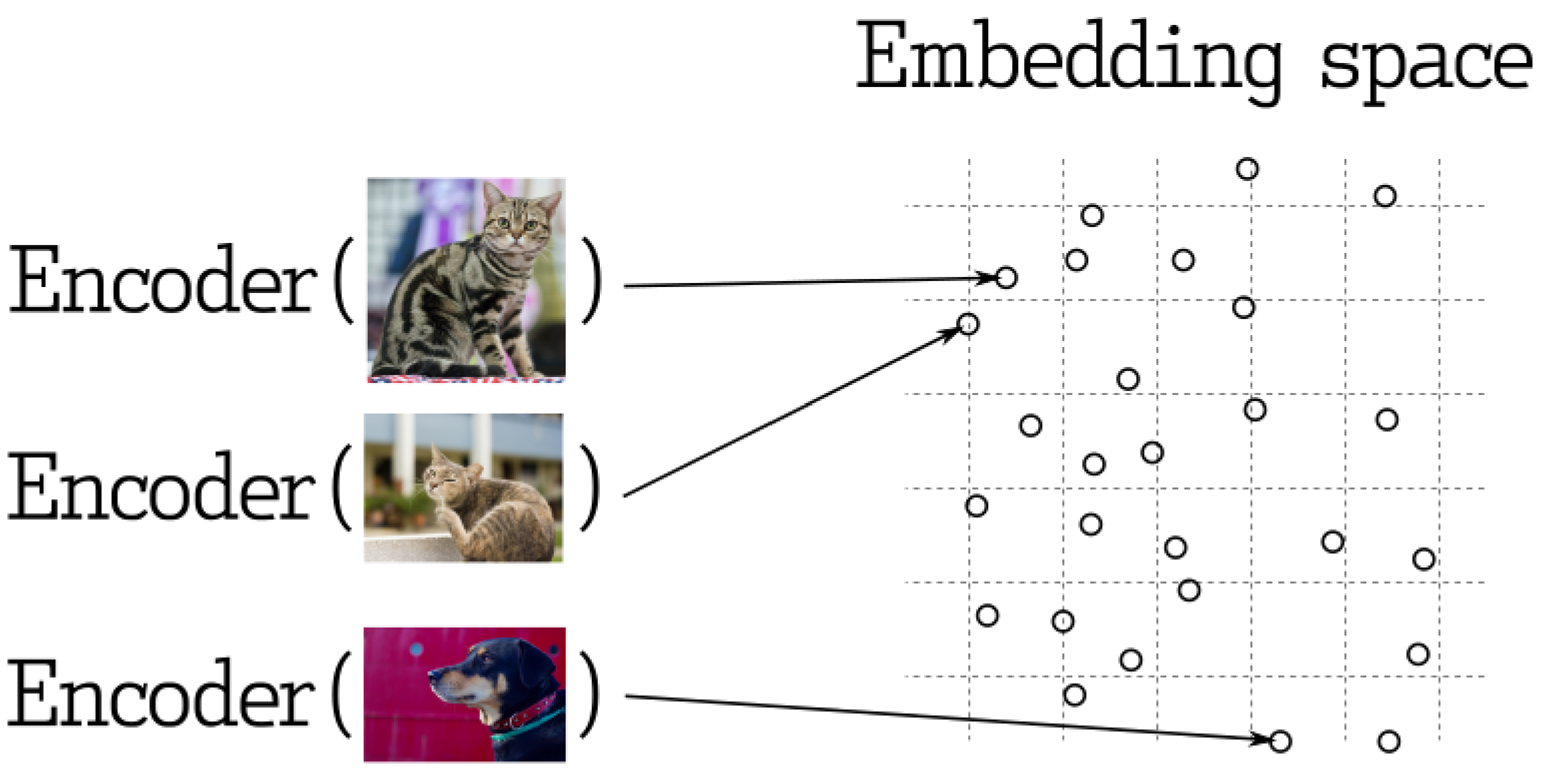
Similarity search

Searching for the nearest vectors is at the core of many representational learning applications. Modern neural networks are trained to transform objects into vectors so that objects close in the real world appear close in vector space. It could be, for example, texts with similar meanings, visually similar pictures, or songs of the same genre.



