Kafka 이벤트 동기화 기반 Elasticsearch 검색 아키텍처: RDB 조인에서 검색 인덱스로 전환

Skill Stack

분야	기술 스택
MSA & Gateway	🟦 Eureka, Spring Cloud Gateway, Spring Cloud Netflix
Backend	🟩 Spring Boot 3.x, Spring Web, Spring Security, Spring Data JPA, Spring Retry, Spring Data Redis, Spring Data Elasticsearch, Spring Kafka, Spring Cloud OpenFeign
Database	🟥 MySQL 8.0 (도메인별 DB 분리), PostgreSQL 15
Messaging	🟧 Apache Kafka 2.8.1, Zookeeper 3.8
Cache & Token	🟨 Redis 7, JWT (jjwt 0.12.5)
Logging & Monitoring	🟪 ELK Stack (Elasticsearch 7.17 운영 로그, Elasticsearch 8.13 검색 전용, Logstash, Kibana, Filebeat), Logstash Logback Encoder
Infra & CI/CD	🐳 Docker (docker-compose 기반 MSA 로컬 인프라 구성), GitHub Actions
Libraries & Tools	🛠️ Lombok, Jakarta Validation API, MySQL JDBC Driver

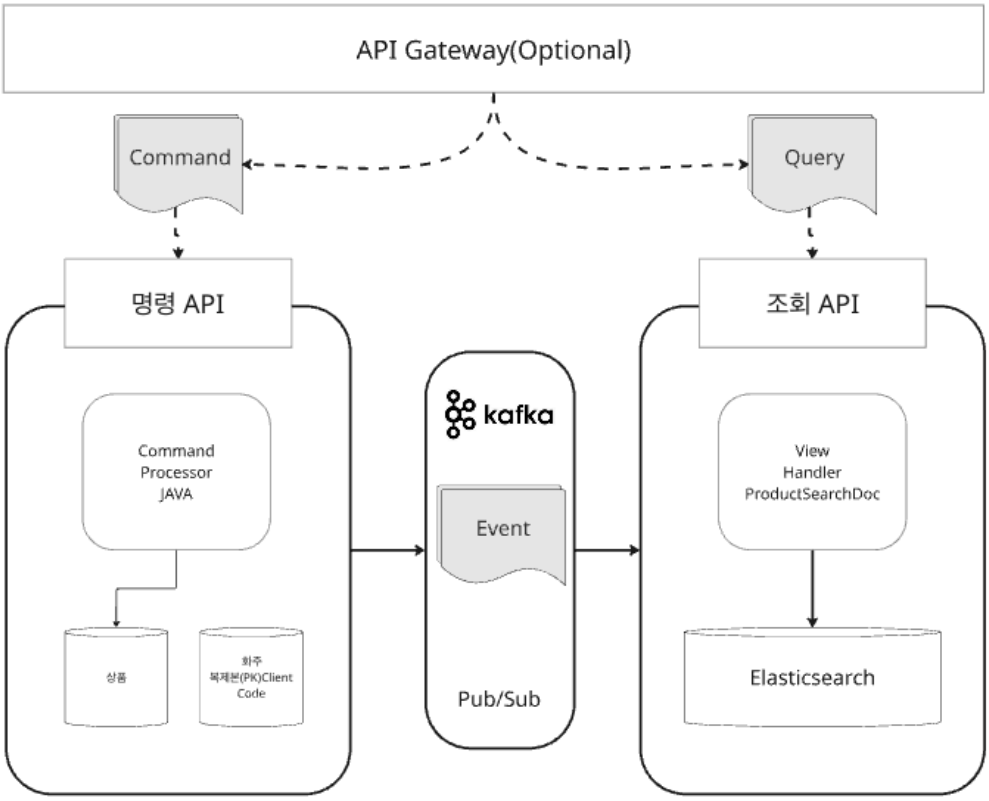
배경

WMS 도메인 특성상 운영자가 **상품 × 화주(고객/거래처) 정보를 한 화면에서** 빠르게 찾아야 합니다.
실제 흐름은 단순 키 조회가 아니라, **다중 필드 조건 + 부분일치(LIKE) + OR 조합 + 정렬 + 페이징**이 동시에 걸리는 형태가 많습니다.

초기에는 **MySQL 단일 조회 경로**로 풀었습니다.
사용자(화주) 마스터는 Kafka로 **상품 서비스 DB에 실시간 복제**해 참조 무결성과 정합성을 확보했고, 서비스 레이어에서는 **RDB 조인 기반 통합 조회**를 제공했습니다. 데이터 일관성 측면에서는 안정적이었지만, **조회 비용**이 문제였습니다. 다중 LIKE/OR와 정렬·페이징이 겹치면 인덱스를 효율적으로 쓰기 어렵고, 조인 후 정렬, 임시 테이블 생성, 광범위한 인덱스 스캔이 빈번히 발생했습니다. 동시 접속자가 늘면 **CPU/IO 스파이크**, **쿼리 급증**, **지연 상승**이 관측되었습니다

이러한 배경으로, **명령(Command)과 조회(Query)의 책임을 분리(CQRS)** 하고, **MySQL은 정합성**, **Elasticsearch는 검색/조회 성능과 사용성**을 담당하도록 아키텍처를 전환하기로 했습니다.

