# BrickLink Scraper: Technical Implementation and Data Flow

## Introduction

This document provides a comprehensive technical overview of the BrickLink Scraper project, explaining how the code works, the data flow, and the approach used to build this system. This scraper collects detailed information about LEGO sets, minifigures, and parts from Bricklink.com and organizes it into a structured SQLite database.

## System Architecture

The BrickLink Scraper is built with a modular, multi-stage architecture that separates concerns and allows for efficient, parallelized processing of large amounts of data.

## File Structure and Purpose

```
BrickLink Scraper/
├── index.py              # Main orchestrator script
├── item_scraper.py       # Retrieves item numbers
├── data_scraper.py       # Downloads HTML and images
├── parser.py             # Parses sets and minifigures data
├── parts_parser.py       # Specialized parser for parts
├── db_connection.py      # Database and connection management
├── Images/               # Downloaded images folder
└── output/               # Excel reports folder
```

### Each file's purpose:

1. **index.py**
   - The central orchestrator that coordinates the entire scraping process
   - Contains configuration for all categories (sets, minifigures, parts)
   - Manages the execution sequence and handles errors
   - Creates necessary folders and database connections
2. **item_scraper.py**
   - Responsible for retrieving all item numbers from catalog pages
   - Uses multi-threading to accelerate catalog page processing
   - Extracts and returns clean lists of item numbers
3. **data_scraper.py**
   - Downloads all HTML data and images for each item
   - Implements respectful web scraping with appropriate headers and rate limiting

- Stores raw HTML in a database for later processing

4. **parser.py**
   - Extracts structured data from HTML for sets and minifigures
   - Parses multiple aspects: item details, prices, relationships
   - Converts raw data into normalized database entries
5. **parts_parser.py**
   - Specialized parser for parts with enhanced capabilities
   - Extracts additional part-specific data like stud dimensions
   - Captures color information including appearance type
6. **db_connection.py**
   - Manages database structure, indexes, and relationships
   - Creates connection tables between sets, minifigures, and parts
   - Optimizes database performance and generates statistics

## Data Flow and Code Execution

The system follows a four-stage process, with each stage building upon the previous one:

### 1. Item Number Collection

**Entry Point**: `index.py` → `get_all_items()` in `item_scraper.py`

This stage collects all the unique item identifiers from Bricklink's catalog pages:

python

```python
# Execution flow in index.py
for category, config in CONFIG.items():
    logger.info(f"Getting {category} items...")
    items = get_all_items(config['pages'], config['cat_type'])
    all_items[category] = items
```

The `get_all_items()` function creates a thread pool to parallelize the scraping of catalog pages:

python

```python
# In item_scraper.py
with ThreadPoolExecutor(max_workers=20) as executor:
    for page in range(1, page_no + 1):
        url = f"https://www.bricklink.com/catalogList.asp?pg={page}&catLike=W&sortBy=I&sortAsc=
        processes.append(executor.submit(get_item, url, cattype))
```

Each thread processes one catalog page, extracting item numbers and returning them to the main process.

### 2. HTML and Image Downloading

**Entry Point**: `index.py` → `process_items()` in `data_scraper.py`

Once we have all item numbers, the system downloads the HTML data and images:

```python
# In index.py
for category, config in CONFIG.items():
    logger.info(f"Processing {category}...")
    table = raw_db[config['table_name']]

    process_items(
        item_nos=all_items[category],
        cat_type=config['cat_type'],
        table=table,
        is_parts=(config['cat_type'] == 'P')
    )
```

The `process_items()` function creates a thread pool to download data for multiple items in parallel:

```python
# In data_scraper.py
with ThreadPoolExecutor(max_workers=20) as executor:
    for item_no in item_nos:
        processes.append(executor.submit(process_item, item_no, cat_type, is_parts))
```

For each item, the system downloads:

- The main item page (with general information)
- The price guide page (with pricing statistics)
- The inventory page (showing what components are in the item)
- The item's image (saved to the Images folder)

All HTML data is stored in a SQLite database for later processing.

