DataHub (Acryl Data) Backend Engineer Interview Preparation Guide

Comprehensive Solutions & Strategies

November 9, 2025

Contents

1 Introduction

1.1 About DataHub

DataHub is the leading open-source metadata platform that helps organizations discover, understand, and trust their data. Founded by LinkedIn engineers and now commercialized by Acryl Data (founded 2020, ˜50 employees), DataHub provides a comprehensive solution for data cataloging, lineage tracking, and data governance.

1.2 Interview Process

Based on typical data infrastructure companies:

1. **Recruiter Screen** - Background and interest (30 min)
2. **Technical Phone Screen** - LeetCode medium + data structures (45-60 min)
3. **Take-Home Assignment** - Metadata API or pipeline (2-4 hours)
4. **Onsite Interviews** (4-5 hours):
    - System Design - Metadata catalog design
    - Coding - Graphs, trees, search algorithms
    - Technical Deep Dive - Past projects
    - Cultural Fit - Team collaboration

1.3 Tech Stack

DataHub's architecture:

- **Backend**: Java (Spring Boot), Python
- **Storage**: MySQL/PostgreSQL, Elasticsearch
- **Graph**: Neo4j for lineage
- **Streaming**: Apache Kafka
- **Frontend**: React, TypeScript
- **Infrastructure**: Kubernetes, Docker

1.4 Key Skills to Demonstrate

- Strong graph algorithm knowledge (critical!)
- Distributed systems and scalability
- Data engineering fundamentals
- Search and indexing expertise
- System design thinking

2 Problem 1: Metadata Lineage Graph

2.1 Problem Statement

DataHub tracks data lineage to understand dependencies between datasets. Implement a system to find all downstream dependencies of a given dataset and detect circular dependencies.

Requirements:

- Build a directed graph of dataset dependencies
- Find all datasets that depend on a given dataset (downstream)
- Detect circular dependencies
- Return dependencies in traversal order

Example:

```
lineage = MetadataLineage()
lineage.add_dependency("table_c", "table_a")  # c depends on a
lineage.add_dependency("table_c", "table_b")  # c depends on b
lineage.add_dependency("table_d", "table_c")  # d depends on c

lineage.get_downstream("table_a")  # Returns ["table_c", "table_d"]
lineage.has_circular_dependency()  # Returns False
```

2.2 Solution

```
from typing import List, Dict, Set
from collections import defaultdict, deque

class MetadataLineage:
    def __init__(self):
        self.graph = defaultdict(list)  # source -> [targets that depend on it]
        self.all_nodes = set()

    def add_dependency(self, target: str, source: str) -> None:
        self.graph[source].append(target)
        self.all_nodes.add(source)
        self.all_nodes.add(target)

    def get_downstream(self, dataset: str) -> List[str]:
        visited = set()
        queue = deque([dataset])

        while queue:
            node = queue.popleft()
            if node in visited:
                continue
            visited.add(node)

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

        visited.discard(dataset)  # Don't include the source dataset
        return list(visited)
```

```
30
31      def has_circular_dependency(self) -> bool:
32          visited = set()
33          rec_stack = set()
34
35          def dfs(node):
36              visited.add(node)
37              rec_stack.add(node)
38
39              for neighbor in self.graph[node]:
40                  if neighbor not in visited:
41                      if dfs(neighbor):
42                          return True
43                  elif neighbor in rec_stack:
44                      return True
45
46              rec_stack.remove(node)
47              return False
48
49          for node in self.all_nodes:
50              if node not in visited:
51                  if dfs(node):
52                      return True
53
54          return False
```

2.3 Complexity Analysis

- **Time Complexity**: O(V + E) for both operations where V is vertices (datasets), E is edges (dependencies)
- **Space Complexity**: O(V + E) for graph storage plus O(V) for traversal

2.4 Key Insights

1. BFS efficiently finds all reachable nodes (downstream dependencies)
2. DFS with recursion stack detects cycles
3. Directed graph naturally models data lineage (A produces B)
4. Production systems need to handle millions of nodes and edges

2.5 Production Enhancements

- **Graph Database**: Use Neo4j for efficient graph queries at scale
- **Caching**: Cache frequently accessed lineage paths
- **Incremental Updates**: Only recompute affected subgraphs
- **Column-Level Lineage**: Track individual column dependencies
- **Visualization**: Generate visual lineage graphs for users

3 Problem 2: Metadata Search Index

3.1 Problem Statement

DataHub needs efficient search across dataset metadata. Implement a search index that supports prefix matching and ranks results by relevance.