전체 아키텍처 설계



설계 의도 (CQRS 관점)

쓰기와 정합성은 **MySQL**이 책임지고, 검색과 조회는 **Elasticsearch(ES)**가 담당하도록 역할을 분리하였습니다. 이는 동일한 데이터베이스에서 **트랜잭션**과 **검색/조회 워크로드**가 공존할 때 발생하는 잠금·인덱스 경합·버퍼 풀 압박 등을 구조적으로 회피하기 위함이었습니다. ES가 텍스트 검색, 정렬·페이징, 샤딩에 강점을 가지므로 **조회 경로를 ES로 이관**하여 **확장성** 확보하였습니다. 이 접근은 **CQRS(Command Query Responsibility Segregation) 패턴**을 적용한 것입니다.

📕CQRS 패턴이란?

CQRS는 **명령(Command)** 과 **조회(Query)** 의 책임을 **논리적으로 분리**하는 아키텍처 패턴을 의미합니다.

핵심 개념을 다음과 같이 정리하였습니다.

1. **명령 모델(Command Model)**
    - 데이터 **생성/수정/삭제**와 같은 상태 변경을 담당합니다.
    - **정합성**과 트랜잭션 경계를 우선시합니다.
    - 본 시스템에서는 **ProductService + MySQL**이 이에 해당하도록 하였습니다.

2. **조회 모델(Query Model)**
    - 화면/검색/리스트/집계 등 **읽기 최적화**를 담당합니다.
    - 스키마를 **읽기 친화적으로 별도 설계**할 수 있습니다.
    - 본 시스템에서는 **Elasticsearch 인덱스(ProductSearchDoc)** 가 이에 해당하도록 하였습니다.
3. **동기화 메커니즘**
    - 명령 모델에서 상태가 변경되면 **도메인 이벤트**를 발행하고, 이를 구독하여 조회 모델을 **비동기 갱신**합니다.
    - 본 시스템에서는 **Kafka 이벤트**로 변경 사실을 전달하고, **버전(version) 기반 멱등성**으로 역순/중복 이벤트를 방지하였습니다.

인덱스(문서) 모델 — "검색 전용 문서"로 재구성

조회 요구에 맞추어 RDB 스키마를 그대로 복제하지 않고, 화면과 검색에 실제로 사용되는 속성만 선별하여 문서를 재구성하였습니다. 문서 식별자는 `productCode`로 고정하여 동일 상품에 대한 색인이 항상 같은 문서를 갱신하도록 하였고, 이로써 Upsert 흐름을 단순화하였습니다. 조회에서 가장 많이 사용하는 필터가 `clientCode`였기 때문에 해당 값을 라우팅 키로 사용하였으며, 저장·삭제 시에는 라우팅이 자동으로 적용되도록 하였습니다. 검색 요청에서는 라우팅 값을 명시적으로 전달하여 쿼리가 불필요하게 모든 샤드로 확산되지 않도록 하였습니다.

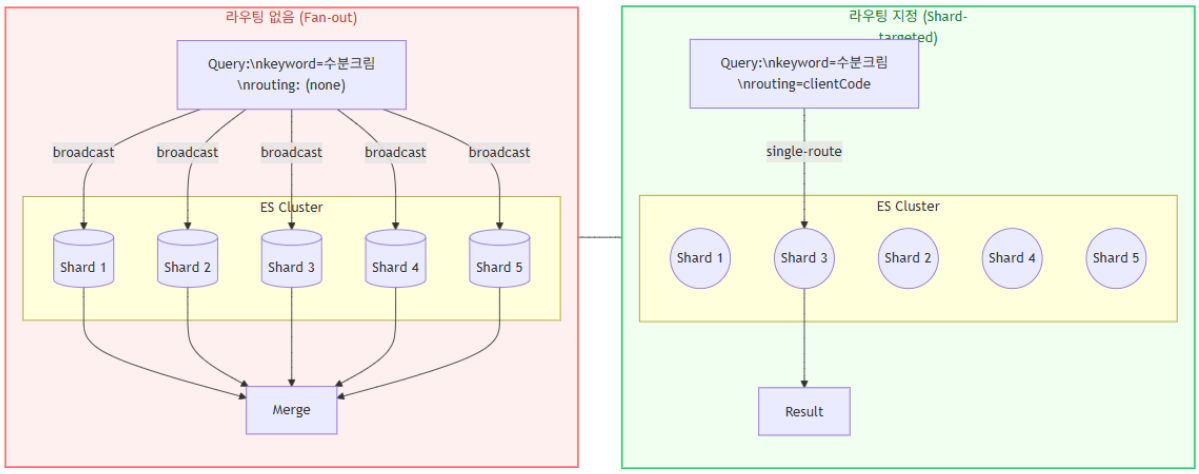
이벤트 처리에서는 역순·중복 상황을 고려하여 애플리케이션 레벨의 이벤트 버전을 문서에 함께 저장하였습니다. 색인 전 단계에서 기존 문서의 버전과 비교하여 더 오래된 이벤트는 건너뛰도록 하였고, 이 방식으로 멱등성을 유지하였습니다. 또한 `lastEventAt`에 기록하여 색인 지연이나 누락 구간을 추적할 수 있도록 하였습니다.