### 3. Data Parsing

**Entry Point**: `index.py` → `parse_all_data()` or `parse_all_parts_data()`

The system now processes the raw HTML to extract structured data:

python

```python
# In index.py
for category, config in CONFIG.items():
    logger.info(f"Parsing {category} data...")
    raw_table = raw_db[config['table_name']]
    parsed_table = parsed_db[config['parse_table']]

    # Use specialized parts parser for parts
    if category == 'parts':
        color_table = parsed_db[config['parse_color_table']]
        parsed_data, color_data = parse_all_parts_data(
            raw_table=raw_table,
            parsed_table=parsed_table,
            color_table=color_table,
            batch_size=1000,
            output_excel=f"output/{config['excel_output']}"
        )
    else:
        # Parse other data with standard parser
        parsed_data = parse_all_data(
            raw_table=raw_table,
            parsed_table=parsed_table,
            is_parts=(config['cat_type'] == 'P'),
            batch_size=1000,
            output_excel=f"output/{config['excel_output']}"
        )
```

The parsing functions process the data in batches to optimize memory usage:

python

```python
# In parser.py or parts_parser.py
for batch in get_records_in_batches(raw_table, batch_size):
    batch_args = [(row, is_parts) for row in batch]

    with Pool(processes=num_workers) as pool:
        results = pool.map(process_row, batch_args)
```

Each row is processed by the `process_row()` function, which:

1. Extracts item details from the main page
2. Extracts pricing information from the price page
3. Extracts relationship data from the inventory page
4. For parts, extracts color information

The parsed data is stored in the appropriate tables and exported to Excel.

### 4. Relationship Building

**Entry Point**: `index.py` → `create_connection_table()` and `populate_connection_table_batch()`

Finally, the system creates relationships between the different entities:

```python
# In index.py
create_connection_table('bricklink_parse.db')
populate_connection_table_batch('bricklink_parse.db', batch_size=1000)
```

The `populate_connection_table_batch()` function processes the sets and minifigures tables to extract relationship data:

```python
# In db_connection.py
# Process sets table
cursor.execute("SELECT item_number, parts_relation, minifigs_relation FROM set_parse")
sets_data = cursor.fetchall()

for set_row in sets_data:
    set_id = set_row[0]
    parts_relation = set_row[1]
    minifigs_relation = set_row[2]

    # Process parts relation
    if parts_relation:
        parts = [p.strip() for p in parts_relation.split(',') if p.strip()]
        for part_id in parts:
            batch_data.append((set_id, None, part_id))
```

This creates a normalized connection table that captures:

- Which parts are in which sets
- Which minifigures are in which sets
- Which parts make up which minifigures

## Key Technical Approaches

### 1. Parallel Processing

The system uses both multi-threading and multi-processing:

- **Multi-threading** for I/O-bound operations (downloading)

```python
with ThreadPoolExecutor(max_workers=20) as executor:
    # Submit download tasks
```

- **Multi-processing** for CPU-bound operations (parsing)

```python
with Pool(processes=num_workers) as pool:
    # Submit parsing tasks
```

This maximizes throughput by efficiently utilizing system resources.

## 2. Batch Processing

The system processes data in batches to manage memory usage:

```python
def get_records_in_batches(table, batch_size=1000):
    offset = 0
    while True:
        records = list(table.find(_limit=batch_size, _offset=offset))
        if not records:
            break
        yield records
        offset += batch_size
```

This allows processing of very large datasets without running out of memory.

## 3. Resilient Error Handling

The system implements retry logic and robust error handling:

python

```python
def process_item(item_no, cat_type, is_parts=False):
    max_retries = 3
    attempt = 0
    while attempt < max_retries:
        attempt += 1
        try:
            # Processing code
            return result
        except Exception as e:
            print(f"❌ Error processing {item_no}: {e}")
            if attempt == max_retries:
                return None
```

This ensures that temporary network failures or server issues don't crash the entire process.