Requirements:

- Index datasets with name, description, and tags
- Support prefix search (e.g., "user" matches "user_events")
- Rank results by relevance (exact match > prefix match)
- Return top K results

Example:

```
index = MetadataSearchIndex()
index.add_dataset("user_events", "User activity events", ["analytics"])
index.add_dataset("user_profile", "User profile data", ["users"])
index.add_dataset("product_sales", "Sales transactions", ["sales"])

index.search("user", k=2)  # Returns ["user_events", "user_profile"]
index.search("sal", k=1)   # Returns ["product_sales"]
```

3.2 Solution

```
from typing import List, Dict

class MetadataSearchIndex:
    def __init__(self):
        self.datasets = {}  # name -> {description, tags}

    def add_dataset(self, name: str, description: str, tags: List[str])
    -> None:
        self.datasets[name] = {
            "description": description,
            "tags": tags
        }

    def search(self, query: str, k: int) -> List[str]:
        query_lower = query.lower()
        scores = []

        for name, metadata in self.datasets.items():
            score = 0
            name_lower = name.lower()

            # Exact name match - highest score
            if name_lower == query_lower:
                score += 100

            # Name starts with query - high score
            elif name_lower.startswith(query_lower):
                score += 50

            # Query in name - medium score
```

```
            elif query_lower in name_lower:
                score += 25

            # Check tags
            for tag in metadata["tags"]:
                tag_lower = tag.lower()
                if tag_lower == query_lower:
                    score += 30
                elif query_lower in tag_lower:
                    score += 10

            # Check description
            desc_lower = metadata["description"].lower()
            if query_lower in desc_lower:
                score += 5

            if score > 0:
                scores.append((score, name))

        # Sort by score descending, then alphabetically
        scores.sort(key=lambda x: (-x[0], x[1]))

        # Return top k results
        return [name for score, name in scores[:k]]
```

3.3 Complexity Analysis

- **Time Complexity**: O(N × M) where N is datasets, M is avg metadata length; O(N log N) for sorting
- **Space Complexity**: O(N × M) for storing all metadata

3.4 Key Insights

1. Simple relevance scoring works for small datasets
2. Exact matches should rank higher than partial matches
3. Production systems use Elasticsearch or similar
4. Consider TF-IDF or BM25 for better ranking

3.5 Production Enhancements

- **Elasticsearch**: Full-text search with analyzers and tokenizers
- **Fuzzy Matching**: Handle typos with edit distance
- **Popularity Boost**: Factor in usage statistics
- **Personalization**: Rank based on user's team or domain
- **Faceted Search**: Filter by type, owner, freshness

4 Problem 3: Schema Evolution Tracking

4.1 Problem Statement

DataHub tracks schema changes over time to ensure compatibility. Implement a system to detect changes and check compatibility between schema versions.

Requirements:

- Store schema versions with metadata
- Detect schema changes (added/removed/modified fields)
- Check backward compatibility (new schema reads old data)
- Check forward compatibility (old schema reads new data)

Example:

```
tracker = SchemaEvolution()

schema_v1 = {"user_id": "int", "name": "string", "email": "string"}
schema_v2 = {"user_id": "int", "name": "string", "email": "string",
             "created_at": "timestamp"}

tracker.add_schema("users", schema_v1, version=1)
tracker.add_schema("users", schema_v2, version=2)

changes = tracker.get_changes("users", version_from=1, version_to=2)
# Returns: {"added": ["created_at"], "removed": [], "modified": []}

tracker.is_backward_compatible("users", 2, 1)  # Returns True
```

