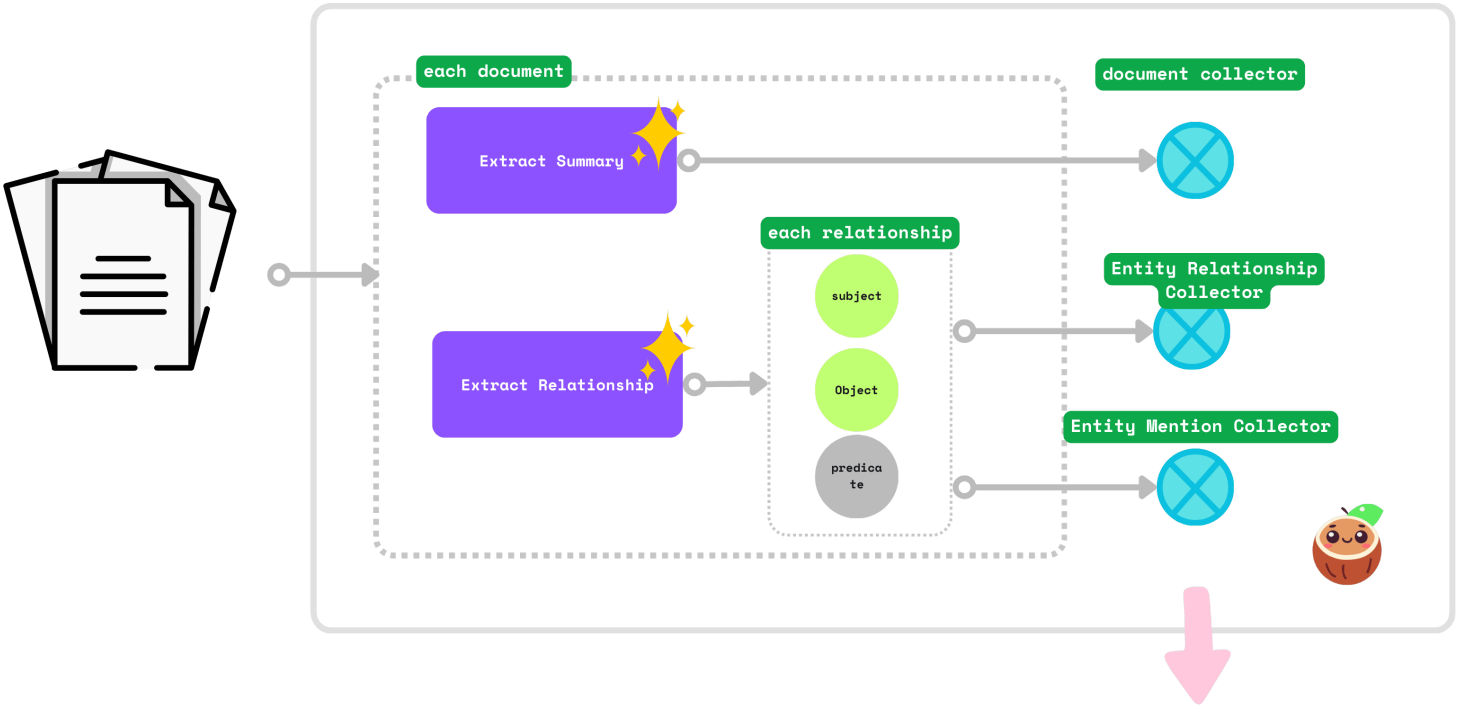
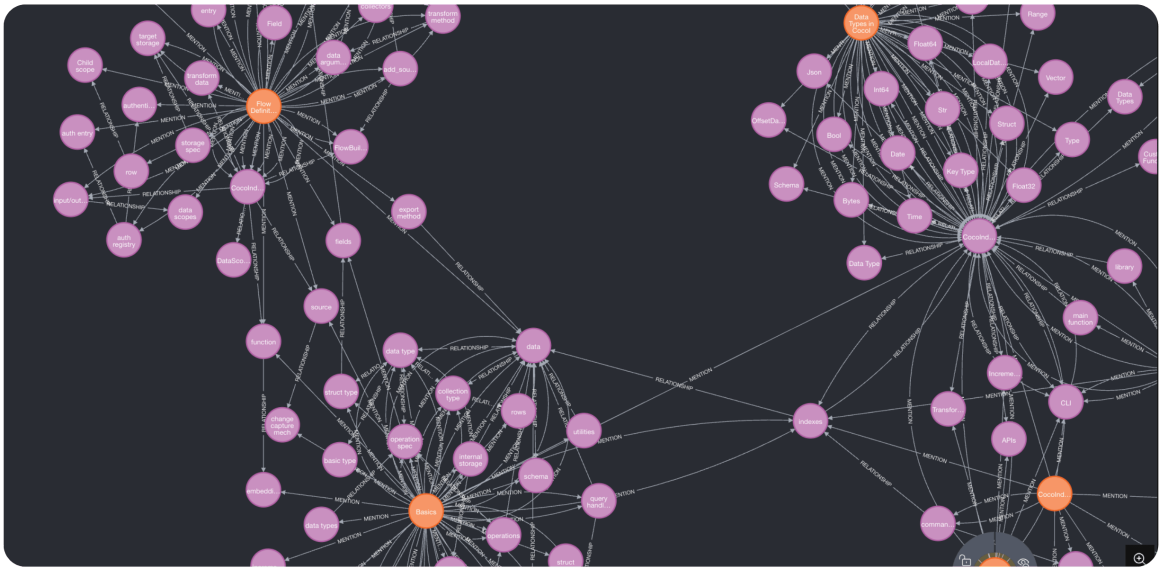
Build Real-Time Knowledge Graph For Documents with LLM