### 4. Database Optimization

The system applies SQLite optimizations for better performance:

python

```python
# Set SQLite pragmas for better performance
cursor.execute("PRAGMA journal_mode=DELETE")  # Avoid huge WAL files
cursor.execute("PRAGMA synchronous=NORMAL")   # Faster writes
cursor.execute("PRAGMA temp_store=MEMORY")    # Use memory for temp tables
cursor.execute("PRAGMA cache_size=-64000")    # 64MB cache
```

It also creates appropriate indexes to speed up queries:

python

```python
cursor.execute("CREATE INDEX IF NOT EXISTS idx_set_id ON connection_table(set_id)")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_minifigure_id ON connection_table(minifigure_id)
cursor.execute("CREATE INDEX IF NOT EXISTS idx_part_id ON connection_table(part_id)")
```

## Advanced Features

### 1. Specialized Parts Parser

The system includes a specialized parser for parts that extracts additional information:

python

```python
def parts_parser(html, image_path, item_no):
    # Basic information extraction

    # Parts-specific extraction
    stud_dim_match = re.search(r"Stud Dim\.:\s*((?:\d+(?:\.\d+)?\s*x\s*)*\d+(?:\.\d+)?\s*(?:in
    if stud_dim_match:
        data["stud_dimensions"] = stud_dim_match.group(1).strip().replace("?", "")

    # Pack Dimensions
    pack_dim_match = re.search(r"Pack\. Dim\.:\s*((?:\d+(?:\.\d+)?\s*x\s*)*\d+(?:\.\d+)?\s*(?:i
    if pack_dim_match:
        data["pack_dimensions"] = pack_dim_match.group(1).strip().replace("?", "")
```

## 2. Color Data Extraction

The system captures detailed color information for parts:

python

```python
def parse_color_summary_table(soup, item_number=None):
    # Find color sections
    for row in table.find_all("tr"):
        # Process color data
        results.append({
            "item_number": item_number,
            "appears_as": current_section,
            "color": color,
            "in_sets": int(in_sets) if in_sets.isdigit() else 0,
            "total_qty": int(total_qty) if total_qty.isdigit() else 0
        })
```

This provides valuable information about which colors each part appears in and how commonly it's used.

## 3. Comprehensive Relationship Tracking

The system builds a complete graph of relationships between sets, minifigures, and parts:

```python
# Set contains Parts
batch_data.append((set_id, None, part_id))

# Set contains Minifigures
batch_data.append((set_id, minifig_id, None))

# Minifigure contains Parts
batch_data.append((None, minifig_id, part_id))
```

This allows for complex queries like:

- Which sets contain a specific part?
- Which minifigures appear in multiple sets?
- What are all the parts needed to build a specific set?

## Database Schema

The system creates a 3rd normal form database with these main tables:

### 1. set_parse

Stores information about LEGO sets including name, category, year, and pricing.

### 2. minifigures_parse

Stores information about minifigures including name, category, year, and pricing.

### 3. part_parse

Stores information about parts including name, category, dimensions, and pricing.

### 4. part_parse_color

Stores color information for parts including appearance type and quantity.

### 5. connection_table

Stores relationships between sets, minifigures, and parts.

## Performance Considerations

The system is designed with performance in mind:

1. **Database Indexes**: Critical fields are indexed for faster queries
2. **Batch Processing**: Data is processed in manageable chunks
3. **Parallel Processing**: Multiple CPU cores and threads are utilized

4. **Memory Management**: Large objects are released when no longer needed
5. **SQLite Optimizations**: Pragmas configured for better performance

## Conclusion

This BrickLink scraper represents a comprehensive solution for collecting, organizing, and analyzing LEGO data. The modular architecture, parallel processing capabilities, and optimization techniques allow it to efficiently handle large datasets while maintaining data integrity and respecting web scraping best practices.

The resulting database provides a valuable resource for understanding the relationships between LEGO sets, minifigures, and parts, enabling sophisticated queries and analyses.