# **Data Engineering Interview Coding Questions**

This document contains logical coding questions commonly asked in data engineering interviews, organized by relevant topic areas. Each section includes questions with increasing complexity levels to help you prepare thoroughly.

## **Table of Contents**

* [Python Fundamentals](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#python-fundamentals)
* [Data Structures & Algorithms](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#data-structures--algorithms)
* [File Processing](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#file-processing)
* [Data Manipulation & Transformation](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#data-manipulation--transformation)
* [Database & SQL](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#database--sql)
* [ETL & Data Pipelines](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#etl--data-pipelines)
* [System Design & Optimization](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#system-design--optimization)
* [Stream Processing](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#stream-processing)
* [Error Handling & Edge Cases](https://claude.ai/chat/39d8e7c7-f2a8-4cf9-9143-4fd65dd57af1#error-handling--edge-cases)

## **Python Fundamentals**

### **Easy**

1. **List Comprehension**
   * Write a list comprehension to extract all even numbers from a list of integers.
   * Write a list comprehension to transform a list of strings to uppercase.
2. **Dictionary Manipulation**
   * Create a function that merges two dictionaries and handles duplicate keys.
   * Write a function to invert a dictionary (swap keys and values).
3. **String Processing**
   * Write a function to count the occurrence of each word in a text.
   * Create a function that validates and reformats date strings from "MM-DD-YYYY" to "YYYY-MM-DD".

### **Medium**

1. **Generator Functions**
   * Write a generator function that yields lines from a large file without loading the entire file into memory.
   * Create a generator that produces Fibonacci numbers up to a given limit.
2. **Lambda Functions**
   * Write a lambda function to sort a list of tuples by the second element.
   * Use map and filter with lambda functions to process a list of data records.
3. **Function Decorators**
   * Create a decorator that measures and logs the execution time of a function.
   * Write a decorator that implements a retry mechanism for functions that may fail.

### **Hard**

1. **Advanced Functions**
   * Implement a function that flattens nested lists to any depth.
   * Create a function that converts a nested dictionary to a flattened dictionary with dot notation for keys.
2. **Context Managers**
   * Write a context manager for safely handling database connections.
   * Create a custom context manager for timing code blocks.

## **Data Structures & Algorithms**

### **Easy**

1. **List Operations**
   * Write a function to find duplicate elements in a list.
   * Implement a function to rotate a list by k positions.
2. **Dictionary Operations**
   * Create a function that groups a list of dictionaries by a specific key.
   * Write a function that finds the most frequent element in a list.

### **Medium**

1. **Efficient Search**
   * Implement a binary search function for a sorted list.
   * Write a function to find the intersection of two unsorted lists efficiently.
2. **Sorting Algorithms**
   * Implement quicksort for sorting a list of integers.
   * Write a function to sort a large file of integers that doesn't fit in memory.
3. **Graphs**
   * Implement a function to detect if a directed graph has cycles.
   * Create a function to find the shortest path between two nodes in a graph.

### **Hard**

1. **Advanced Data Structures**
   * Implement a trie data structure for efficiently storing and retrieving strings.
   * Create a least-recently-used (LRU) cache with get and put operations.
2. **Priority Algorithms**
   * Implement a priority queue for task scheduling in a data pipeline.
   * Write an algorithm to find the top K frequent elements in a stream of data.

## **File Processing**

### **Easy**

1. **Basic File Operations**
   * Write a function that counts lines, words, and characters in a text file.
   * Create a function to find and replace text in a file.
2. **CSV Processing**
   * Write a function to extract specific columns from a CSV file.
   * Create a function that validates CSV data against a schema.

### **Medium**

1. **Large File Handling**
   * Implement a function to process a large CSV file line by line efficiently.
   * Write a function to merge multiple sorted files into a single sorted file.
2. **Advanced Formats**
   * Create a function that converts data between CSV, JSON, and Parquet formats.
   * Write code to extract specific fields from nested JSON structures.

### **Hard**

1. **Complex File Operations**
   * Implement chunked processing for a multi-gigabyte file with memory constraints.
   * Create a function that performs incremental processing on appending log files.
2. **Custom Parsers**
   * Write a custom parser for a complex log file format.
   * Implement a function to efficiently parse and validate XML data.

## **Data Manipulation & Transformation**

### **Easy**

1. **Basic Transformations**
   * Write a function to normalize values in a list (scale to 0-1 range).
   * Create a function that pivots tabular data.
2. **Data Cleaning**
   * Implement a function to handle missing values in a dataset.
   * Write code to detect and remove outliers from a dataset.