April 29, 2025 · 8 min read

Linghua Jin



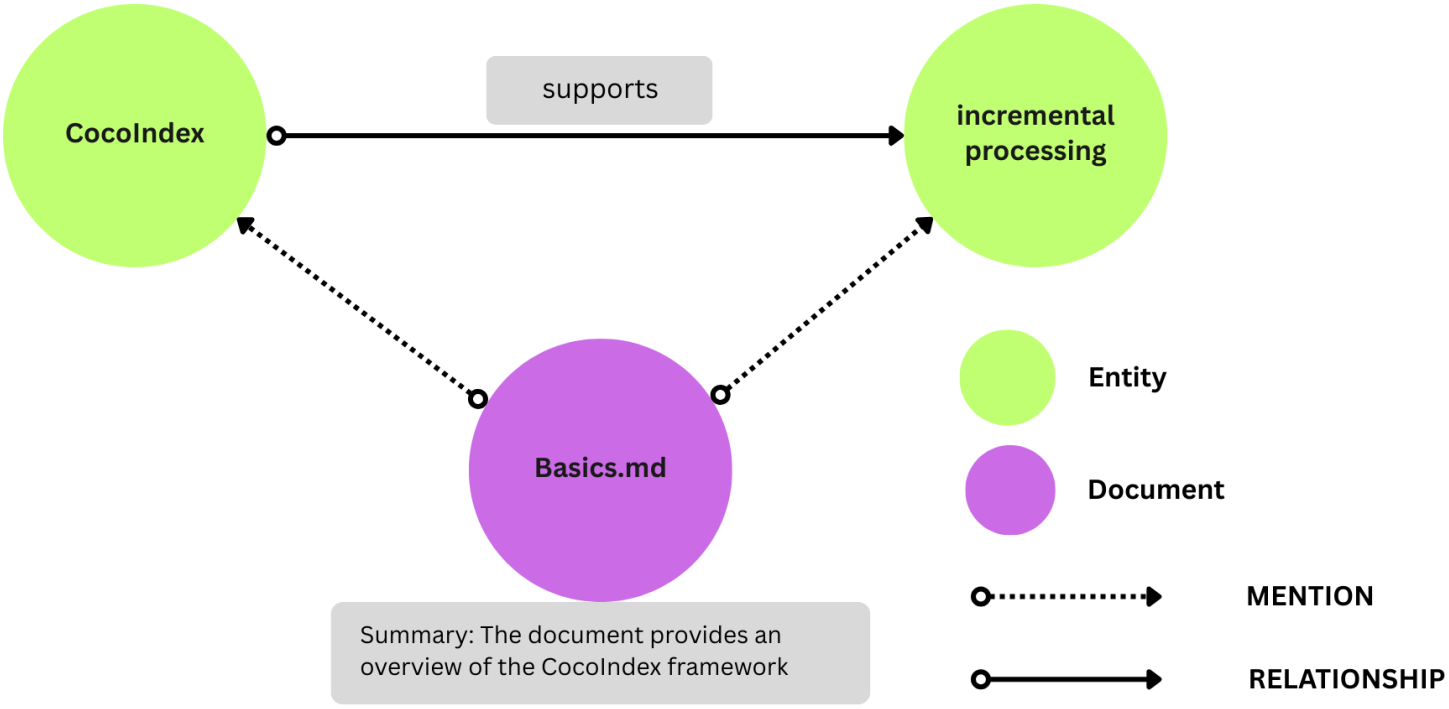


CocoIndex makes it easy to build and maintain knowledge graphs with continuous source updates. In this blog, we will process a list of documents (using CocoIndex documentation as an example). We will use LLM to extract relationships between the concepts in each document.

We will generate two kinds of relationships:

. Relationships between subjects and objects. E.g., "CocoIndex supports Incremental Processing"
. Mentions of entities in a document. E.g., "core/basics.mdx" mentions `CocoIndex` and `Incremental Processing`.

The source code is available at CocoIndex Examples - docs_to_knowledge_graph.



We are constantly improving, and more features and examples are coming soon. Stay tuned and follow our progress by starring our GitHub repo.

