JOHN BRYCE
Leading in IT Education
matrix company

UPDATED TO ELK V.8.3.3

Deep Dive: Adv Data Modeling & Relationships

●●●

Provided by DevOpShift & Autherd by Yaniv Cohen
DevopShift CEO and Head of Devops JB

- Data Relationships -

- Normalization vs. Denormalization - Introduction -

A

Normalization vs. Denormalization - Introduction

Understanding Data Structures in Elasticsearch

Description: Before diving into how Elasticsearch handles data, it's crucial to grasp the foundational concepts of data structuring: normalization and denormalization.

A

Normalization vs. Denormalization - Introduction

What is Normalization?

Definition:
- The process of efficiently organizing data in databases.

Goal:
- Minimize redundancy and dependency by organizing fields and table of data.

Use-Cases:
- Relational databases where data integrity and ACID (atomicity, consistency, isolation, durability) properties are vital.

Example:
- Splitting a table into two tables to eliminate data redundancy.

Normalization vs. Denormalization - Introduction

What is Denormalization?

Definition:

- The process of integrating normalized datasets for querying and analytical purposes.

Goal:

- Improve performance by adding redundant copies of data

Use-Cases:

- Data warehousing, NoSQL databases, and Elasticsearch for faster read operations.

Example:

- Combining multiple tables into one table/view to avoid joins

Normalization vs. Denormalization - Introduction

Trade-offs -

Normalization

Pros:

- Data integrity, minimizes redundancy, smaller data size.

Cons:

- Increased complexity, potentially slower read performance due to joins.

Denormalization

Pros:

- Faster reads, simpler query patterns, no joins needed.

Cons:

- Potential for data inconsistency, larger data size, complex write operations

A

Normalization vs. Denormalization - Introduction

Elasticsearch's Approach

Primarily Denormalized:

- Elasticsearch, as a search engine, is optimized for read-heavy operations.

- Nested Objects & Parent-Child: While denormalization is preferred, ES offers ways to maintain relationships.

Example: E-commerce Products

- **Normalized** (Relational DBs): Separate tables for products, categories, and manufacturers.

- **Denormalized** (Elasticsearch): A single document for a product containing all relevant information, including category details and manufacturer data.

A

Normalization vs. Denormalization - Introduction

Strategy in Elasticsearch

Default to Denormalization:
- Given its nature, start with a denormalized approach.

Consider Relationships:
- Use tools like nested objects and parent-child only when necessary to maintain data relationships (TBD)

Evaluate Use Case:
- Understand the specific requirements of your application before deciding on a data model.

A

Normalization vs. Denormalization - Introduction

Conclusion

While relational databases typically emphasize normalization, Elasticsearch's primary goal is speed and efficiency, leading to a more denormalized approach. However, with the tools provided by Elasticsearch, you can find a balance that fits your specific needs.

A

Some serious issues up ahead :-)

- QUESTION... -

A

Normalization vs. Denormalization - QUESTION

- How do we handle Updates in Denormalized (IN ES) vs Normalization in DB?

Updates in databases, whether following a normalized or denormalized approach, have distinct challenges and mechanisms and it require careful planning. Let's delve into it:

- Handling Updates in Databases -

U

Handling Updates in Databases

Differences in Approach for Normalized and Denormalized Systems

Updates in databases can be straightforward or complex, depending on the architecture and the data modeling. The approach varies distinctly between normalized relational databases and denormalized systems like Elasticsearch and also distributed databases.

U

Handling Updates in Databases

Updates in a Normalized Database

1. Efficiency in Redundancy: Since data is not redundant, an update typically affects a single location. Once you change a record, all references to that record are inherently up-to-date.

2. Complexity: Due to the interconnected nature of tables in relational databases, updates might involve multiple tables, potentially triggering cascades or requiring joins.

3. Consistency: The *__ACID (see next slide)__ properties of relational databases ensure atomic updates, making sure the data remains consistent before and after the update.

Example: Updating a user's address in a Users table will reflect wherever that user's data is fetched, without needing to change multiple tables.

U

Handling Updates in Databases

Updates in a Normalized Database

ACID is an acronym that stands for atomicity, consistency, isolation, and durability. Together, these ACID properties ensure that a set of database operations (grouped together in a transaction) leave the database in a valid state even in the event of unexpected errors.