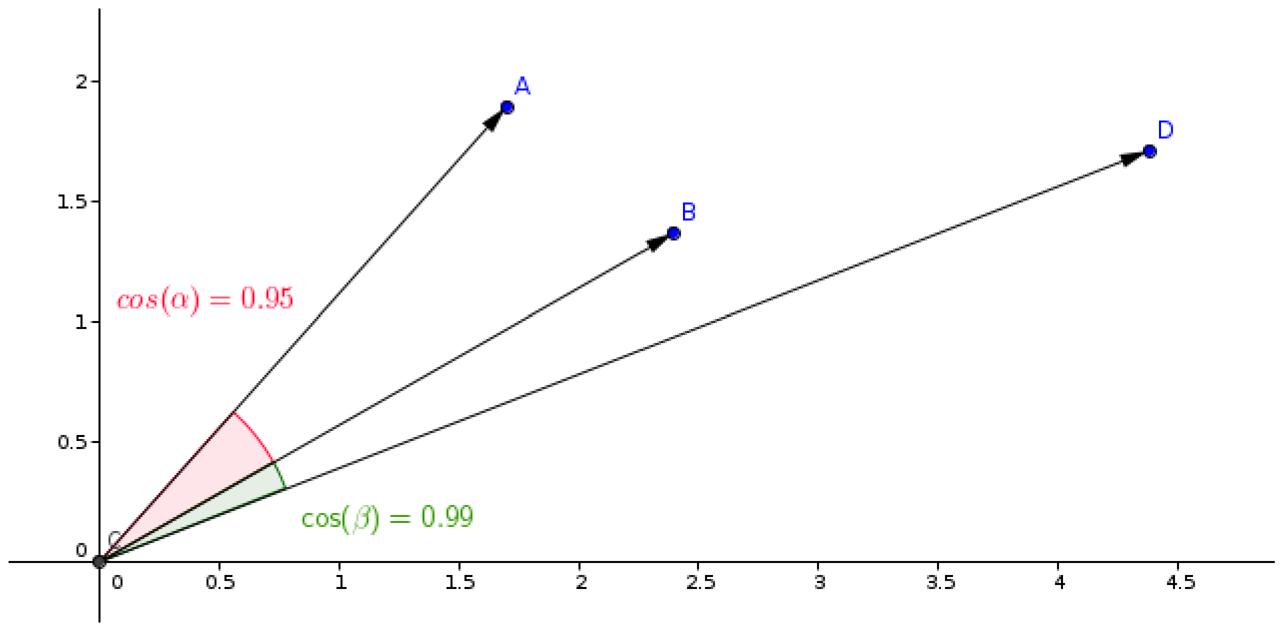
Metrics

There are many ways to estimate the similarity of vectors with each other. In Qdrant terms, these ways are called metrics. The choice of metric depends on vectors obtaining and, in particular, on the method of neural network encoder training.

Qdrant supports these most popular types of metrics:

- Dot product: `Dot` - https://en.wikipedia.org/wiki/Dot_product
- Cosine similarity: `Cosine` - https://en.wikipedia.org/wiki/Cosine_similarity
- Euclidean distance: `Euclid` - https://en.wikipedia.org/wiki/Euclidean_distance
- Manhattan distance: `Manhattan`* - https://en.wikipedia.org/wiki/Taxicab_geometry **Available as of v1.7*

The most typical metric used in similarity learning models is the cosine metric.



Qdrant counts this metric in 2 steps, due to which a higher search speed is achieved. The first step is to normalize the vector when adding it to the collection. It happens only once for each vector.

The second step is the comparison of vectors. In this case, it becomes equivalent to dot production - a very fast operation due to SIMD.

Query planning

Depending on the filter used in the search - there are several possible scenarios for query execution. Qdrant chooses one of the query execution options depending on the available indexes, the complexity of the conditions and the cardinality of the filtering result. This process is called query planning.