4.2 Solution

```
from typing import Dict, List

class SchemaEvolution:
    def __init__(self):
        self.schemas = {}  # (table_name, version) -> schema dict

    def add_schema(self, table_name: str, schema: Dict[str, str],
    version: int) -> None:
        self.schemas[(table_name, version)] = schema.copy()

    def get_changes(self, table_name: str, version_from: int,
    version_to: int) -> Dict[str, List[str]]:
        schema_old = self.schemas.get((table_name, version_from), {})
        schema_new = self.schemas.get((table_name, version_to), {})

        added = []
        removed = []
        modified = []

        # Find added and modified fields
        for field, new_type in schema_new.items():
            if field not in schema_old:
                added.append(field)
            elif schema_old[field] != new_type:
```

```
                modified.append(field)

        # Find removed fields
        for field in schema_old:
            if field not in schema_new:
                removed.append(field)

        return {
            "added": added,
            "removed": removed,
            "modified": modified
        }

    def is_backward_compatible(self, table_name: str, new_version: int, old_version: int) -> bool:
        changes = self.get_changes(table_name, old_version, new_version)

        # Backward compatible if:
        # 1. No fields removed (new schema has all old fields)
        # 2. No field types changed
        if changes["removed"] or changes["modified"]:
            return False

        return True
```

4.3 Complexity Analysis

- **Time Complexity**: O(n) where n is number of fields in schemas
- **Space Complexity**: O(v × n) where v is versions, n is fields per version

4.4 Key Insights

1. Backward compatibility: new code reads old data (adding optional fields OK)
2. Forward compatibility: old code reads new data (more restrictive)
3. Schema evolution is critical for zero-downtime deployments
4. Track not just schema but also statistics and sample values

4.5 Schema Registry Integration

In production DataHub:

- **Avro/Protobuf**: Integration with schema registries
- **Compatibility Modes**: BACKWARD, FORWARD, FULL, NONE
- **Version History**: Complete audit trail of all changes
- **Impact Analysis**: Show which pipelines affected by schema change
- **Alerts**: Notify owners of breaking changes

5 Problem 4: Data Quality Monitoring

5.1 Problem Statement

DataHub monitors data quality metrics to detect anomalies. Implement a system that tracks metrics over time and detects significant deviations using statistical methods.

Requirements:

- Track metrics over time (row count, null percentage, distinct values)
- Calculate rolling statistics (mean, standard deviation)
- Detect anomalies using Z-score (values beyond threshold)
- Return alerts for anomalous metrics

Example:

```
monitor = DataQualityMonitor(window_size=5, threshold=2.0)

# Normal metrics
monitor.add_metric("table1", "row_count", 1000)
monitor.add_metric("table1", "row_count", 1050)
monitor.add_metric("table1", "row_count", 980)
monitor.add_metric("table1", "row_count", 1020)
monitor.add_metric("table1", "row_count", 1010)

# Anomalous metric
result = monitor.add_metric("table1", "row_count", 5000)
# Returns: {"is_anomaly": True, "z_score": 3.5, ...}
```

5.2 Solution

```
from typing import Dict
from collections import deque
import statistics

class DataQualityMonitor:
    def __init__(self, window_size: int, threshold: float):
        self.window_size = window_size
        self.threshold = threshold
        self.metrics = {}  # (table, metric_name) -> deque of values

    def add_metric(self, table_name: str, metric_name: str, value:
    float) -> Dict:
        key = (table_name, metric_name)

        if key not in self.metrics:
            self.metrics[key] = deque(maxlen=self.window_size)

        history = self.metrics[key]

        # Calculate stats before adding new value
        is_anomaly = False
        z_score = 0.0
        mean = 0.0
        std_dev = 0.0