U

Handling Updates in Databases

Updates in a Denormalized System (Elasticsearch)

1. Redundancy Challenges: Since the same piece of data might exist in multiple documents, you might need to update multiple documents to keep data consistent.

2. Performance Implications: While reading is fast in denormalized systems, updates might be slower due to the need to update multiple documents.

3. Version Control: Elasticsearch provides versioning for documents, which helps handle concurrent updates and potential conflicts.

4. Partial Updates: Elasticsearch supports the _update endpoint, which allows for partial updates to documents. However, under-the-hood, this is still a full reindex of the document.

Example: If a product's price changes and that product data exists in multiple documents (due to denormalization), all those documents need updating.

U

Handling Updates in Databases

Updates in a Denormalized System (Elasticsearch)

BUT THAT SOUNDS BAD!
Is that the only option ?

Before we talk about our options, Let understand the required balance

U

Handling Updates in Databases

Striking a Balance in Elasticsearch

1. Reindexing Strategy: Given the challenges with updates in a denormalized system, sometimes it's more efficient to reindex data rather than updating individual documents.

2. Modeling Decisions: Consider the frequency of updates when designing your Elasticsearch index. If certain fields update frequently, you might reconsider how they're modeled or stored... **note this part.**

U

Handling Updates in Databases

Question

- what about in ES to add new document with the updated information instead of updating one or many documents ?

Answer: Adding a new document with the updated information rather than updating an existing one is a common strategy in Elasticsearch, especially when considering the immutability paradigm. Let's delve into this

U

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Adding New Documents vs Updating Existing One(s)

In many situations, especially when dealing with time-based data like logs or events, it's common to leverage the immutability paradigm in Elasticsearch. This involves adding new documents for updated information rather than altering existing ones.

U

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Benefits of Immutability in Elasticsearch

1. Performance: Indexing a new document is typically faster than updating an existing one because updates in Elasticsearch involve a delete and reindex process under the hood.

2. Data Integrity: Immutability ensures that historical data remains untouched, providing a reliable audit trail.

3. Simplified Concurrency: By always writing new documents, you avoid potential versioning conflicts that arise from concurrent updates.

4. Optimized for Write-Once Patterns: Elasticsearch, especially in its use as a logging solution, is optimized for situations where data is written once and rarely updated.

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Drawbacks and Considerations

1. Data Volume: Continually indexing new documents can increase the volume of data. This might have storage implications.

2. Complex Queries: If there are multiple versions of a "logical" document, querying for the most recent or relevant one can be more complex.

3. Data Cleanup: Older or obsolete versions of documents might need to be cleaned up to free up space and maintain performance.

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Use Cases in Elasticsearch

1. Event Logging: Systems that record events, like server logs, where each entry is a unique event and previous events are immutable.

2. Time Series Data: Measurements or metrics that are recorded over time where historical data remains static.

3. Audit Trails: Situations where you need a record of all changes, not just the current state.

U

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Implementation in Elasticsearch

1. Document IDs: Consider how you assign document IDs. Letting Elasticsearch auto-generate IDs is straightforward, but if you want control over overwriting specific documents, you'd need a strategy for generating consistent IDs.

2. Aliases and Rollover: For time-based indices, use index aliases and the rollover API to manage when to write to new indices and how to query across them.

Conclusion

Opting for immutability by indexing new documents in Elasticsearch can offer significant performance and data integrity benefits, especially for certain use cases. However, it's crucial to be aware of the associated challenges and design your indexing and querying strategies accordingly.

- Example Scenario: User Profile Updates -

U

Handling Updates in Databases

The Immutability Paradigm in Elasticsearch

Implementation in Elasticsearch

In this example, let's assume we have an Elasticsearch index storing user profiles. When a user's profile is updated, instead of modifying the existing document, we'll add a new document with the latest data.

U

Handling Updates in Databases

Creating the Index

Create the following Index structure:

Index name: user_profiles

Structure:

```
{
 "mappings": {
   "properties": {
     "user_id": {"type": "keyword"},
     "name": {"type": "text"},
     "email": {"type": "text"},
     "update_timestamp": {"type": "date"}
   }
 }
}
```

In this mapping, user_id is a keyword used to identify each user uniquely. The update_timestamp field records the time of the profile update.

