ELASTICSEARCH - ELK

```
{
    "type": "Training",
    "duration": "2d",
    "objective": "Mastering ElasticSearch"
}
```

elastic

DOCDOKU
digital enterprise applications

LEARNING OBJECTIVES

- Target audience: Developers
- Goal: Harness the full power of Elasticsearch
- Skills gained:
    - Develop Elasticsearch architecture effectively
    - Formulate powerful and relevant searches
    - Understand how Elasticsearch works internally
    - Optimize Elasticsearch performance

INTRODUCTION

- About the trainer
- About DocDoku

ABOUT YOU

- Experience with Elasticsearch
- Your expectations for the training
- Your upcoming projects using ElasticSearch

LOGISTICS

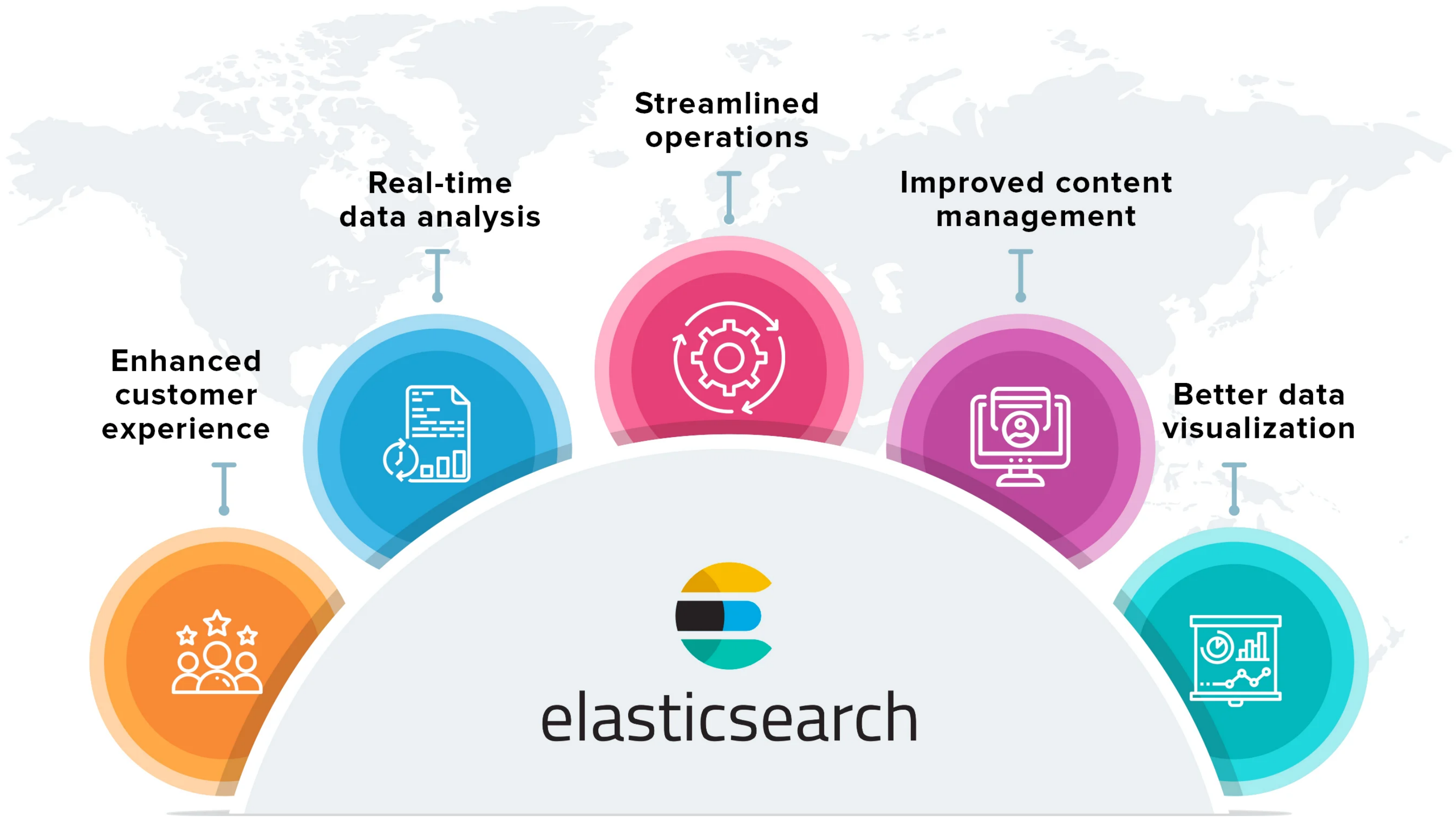
- Schedule: 9:00 AM → 5:00 PM
- Breaks: 15 minutes in the morning and afternoon
- Lunch: 1 hour 30 minutes

AGENDA

- [1] Introduction
- [2] Architecture of Elasticsearch
- [3] Configuration
- [4] Mappings
- [5] Search
- [6] Aggregations
- [7] Analyzers
- [8] Data Modelling and Relational Data
- [9] Indexing Strategies
- [10] Elasticsearch Best Practises
- [11] Backup / Disaster Recovery

[1] - INTRODUCTION

Why Elasticsearch ?



NEEDS

We are increasingly required to store more and more information, and we need to search through it in an **efficient** and **fast** way.

Relational databases may be suitable for some features, but they can be very slow when handling certain **advanced queries**.

ElasticSearch aims to meet this need while offering a range of **additional features**.

WHAT IS ELASTICSEARCH?