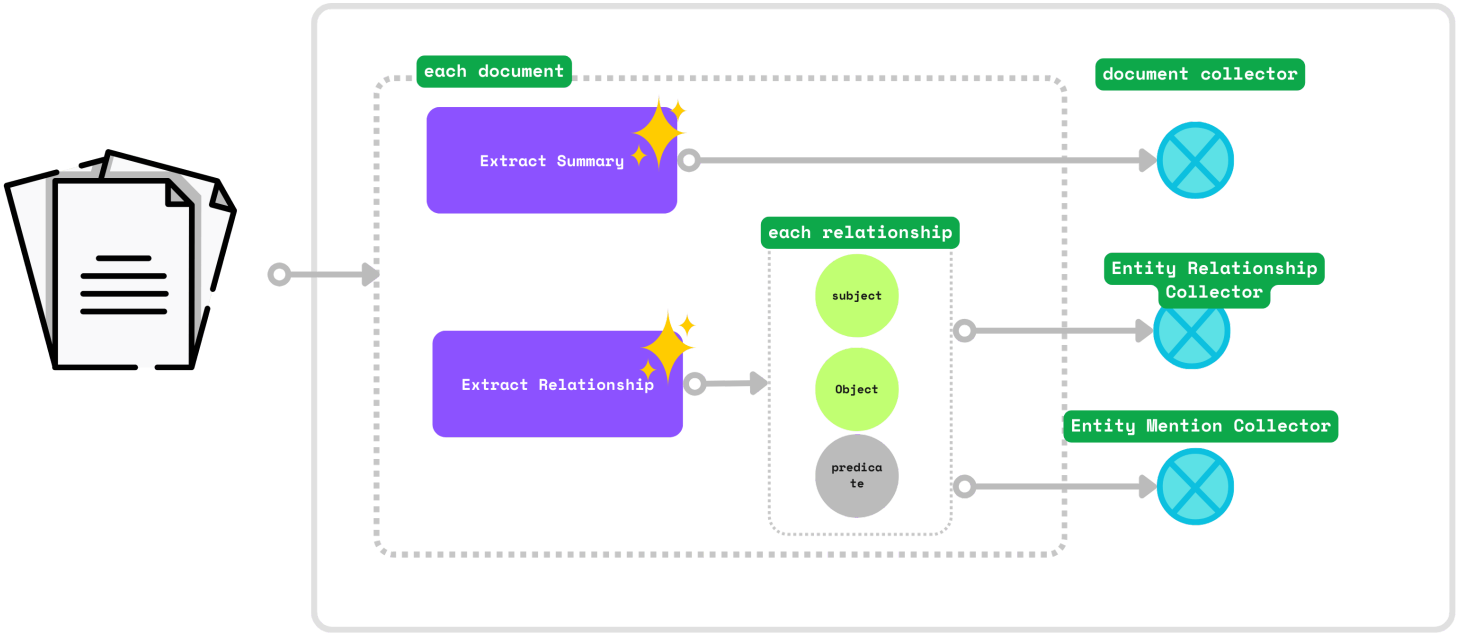
Prerequisites

Install PostgreSQL. CocoIndex uses PostgreSQL internally for incremental processing.
Install Neo4j, a graph database.
Configure your OpenAI API key. Alternatively, you can switch to Ollama, which runs LLM models locally - guide.

Documentation

You can read the official CocoIndex Documentation for Property Graph Targets here.

Data flow to build knowledge graph

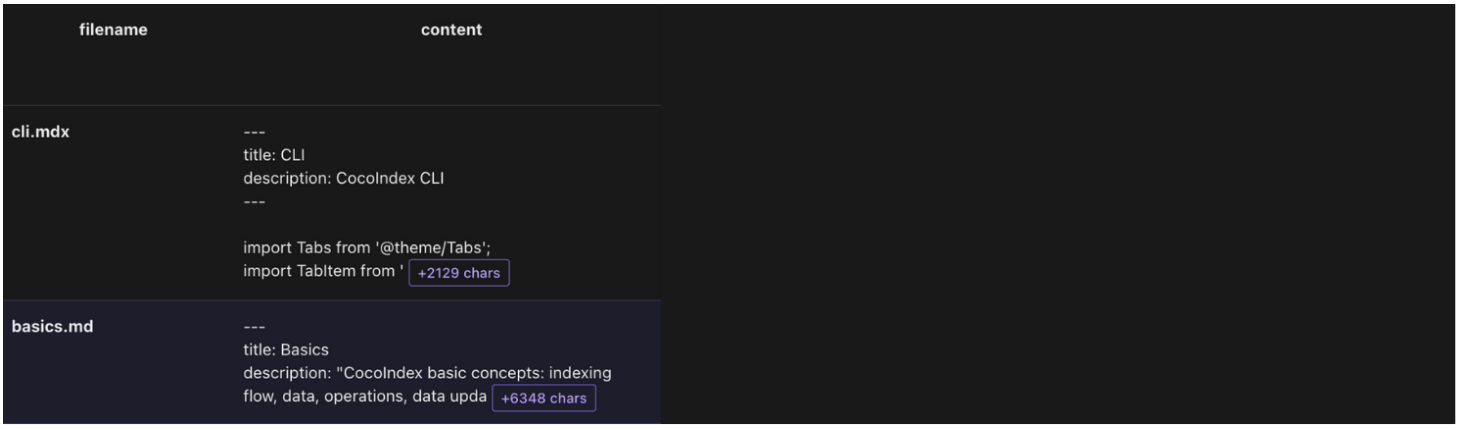


Add documents as source

We will process CocoIndex documentation markdown files (`.md`, `.mdx`) from the `docs/core` directory (markdown files, deployed docs).

```
@cocoindex.flow_def(name="DocsToKG")
def docs_to_kg_flow(flow_builder: cocoindex.FlowBuilder, data_scope: cocoindex.DataScope):
    data_scope["documents"] = flow_builder.add_source(
        cocoindex.sources.LocalFile(path="../../docs/docs/core",
                                    included_patterns=["*.md", "*.mdx"]))
```

Here `flow_builder.add_source` creates a KTable. `filename` is the key of the KTable.



Add data collectors

Add collectors at the root scope:

```
document_node = data_scope.add_collector()
entity_relationship = data_scope.add_collector()
entity_mention = data_scope.add_collector()
```

`document_node` collects documents. E.g., `core/basics.mdx` is a document.
`entity_relationship` collects relationships. E.g., "CocoIndex supports Incremental Processing" indicates a relationship between `CocoIndex` and `Incremental Processing`.

`entity_mention` collects mentions of entities in a document.
E.g., `core/basics.mdx` mentions `CocoIndex` and `Incremental Processing`.