U

Handling Updates in Databases

Indexing Documents

When a user's profile is created or updated, we index a new document:

```
POST /user_profiles/_doc
{
  "user_id": "user123",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "update_timestamp": "2023-03-15T12:00:00"
}
```

U

Handling Updates in Databases

Indexing Documents

When a user's profile is created or updated, we index a new document:

```
POST /user_profiles/_doc
{
  "user_id": "user123",
  "name": "John Doe",
  "email": "john.newemail@example.com",
  "update_timestamp": "2023-03-16T15:00:00"
}
```

If John Doe updates his email, we index a new document with the current timestamp:

U

Handling Updates in Databases

Querying the Latest Profile Information

To retrieve the latest profile information for a user, we use a combination of term and sort:

```
GET /user_profiles/_search
{
  "query": {
    "term": {
      "user_id": {
        "value": "user123"
      }
    }
  },
  "sort": [
    { "update_timestamp": "desc" }
  ],
  "size": 1
}
```

This query returns the most recent document for user123, showing their latest profile information.

Benefits: This method ensures that all historical states of the user profile are preserved. It's also faster than updating a document as Elasticsearch doesn't have to delete and reindex the document.

Considerations: This approach increases the index size over time. It's important to have a strategy for archiving or cleaning up old documents if they become too numerous.

- PROBLEM! -

Partial Update from Different Application Views

U

Handling Updates in Databases

Scenario: Partial Update from Different Application Views

What will happen and how we would handle when an application or API only updates a subset of a document's fields (like just the email in a user profile), and doesn't have access to the entire document?

there are a few strategies you can consider

U

Handling Updates in Databases

Scenario: Partial Update from Different Application Views

Retrieve and Update:
first we retrieve the full document, apply the updates in our application, and then index the entire document.

Pros:

Straightforward and ensures document integrity.
Easy to manage and understand.

Cons:

Additional read operation required.
Slightly slower due to the read-modify-write cycle.

Handling Updates in Databases

In summary

indexing a new document for each update in Elasticsearch can be a more scalable and efficient approach, especially when dealing with frequent updates across many documents. This method aligns well with Elasticsearch's strengths in handling immutable data and can lead to better performance and simpler data management in certain scenarios.

U

Handling Updates in Databases

In summary

Approach: Indexing New Document for Each Update

Advantages:

Performance: Indexing new documents is generally faster in Elasticsearch than updating existing ones, as updates internally are a delete-then-index operation.

Immutability: This approach maintains historical data, allowing for time-based analysis and auditing.

Simplicity: Avoids the complexity of scripted updates or partial updates, especially when the full document context isn't available.

U

Handling Updates in Databases

Implementation Strategy

Unique Document IDs: Rather than using a static document ID, each update could be indexed under a unique ID or a combination of fields (like user_id and timestamp).

Latest Document Retrieval: To retrieve the latest state of a user's data, use query sorting based on a timestamp field. For instance:

Housekeeping: Periodically, you might need to clean up or archive older documents to manage the index size and maintain performance.

U

Handling Updates in Databases

Implementation Strategy

This approach, while introducing some complexity in terms of managing data volume and application logic, leverages Elasticsearch's strengths in indexing and searching large volumes of data efficiently. It can be particularly advantageous in scenarios with frequent and varied updates.

- Handling Arrays in Elasticsearch -

A

Handling Arrays in Elasticsearch

Understanding and Working with Arrays

Description: In Elasticsearch, any field can contain zero or more values by default. However, there is no dedicated array data type. A field's data type is determined by the type of the first value you index

A

Handling Arrays in Elasticsearch

Basic Concepts

No Special Array Type:

- Unlike some other systems, you don't explicitly define a field as an array. If you provide multiple values, Elasticsearch recognizes it as an array.

Same Data Type:

- All values in the array must be of the same data type. You cannot mix, for instance, strings and numbers in the same array.

A

Handling Arrays in Elasticsearch

Indexing Arrays

Example Document:

```
{
  "movie": "Inception",
  "tags": ["sci-fi", "thriller", "dream"]
}
```

Description: The tags field is recognized as an array because it holds multiple string values.