### **Medium**

1. **Advanced Transformations**
   * Create a function that performs a rolling window aggregation on time series data.
   * Write a function to reformat nested JSON data into a flat structure.
2. **Pandas Operations**
   * Implement an efficient function to merge and aggregate large DataFrames.
   * Write a function that performs complex group-by operations with multiple conditions.

### **Hard**

1. **Complex Aggregations**
   * Create a function to compute moving averages with variable windows.
   * Implement a solution for sessionization of event data based on time thresholds.
2. **Data Reshaping**
   * Write a function to efficiently transpose a large sparse matrix.
   * Create a solution for de-duplicating records based on fuzzy matching criteria.

## **Database & SQL**

### **Easy**

1. **Basic Queries**
   * Write SQL to find the top 5 customers by purchase amount.
   * Create a query to find duplicate records in a table.
2. **Simple Joins**
   * Write a SQL query that joins multiple tables to get product sales by category.
   * Create a SQL query that finds employees who earn more than their managers.

### **Medium**

1. **Complex Joins**
   * Write a SQL query using window functions to rank products by sales within each category.
   * Create a query that performs a self-join to find all employee hierarchies.
2. **Advanced Aggregations**
   * Implement a SQL query for running totals by date.
   * Write a query for calculating month-over-month percentage change.
3. **Subqueries**
   * Create a SQL query using subqueries to find departments with above-average salaries.
   * Write a query using common table expressions (CTEs) to analyze customer purchase patterns.

### **Hard**

1. **Performance Optimization**
   * Rewrite a slow-performing SQL query to improve execution time.
   * Create an indexing strategy for a set of queries on a large table.
2. **Complex Analytics**
   * Implement a SQL solution for market basket analysis.
   * Write a query for calculating cohort retention rates.

## **ETL & Data Pipelines**

### **Easy**

1. **Simple Pipelines**
   * Write a function that extracts data from a CSV, transforms date formats, and loads to a new file.
   * Create a script that monitors a directory for new files and processes them.
2. **Data Validation**
   * Implement validation checks for data quality in a pipeline.
   * Write code to handle and log data exceptions during processing.

### **Medium**

1. **Pipeline Components**
   * Create a configurable data extraction function that supports multiple sources.
   * Implement a transformation system with pluggable processors.
2. **Scheduling & Orchestration**
   * Write code that manages dependencies between pipeline tasks.
   * Create a solution for retry logic with exponential backoff.

### **Hard**

1. **Advanced Pipeline Design**
   * Design and implement a streaming ETL pipeline with exactly-once processing guarantees.
   * Create a data pipeline that handles late-arriving data and updates historical aggregations.
2. **Performance Optimization**
   * Write code to parallelize data processing tasks efficiently.
   * Implement backpressure handling in a data streaming pipeline.

## **System Design & Optimization**

### **Easy**

1. **Basic Optimization**
   * Write code to profile and identify bottlenecks in a Python function.
   * Implement memory usage optimization for a data processing task.
2. **Simple Concurrency**
   * Create a solution using threading for concurrent API requests.
   * Write code that uses connection pooling for database operations.

### **Medium**

1. **Advanced Concurrency**
   * Implement a producer-consumer pattern for processing a stream of data.
   * Create a solution using multiprocessing for CPU-bound data tasks.
2. **Resource Management**
   * Write code that implements rate limiting for external API calls.
   * Create a function that manages database connection resources efficiently.

### **Hard**

1. **Distributed Processing**
   * Design a distributed system for processing large datasets across multiple nodes.
   * Implement a data partitioning strategy for balanced load distribution.
2. **Complex System Design**
   * Create a design for a fault-tolerant data pipeline with automatic recovery.
   * Implement a solution for near-real-time data synchronization between systems.

## **Stream Processing**

### **Easy**

1. **Basic Streaming**
   * Write a function that processes data in a streaming fashion.
   * Implement a simple message consumer that processes records sequentially.
2. **Windowing**
   * Create code that computes metrics over a sliding window of time.
   * Write a function to detect patterns in a stream of events.

### **Medium**

1. **Advanced Stream Processing**
   * Implement a stateful stream processor that maintains aggregations.
   * Create a solution for handling out-of-order events in a data stream.
2. **Time-Based Processing**
   * Write code that processes events based on event time versus processing time.
   * Implement watermark-based processing for streaming data.