Process each document and extract summary

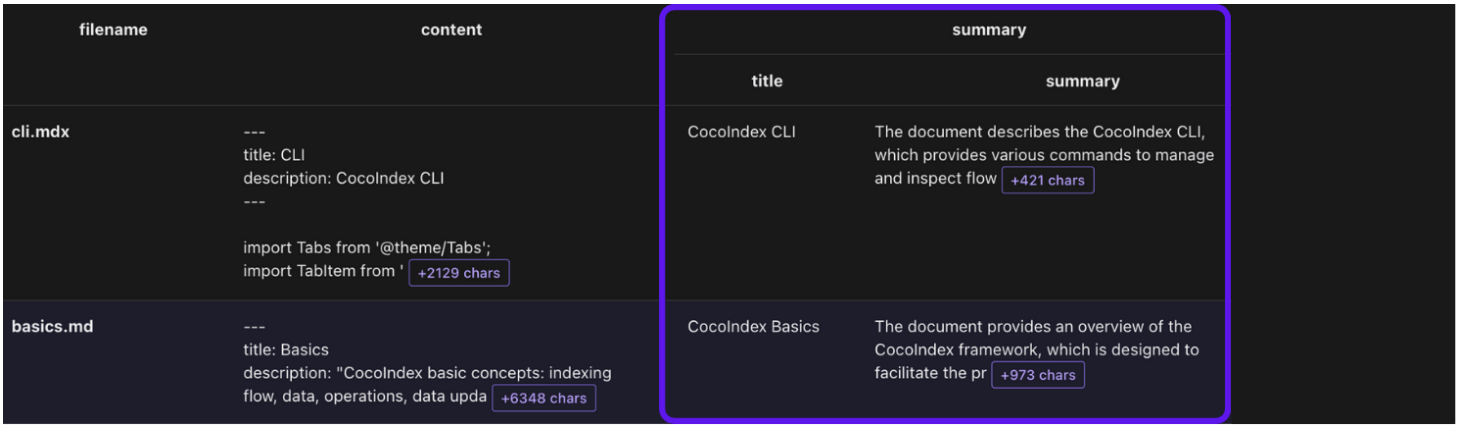
Define a `DocumentSummary` data class to extract the summary of a document.

```
@dataclasses.dataclass
class DocumentSummary:
    title: str
    summary: str
```

Within the flow, use `cocoindex.functions.ExtractByLlm` for structured output.

```
with data_scope["documents"].row() as doc:
    doc["summary"] = doc["content"].transform(
            cocoindex.functions.ExtractByLlm(
                llm_spec=cocoindex.LlmSpec(
                    api_type=cocoindex.LlmApiType.OPENAI, model="gpt-4o"),
                output_type=DocumentSummary,
                instruction="Please summarize the content of the document."))
    document_node.collect(
        filename=doc["filename"], title=doc["summary"]["title"],
        summary=doc["summary"]["summary"])
```

`doc["summary"]` adds a new column to the KTable `data_scope["documents"]`.



Extract relationships from the document using LLM

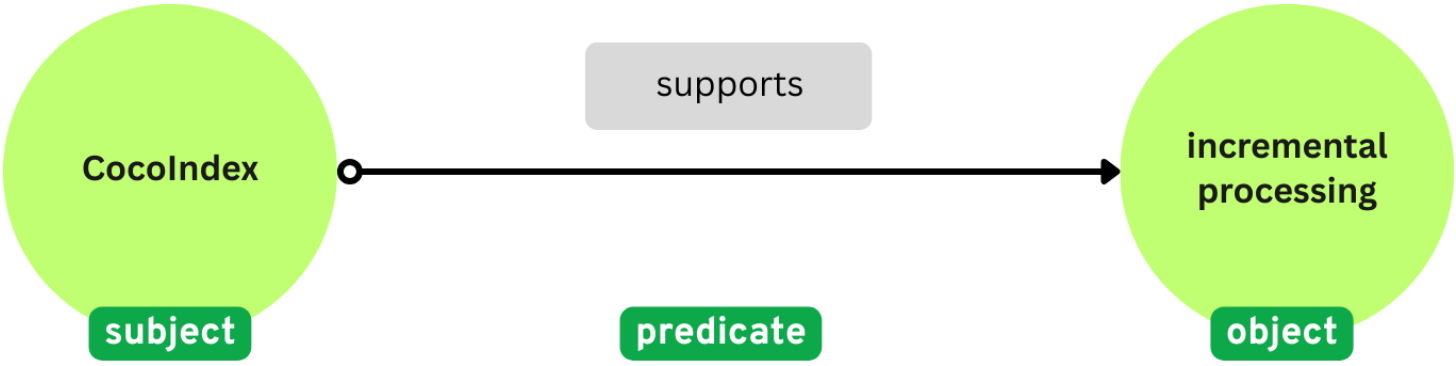
Define a data class to represent relationship for the LLM extraction.

```
@dataclasses.dataclass
class Relationship:
    """
    Describe a relationship between two entities.
    Subject and object should be Core CocoIndex concepts only, should be nouns. For example, `CocoIndex`, `Incremental Processing`, `ETL`, `Data` etc.
    """
    subject: str
    predicate: str
    object: str
```

The Data class defines a knowledge graph relationship. We recommend putting detailed instructions in the class-level docstring to help the LLM extract relationships correctly.

`subject`: Represents the entity the statement is about (e.g., 'CocoIndex').
`predicate`: Describes the type of relationship or property connecting the subject and object (e.g., 'supports').
`object`: Represents the entity or value that the subject is related to via the predicate (e.g., 'Incremental Processing').