A

Handling Arrays in Elasticsearch

Searching in Arrays

Simple Queries: Searching arrays is as straightforward as searching single-value fields.

```
{
  "query": {
    "match": {
      "tags": "dream"
    }
  }
}
```

Exact Match: If you're looking for documents where an array field contains multiple specific values (e.g., both "sci-fi" and "dream"), you can use the terms query.

```
{
  "query": {
    "terms": {
      "tags": ["sci-fi", "dream"]
    }
  }
}
```

- Nested Objects in Elasticsearch -

Nested Objects in Elasticsearch

Handling Complex Data Structures

Introduction to Nested Objects:

Unlike arrays, nested objects allow you to index, search, and manage arrays of complex objects as independent documents.
Each object in the array is indexed as a separate hidden document, allowing for more precise querying.

Why Use Nested Objects?:

Useful for handling data that has an inherent hierarchical structure, like product catalogs, where products have multiple variants.

A

Nested Objects in Elasticsearch

Structuring and Indexing

Defining Nested Mappings:
Nested fields must be explicitly marked as such in the index mappings.

```
PUT /products
{
 "mappings": {
  "properties": {
   "variants": {
    "type": "nested"
   }
  }
 }
}
```

Indexing Nested Documents:
Nested objects are indexed in the same way as other fields but maintain their hierarchy.

```
POST /products/_doc
{
 "name": "Widget",
 "variants": [
  { "color": "blue", "price": 20 },
  { "color": "red", "price": 15 }
 ]
}
```

A

Querying Nested Objects

Searching Within Complex Structures

Nested Query Usage

Use the nested query to search within nested objects.

```
GET /products/_search
{
 "query": {
  "nested": {
   "path": "variants",
   "query": {
    "bool": {
     "must": [
      { "match": { "variants.color": "blue" }},
      { "range": { "variants.price": { "lte": 25 }}}
     ]
    }
   }
  }
 }
}
```

This query searches for products that have a blue variant priced at $25 or less.

A

Nested Objects

Limitations and Best Practices

Performance Considerations:
Nested objects can increase index size and affect performance. Use them judiciously.

Modeling Data:
Prefer flat structures where possible. Use nested objects only when necessary for preserving relationships.

Query Complexity:
Nested queries can be more complex and may require more careful construction.

- The Flattened Data Type in Elasticsearch -

A

Implementing the Flattened Data Type

Indexing Dynamic Content

Introduction to the Flattened Data Type:
The flattened data type is used for fields that contain an arbitrary set of key-value pairs, offering a way to index and search through them without pre-defining a strict mapping.
Ideal for situations with dynamic or unpredictable data structures.

Use Cases:
Handling semi-structured or schema-less data, such as user-generated metadata.
Flexible indexing where the data schema may evolve over time.

A

The Flattened Data Type in Elasticsearch

Managing Dynamic or Heterogeneous Object Fields

Defining a Flattened Field:
In your mappings, you can define a field as flattened to store varied key-value pairs.

```
PUT /dynamic_data
{
  "mappings": {
    "properties": {
      "attributes": {
        "type": "flattened"
      }
    }
  }
}
```

Indexing Flattened Data:
You can then index documents with diverse or unpredictable structures in this field.

```
POST /dynamic_data/_doc
{
  "attributes": {
    "color": "blue",
    "size": "large",
    "region": "northwest"
  }
}
```

A

Querying Flattened Fields

Searching Through Diverse Data Structures

Querying a Flattened Field

Queries on flattened fields can match any key-value pair stored in them

```
GET /dynamic_data/_search
{
  "query": {
    "match": {
      "attributes.color": "blue"
    }
  }
}
```

This example will match documents where the "attributes" field contains a key "color" with the value "blue".

Best Practices and Limitations

Effective Use of Flattened Data Type

When to Use Flattened Type:

1. Best suited for unstructured or semi-structured data with a large number of distinct attributes.

Performance Considerations:

1. Flattened fields are not as performant as regular fields for search operations and can consume more disk space.

Limitations:

1. Less flexibility in querying (e.g., no nested queries) and limited support for aggregations compared to regular object fields.

