Azure Data Engineering - Comprehensive Q&A Guide

Table of Contents

---

1. Integration Runtime

Q1: Integration Runtime - Types

Answer: Integration Runtime (IR) is the compute infrastructure used by Azure Data Factory to provide data integration capabilities across different network environments.

Types of Integration Runtime:

1. **Azure Integration Runtime (Azure IR)**
    - Default compute infrastructure in Azure
    - Supports copy activities between cloud data stores
    - Supports data flow activities
    - Provides dispatch activities like lookup, get metadata
    - Automatically managed by Microsoft
    - No setup required
2. **Self-hosted Integration Runtime (Self-hosted IR)**
    - Customer-managed compute infrastructure
    - Installed on on-premises machines or VMs
    - Enables hybrid data integration scenarios
    - Supports copying data between cloud and on-premises
    - Provides secure data movement
    - Requires manual installation and configuration
3. **Azure-SSIS Integration Runtime (Azure-SSIS IR)**
    - Managed cluster of Azure VMs

    - Dedicated to running SSIS packages
    - Lift-and-shift SSIS workloads to Azure
    - Supports both Azure and on-premises data sources
    - Fully managed service

Use Cases: - Azure IR: Cloud-to-cloud data movement - Self-hosted IR: Hybrid scenarios, on-premises connectivity - Azure-SSIS IR: SSIS package execution in Azure

---

2. Tables and Storage

Q2: Physical & External Table

Answer:

Physical Tables: - Data is physically stored in the database/storage system - Tables own the data and storage - Data is managed by the database engine - Schema and data are tightly coupled - Examples: Regular SQL Server tables, managed Delta tables

External Tables: - Metadata definition pointing to external data - Data is stored outside the database/warehouse - Table definition is separate from data storage - Schema is defined but data remains in original location - Examples: Synapse external tables, Databricks external tables

Key Differences:

Aspect	Physical Table	External Table
Data Location	Inside database	External storage
Data Ownership	Database manages	External system
Performance	Optimized	Depends on external source
Data Movement	Required	Not required
Storage Cost	Database storage	External storage
Schema Evolution	Database controlled	External source dependent

Examples:

```sql
-- Physical Table (Synapse) CREATE TABLE PhysicalTable ( ID INT, Name VARCHAR(100) );

-- External Table (Synapse) CREATE EXTERNAL TABLE ExternalTable ( ID INT, Name VARCHAR(100) ) WITH ( LOCATION = '/data/table/', DATASOURCE = ExternalDataSource, FILEFORMAT = ParquetFormat );
```

---

3. Azure Data Lake Storage

Q3: ADLS (Azure Data Lake Storage)

Answer:

Azure Data Lake Storage (ADLS) is a scalable data storage service optimized for big data analytics workloads.

Key Features:

1. **Hierarchical Namespace**

    - Directory and file-based organization
    - POSIX-compliant file system semantics
    - Efficient operations on directories

2. **Security**

    - Role-based access control (RBAC)
    - Access Control Lists (ACLs)
    - Integration with Azure Active Directory
    - Encryption at rest and in transit

3. **Performance**

    - Optimized for analytics workloads
    - High throughput and low latency
    - Parallel processing capabilities

4. **Integration**

    - Native integration with Azure services
    - Hadoop-compatible (HDFS)
    - Support for various analytics frameworks

ADLS Gen2 Architecture: `Storage Account ├── Container (File System) │ ├── Directory 1 │ │ ├── file1.parquet │ │ └── file2.json │ └── Directory 2 │ ├── Subdirectory │ └── file3.csv`

Access Methods: - REST APIs - Azure Storage SDKs - HDFS drivers - Azure Storage Explorer - Various analytics tools (Databricks, Synapse, etc.)

---

4. PySpark Optimization

Q4: Optimization Techniques - PySpark

Answer:

1. Data Serialization ```python

Use efficient serialization

spark.conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer") ```

2. Partitioning Strategies ```python

Partition by frequently filtered columns

df.write.partitionBy("year", "month").parquet("path")

Optimal partition size (128MB - 1GB)

df.coalesce(10).write.parquet("path") ```

3. Caching and Persistence ```python

Cache frequently accessed DataFrames

df.cache() df.persist(StorageLevel.MEMORYANDDISK)

Unpersist when done

df.unpersist() ```

4. Broadcast Variables ```python

Broadcast small lookup tables

broadcastdict = spark.sparkContext.broadcast(lookupdict) ```

5. Column Pruning and Predicate Pushdown ```python

Select only required columns

df.select("col1", "col2").filter(df.col1 > 100) ```

6. Data Format Optimization ```python

Use columnar formats

df.write.format("delta").save("path") # Delta Lake df.write.format("parquet").save("path") # Parquet ```

7. Join Optimization ```python

Use broadcast joins for small tables

from pyspark.sql.functions import broadcast result = largedf.join(broadcast(smalldf), "key") ```

8. Resource Configuration ```python

Optimize Spark configuration

spark.conf.set("spark.sql.adaptive.enabled", "true") spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true") ```

9. Bucketing ```python

Pre-partition data for joins

df.write.bucketBy(10, "joinkey").saveAsTable("bucketedtable") ```

---

5. Cluster Management

Q5: Types of Modes of Cluster

Answer:

Databricks Cluster Modes:

1. Standard Cluster - General-purpose clusters - Support multiple languages (Python, Scala, R, SQL) - Can be shared by multiple users - Suitable for interactive analytics - Supports auto-scaling

2. High Concurrency Cluster - Optimized for concurrent users - Enhanced security and isolation - Table access control - Process isolation for different users - Supports SQL, Python, R (limited Scala)

3. Single Node Cluster - No worker nodes, only driver - Cost-effective for small workloads - Good for development and testing - Limited by single machine resources

Cluster Access Modes:

1. No Isolation Shared - Multiple users can share cluster - No process isolation - Users can see each other's data

2. Shared - Process isolation between users - Enhanced security - Table access control - Multiple users can access safely

3. Single User - Assigned to specific user - Maximum performance for single user - No sharing restrictions

Azure Synapse Spark Pool Modes:

1. Auto-scale - Dynamically adjusts node count - Based on workload requirements - Cost optimization

2. Fixed Size - Predefined number of nodes - Consistent performance - Predictable costs

---

6. Data Types and Variables

Q6: var & nvar difference

Answer:

In SQL Server context:

VARCHAR (Variable Character) - Stores non-Unicode character data - 1 byte per character - Maximum length: 8,000 characters - Uses default database collation - More storage efficient for English text

NVARCHAR (National Variable Character) - Stores Unicode character data - 2 bytes per character - Maximum length: 4,000 characters - Supports international characters - Required for multilingual applications

Comparison:

Aspect	VARCHAR	NVARCHAR
Character Set	Non-Unicode	Unicode
Storage	1 byte/char	2 bytes/char
Max Length	8,000 chars	4,000 chars
International Support	Limited	Full
Storage Efficiency	Higher	Lower
Use Case	English only	Multilingual

Examples:
```sql
-- VARCHAR example
DECLARE @name VARCHAR(50) = 'John Doe';

-- NVARCHAR example
DECLARE @international_name NVARCHAR(50) = N'José García';
```

In Programming contexts (like JavaScript): - `var`: Function-scoped, can be redeclared, hoisted - `let/const`: Block-scoped, cannot be redeclared

---

7. Job Cluster

Q7: Job Cluster

Answer:

A Job Cluster is a cluster that is created specifically to run a particular job and is terminated when that job completes.

Key Characteristics:

1. Lifecycle Management - Created when job starts - Terminated when job completes - Ephemeral and dedicated

2. Cost Optimization - Pay only for job execution time - No idle cluster costs - Automatic resource cleanup

3. Isolation - Dedicated resources for specific job - No resource contention - Enhanced security

4. Configuration - Job-specific cluster sizing - Optimized for particular workload - Custom libraries and configurations

Job Cluster vs Interactive Cluster:

Aspect	Job Cluster	Interactive Cluster
Lifecycle	Job duration	Long-running
Cost	Job execution only	Continuous
Sharing	Single job	Multiple users
Use Case	Production jobs	Development/Analysis
Startup Time	Cluster creation overhead	Immediate

Example Configuration (Databricks):
```json
{ "new_cluster": { "spark_version": "11.3.x-scala2.12", "node_type_id": "Standard_DS3_v2", "num_workers": 2, "cluster_log_conf": { "dbfs": { "destination": "dbfs:/cluster-logs" } } }, "notebook_task": { "notebook_path": "/Users/user@example.com/my-notebook" } }
```

Best Practices: - Use for production ETL jobs - Configure appropriate cluster size - Enable logging for troubleshooting - Use spot instances for cost optimization

---

8. Delta Lake

Q8: Delta Table -- delta_df.history()

Answer:

The `history()` method in Delta Lake provides access to the transaction log and version history of a Delta table.

Purpose: - Track all changes made to the table - Enable time travel queries - Audit data modifications - Debug data pipeline issues

Syntax: ```python

Get full history

historydf = deltatable.history()

Get limited history

historydf = deltatable.history(limit=10)

Display history

display(history_df) ```

History Information Includes:

| Column | Description | |--------|-------------| | version | Table version number | | timestamp | When operation occurred | | userId | User who performed operation | | userName | User name | | operation | Type of operation (WRITE, DELETE, etc.) | | operationParameters | Parameters used in operation | | job | Job information | | notebook | Notebook information | | clusterId | Cluster that performed operation | | readVersion | Version read during operation | | isolationLevel | Transaction isolation level |

Example Usage: ```python from delta.tables import DeltaTable

Create Delta table reference

delta_table = DeltaTable.forPath(spark, "/path/to/delta/table")

Get history

history = delta_table.history()

Show recent operations

history.select("version", "timestamp", "operation", "operationParameters").show()

Time travel using version

df_v1 = spark.read.format("delta").option("versionAsOf", 1).load("/path/to/delta/table")

Time travel using timestamp

df_yesterday = spark.read.format("delta").option("timestampAsOf", "2023-01-01").load("/path/to/delta/table") ```

Common Operations in History: - WRITE: Data insertion/overwrite - DELETE: Row deletion - UPDATE: Row updates - MERGE: Merge operations - OPTIMIZE: File compaction - VACUUM: File cleanup

---

This is the first section of the comprehensive guide. Would you like me to continue with the remaining questions (9-40) in the next sections?