then use).

For now, the general algorithm is simple:

1. Pick fields that work both for `filter` and `sort` (as described in the section above)
2. When it's not possible - pick `filter` fields (sorted by predicate specificity) if:
    1. there are filters on multiple fields
    2. there is a single `eq` or `in` filter
    3. there is a filter with two predicates (e.g. `gt` and `lt`)
3. In all other cases - pick `sort` fields for index

The caveat is that there are actually 3 costly operations:

1. Filter
2. Sort
3. Count total number of results (used for pagination)

So when we prefer `sort` fields for index we de-optimize not just `filter`, but also `count`.

> *In general, selecting the best index to build and use is exceptionally hard. Databases use sophisticated heuristics and statistics as a part of this process. Our current approach is rather naive and has plenty of room for improvement.*

> *TODO: try to avoid fields on array values (so avoid MultiKey indexes when possible).*

> *TODO: Filter and sort fields that can not be used for range scans directly are added to index values. This allows us to avoid additional expensive `getNode` operations.*

Running a query

There are several codepaths here

Can not use index

> *Can happen for `regex` / `glob` filters and `ne` / `nin` with MultiKey index.*

1. Iterate through all nodes and apply filters.
2. 🐌 If sorting is needed - load all filtered nodes in memory and sort
3. Slice results (it is applied lazily to final iterable)

Can use index

Running a query consists of several steps:

1. Select an index
2. Create this index (if it does not exist yet)
3. Filter using index
4. For each entry returned from index - load full node object
5. Apply any remaining not applied filters

If index satisfies sorting requirements (or no sorting requirements):

6. Apply `skip` / `limit`
7. Done

If index doesn't satisfy sorting requirements:

6. 🐌 Load all resulting nodes and sort in-memory.
7. Apply `skip` / `limit`
8. Done

> ***Note:*** *Every step uses iterators and generators, so essentially it is a single traversal defined lazily. It doesn't actually require double traversal (except when in-memory sorting is actually needed).*
>
> *For example `skip` and `limit` are added at the last step, but they will be actually applied to the whole iterable as we traverse it, so no unnecessary iteration will happen.*

> *TODO: we can play with streamed sorting via temporary tables in LMDB: https://github.com/DoctorEvidence/lmdb-store/discussions/70*

Caveats

1. Counting is slow

In-memory store can do counts in `O(1)` using `results.length`. In case of querying LMDB - counting basically requires a separate query without `skip` / `limit`. So it is at least `O(N)`.

What's worse: when counting cannot be fully satisfied by the filter - we literally must traverse the full resultset (so can't apply `limit` / `skip`).

Fortunately `lmdb-store@1.6.0` introduces fast counters in C++, so it performs pretty well in this scenario: https://github.com/DoctorEvidence/lmdb-store/discussions/68#discussioncomment-963542

> *TODO: use this feature of lmdb-store when ready: https://github.com/DoctorEvidence/lmdb-store/issues/66 it will return counts in O(1) for 90% of common cases.*

2. MultiKey indexes and count

Multikey index cannot reliably count the number of elements returned by some query The following node has two entries in index `{ a: 1 }`.

```
const node = { id: 1, a: [`foo`, `bar`] }
```

It may show up multiple times in results for range queries (like `in` or `gt`).

The only case when it returns reliable count is when *all* multikey fields are filtered with `eq` predicate (field `a` in this example).

So in the worst case we must traverse all index results and deduplicate to get the actual count.

3. MultiKey index and limit, offset

Limit and offset are also unreliable with MultiKey indexes (unless all multiKey fields have `eq` predicate).

4. Mixed sort order requires full in-memory sort (yet)

Currently, we cannot scan index in mixed order. This feature requires binary inversion for key elements in `lmdb-store` which is not yet available.

5. Key size limit

1978 bytes hard limit.

6. Using node counter for default sorting vs node id

In the index structure section we mention that the last column in the index is node id. But in reality we use an internal node counter to have smaller keys and get sorting by insertion order by default.

> *This value is stored in `node.internal.counter` and added by `createNode` action creator*

7. Materialization

There is a special case when we index fields having custom GraphQL resolvers. Values for such fields require additional resolution step. We call it "materialization".

For example, imaging this node:

```
const a1 = {
  id: "a1",
  a: "foo",
  b: ["bar", "baz"],
  internal: { type: "A" },
}
```

Also, imagine a custom resolver defined for field `b`:

```
exports.createResolvers = ({ createResolvers }) => {
  createResolvers({
    A: {
      b: source => source.join(`-`),
    },
  })
}
```

Then the actual value added to index will be

```
"bar-baz"
```

Notes:

1. Currently, materialization does another full pass on all nodes before `createIndex`.
2. Materialization is optional - it doesn't occur if field has no custom resolver.
3. The assumption about materialization is that it is deterministic. We do not invalidate materialization results in index unless a corresponding node itself was updated or deleted.

8. Any aggregations

9. `null` and `undefined` values

Imagine we want to add another node `"a3"` to the index which has no `b` field:

```
const a3 = {
  id: "a3",
  a: null,
  internal: { type: "A" },
}
```

We cannot exclude it from the index because it still has value for `a` field and should be returned for a query like this: `{ filter: { a: { in: [null, "foo"] } } }`.

So it has to be added to index too. lmdb-store supports `null` values in keys but not `undefined`. Thankfully it also supports symbols! So we have a `Symbol("undef")` to represent `undefined`:

```
["A/a:1/b:1", null, Symbol("undef"), "a3"]: "a3"
["A/a:1/b:1", "foo", "bar", "a1"]: "a1"
["A/a:1/b:1", "foo", "bar", "a2"]: "a2"
["A/a:1/b:1", "foo", "baz", "a1"]: "a1"
```

We add a node to index even if the very first attribute `a` is `undefined` because index may be also used just for sorting: `{ sort: { fields: ["a"] } }`.

> *Our indexes are essentially similar to non-sparse MongoDB indexes*