- Parent-Child Relationships in Elasticsearch -

P

Parent-Child Relationships in Elasticsearch

Modeling Complex Data Associations

Introduction to Parent-Child Relationships:

- Elasticsearch can model data that has one-to-many relationships (parent-child) within the same index.
- Useful for data models where entities are related but have independent lifecycles and characteristics.

Deep Dive into Parent-Child Concept:

- In Elasticsearch, the parent-child relationship is a way to associate one entity (the parent) with other related entities (the children) within a single index.
- Unlike traditional relational databases, this relationship in Elasticsearch is less about enforcing data integrity and more about enabling complex search scenarios.

Distinct Characteristics:

- Parent and child documents are indexed separately, allowing for independent updates and deletions, which is particularly beneficial in scenarios where child entities change more frequently than the parents.

P

Parent-Child Relationships in Elasticsearch

Modeling Complex Data Associations

Use Cases:

1. **Products (parent) and Reviews (child),** where reviews can be frequently added or updated independently of the product.

2. **User Profiles and User Actions**: A user profile (parent) can have associated actions or logs (children), like posts, comments, or transactions.

3. **Blog Posts and Comments**: Blog entries as parents with user comments as children, facilitating efficient querying of posts based on comment content.

P

Parent-Child Relationships in Elasticsearch

Modeling Complex Data Associations

Use Cases continue:

1. **E-Commerce:** Products with their Q&A sections, where each question can have multiple answers and discussions.

2. **Organizational Data:** Employees (children) associated with departments (parents), allowing searches like finding departments based on employee skills or roles.

3. **Geographical Data:** Cities or locations (parents) with associated events or listings (children).

4. **Healthcare Records:** Patient records (parent) with various medical reports or tests (children).

- Parent-Child Relationships
- Question -

P

Implementing Parent-Child Relationships

Setting Up Relational Data

Why Parent-Child and Not Nested?
The parent-child model is chosen over nested when child documents are large in number or frequently updated, as nested objects would require reindexing the entire document.

- Parent-Child Relationships -

Implementing Parent-Child Relationships

Setting Up Relational Data

Defining Parent-Child Mappings:

- Use the **join** field to establish parent-child relationships.

```
PUT /products
{
 "mappings": {
  "properties": {
   "my_join_field": {
    "type": "join",
    "relations": {
     "product": "review"
    }
   }
  }
 }
}
```

Here, product is the parent, and review is the child.

P

Implementing Parent-Child Relationships

Indexing

Index a parent document:
without specifying a parent

```
POST /products/_doc
{
  "name": "Coffee Maker",
  "my_join_field": "product"
}
```

Indexing Child Documents:
When indexing a child document, specify the ID of the parent.

```
POST /products/_doc?routing=1
{
  "text": "Great coffee maker!",
  "my_join_field": {
    "name": "review",
    "parent": "1"
  }
}
```

****routing** parameter ensures the child document is stored in the same shard as the parent. (see next slide)

- Parent-Child Relationships
- Question -

P

Shard Allocation in Parent-Child Relationships

Why Parent and Child Must Reside on the Same Shard?

Co-Location Requirement:

- In Elasticsearch, a parent and its children must be stored on the same shard because the parent-child relationship is implemented using a "join" at the shard level.

- This co-location is crucial to ensure that the join operation can be performed efficiently without the overhead of inter-shard communication.

Shard Allocation in Parent-Child Relationships

Why Parent and Child Must Reside on the Same Shard?

Query Execution Mechanics:

- When a query involving a parent-child relationship is executed, Elasticsearch needs to check the relationship between these documents.

- If parent and child documents were stored on different shards, it would require network calls between shards for each query, significantly impacting performance.

Routing Strategy:

- To ensure parents and children are co-located, Elasticsearch uses a routing mechanism.

- Typically, the parent's ID is used to route both parent and child documents to the same shard.

P

Shard Allocation in Parent-Child Relationships

Why Parent and Child Must Reside on the Same Shard?

Example of Routing:

```
POST /user_profiles/_doc?routing=user123
{
  "name": "John Doe",
  "type": "parent"
}

POST /user_profiles/_doc?routing=user123
{
  "comment": "This is a child document.",
  "parent_id": "user123",
  "type": "child"
}
```

Here, both documents are routed using the parent's ID (user123), ensuring they reside on the same shard.