- A data search and indexing engine
- **Open source** (Apache 2 License)
- Written in **Java**
- Clustered, with data backup and replication
- Near real-time search (**NRT**) (low latency)
- Multiple features:
    - Indexing, Searching, Analytics, Relevance scoring

FEATURES

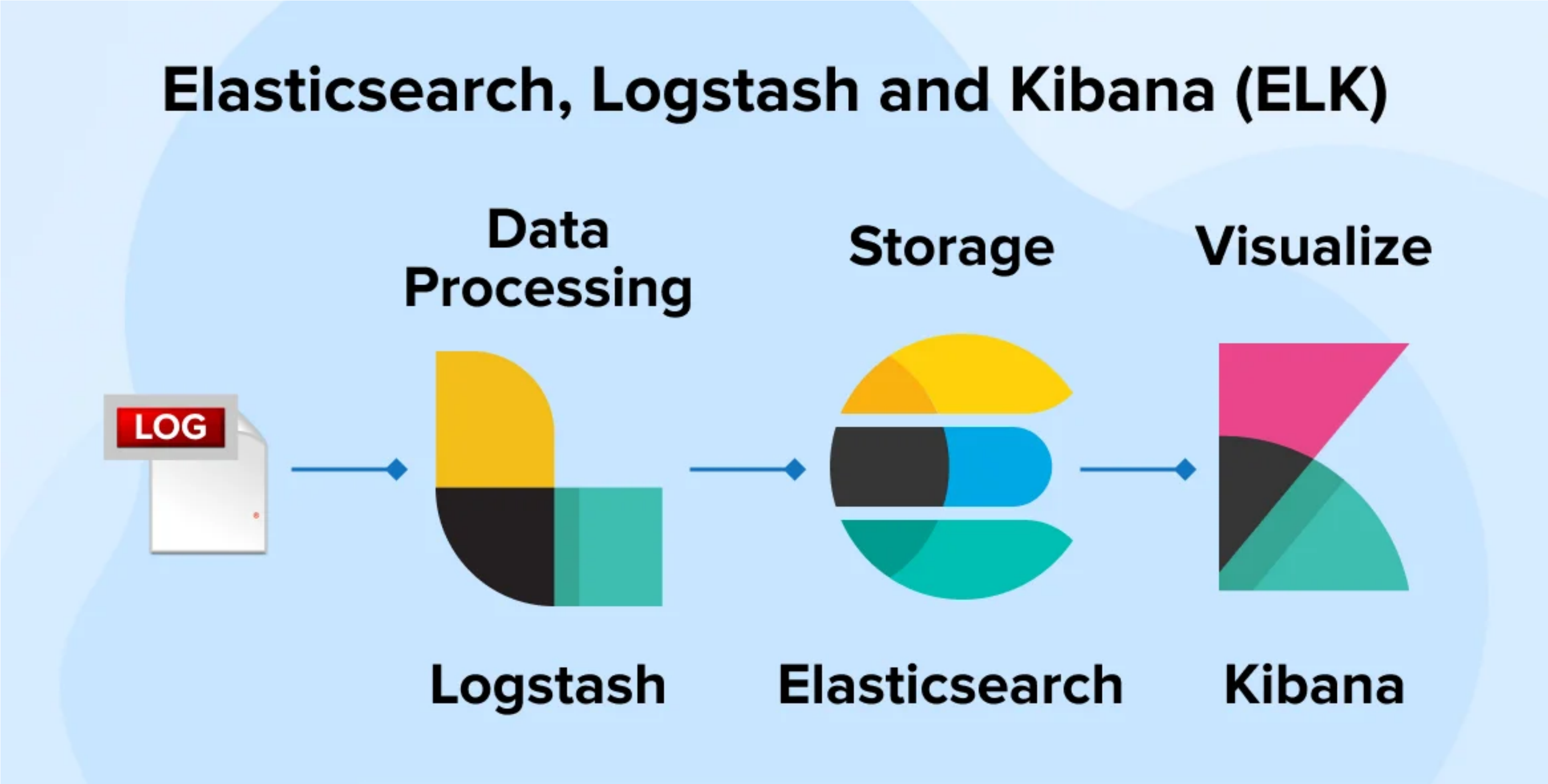
- Distributed system, **RESTful** architecture
- Based on Apache Lucene (Apache Software Foundation)
- APIs available for multiple programming languages
- **JSON** format for the REST API
- Schema-less data model
- Scalable:
    - Can run on a laptop, or on hundreds of servers

ECOSYSTEM

In addition to ElasticSearch, the company Elastic also provides:

- Kibana (visualization)
- Logstash/Beats (data ingestion)
- Agents (application integration)
- Monitoring tools
- SDKs (libraries for different languages)

ELK SUITE


Elasticsearch, Logstash and Kibana (ELK)
Data Processing
Storage
Visualize
LOG
Logstash
Elasticsearch
Kibana

HISTORY

- Created in 2004 by Shay Banon
- Version 0.4.0 released in 2010 (considered the first version)
- Version 1.0 released in 2012
- Version 2.0 in 2015
- Version 5.0 in 2016
- Version 6.0 in 2017
- Version 7.0 in 2019
- Version 8.0 in 2022
- Currently at version **9.1.5** (October 06, 2025)

[2] - ARCHITECTURE OF ELASTICSEARCH

- Shards and Replication
- Types of Node
- Cluster Topologies
- Hot Warm Cold Architecture

SHARDS AND REPLICATION

- **Shards**: Elasticsearch splits indices into smaller units called *shards*. Each shard is a fully functional and independent "sub-index" that can be hosted on any node.
- **Replication**: For fault tolerance, Elasticsearch allows you to configure **replica shards**. These are copies of primary shards and provide high availability and load balancing.

TYPES OF NODES

- **Master Node**: Manages cluster-wide operations like creating/deleting indices and tracking which nodes are part of the cluster.
- **Data Node**: Stores data and executes data-related operations such as CRUD, search, and aggregations.
- **Ingest Node**: Preprocesses documents before indexing (e.g., applying pipelines).
- **Machine Learning (ML) Node**: Handles ML jobs if using Elastic’s ML features.
- **Transform Node**: Performs data transformation operations.

CLUSTER TOPOLOGIES

- **Single-Node Cluster**: Simple setup for development or testing environments.
- **Multi-Node Cluster**: Production setup where different nodes take on specific roles for scalability and reliability.
- **Dedicated Node Roles**: Separate roles across different machines for performance optimization (e.g., dedicated master, data, ingest nodes).
- **High Availability**: Requires at least three master-eligible nodes to ensure quorum and avoid split-brain scenarios.