```

```
        if len(history) >= 2:
            mean = statistics.mean(history)
            std_dev = statistics.stdev(history)

            if std_dev > 0:
                z_score = abs((value - mean) / std_dev)
                is_anomaly = z_score > self.threshold

        # Add current value to history
        history.append(value)

        return {
            "is_anomaly": is_anomaly,
            "z_score": z_score,
            "mean": mean,
            "std_dev": std_dev
        }
```

5.3 Complexity Analysis

- **Time Complexity**: O(w) where w is window size (for statistics calculation)
- **Space Complexity**: O(t × m × w) where t is tables, m is metrics, w is window size

5.4 Key Insights

1. Z-score detects outliers based on historical pattern
2. Rolling window adapts to changing baselines
3. Threshold of 2.0-3.0 balances sensitivity vs false positives
4. Production systems use more sophisticated methods (ML-based)

5.5 Production Enhancements

- **Multiple Algorithms**: Combine Z-score, IQR, ARIMA, ML models
- **Seasonality**: Account for daily/weekly patterns
- **Multi-metric Correlation**: Detect related anomalies
- **Alert Routing**: Notify data owners via Slack/email
- **Historical Analysis**: Store all anomalies for pattern analysis

6 IMPORTANT: Java Concurrency Problems

6.1 Why Java for DataHub?

DataHub explicitly tests multithreading and concurrency in their first interview round.

Key reasons to use Java (not Python) for concurrency:

- **DataHub's backend is Java** (Spring Boot)
- Java has robust concurrency primitives
- Python's GIL limits true parallelism
- Interview will expect Java concurrency knowledge

Problems 5-7 are in Java and focus specifically on concurrency patterns used in DataHub's production systems.

7 Problem 5: Thread-Safe Metadata Cache

7.1 Problem Statement

DataHub's metadata service needs a thread-safe cache for frequently accessed metadata. Implement a thread-safe LRU cache that handles concurrent reads and writes.

Requirements:

- Thread-safe get() and put() operations
- LRU eviction policy when capacity is reached
- Support concurrent reads (multiple threads simultaneously)
- Writes must be exclusive (only one writer at a time)
- No data corruption under concurrent access

Example:

```
MetadataCache cache = new MetadataCache(2);

// Thread 1
cache.put("dataset1", "metadata1");

// Thread 2 (concurrent)
cache.put("dataset2", "metadata2");

// Thread 3 (concurrent read)
String m = cache.get("dataset1");
```

7.2 Solution

```
import java.util.*;
import java.util.concurrent.locks.*;

class MetadataCache {
    private final int capacity;
    private final LinkedHashMap<String, String> cache;
    private final ReadWriteLock lock;
    private final Lock readLock;
    private final Lock writeLock;

    public MetadataCache(int capacity) {
        this.capacity = capacity;
        this.cache = new LinkedHashMap<String, String>(capacity, 0.75f,
    true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                return size() > MetadataCache.this.capacity;
            }
        };
        this.lock = new ReentrantReadWriteLock();
        this.readLock = lock.readLock();
        this.writeLock = lock.writeLock();
    }

    public String get(String key) {
```

```
        readLock.lock();
        try {
            return cache.get(key);
        } finally {
            readLock.unlock();
        }
    }

    public void put(String key, String value) {
        writeLock.lock();
        try {
            cache.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }

    public int size() {
        readLock.lock();
        try {
            return cache.size();
        } finally {
            readLock.unlock();
        }
    }
}
```

7.3 Complexity Analysis

- **Time Complexity**: O(1) for both get and put operations
- **Space Complexity**: O(capacity)

7.4 Key Insights

1. **ReadWriteLock**: Allows multiple concurrent readers, but exclusive writer
2. **LinkedHashMap**: Maintains insertion/access order for LRU
3. **accessOrder=true**: Moves accessed entries to end (LRU behavior)
4. **removeEldestEntry**: Automatic eviction when capacity exceeded
5. **Always unlock in finally**: Ensures lock is released even on exception

7.5 Follow-up Questions

1. Why use ReadWriteLock instead of synchronized?
2. What if we needed distributed caching across multiple servers?
3. How would you handle cache invalidation?
4. What are the trade-offs between LRU, LFU, and FIFO?