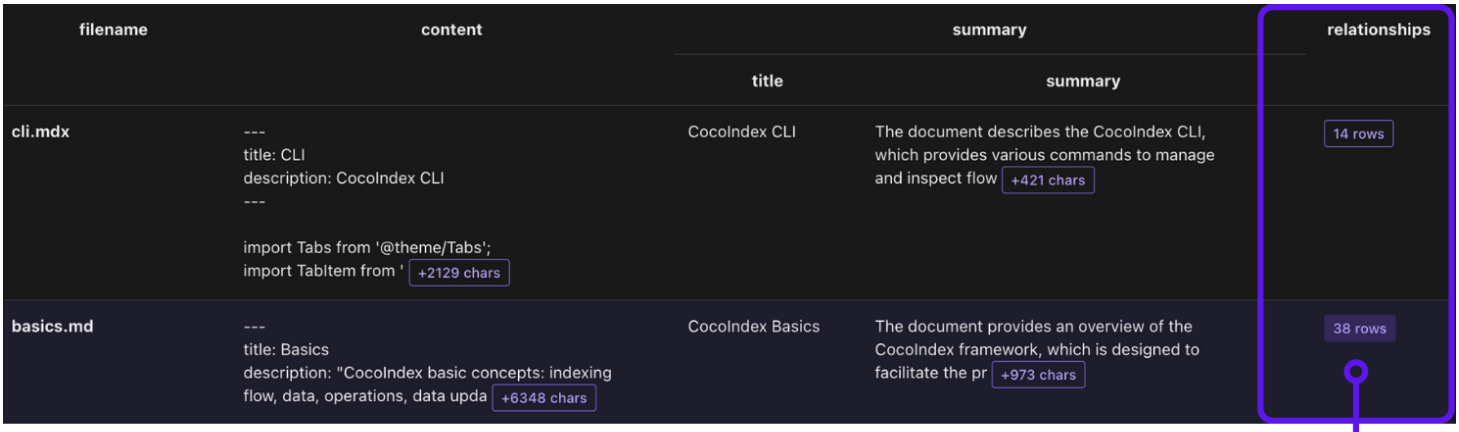
This structure represents facts like "CocoIndex supports Incremental Processing". Its graph representation is:



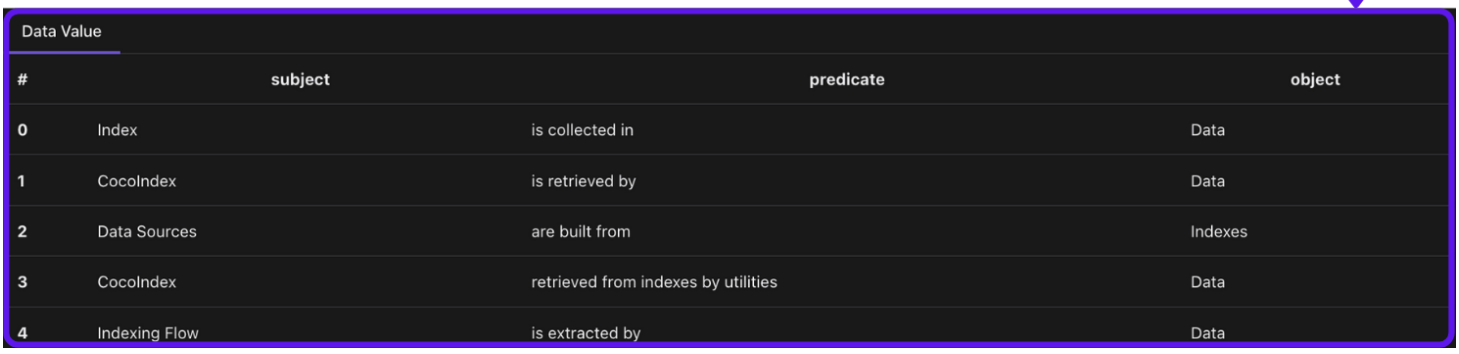
Next, we will use `cocoindex.functions.ExtractByLlm` to extract the relationships from the document.

```
doc["relationships"] = doc["content"].transform(
    cocoindex.functions.ExtractByLlm(
        llm_spec=cocoindex.LlmSpec(
            api_type=cocoindex.LlmApiType.OPENAI,
            model="gpt-4o"
        ),
        output_type=list[Relationship],
        instruction=(
            "Please extract relationships from CocoIndex documents. "
            "Focus on concepts and ignore examples and code. "
        )
    )
)
```

`doc["relationships"]` adds a new field `relationships` to each document. `output_type=list[Relationship]` specifies that the output of the transformation is a LTable.



filename	content	summary		relationships
		title	summary	
cli.mdx	--- title: CLI description: CocoIndex CLI --- import Tabs from '@theme/Tabs'; import TabItem from ' +2129 chars	CocoIndex CLI	The document describes the CocoIndex CLI, which provides various commands to manage and inspect flow +421 chars	14 rows
basics.md	--- title: Basics description: "CocoIndex basic concepts: indexing flow, data, operations, data upda +6348 chars	CocoIndex Basics	The document provides an overview of the CocoIndex framework, which is designed to facilitate the pr +973 chars	38 rows



Data Value

#	subject	predicate	object
0	Index	is collected in	Data
1	CocoIndex	is retrieved by	Data
2	Data Sources	are built from	Indexes
3	CocoIndex	retrieved from indexes by utilities	Data
4	Indexing Flow	is extracted by	Data

Collect relationships

```
with doc["relationships"].row() as relationship:
    # relationship between two entities
    entity_relationship.collect(
        id=cocoindex.GeneratedField.UUID,
        subject=relationship["subject"],
        object=relationship["object"],
        predicate=relationship["predicate"],
    )
    # mention of an entity in a document, for subject
    entity_mention.collect(
        id=cocoindex.GeneratedField.UUID, entity=relationship["subject"],
        filename=doc["filename"],
    )
    # mention of an entity in a document, for object
    entity_mention.collect(
        id=cocoindex.GeneratedField.UUID, entity=relationship["object"],
        filename=doc["filename"],
    )
```

`entity_relationship` collects relationships between subjects and objects.
`entity_mention` collects mentions of entities (as subjects or objects) in the document separately. For example, `core/basics.mdx` has a sentence `CocoIndex supports Incremental Processing`. We want to collect:
`core/basics.mdx` mentions `CocoIndex`.
`core/basics.mdx` mentions `Incremental Processing`.