### **Hard**

1. **Complex Event Processing**
   * Create a system that detects complex patterns across multiple event streams.
   * Implement a solution for anomaly detection in real-time data streams.
2. **Stream Joins**
   * Write code that joins multiple data streams with different arrival rates.
   * Implement a solution for enriching streaming data with slowly changing reference data.

## **Error Handling & Edge Cases**

### **Easy**

1. **Basic Error Handling**
   * Write functions with appropriate exception handling for common data processing errors.
   * Implement data validation with custom exceptions.
2. **Input Validation**
   * Create robust input validation for a data processing function.
   * Write code that handles and standardizes different date formats.

### **Medium**

1. **Advanced Error Handling**
   * Implement a comprehensive error handling strategy for a multi-stage pipeline.
   * Create a solution for graceful degradation when external systems fail.
2. **Fault Tolerance**
   * Write code that implements circuit breaker patterns for external API calls.
   * Implement checkpointing in a long-running data process.

### **Hard**

1. **Recovery Mechanisms**
   * Design and implement a system for automatic recovery from pipeline failures.
   * Create a solution for exactly-once processing guarantees despite failures.
2. **Complex Edge Cases**
   * Write code that handles timezone complexities in global data.
   * Implement a solution for managing slowly changing dimensions in a data warehouse.

## **Example Solutions**

Here are a few examples of solutions to some of the problems mentioned above:

### **Example 1: Processing a large file efficiently**

def process\_large\_file(filepath, process\_line\_func):

"""Process a large file line by line without loading it all into memory."""

with open(filepath, 'r') as file:

for line in file:

process\_line\_func(line.strip())

# Example usage

def count\_words\_per\_line(line):

word\_count = len(line.split())

print(f"Line has {word\_count} words")

process\_large\_file('large\_data.txt', count\_words\_per\_line)

### **Example 2: Finding duplicates in a list efficiently**

def find\_duplicates(items):

"""Find duplicate elements in a list and return them."""

seen = set()

duplicates = set()

for item in items:

if item in seen:

duplicates.add(item)

else:

seen.add(item)

return list(duplicates)

# Example usage

items = [1, 2, 3, 2, 4, 5, 3, 6]

duplicates = find\_duplicates(items)

print(f"Duplicate items: {duplicates}") # Output: Duplicate items: [2, 3]

### **Example 3: Implementing a retry decorator**

import time

import random

from functools import wraps

def retry(max\_attempts=3, delay=1, backoff=2, exceptions=(Exception,)):

"""

Retry decorator with exponential backoff for functions that might fail.

Parameters:

- max\_attempts: Maximum number of retry attempts

- delay: Initial delay between retries in seconds

- backoff: Multiplier for delay between retries

- exceptions: Tuple of exceptions to catch and retry

"""

def decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

attempts = 0

current\_delay = delay

while attempts < max\_attempts:

try:

return func(\*args, \*\*kwargs)

except exceptions as e:

attempts += 1

if attempts == max\_attempts:

raise

print(f"Attempt {attempts} failed with error: {str(e)}")

print(f"Retrying in {current\_delay} seconds...")

time.sleep(current\_delay)

current\_delay \*= backoff

return wrapper

return decorator

# Example usage

@retry(max\_attempts=5, delay=1, backoff=2, exceptions=(ConnectionError, TimeoutError))

def fetch\_data\_from\_api(url):

# Simulate API call that sometimes fails

if random.random() < 0.7: # 70% chance of failure

raise ConnectionError("API connection failed")

return {"data": "Success!"}

# Testing the function

try:

result = fetch\_data\_from\_api("https://api.example.com/data")

print(f"Result: {result}")

except Exception as e:

print(f"All attempts failed with error: {str(e)}")

### **Example 4: Custom context manager for timing code blocks**

import time

from contextlib import contextmanager

@contextmanager

def timing\_block(name):

"""

Context manager to measure and print execution time of a code block.

Parameters:

- name: Name of the code block for reporting

Usage:

with timing\_block("data processing"):

# code to be timed

"""

start\_time = time.time()

try:

yield

finally:

elapsed\_time = time.time() - start\_time

print(f"[{name}] completed in {elapsed\_time:.4f} seconds")

# Example usage

with timing\_block("data processing"):

# Simulate some data processing

total = 0

for i in range(10000000):

total += i

print(f"Processing result: {total}")