8 Problem 6: Producer-Consumer Metadata Ingestion

8.1 Problem Statement

DataHub ingests metadata from multiple sources concurrently. Implement a producer-consumer pattern where producers add metadata events to a queue and consumers process them.

Requirements:

- Thread-safe queue for metadata events
- Multiple producers add events concurrently
- Multiple consumers process events concurrently
- Graceful shutdown (process all pending events)
- No events lost or processed twice

Example:

```
MetadataIngestionPipeline pipeline = new MetadataIngestionPipeline(10);

// Start 3 consumer threads
pipeline.startConsumers(3);

// Producers add events
pipeline.addEvent(new MetadataEvent("dataset1", "SCHEMA_CHANGE"));
pipeline.addEvent(new MetadataEvent("dataset2", "LINEAGE_UPDATE"));

// Shutdown and wait for completion
pipeline.shutdown();
```

8.2 Solution

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

class MetadataIngestionPipeline {
    private final BlockingQueue<MetadataEvent> queue;
    private ExecutorService executorService;
    private final AtomicInteger processedCount;
    private volatile boolean isShutdown;
    private static final MetadataEvent POISON_PILL =
        new MetadataEvent("POISON", "PILL");

    public MetadataIngestionPipeline(int queueSize) {
        this.queue = new ArrayBlockingQueue<>(queueSize);
        this.processedCount = new AtomicInteger(0);
        this.isShutdown = false;
    }

    public void startConsumers(int numConsumers) {
        executorService = Executors.newFixedThreadPool(numConsumers);

        for (int i = 0; i < numConsumers; i++) {
            executorService.submit(() -> {
                try {
                    while (true) {
```

```
                        MetadataEvent event = queue.take();
                        if (event == POISON_PILL) {
                            break;
                        }
                        // Process event
                        Thread.sleep(10); // Simulate processing
                        processedCount.incrementAndGet();
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
    }

    public boolean addEvent(MetadataEvent event) throws
   InterruptedException {
        if (isShutdown) {
            return false;
        }
        queue.put(event);  // Blocks if queue is full
        return true;
    }

    public void shutdown() throws InterruptedException {
        isShutdown = true;

        // Add poison pills for each consumer
        int consumers = ((ThreadPoolExecutor) executorService).
   getActiveCount();
        for (int i = 0; i < consumers; i++) {
            queue.put(POISON_PILL);
        }

        executorService.shutdown();
        executorService.awaitTermination(10, TimeUnit.SECONDS);
    }

    public int getProcessedCount() {
        return processedCount.get();
    }
}
```

8.3 Complexity Analysis

- **Time Complexity**: O(1) for add/poll operations
- **Space Complexity**: O(queue_size)

8.4 Key Insights

1. **BlockingQueue**: Thread-safe, handles blocking automatically
2. **Poison Pill**: Sentinel value to signal consumer shutdown
3. **AtomicInteger**: Lock-free counter for thread safety
4. **ExecutorService**: Manages thread lifecycle

5. **Graceful Shutdown**: Process pending events before stopping

8.5 Production Considerations

- **Backpressure**: ArrayBlockingQueue naturally provides backpressure
- **Error Handling**: Wrap event processing in try-catch
- **Monitoring**: Track queue depth, processing rate, error rate
- **Distributed**: Use Kafka for distributed producer-consumer

9 Problem 7: Concurrent Batch Processor

9.1 Problem Statement

DataHub needs to process large batches of metadata updates efficiently. Implement a concurrent batch processor using thread pools.

Requirements:

- Use ExecutorService with thread pool
- Process batches concurrently
- Collect results from all threads safely
- Handle exceptions without crashing
- Proper shutdown and cleanup

Example:

```
BatchProcessor processor = new BatchProcessor(4);