The strategy selection process relies heavily on heuristics and can vary from release to release. However, the general principles are:

- planning is performed for each segment independently (see storage for more information about segments)
- prefer a full scan if the amount of points is below a threshold
- estimate the cardinality of a filtered result before selecting a strategy
- retrieve points using payload index (see indexing) if cardinality is below threshold
- use filterable vector index if the cardinality is above a threshold

You can adjust the threshold using a configuration file, as well as independently for each collection.

Search API

Let's look at an example of a search query.

REST API - API Schema definition is available here

http python typescript rust java

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_filter=models.Filter(
        must=[
            models.FieldCondition(
                key="city",
                match=models.MatchValue(
                    value="London",
                ),
            )
        ]
    ),
    search_params=models.SearchParams(hnsw_ef=128, exact=False),
    query_vector=[0.2, 0.1, 0.9, 0.7],
    limit=3,
)
```

In this example, we are looking for vectors similar to vector `[0.2, 0.1, 0.9, 0.7]`. Parameter `limit` (or its alias - `top`) specifies the amount of most similar results we would like to retrieve.

Values under the key `params` specify custom parameters for the search. Currently, it could be:

- `hnsw_ef` - value that specifies `ef` parameter of the HNSW algorithm.
- `exact` - option to not use the approximate search (ANN). If set to true, the search may run for a long as it performs a full scan to retrieve exact results.
- `indexed_only` - With this option you can disable the search in those segments where vector index is not built yet. This may be useful if you want to minimize the impact to the search performance whilst the collection is also being updated. Using this option may lead to a partial result if the collection is not fully indexed yet, consider using it only if eventual consistency is acceptable for your use case.

Since the `filter` parameter is specified, the search is performed only among those points that satisfy the filter condition. See details of possible filters and their work in the filtering section.

Example result of this API would be

```
{
  "result": [
    { "id": 10, "score": 0.81 },
    { "id": 14, "score": 0.75 },
    { "id": 11, "score": 0.73 }
  ],
  "status": "ok",
  "time": 0.001
}
```

The `result` contains ordered by `score` list of found point ids.

Note that payload and vector data is missing in these results by default. See payload and vector in the result on how to include it.

Available as of v0.10.0

If the collection was created with multiple vectors, the name of the vector to use for searching should be provided:

http python typescript rust java

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_vector=("image", [0.2, 0.1, 0.9, 0.7]),
    limit=3,
)
```

Search is processing only among vectors with the same name.

Available as of v1.7.0

If the collection was created with sparse vectors, the name of the sparse vector to use for searching should be provided:

You can still use payload filtering and other features of the search API with sparse vectors.

There are however important differences between dense and sparse vector search:

Index	Sparse Query	Dense Query
Scoring Metric	Default is `Dot product`, no need to specify it	`Distance` has supported metrics e.g. Dot, Cosine
Search Type	Always exact in Qdrant	HNSW is an approximate NN
Return	Returns only vectors with non-zero values in the same indices as the	Returns `limit` vectors

In general, the speed of the search is proportional to the number of non-zero values in the query vector.

http python typescript rust java

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_vector=models.NamedSparseVector(
        name="text",
        vector=models.SparseVector(
            indices=[1, 7],
            values=[2.0, 1.0],
        ),
    ),
    limit=3,
)
```

Filtering results by score

In addition to payload filtering, it might be useful to filter out results with a low similarity score. For example, if you know the minimal acceptance score for your model and do not want any results which are less similar than the threshold. In this case, you can use `score_threshold` parameter of the search query. It will exclude all results with a score worse than the given.

This parameter may exclude lower or higher scores depending on the used metric. For example, higher scores of Euclidean metric are considered more distant and, therefore, will be excluded.

Payload and vector in the result

By default, retrieval methods do not return any stored information such as payload and vectors. Additional parameters `with_vectors` and `with_payload` alter this behavior.

Example:

http python typescript rust java