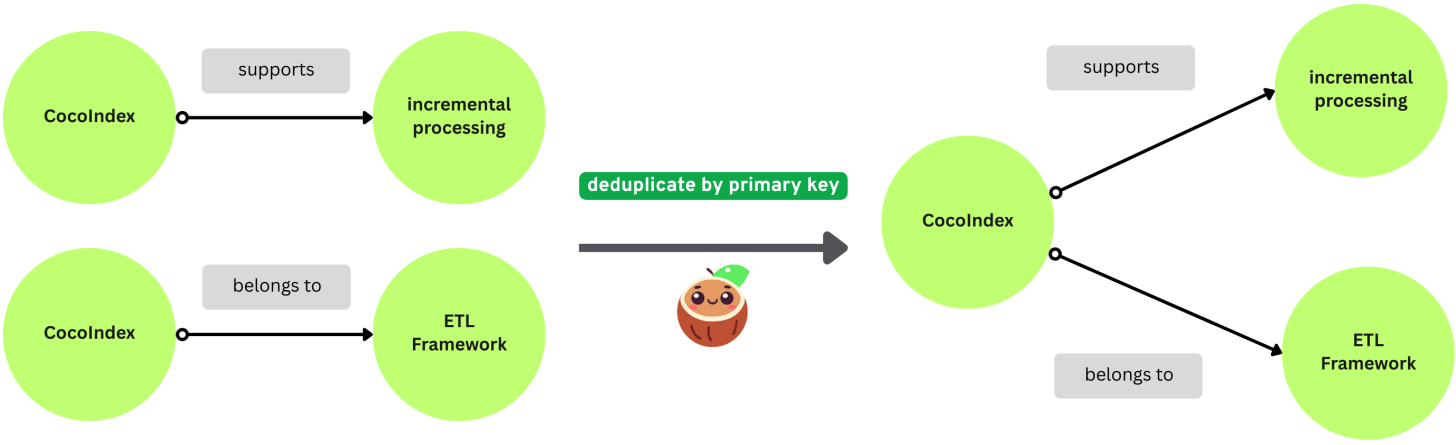
Build knowledge graph

Basic concepts

All nodes for Neo4j need two things:

. Label: The type of the node. E.g., `Document`, `Entity`.
. Primary key field: The field that uniquely identifies the node. E.g., `filename` for `Document` nodes.

CocoIndex uses the primary key field to match the nodes and deduplicate them. If you have multiple nodes with the same primary key, CocoIndex keeps only one of them.

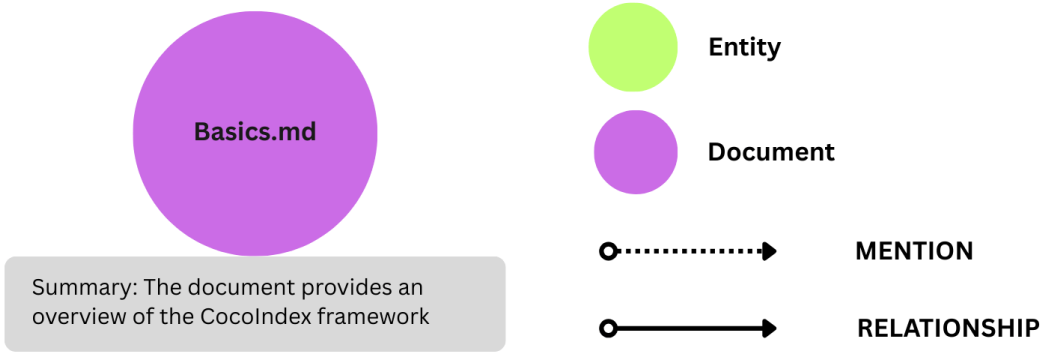


There are two ways to map nodes:
. When you have a collector just for the node, you can directly export it to Neo4j.
. When you have a collector for relationships connecting to the node, you can map nodes from selected fields in the relationship collector. You must declare a node label and primary key field.

Configure Neo4j connection:

```
conn_spec = cocoindex.add_auth_entry(
    "Neo4jConnection",
    cocoindex.storages.Neo4jConnection(
        uri="bolt://localhost:7687",
        user="neo4j",
        password="cocoindex",
))
```

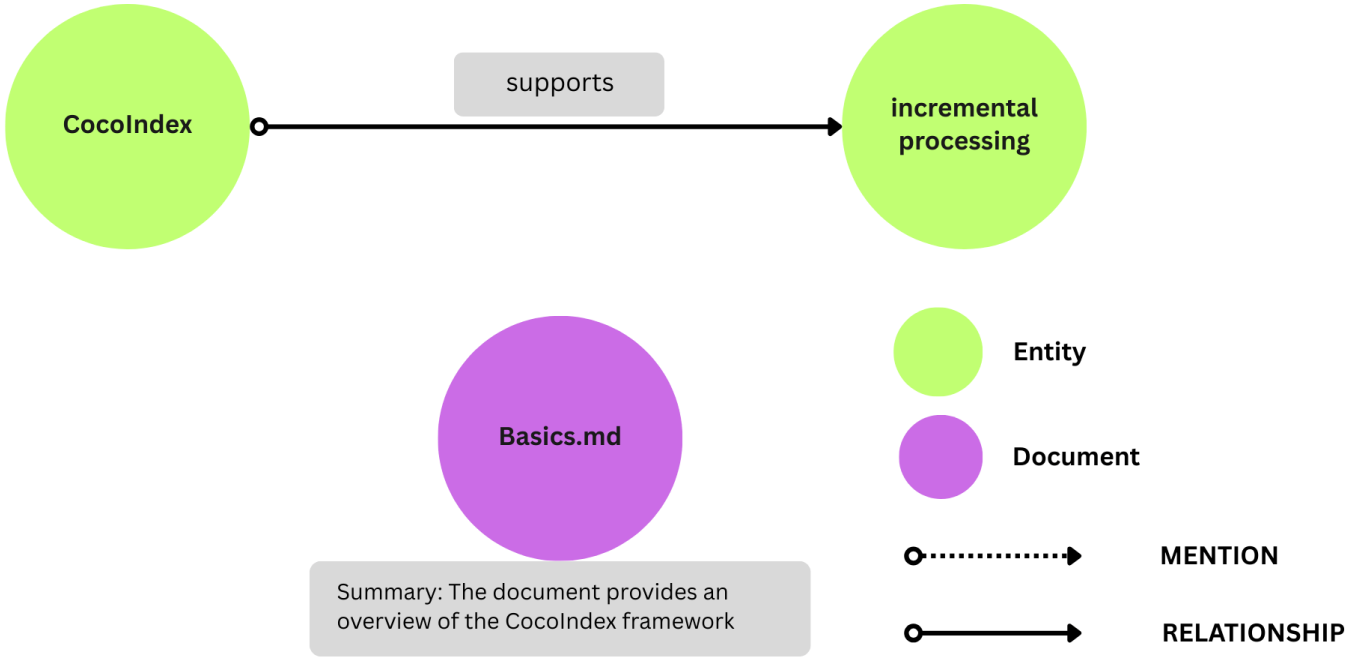
Export `Document` nodes to Neo4j




```
document_node.export(
    "document_node",
    cocoindex.storages.Neo4j(
        connection=conn_spec,
        mapping=cocoindex.storages.Nodes(label="Document")),
    primary_key_fields=["filename"],
)
```