List<String> datasets = Arrays.asList("ds1", "ds2", ..., "ds100");
List<String> results = processor.processBatch(datasets);

processor.shutdown();
```

9.2 Solution

```
import java.util.*;
import java.util.concurrent.*;

class BatchProcessor {
    private final ExecutorService executorService;

    public BatchProcessor(int threadPoolSize) {
        this.executorService = Executors.newFixedThreadPool(
   threadPoolSize);
    }

    public List<String> processBatch(List<String> datasets)
            throws InterruptedException, ExecutionException {
        List<Future<String>> futures = new ArrayList<>();

        // Submit all tasks
        for (String dataset : datasets) {
            Future<String> future = executorService.submit(() -> {
                return processDataset(dataset);
            });
            futures.add(future);
        }

        // Collect results
        List<String> results = new ArrayList<>();
        for (Future<String> future : futures) {
            results.add(future.get());  // Blocking
        }

```

```
        return results;
    }

    public ProcessResult processBatchWithFailures(List<String> datasets
    )
            throws InterruptedException {
        List<Future<String>> futures = new ArrayList<>();

        for (String dataset : datasets) {
            futures.add(executorService.submit(() -> processDataset(
    dataset)));
        }

        List<String> successes = new ArrayList<>();
        int failures = 0;

        for (Future<String> future : futures) {
            try {
                successes.add(future.get());
            } catch (ExecutionException e) {
                failures++;
            }
        }

        return new ProcessResult(successes, failures);
    }

    public void shutdown() throws InterruptedException {
        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);
    }