```python
client.search(
    collection_name="{collection_name}",
    query_vector=[0.2, 0.1, 0.9, 0.7],
    with_vectors=True,
    with_payload=True,
)
```

You can use `with_payload` to scope to or filter a specific payload subset. You can even specify an array of items to include, such as `city`, `village`, and `town`:

http python typescript rust java

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_vector=[0.2, 0.1, 0.9, 0.7],
    with_payload=["city", "village", "town"],
)
```

Or use `include` or `exclude` explicitly. For example, to exclude `city`:

http python typescript rust java

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_vector=[0.2, 0.1, 0.9, 0.7],
    with_payload=models.PayloadSelectorExclude(
        exclude=["city"],
    ),
)
```

It is possible to target nested fields using a dot notation:

- `payload.nested_field` - for a nested field
- `payload.nested_array[].sub_field` - for projecting nested fields within an array

Accessing array elements by index is currently not supported.

Batch search API

Available as of v0.10.0

The batch search API enables to perform multiple search requests via a single request.

Its semantic is straightforward, `n` batched search requests are equivalent to `n` singular search requests.

This approach has several advantages. Logically, fewer network connections are required which can be very beneficial on its own.

More importantly, batched requests will be efficiently processed via the query planner which can detect and optimize requests if they have the same `filter`.

This can have a great effect on latency for non trivial filters as the intermediary results can be shared among the request.

http python typescript rust java

```
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

filter = models.Filter(
    must=[
        models.FieldCondition(
            key="city",
            match=models.MatchValue(
                value="London",
            ),
        )
    ]
)

search_queries = [
    models.SearchRequest(vector=[0.2, 0.1, 0.9, 0.7], filter=filter, limit=3),
    models.SearchRequest(vector=[0.5, 0.3, 0.2, 0.3], filter=filter, limit=3),
]

client.search_batch(collection_name="{collection_name}", requests=search_queries)
```

The result of this API contains one array per search requests.

```
{
  "result": [
    [
        { "id": 10, "score": 0.81 },
        { "id": 14, "score": 0.75 },
        { "id": 11, "score": 0.73 }
    ],
    [
        { "id": 1, "score": 0.92 },
        { "id": 3, "score": 0.89 },
        { "id": 9, "score": 0.75 }
    ]
  ],
  "status": "ok",
  "time": 0.001
}
```

Pagination

Available as of v0.8.3

Example:

http python typescript rust java

```python
from qdrant_client import QdrantClient

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_vector=[0.2, 0.1, 0.9, 0.7],
    with_vectors=True,
    with_payload=True,
    limit=10,
    offset=100,
)
```

Is equivalent to retrieving the 11th page with 10 records per page.

Large offset values may cause performance issues

Vector-based retrieval in general and HNSW index in particular, are not designed to be paginated. It is impossible to retrieve Nth closest vector without retrieving the first N vectors first.

However, using the offset parameter saves the resources by reducing network traffic and the number of times the storage is accessed.

Using an `offset` parameter, will require to internally retrieve `offset + limit` points, but only access payload and vector from the storage those points which are going to be actually returned.

Grouping API

Available as of v1.2.0

It is possible to group results by a certain field. This is useful when you have multiple points for the same item, and you want to avoid redundancy of the same item in the results.

For example, if you have a large document split into multiple chunks, and you want to search or recommend on a per-document basis, you can group the results by the document ID.

Consider having points with the following payloads:

```
[
    {
        "id": 0,
        "payload": {
            "chunk_part": 0,
            "document_id": "a"
        },
        "vector": [0.91]
    },
    {
        "id": 1,
        "payload": {
            "chunk_part": 1,
            "document_id": ["a", "b"]
        },
        "vector": [0.8]
    },
    {
        "id": 2,
        "payload": {
            "chunk_part": 2,
            "document_id": "a"
        },
        "vector": [0.2]
    },
    {
        "id": 3,
        "payload": {
            "chunk_part": 0,
            "document_id": 123
        },
        "vector": [0.79]
    },
    {
        "id": 4,
        "payload": {
            "chunk_part": 1,
            "document_id": 123
        },
        "vector": [0.75]
    },
    {
        "id": 5,
        "payload": {
            "chunk_part": 0,
        },
        "vector": [0.6]
    }
]
```

With the ***groups*** API, you will be able to get the best *N* points for each document, assuming that the payload of the points contains the document ID. Of course there will be times where the best *N* points cannot be fulfilled due to lack of points or a big distance with respect to the query. In every case, the `group_size` is a best-effort parameter, akin to the `limit` parameter.

Search groups

REST API (Schema):

http python typescript rust java