This exports Neo4j nodes with label `Document` from the `document_node` collector.

It declares Neo4j node label `Document`. It specifies `filename` as the primary key field.
It carries all the fields from `document_node` collector to Neo4j nodes with label `Document`.
Export `RELATIONSHIP` and `Entity` nodes to Neo4j



We don't have explicit collector for `Entity` nodes. They are part of the `entity_relationship` collector and fields are collected during the relationship extraction.

To export them as Neo4j nodes, we need to first declare `Entity` nodes.

```
flow_builder.declare(
    cocoindex.storages.Neo4jDeclaration(
        connection=conn_spec,
        nodes_label="Entity",
        primary_key_fields=["value"],
    )
)
```

Next, export the `entity_relationship` to Neo4j.

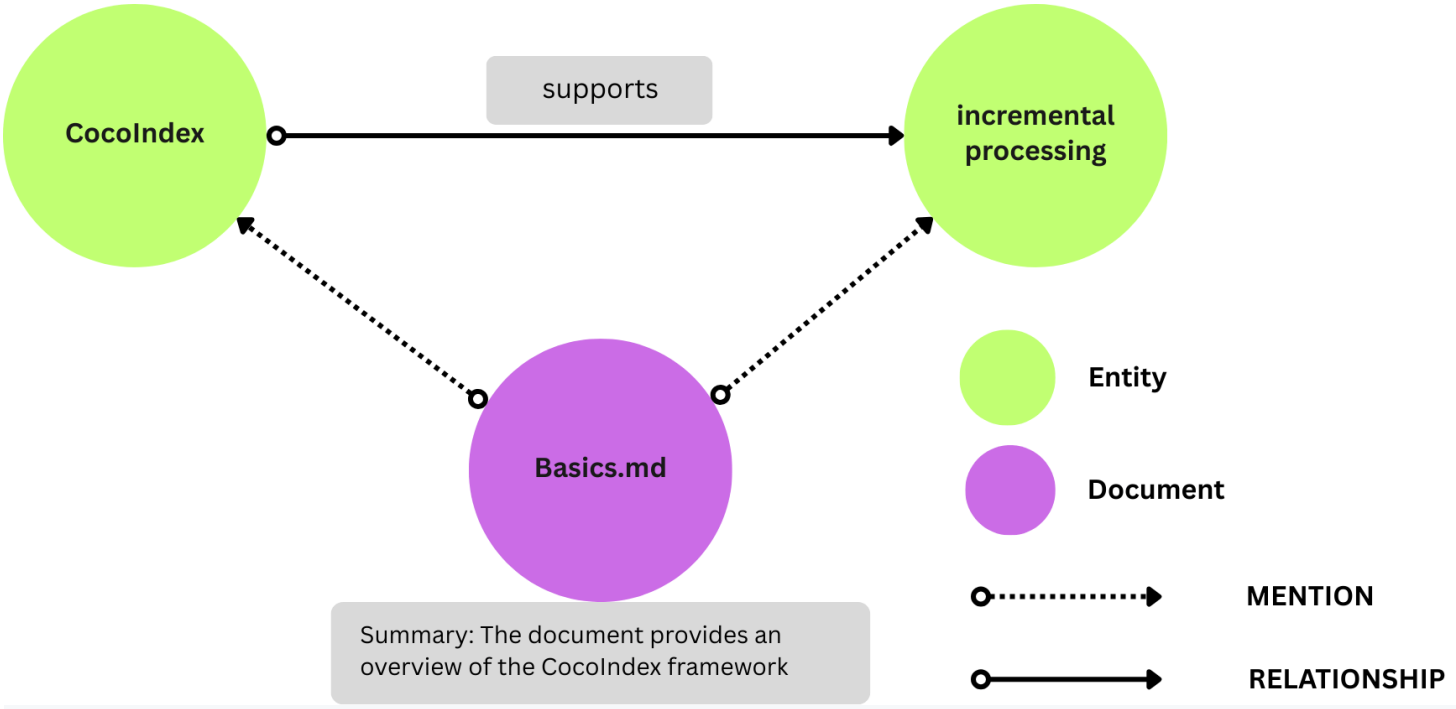
```
entity_relationship.export(
    "entity_relationship",
    cocoindex.storages.Neo4j(
        connection=conn_spec,
        mapping=cocoindex.storages.Relationships(
            rel_type="RELATIONSHIP",
            source=cocoindex.storages.NodeFromFields(
                label="Entity",
                fields=[
                    cocoindex.storages.TargetFieldMapping(
                        source="subject", target="value"),
                ]
            ),
            target=cocoindex.storages.NodeFromFields(
```

```
            label="Entity",
            fields=[
                cocoindex.storages.TargetFieldMapping(
                    source="object", target="value"),
            ]
        ),
    ),
  ),
  primary_key_fields=["id"],
)
)
```

The `cocoindex.storages.Relationships` declares how to map relationships in Neo4j.