HOT-WARM-COLD ARCHITECTURE

- **Hot Nodes**: Handle heavy indexing and frequent searches. Use fast storage (e.g., SSD).
- **Warm Nodes**: Store less-frequently queried data; optimized for cost-efficiency over performance.
- **Cold Nodes**: Store read-only, infrequently accessed data.
- **Frozen**: Additional layer introduced by Elastic for even cheaper storage options (e.g., searchable snapshots).
- Used for **data lifecycle management (ILM)** to optimize cost vs. performance.

DATA LIFECYCLE FLOW 1

Hot nodes

- Receive all new writes (indexing).
- Data is frequently queried (e.g., dashboards, alerts).
- After a defined period, indices are **moved to warm nodes**.

DATA LIFECYCLE FLOW 2

Warm nodes

- Handle **read-only data** (no new writes).
- Still queried occasionally.
- Often use **shard allocation filtering** or **index lifecycle management (ILM)** to move data here automatically.

DATA LIFECYCLE FLOW 3

Cold nodes

- Store old data for **compliance or historical analysis**.
- Rarely queried.
- ILM transitions data here or even to **frozen/object storage (S3, Azure Blob, etc.)**.

[3] - CONFIGURATION

- Important Configurations
- Node discovery
- Cluster Settings

IMPORTANT CONFIGURATIONS 1 - 2

- **elasticsearch.yml**: Main configuration file located in the config directory.
- **cluster.name**: Defines the name of your cluster. All nodes in the same cluster must share this name.
- **node.name**: Unique identifier for each node in the cluster.
- **path.data / path.logs**: File system paths for storing index data and logs.

IMPORTANT CONFIGURATIONS 2 - 2

- **network.host**: IP address or hostname the node binds to (e.g., 0.0.0.0 for all interfaces).
- **http.port**: Port for REST API access (default is 9200).
- **transport.port**: Port for internal communication between nodes (default is 9300).
- **xpack.security.enabled**: Enables security features (e.g., authentication, encryption).

NODE DISCOVERY

- **Discovery Mechanism**: Elasticsearch uses **Zen Discovery** (or **Zen2**) for node discovery and master election.
- **discovery.seed_hosts**: List of known hosts for discovery during startup (used to bootstrap cluster).
- **cluster.initial_master_nodes**: Required for initial master election in new clusters (especially with multiple nodes).

CLUSTER SETTINGS 1 - 2

- Configurable via the **Cluster API** or **elasticsearch.yml** (depending on type).
- Two types:
- **Persistent Settings**: Saved across restarts.
- **Transient Settings**: Temporary and reset after restart.

CLUSTER SETTINGS 2 - 2

- `cluster.routing.allocation.enable`: Controls shard allocation behavior.
- `indices.recovery.max_bytes_per_sec`: Limits recovery bandwidth.
- `cluster.max_shards_per_node`: Limits number of shards a node can hold.
- Updated using the **Cluster Settings API**:

```
PUT _cluster/settings
{
    "persistent": {
        "cluster.routing.allocation.enable": "all"
    }
}
```

[4] - MAPPINGS

- Mapping
- Data Types
- Meta Data
- Dynamic mapping
- Resolving Mapping Conflict

MAPPING

- A **mapping** defines the structure of documents and how fields are stored and indexed in an index.
- It is similar to a **schema** in relational databases.
- Determines:
  - **Field types** (e.g., text, keyword, date)
  - **Analyzers**
  - **Indexing behavior**
  - **Field properties** (e.g., searchable, aggregatable)

DATA TYPES 1 - 2

Elasticsearch supports various **data types**, grouped into:

- **Core** Data Types:
  - `text`, `keyword`, `long`, `integer`, `short`, `double`, `float`, `boolean`, `date`
- **Complex** Data Types:
  - `object`, `nested`, `array`

DATA TYPES 2 - 2

- **Geo** and Specialized Types:
  - `geo_point`, `geo_shape`, `ip`, `completion`, `token_count`
- **Range** Types:
  - `integer_range`, `float_range`, `date_range`

META DATA

- Metadata fields provide information about the document and its indexing:
  - `_index`: Index name
  - `_id`: Document ID
  - `_source`: Original JSON document
  - `_type`: (deprecated, always `_doc` in modern versions)
  - `_routing`: Used for custom shard routing
- Meta fields can be used in queries and settings (e.g., `_source.enabled: false` to save space).

DYNAMIC MAPPING

- By default, Elasticsearch uses **dynamic mapping**, automatically detecting and adding new fields.
- Pros: Flexible and fast development.
- Cons: Can lead to unexpected mapping definitions or **mapping explosion**.
- Can be controlled via:
  - `dynamic: true` (default)
  - `dynamic: false` (ignore new fields)
  - `dynamic: strict` (throws error on unknown fields)

RESOLVING MAPPING CONFLICTS

- Mapping conflicts occur when the same field has **inconsistent types** across indices.
- Common causes:
- Automatic field type detection
- Inconsistent data input
- Solutions:
- Use **explicit mappings** instead of dynamic ones.
- Align field types before indexing.
- Use **index templates** to enforce consistent mappings.
- Reindex data after correcting mappings.

[5] - SEARCH

- How to optimize search relevance
- Match and Term Queries,fuzzy search
- Compound Queries Bool, function score
- Nested Queries,
- Geo Queries
- Pagination and Sorting

OPTIMIZE SEARCH RELEVANCE

- Elasticsearch uses a **scoring system** (default: BM25) to rank results.
- Key techniques to improve relevance:
- Use the **right analyzer** (e.g., standard, custom, language-specific)
- Boost important fields using ^ (e.g., `title^2`)

OPTIMIZE SEARCH RELEVANCE

- Combine multiple queries using **function score** or **bool**
- Leverage **synonyms** for broader matching
- Tune **BM25 parameters** (e.g., k1, b) if needed
- Use **user behavior data** (clicks, conversions) for scoring adjustments

MATCH AND TERM QUERIES, FUZZY SEARCH

- **Match Query**:
- Full-text search
- Uses analyzer and scoring
- Example:

```
{ "match": { "title": "quick brown fox" } }
```