    private String processDataset(String dataset) throws Exception {
        Thread.sleep(10);  // Simulate work
        if (dataset.contains("fail")) {
            throw new Exception("Failed: " + dataset);
        }
        return "processed_" + dataset;
    }
}
```

9.3 Complexity Analysis

- **Time Complexity**: O(n/p) where n is items, p is threads (parallelization)
- **Space Complexity**: O(n) for results

9.4 Key Insights

1. **Future**: Represents async computation result
2. **ExecutorService**: Reuses threads, avoids overhead
3. **Thread Pool Sizing**: CPU-bound = cores + 1; I/O-bound = cores × 2
4. **Exception Handling**: ExecutionException wraps task exceptions
5. **Shutdown**: Always call shutdown() and awaitTermination()

9.5 Optimizations

- **CompletionService**: Process results as they complete (not in order)
- **ForkJoinPool**: For recursive divide-and-conquer tasks
- **Timeout**: Use future.get(timeout, unit) to avoid hanging
- **Batch Size**: Tune for optimal throughput vs latency

10 Java Concurrency Quick Reference

10.1 Essential Concurrency Primitives

- **synchronized**: Basic mutual exclusion (method or block level)
- **ReentrantLock**: Explicit lock with tryLock, fairness options
- **ReadWriteLock**: Many readers OR one writer (cache scenarios)
- **Semaphore**: Limit concurrent access to N threads
- **CountDownLatch**: Wait for N events (one-time barrier)
- **CyclicBarrier**: Synchronize N threads repeatedly
- **Atomic Classes**: Lock-free thread-safe variables

10.2 Thread-Safe Collections

- **ConcurrentHashMap**: Thread-safe map with fine-grained locking
- **CopyOnWriteArrayList**: Thread-safe list (read-heavy workloads)
- **BlockingQueue**: Producer-consumer pattern
- **ConcurrentLinkedQueue**: Non-blocking concurrent queue

10.3 Common Pitfalls

1. **Forgetting unlock**: Always use try-finally with explicit locks
2. **Deadlock**: Acquire locks in consistent order
3. **Race Conditions**: Use atomic operations or synchronization
4. **Ignoring InterruptedException**: Restore interrupt flag
5. **Shared Mutable State**: Minimize or use immutable objects

11 System Design Topics

11.1 Metadata Catalog Architecture

Key components of a metadata platform:

1. **Metadata Store**: Primary database for all metadata
2. **Search Index**: Elasticsearch for fast discovery
3. **Graph Database**: Neo4j for lineage queries
4. **Ingestion**: Kafka for real-time metadata events
5. **API Layer**: RESTful and GraphQL APIs
6. **UI**: React frontend for data discovery

11.2 Scalability Considerations

Critical for enterprise deployments:

- **Sharding**: Partition metadata by domain or data source
- **Caching**: Redis for frequently accessed metadata
- **Read Replicas**: Scale read queries independently
- **Async Processing**: Background jobs for lineage computation
- **Rate Limiting**: Protect from ingestion spikes

11.3 Data Lineage at Scale

Challenges and solutions:

- **Graph Size**: Billions of nodes, trillions of edges
- **Query Performance**: Sub-second lineage traversal
- **Real-time Updates**: Incremental lineage computation
- **Multi-hop Queries**: Efficient path finding algorithms
- **Visualization**: Render large graphs without overwhelming UI

12 DataHub-Specific Knowledge

12.1 Core Entities

DataHub models these entities:

- **Datasets**: Tables, files, topics
- **Charts**: BI visualizations
- **Dashboards**: Collections of charts
- **Data Jobs**: Pipelines, workflows
- **ML Models**: Machine learning models
- **Containers**: Databases, schemas

12.2 Metadata Aspects

Each entity has multiple aspects:

- **Schema**: Column names, types, descriptions
- **Ownership**: Individuals and teams
- **Tags**: Free-form labels
- **Glossary Terms**: Business vocabulary
- **Lineage**: Upstream and downstream dependencies
- **Usage Stats**: Query frequency, users
- **Properties**: Custom key-value pairs

12.3 Ingestion Framework

DataHub supports 100+ data sources:

- **Databases**: MySQL, PostgreSQL, Oracle, SQL Server
- **Warehouses**: Snowflake, BigQuery, Redshift
- **BI Tools**: Looker, Tableau, PowerBI
- **Orchestration**: Airflow, Dagster
- **Streaming**: Kafka, Pulsar

13 Behavioral Interview Prep

13.1 DataHub/Acryl Data Values

- **Data Democratization**: Make data accessible to all
- **Open Source First**: Community-driven development
- **Engineering Excellence**: Scalable, reliable systems
- **Customer Success**: Solve real data discovery problems

13.2 Common Questions

1. Why are you interested in data infrastructure?
2. Tell me about a time you built a scalable distributed system
3. How do you approach system design with conflicting requirements?
4. Describe your experience with graph algorithms
5. What's your approach to contributing to open-source projects?

13.3 Questions to Ask

- What are the biggest technical challenges in scaling DataHub?
- How do you balance open-source community vs commercial features?
- What's the roadmap for column-level lineage?
- How does the team handle on-call and production incidents?
- What opportunities exist for technical leadership growth?

14 Additional Resources

14.1 Technical Learning

- **DataHub GitHub**: Study the codebase and architecture docs
- **Algorithms**: Focus on graph algorithms (BFS, DFS, topological sort)
- **System Design**: "Designing Data-Intensive Applications"
- **Data Engineering**: "Fundamentals of Data Engineering"
- **Similar Projects**: Apache Atlas, Amundsen, LinkedIn WhereHows

14.2 Practice Platforms

- **LeetCode**: Graph problems (medium/hard), trees, search
- **System Design**: Grokking courses, YouTube channels
- **Open Source**: Contribute to DataHub to understand deeply

14.3 Data Infrastructure Blogs

- DataHub Blog (blog.datahubproject.io)
- Acryl Data Engineering Blog
- LinkedIn Engineering Blog
- Netflix Tech Blog (data infrastructure posts)

15 Final Tips

1. **Study DataHub**: Clone the repo, run locally, understand architecture
2. **Graph Mastery**: DataHub is heavily graph-based - know your graph algorithms!
3. **Scalability Mindset**: Think about millions of datasets from day one
4. **Open Source**: Show familiarity with open-source contribution process
5. **Data Passion**: Demonstrate genuine interest in solving data discovery problems
6. **Ask Questions**: Show curiosity about technical decisions and trade-offs

Good luck with your DataHub/Acryl Data interview!