Performance Implications:

- While this ensures efficient query execution, it also implies that all the load related to a particular parent-child relationship is concentrated on a single shard.
-
- Proper planning and sharding strategy are essential to prevent hotspots and ensure balanced data distribution across the cluster.

- Parent-Child Relationships -

P

Querying Parent-Child Data
INDEX DOCUMENT

Parent

```
POST /products/_doc
{
  "name": "Coffee Maker",
  "my_join_field": "product"
}
```

Child

```
POST /products/_doc?routing=1
{
  "text": "Great coffee maker!",
  "my_join_field": {
    "name": "review",
    "parent": "1"
  }
}
```

P

Querying Parent-Child Data

HOWTO

Use has_child or has_parent queries

```
GET /products/_search
{
 "query": {
  "has_child": {
   "type": "review",
   "query": {
    "match": {
     "text": "great"
    }
   }
  }
 }
}
```

Here, both documents are routed using the parent's ID (user123), ensuring they reside on the same shard.

Performance Implications:

- While this ensures efficient query execution, it also implies that all the load related to a particular parent-child relationship is concentrated on a single shard.
-
- Proper planning and sharding strategy are essential to prevent hotspots and ensure balanced data distribution across the cluster.

- Parent-Child Relationships -
- LAB -

P

Creating an IMDB-like Index with Parent-Child Relationships

Create an Elasticsearch index to model movies, their associated actors, and reviews using parent-child relationships.

Instructions:

Step 1: Set Up the Movie Index with Parent-Child Mappings

1. Define the index with a join field to establish parent-child relationships.

2. **Tip** - the relations needs to be as followed :

```
    "relations": {
     "movie": ["actor", "review"]
```

3. The schema fields for the index can be as followed:

```
"title": { "type": "text" },
 "release_year": { "type": "date" },
   // Add other movie fields as needed
  "actor_name": { "type": "text" },
  "review_text": { "type": "text" },
  "review_date": { "type": "date" }
```

P

Creating an IMDB-like Index with Parent-Child Relationships

Create an Elasticsearch index to model movies, their associated actors, and reviews using parent-child relationships.

Step 2: Indexing Parent Documents (Movies)

- Index a movie as a parent document.

Step 3: Indexing Child Documents (Actors and Reviews)

- Index an actor as a child document related to the movie.
- Similarly, index a review for the movie

P

Creating an IMDB-like Index with Parent-Child Relationships

Create an Elasticsearch index to model movies, their associated actors, and reviews using parent-child relationships.

Step 4: Querying the Data

- Craft a query to find all movies with reviews containing the word "great."

Deliverables:

- An Elasticsearch index with parent-child mappings.
- Sample indexed data for movies, actors, and reviews.
- Queries to retrieve data based on parent-child relationships.

- Parent-Child Relationships -
- LAB SOLUTION -

Movie Index with Parent-Child Mappings

```
PUT /imdb
{
 "mappings": {
  "properties": {
   "movie_join_field": {
    "type": "join",
    "relations": {
     "movie": ["actor", "review"]
    }
   },
   "title": { "type": "text" },
   "release_year": { "type": "date" },
   // Add other movie fields as needed
   "actor_name": { "type": "text" },
   "review_text": { "type": "text" },
   "review_date": { "type": "date" }
  }
 }
}
```

Index a movie as a parent document

```
POST /imdb/_doc
{
 "title": "Inception",
 "release_year": "2010",
 "movie_join_field": "movie"
}
```

Indexing Child Documents Actors

```
POST /imdb/_doc?routing=1
{
 "actor_name": "Leonardo DiCaprio",
 "movie_join_field": {
  "name": "actor",
  "parent": "1"
 }
}
```

Indexing Child Documents Review

```
POST /imdb/_doc?routing=1
{
 "review_text": "Great visual effects!",
 "review_date": "2023-03-21",
 "movie_join_field": {
  "name": "review",
  "parent": "1"
 }
}
```

Querying the Data

```
GET /imdb/_search
{
  "query": {
    "has_child": {
      "type": "review",
      "query": {
        "match": {
          "review_text": "great"
        }
      }
    }
  }
}
```

- END -