MATCH AND TERM QUERIES, FUZZY SEARCH

- **Term Query**:
- Exact match, not analyzed
- Used for keyword fields or exact values (e.g., IDs, statuses)

```
{ "term": { "status": "active" } }
```

MATCH AND TERM QUERIES, FUZZY SEARCH

- **Fuzzy Query**:
- Handles typos and misspellings (based on Levenshtein distance)

```
{ "fuzzy": { "name": { "value": "robrt", "fuzziness": "AUTO" } } }
```

COMPOUND QUERIES: BOOL, FUNCTION SCORE

- **Bool Query**: Combines multiple queries using:
- `must` (AND)
- `should` (OR)
- `must_not` (NOT)
- `filter` (no scoring)

```
{
    "bool": {
        "must": [ { "match": { "title": "search" } } ],
        "filter": [ { "term": { "status": "active" } } ]
    }
}
```

FUNCTION SCORE QUERY:

- Alters document score based on custom logic
- Example: boost by recency, popularity, or a numeric field

```
{
  "function_score": {
      "query": { "match": { "title": "search" } },
      "functions": [
          {
              "field_value_factor": {
                  "field": "popularity",
                  "factor": 1.2,
                  "modifier": "sqrt"
              }
          }
      ],
      "boost_mode": "multiply"
  }
}
```

NESTED QUERIES

- Required when querying fields inside **nested objects**
- Maintains correct parent-child relationship during matching
- Example:

```
{
  "nested": {
      "path": "comments",
      "query": {
          "bool": {
              "must": [
                  { "match": { "comments.author": "John" } },
                  { "match": { "comments.text": "great" } }
              ]
          }
      }
  }
}
```

GEO QUERIES

- Used to search based on geographic data (`geo_point`, `geo_shape`)
- Common query types:
- `geo_distance` (e.g., find points within 5km)
- `geo_bounding_box`
- `geo_polygon`
- Example:

```
{
  "geo_distance": {
        "distance": "10km",
        "location": { "lat": 40.715, "lon": -74.011 }
  }
}
```

PAGINATION AND SORTING 1 - 2

- **Pagination**:
- Use `from` and `size` parameters

```
{ "from": 0, "size": 10 }
```

- Alternatives for large data sets:
- **Search After** (better performance for deep paging)
- **Scroll API** (for exporting large result sets)

PAGINATION AND SORTING 1 - 2

- **Sorting**:
- Use `sort` field

```
{
  "sort": [
      { "date": { "order": "desc" } },
      { "_score": "desc" }
  ]
}
```

[6] - AGGREGATIONS

- Introduction
- Aggregation Types
- Metric Aggregations
- Bucket Aggregations
- Metric + Bucket Aggregations
- Range and Date Aggregations

INTRODUCTION

- Aggregations in Elasticsearch provide powerful **analytics capabilities** over indexed data.
- They are similar to **GROUP BY** in SQL and allow you to compute **metrics**, **statistics**, and **breakdowns** over your data.
- Common use cases:
  - Generating dashboards
  - Statistical reports
  - Faceted search (filterable UI categories)

AGGREGATION TYPES

- Two main categories:
  - **Metric Aggregations**: Return numeric values (e.g., avg, sum)
  - **Bucket Aggregations**: Group documents into buckets (e.g., terms, ranges)
- Aggregations can be **nested** to build complex analytical queries.

METRIC AGGREGATIONS 1 - 2

Used to calculate **numerical metrics** over a field:

- `avg`: Average value
- `sum`: Total sum
- `min` / `max`: Minimum and maximum values
- `value_count`: Number of values
- `stats`: Summary (count, min, max, avg, sum)
- `extended_stats`: Includes variance, std deviation

METRIC AGGREGATIONS 2 - 2

- Example:

```
{
    "aggs": {
        "average_price": {
            "avg": { "field": "price" }
        }
    }
}
```

BUCKET AGGREGATIONS 1 - 2

Group documents based on field values or conditions:

- `terms`: Group by unique values
- `range`: Group into numeric ranges
- `date_histogram`: Group by time intervals (day, month, etc.)
- `filters`: Group by multiple filter conditions
- `histogram`: Fixed-size numeric intervals

BUCKET AGGREGATIONS 2 - 2

- Example (terms):

```
{
    "aggs": {
        "top_categories": {
            "terms": { "field": "category.keyword" }
        }
    }
}
```

METRIC + BUCKET AGGREGATIONS

Nest metric aggregations inside buckets to compute **metrics per group**.

- Example: Average price per category

```
{
    "aggs": {
        "by_category": {
            "terms": { "field": "category.keyword" },
            "aggs": {
                "avg_price": {
                    "avg": { "field": "price" }
                }
            }
        }
    }
}
```

RANGE AND DATE AGGREGATIONS 1 - 2

Useful for **grouping by numeric ranges** or **time intervals**:

```
{
    "aggs": {
        "price_ranges": {
            "range": {
                "field": "price",
                "ranges": [
                    { "to": 100 },
                    { "from": 100, "to": 500 },
                    { "from": 500 }
                ]
            }
        }
    }
}
```

RANGE AND DATE AGGREGATIONS 2 - 2

- **Date Histogram**:

```
{
  "aggs": {
      "sales_over_time": {
          "date_histogram": {
              "field": "sale_date",
              "calendar_interval": "month"
          }
      }
  }
}
```

- Can combine with metrics (e.g., total sales per month)

[7] - ANALYZERS

- What are analyzers and why do we use them
- Custom Analyzers,Tokenizers and Filters
- Telephone numbers
- Autocomplete

INTRODUCTION

- **Analyzers** are responsible for processing text during indexing and search.
- They **break down** text into tokens (words) and **normalize** them (e.g., lowercase, remove punctuation).