```
client.search_groups(
    collection_name="{collection_name}",
    # Same as in the regular search() API
    query_vector=g,
    # Grouping parameters
    group_by="document_id",  # Path of the field to group by
    limit=4,  # Max amount of groups
    group_size=2,  # Max amount of points per group
)
```

The output of a ***groups*** call looks like this:

```
{
    "result": {
        "groups": [
            {
                "id": "a",
                "hits": [
                    { "id": 0, "score": 0.91 },
                    { "id": 1, "score": 0.85 }
                ]
            },
            {
                "id": "b",
                "hits": [
                    { "id": 1, "score": 0.85 }
                ]
            },
            {
                "id": 123,
                "hits": [
                    { "id": 3, "score": 0.79 },
                    { "id": 4, "score": 0.75 }
                ]
            },
            {
                "id": -10,
                "hits": [
                    { "id": 5, "score": 0.6 }
                ]
            }
        ]
    },
    "status": "ok",
    "time": 0.001
}
```

The groups are ordered by the score of the top point in the group. Inside each group the points are sorted too.

If the `group_by` field of a point is an array (e.g. `"document_id": ["a", "b"]`), the point can be included in multiple groups (e.g. `"document_id": "a"` and `document_id: "b"`).

> ℹ This feature relies heavily on the `group_by` key provided. To improve performance, make sure to create a dedicated index for it.

Limitations:

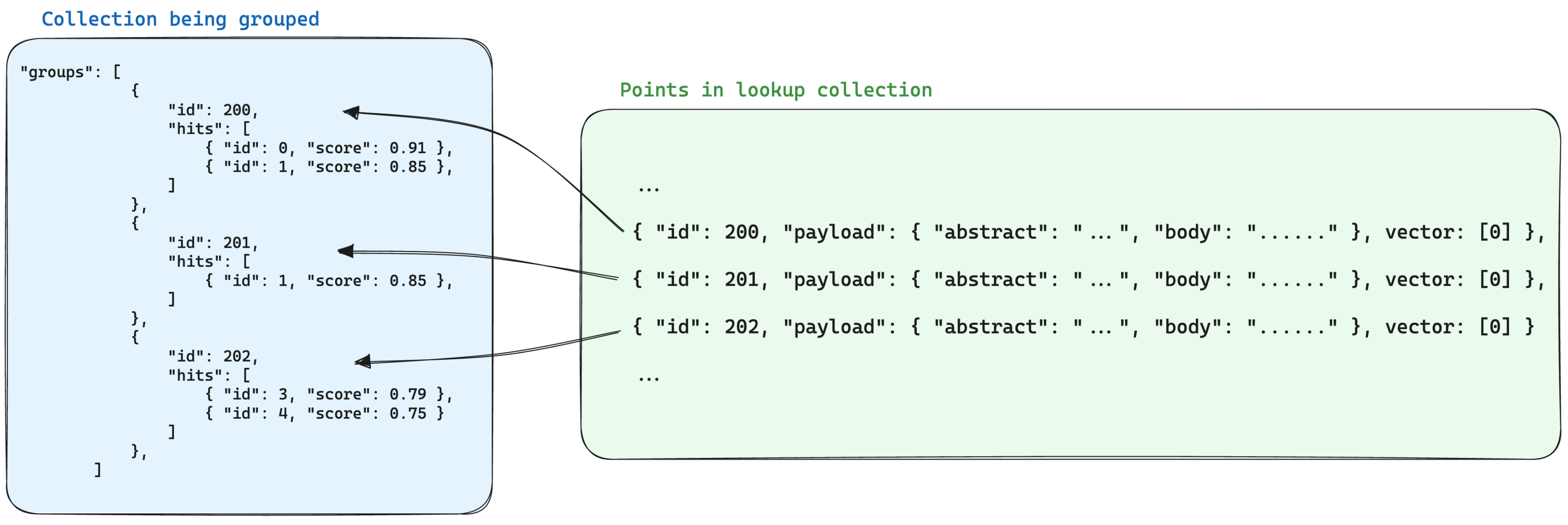
- Only keyword and integer payload values are supported for the `group_by` parameter. Payload values with other types will be ign

Lookup in groups

Available as of v1.3.0

Having multiple points for parts of the same item often introduces redundancy in the stored data. Which may be fine if the information shared by the points is small, but it can become a problem if the payload is large, because it multiplies the storage space needed to store the points by a factor of the amount of points we have per group.

One way of optimizing storage when using groups is to store the information shared by the points with the same group id in a single point in another collection. Then, when using the **groups** API, add the `with_lookup` parameter to bring the information from those points into each group.