In a relationship, there's:
. A source node and a target node.
. A relationship connecting the source and target. Note that different relationships may share the same source and target nodes.

`NodeFromFields` takes the fields from the `entity_relationship` collector and creates `Entity` nodes.
Export the `entity_mention` to Neo4j.




```
entity_mention.export(
  "entity_mention",
  cocoindex.storages.Neo4j(
    connection=conn_spec,
    mapping=cocoindex.storages.Relationships(
      rel_type="MENTION",
      source=cocoindex.storages.NodesFromFields(
        label="Document",
        fields=[cocoindex.storages.TargetFieldMapping("filename")],
      ),
      target=cocoindex.storages.NodesFromFields(
```

```
            label="Entity",
            fields=[cocoindex.storages.TargetFieldMapping(
                source="entity", target="value")],
        ),
    ),
  ),
  primary_key_fields=["id"],
)
```

Similarly here, we export `entity_mention` to Neo4j Relationships using `cocoindex.storages.Relationships`. It creates relationships by:

Creating `Document` nodes and `Entity` nodes from the `entity_mention` collector.
Connecting `Document` nodes and `Entity` nodes with relationship `MENTION`.

Query and test your index

🎉 Now you are all set!

. Install the dependencies:

```
pip install -e .
```

. Run following commands to setup and update the index.

```
cocoindex update --setup main.py
```

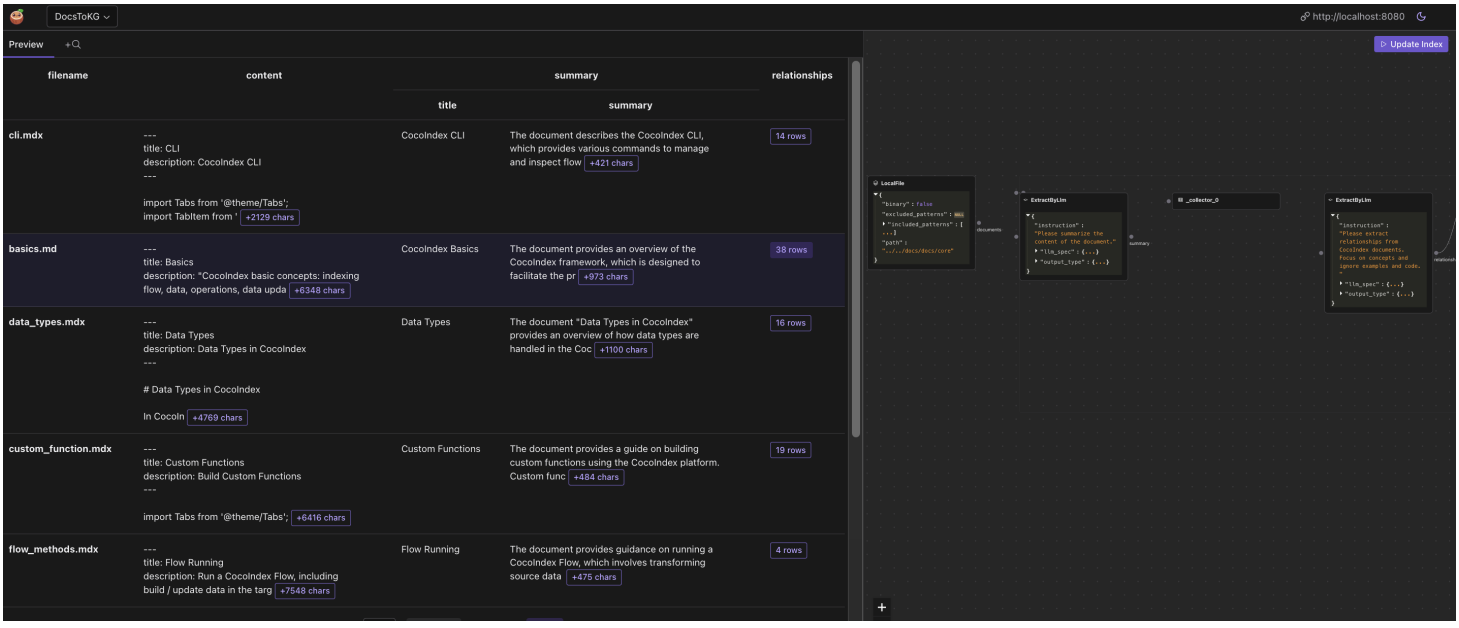
You'll see the index updates state in the terminal. For example, you'll see the following output:

```
documents: 7 added, 0 removed, 0 updated
```

. (Optional) I used CocoInsight to troubleshoot the index generation and understand the data lineage of the pipeline. It is in free beta now, you can give it a try. Run following command to start CocoInsight:

```
cocoindex server -ci main
```

And then open the url https://cocoindex.io/cocoinsight. It just connects to your local CocoIndex server, with Zero pipeline data retention.



Browse the knowledge graph

After the knowledge graph is built, you can explore the knowledge graph you built in Neo4j Browser.

For the dev environment, you can connect to Neo4j browser using credentials:

username: `Neo4j`
password: `cocoindex` which is pre-configured in our docker compose config.yaml.

You can open it at http://localhost:7474, and run the following Cypher query to get all relationships:

```
MATCH p=()-->() RETURN p
```

Support us

We are constantly improving, and more features and examples are coming soon. If you love this article, please give us a star ⭐ at GitHub repo to help us grow.

Thanks for reading!