Try it!

```
GET _analyze
{
  "text" : "Hello Elastic-Search WORLD!!!"
}
```

USE CASES

- Used to:
  - Enable **full-text search**
  - Improve **search relevance**
  - Handle **language-specific rules**
- Example: `"The Quick Brown Fox"` → `["quick", "brown", "fox"]` (after lowercasing and removing stop words)

CUSTOM ANALYZERS

- A **custom analyzer** is composed of:
  - A **tokenizer**: splits text into tokens (e.g., `standard`, `whitespace`, `edge_ngram`)
  - Zero or more **token filters**: modify tokens (e.g., lowercase, stemming)
  - Optionally a **char filter**: pre-processes characters (e.g., HTML stripping)

CUSTOM ANALYZERS

Definition of a custom analyzer:

```
{
  "analysis": {
    "analyzer": {
      "my_custom_analyzer": {
        "type": "custom",
        "tokenizer": "standard",
        "filter": ["lowercase", "asciifolding", "stop"]
      }
    }
  }
}
```

TOKENIZERS EXAMPLES

- `standard`: smart word tokenizer
- `whitespace`: splits on spaces
- `keyword`: returns the whole input as a single token
- `edge_ngram`: useful for autocomplete

TOKEN FILTERS EXAMPLES

- `lowercase`: converts to lowercase
- `stop`: removes common stop words (like "the", "and")
- `stemmer`: reduces words to root form (e.g., "running" → "run")

SPECIAL CASES

- **Telephone Numbers**
  - By default, phone numbers may be split or treated as text.
  - For exact matching or partial search:
    - Use `keyword` or `custom analyzers` without tokenization.
    - For partial matching (e.g., last 4 digits), use `edge_ngram`.

SPECIAL CASES

- Example mapping:

```
{
"mappings": {
  "properties": {
    "phone": {
      "type": "text",
      "analyzer": "edge_ngram_phone"
    }
  }
},
"settings": {
  "analysis": {
    "analyzer": {
      "edge_ngram_phone": {
        "tokenizer": "edge_ngram_tokenizer",
        "filter": ["lowercase"]
```

SPECIAL CASES

- **Autocomplete**
  - Autocomplete suggests results as the user types.
  - Implemented using:
    - **Edge NGram Analyzer**: Tokenizes prefixes
    - **Completion Suggester**: A separate data structure optimized for speed
  - **Edge NGram** approach:
    - Index “`search`” as: `["s", "se", "sea", "sear", "searc", "search"]`
    - Fast prefix matches

SPECIAL CASES

- Example mapping for autocomplete:

```
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "autocomplete",
        "search_analyzer": "standard"
      }
    }
  },
  "settings": {
    "analysis": {
      "analyzer": {
        "autocomplete": {
          "tokenizer": "edge_ngram_tokenizer",
```

[8] - DATA MODELLING AND RELATIONAL DATA

- How to structure your indices
- Relational vs non relational databases
- Denormalizing data
- Nested data

WHY MODELLING?

Proper data modelling in Elasticsearch is **crucial** for ensuring:

- Performance,
- Scalability,
- Maintainability.

HOW TO STRUCTURE YOUR INDICES

- **Design indices around use cases**, not around entities. Elasticsearch is optimized for search and read-heavy workloads, so model your indices for fast query performance.
- **Avoid creating an index per user or customer**, as too many indices can hurt cluster performance. Instead, use a shared index with a field to separate users or tenants.

HOW TO STRUCTURE YOUR INDICES

- **Use index templates** to standardize settings and mappings across similar indices.
- **Leverage time-based indices** (e.g., daily or monthly) for log or event data. Combine with index lifecycle management (ILM) for automated rollover and retention.
- **Plan for reindexing**—data models might evolve over time, and reindexing strategies are important for long-term flexibility.

RELATIONAL VS NON-RELATIONAL DATABASES

- **Relational Databases (RDBMS):**
    - Data is normalized and stored across multiple tables.
    - Relationships are enforced via foreign keys and joins.
    - Best for transactional systems and strict schema enforcement.

RELATIONAL VS NON-RELATIONAL DATABASES

- **Elasticsearch (Non-Relational / Document-Based):**
    - Data is stored as JSON documents in indices.
    - No native support for joins (limited support via `join` or application-side joins).
    - Optimized for full-text search and analytics rather than relational integrity.
    - Encourages denormalization for performance and query simplicity.

DENORMALIZING DATA

- **What is denormalization?**
  - The process of embedding related data into a single document to avoid joins.
- **Benefits:**
  - Faster queries (no need for joins).
  - Simpler data retrieval—one document can contain all necessary context.
  - Better suited for Elasticsearch's distributed architecture.

DENORMALIZING DATA

- **Considerations:**
    - Data duplication can increase storage needs.
    - Updates require careful handling to avoid inconsistency across documents.
    - Denormalize only the fields necessary for search or aggregation.

NESTED DATA

- **Nested objects** are used when storing arrays of objects within a document where each object has its own set of fields.
- Example use case: a `user` document with multiple `addresses`, each having its own `city`, `state`, and `zip`.

```
{
  "name": "John Doe",
  "addresses": [
    { "city": "New York", "state": "NY", "zip": "10001" },
    { "city": "Los Angeles", "state": "CA", "zip": "90001" }
  ]
}
```

NESTED DATA

- **Why use nested fields?**
  - Without nesting, Elasticsearch flattens the objects, potentially causing incorrect matches during queries.
  - Nested fields allow accurate matching of subdocuments with nested queries.

NESTED DATA

- **Performance impact:**
  - Nested fields are more resource-intensive than flat fields.
  - Use them only when strict parent-child relationships within a document are required.

SUMMARY

- Model your Elasticsearch data based on access patterns, not traditional database schemas.
- Avoid over-indexing or over-normalizing; instead, embrace denormalization where it improves performance.
- Understand when to use nested data and the trade-offs it brings.
- Elasticsearch is not a relational database—embrace its strengths in distributed search and analytics.

[9] - INDEXING STRATEGIES

- Index Settings
- Index Aliases
- Index Templates
- Index Lifecycle Management
- DataStreams

INDEX SETTINGS

- Index settings define how an index behaves at creation time.
- **Key settings include:**
  - `number_of_shards`: Controls how data is distributed across the cluster.
  - `number_of_replicas`: Defines the number of redundant copies of each shard.
  - `refresh_interval`: Determines how often the index is refreshed and made searchable.
  - `analysis`: Custom analyzers, tokenizers, and filters can be configured per index.