```
"groups": [
        {
            "id": 200,
            "hits": [
                { "id": 0, "score": 0.91 },
                { "id": 1, "score": 0.85 },
            ]
        },
        {
            "id": 201,
            "hits": [
                { "id": 1, "score": 0.85 },
            ]
        },
        {
            "id": 202,
            "hits": [
                { "id": 3, "score": 0.79 },
                { "id": 4, "score": 0.75 }
            ]
        },
    ]
```

```
...
{ "id": 200, "payload": { "abstract": " ... ", "body": "......" }, vector: [0] },
{ "id": 201, "payload": { "abstract": " ... ", "body": "......" }, vector: [0] },
{ "id": 202, "payload": { "abstract": " ... ", "body": "......" }, vector: [0] }
...
```

This has the extra benefit of having a single point to update when the information shared by the points in a group changes.

For example, if you have a collection of documents, you may want to chunk them and store the points for the chunks in a separate collection, making sure that you store the point id from the document it belongs in the payload of the chunk point.

In this case, to bring the information from the documents into the chunks grouped by the document id, you can use the `with_lookup` parameter:

http python typescript rust java

```python
client.search_groups(
    collection_name="chunks",
    # Same as in the regular search() API
    query_vector=[1.1],
    # Grouping parameters
    group_by="document_id",  # Path of the field to group by
    limit=2,  # Max amount of groups
    group_size=2,  # Max amount of points per group
    # Lookup parameters
    with_lookup=models.WithLookup(
        # Name of the collection to look up points in
        collection="documents",
        # Options for specifying what to bring from the payload
        # of the looked up point, True by default
        with_payload=["title", "text"],
        # Options for specifying what to bring from the vector(s)
        # of the looked up point, True by default
        with_vectors=False,
    ),
)
```

For the `with_lookup` parameter, you can also use the shorthand `with_lookup="documents"` to bring the whole payload and vector(s) without explicitly specifying it.

The looked up result will show up under `lookup` in each group.

```json
{
    "result": {
        "groups": [
            {
                "id": 1,
                "hits": [
                    { "id": 0, "score": 0.91 },
                    { "id": 1, "score": 0.85 }
                ],
                "lookup": {
                    "id": 1,
                    "payload": {
                        "title": "Document A",
                        "text": "This is document A"
                    }
                }
            },
            {
                "id": 2,
                "hits": [
                    { "id": 1, "score": 0.85 }
                ],
                "lookup": {
                    "id": 2,
                    "payload": {
                        "title": "Document B",
                        "text": "This is document B"
                    }
                }
            }
        ]
    },
    "status": "ok",
    "time": 0.001
}
```

Since the lookup is done by matching directly with the point id, any group id that is not an existing (and valid) point id in the lookup collection will be ignored, and the `lookup` field will be empty.