INDEX SETTINGS

```
PUT some-index
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1,
    "refresh_interval": "1s"
  }
}
```

- Settings can be **static** (require index to be closed for changes) or **dynamic** (modifiable on the fly).

INDEX ALIASES

- Aliases are logical names that can point to one or more indices.
- Benefits of using aliases:
  - Abstract index names from client applications.
  - Enable zero-downtime reindexing and index rollovers.
  - Support filtered or write-only aliases.

INDEX ALIASES

Alias API

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "logs-2025-10",
        "alias": "logs-latest"
      }
    }
  ]
}
```

- Aliases allow for smooth transitions between indices without breaking queries.

INDEX TEMPLATES

- Index templates define **settings, mappings, and aliases** for indices that match a naming pattern.
- Automatically applied when a new index is created.
- Useful for standardizing index configurations across time-based or dynamically created indices.

INDEX TEMPLATES

Index Templates API

```
PUT _index_template/logs_template
{
  "index_patterns": ["logs-*"],
  "template": {
    "settings": { ... },
    "mappings": { ... },
    "aliases": { ... }
  }
}
```

- Templates improve consistency, reduce manual errors, and enforce structure.

INDEX LIFECYCLE MANAGEMENT (ILM)

- **ILM** automates index operations over time, based on lifecycle phases:
  - **Hot** – Active write and search.
  - **Warm** – Less frequent access, optimized for storage.
  - **Cold** – Rarely accessed, stored on low-cost hardware.
  - **Delete** – Remove old data when it's no longer needed.

INDEX LIFECYCLE MANAGEMENT (ILM)

- Policies define rules for transitions between phases:

```
PUT _ilm/policy/log_retention
{
  "policy": {
    "phases": {
      "hot": { "actions": { "rollover": {
            "max_age": "7d", "max_size": "50gb" } } },
      "delete": { "min_age": "30d", "actions": { "delete": {} } }
    }
  }
}
```

- Reduces manual intervention and helps manage large volumes of time-series data.

DATA STREAMS

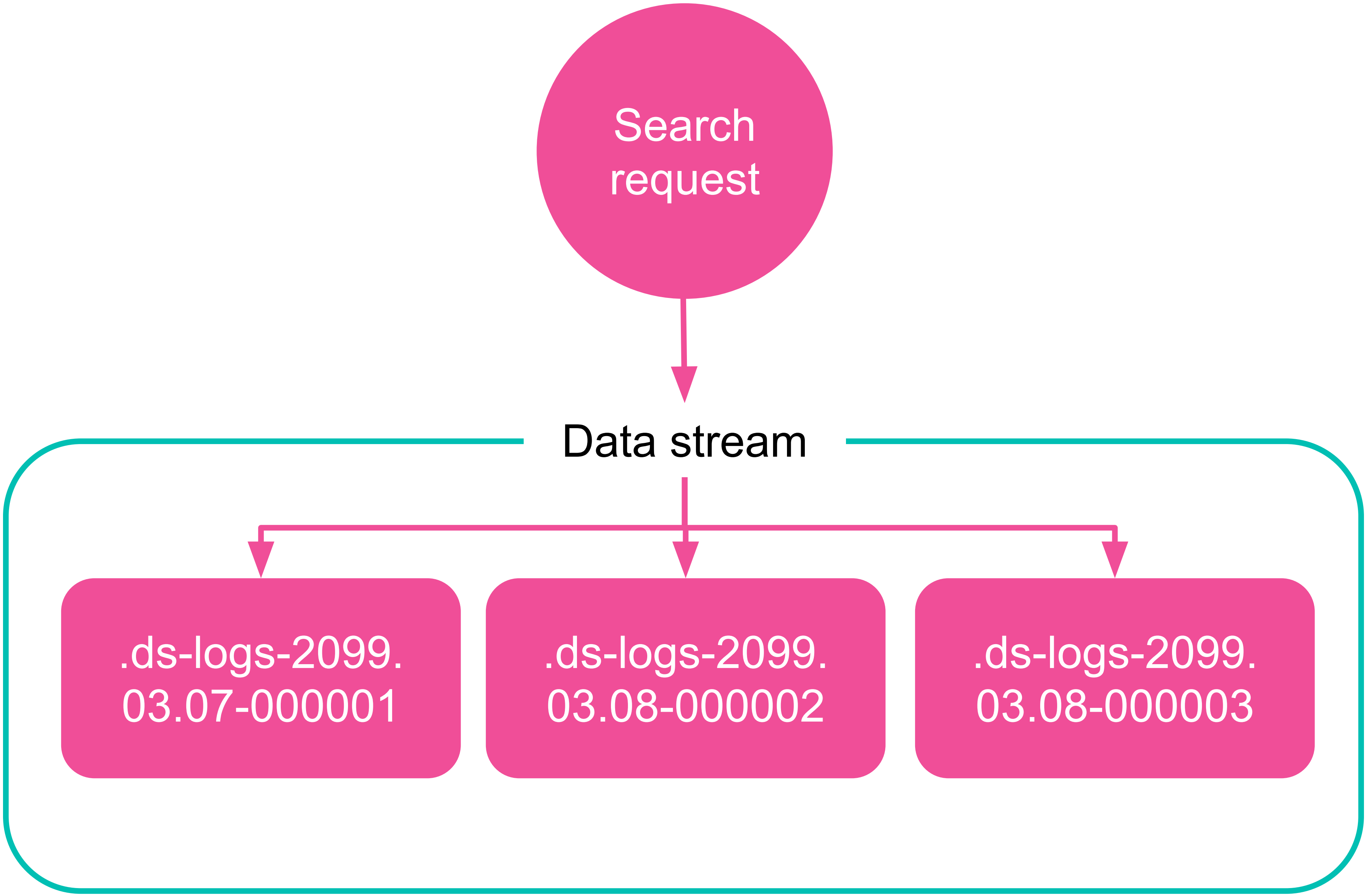
- **Data streams** are purpose-built for indexing continuously generated time-based data (e.g., logs, metrics).
- Abstracts away index creation and rollover—managed automatically via ILM policies.
- Backed by a series of **hidden backing indices**.

```
PUT _index_template/logs_template
{
  "index_patterns": ["logs-*"],
  "data_stream": { }
}
```

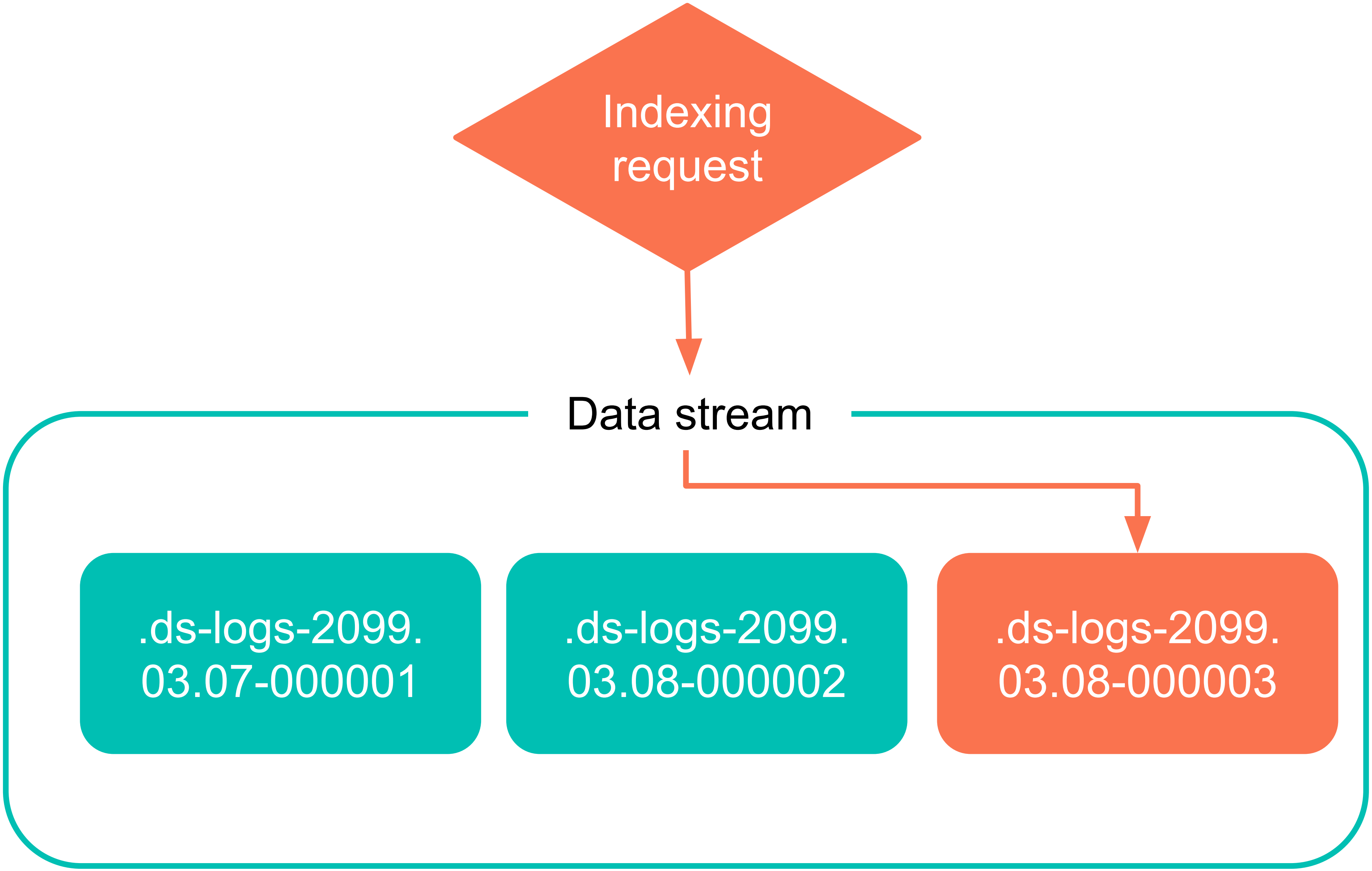
DATA STREAMS

- Use `POST /logs-myapp/_doc` to index into a data stream.
- Ideal for observability, log management, and metric collection.

DATA STREAMS



DATA STREAMS



SUMMARY

- Choosing the right **indexing strategy** is foundational for Elasticsearch performance and maintenance.
- Use **settings** and **templates** to enforce structure and consistency.
- Leverage **aliases** for flexibility and zero-downtime changes.
- Implement **ILM** and **data streams** for scalable, hands-off management of time-series data.

[10] - ELASTICSEARCH BEST PRACTISES

- Sharding
- Aliases
- Re-indexing
- Optimizing Indexing and Search Queries
- Kibana and Dashboards
- Discovery Panel
- Visualisations and Dashboards
- Development Panel

SHARDING

Shards are the fundamental units of storage and distribution in Elasticsearch.

Best practices:

- Choose an optimal **number of shards** based on index size and expected growth. Too many small shards waste resources; too few large shards reduce parallelism.
- Use **shard size guidelines** (e.g., 10–50 GB per shard) to balance performance and manageability.
- Use **Index Lifecycle Management (ILM)** with rollover to manage shard growth over time.

ALIASES

- **Index aliases** provide an abstraction layer between your application and the physical index name.
- Use aliases to:
  - Support **zero-downtime reindexing** by switching aliases to new indices.
  - Implement **filtered aliases** for multi-tenant or secured access patterns.
  - Define **write aliases** for ingestion workflows using ILM rollover.

RE-INDEXING

- **Re-indexing** is necessary when mappings or analyzers change.
- Best practices:
  - Use the `_reindex` API to copy data from one index to another with new mappings.
  - Reindex into a new index with a temporary name, then switch the alias.
  - For large datasets, consider **scroll + bulk reindexing** to avoid timeouts.
  - Always **test reindexing** in a non-production environment when possible.

OPTIMIZING INDEXING

- Keep indexing fast and efficient by:
  - **Disabling `_source` or storing minimal fields** when full document retrieval isn't needed.
  - Reducing `refresh_interval` during heavy indexing to avoid constant segment merges.
  - Using **bulk API** for large-scale ingestion.
  - Avoiding unnecessary fields, large nested structures, or deeply nested documents.
  - Turning off **indexing for non-searchable fields** using `"index": false`.

OPTIMIZING SEARCH QUERIES

- Efficient search is key to good user experience and cluster performance.
- Best practices:
  - Use **filters instead of queries** for boolean conditions—they're cached.
  - Avoid wildcard searches (`*term*`) on large fields—use edge n-grams or keyword subfields.
  - Use **doc values** for aggregations and sorting.
  - Paginate large result sets using `search_after` instead of deep paging with `from/size`.

KIBANA AND DASHBOARDS

- Kibana is the primary UI for interacting with Elasticsearch data.
- Best practices for using dashboards:
  - Design dashboards around **specific use cases or KPIs**.
  - Avoid overloading dashboards with too many visualizations or data-heavy panels.
  - Use **filters and time selectors** to reduce data volume and improve performance.
  - Save frequently used queries and visualizations as **Reusable Panels**.

DISCOVERY PANEL

- The **Discovery Panel** allows raw exploration of data indexed in Elasticsearch.
- Tips:
  - Use saved searches for commonly explored datasets.
  - Leverage field filters to reduce visible columns and focus on relevant fields.
  - Use time range and query bar effectively to avoid querying large datasets unnecessarily.

VISUALIZATIONS AND DASHBOARDS

- Visualizations provide insight into trends, metrics, and anomalies.
- Best practices:
  - Use the **Lens** editor for quick and flexible visualizations.
  - Choose appropriate visualization types (e.g., bar for categories, line for time series).
  - Aggregate data appropriately—e.g., `avg` for continuous values, `count` for events.
  - Limit time ranges or sample data to improve load times on high-volume dashboards.

DEVELOPMENT PANEL

- The **Dev Tools panel** in Kibana is a powerful interface for working directly with Elasticsearch via the Console.
- Use cases:
    - Running ad hoc queries and indexing commands.
    - Testing mapping, analyzers, and search DSL.
    - Reviewing response structures and debugging.

DEVELOPMENT PANEL

- Best practices:
  - Keep common queries saved for reuse.
  - Use comments and formatting to document and organize queries.
  - Test and prototype in Dev Tools before integrating into production applications.

SUMMARY

Area	Key Practices
Sharding	Balance shard count with data volume and growth
Aliases	Enable abstraction and zero-downtime changes
Re-indexing	Plan for schema evolution and minimize downtime
Indexing	Use bulk operations, disable unneeded features

SUMMARY

Area	Key Practices
Search Queries	Use filters, avoid deep pagination, optimize DSL
Kibana	Keep dashboards focused and performant
Discovery Panel	Use filters and saved searches effectively

SUMMARY

Area	Key Practices
Visualizations	Match chart types to data and use time filters
Dev Tools	Prototype, test, and document queries efficiently

[11] - BACKUP / DISASTER RECOVERY

- What is a Snapshot?
- Creating a Snapshot Repository
- Taking a Snapshot
- Restoring a Snapshot
- Best Practices
- Snapshot Security Considerations

WHAT IS A SNAPSHOT?

- A **snapshot** is a backup of:
  - One or more indices, or
  - The entire cluster state (including index settings, mappings, and templates).
- Snapshots are **incremental**—only changed data is stored, minimizing storage and time requirements.

SNAPSHOT TYPES

- Snapshots are stored in a **repository**, which can be:
  - Shared filesystem
  - Amazon S3
  - Google Cloud Storage
  - Azure Blob Storage
  - HDFS or other custom plugins

CREATING A SNAPSHOT REPOSITORY

Before taking snapshots, a repository must be registered:

```
PUT _snapshot/my_backup_repo
{
  "type": "fs",
  "settings": {
    "location": "/mount/backups",
    "compress": true
  }
}
```

The path must be accessible by all nodes in the cluster.

TAKING A SNAPSHOT

You can take a snapshot of specific indices or the entire cluster:

```
PUT _snapshot/my_backup_repo/snapshot_2025_10_13
{
  "indices": "logs-*,metrics-*",
  "include_global_state": true
}
```

- Snapshots are non-blocking but should be scheduled during off-peak times.
- Snapshot policies (Kibana) allows to schedule snapshots.

RESTORING A SNAPSHOT

Restoring from a snapshot can be done to recover lost data or migrate environments:

```
POST _snapshot/my_backup_repo/snapshot_2025_10_13/_restore
{
  "indices": "logs-*",
  "rename_pattern": "logs-(.+)",
  "rename_replacement": "restored-logs-$1"
}
```

- You can choose to:
    - Restore specific indices
    - Restore without changing index names
    - Restore only the cluster state or mappings
- Restoring can be done to the same or a different cluster (useful for DR environments).

BEST PRACTICES

- **Automate snapshots** regularly (daily/hourly based on data criticality).
- **Test restores** periodically to ensure recoverability.
- Monitor snapshot duration and size.
- Enable **security and access controls** on your backup storage.
- Consider **cross-cluster replication** (CCR) for active-active disaster recovery scenarios.
- Store snapshots in a **different availability zone or region** for true DR.

SNAPSHOT SECURITY CONSIDERATIONS

- Use **role-based access control (RBAC)** to restrict snapshot and restore permissions.
- Encrypt snapshot storage (most cloud providers support encryption at rest).
- Keep credentials and repository access keys secure and rotated regularly.

SUMMARY

Feature	Description
Snapshot	Incremental backup of indices and metadata
Repository	Storage location for snapshots
Restore	Recover data and cluster settings
Best Practice	Automate, test, secure, and monitor backups
Use Case	Disaster recovery, environment migration, data archival

END OF THE TRAINING

Thank you for your participation