텍스트 검색 품질을 높이기 위해 `productName`, `brand`, `style`, `color`, `size`를 결합한 `keywords` 보조 필드를 두었습니다.

아래는 실제 문서 클래스의 예시입니다.

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Document(indexName = "product_search_v1")
public class ProductSearchDoc {

    /** 동일 상품에 대한 업서트 기준이 되는 문서 식별자 */
    @Id
    private Long productCode;

    /** 저장/삭제 시 라우팅 키로 사용 */
    @Routing
    private Long clientCode;

    /** 화면 표시 및 다중 필드 검색에 사용되는 주요 속성 */
    private String clientName;
    private String productName;
    private String brand;
```

```
    private String style;
    private String color;
    private String size;

    /** 정렬/필터에 활용되는 보조 속성 */
    private String productYear;
    private String productSeason;
    private BigDecimal retailPrice;
    private String useYn;

    /** 부분 검색·오타 내성 대응을 위한 보조 문자 */
    private String keywords;

    /** 운영 진단을 위한 마지막 이벤트 반영 시각 */
    @Field(type = FieldType.Date, format = DateFormat.epoch_millis)
    private Instant lastEventAt;

    /** 애플리케이션 레벨 멱등 체크를 위한 이벤트 버전 */
    private Long version;
}
```

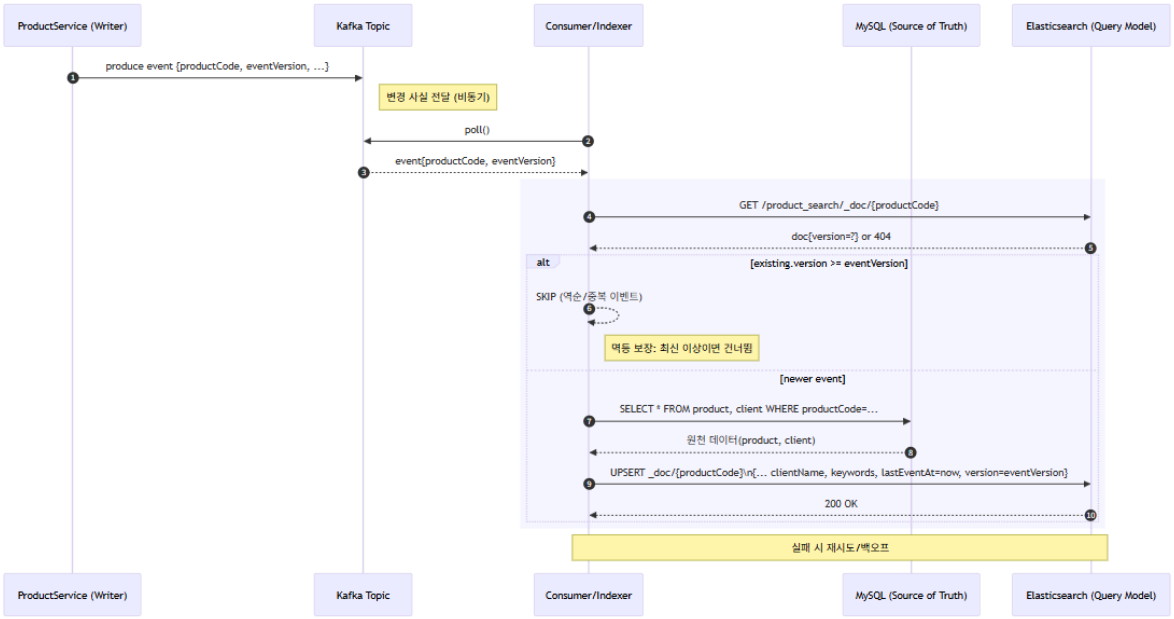


검색 시에는 `clientCode` 값을 라우팅 키로 함께 전달하여, **불필요하게 모든 샤드를 뒤지는 상황** 을 줄였습니다. 또한 매핑 변경이나 분석기 조정이 필요할 때는 새 버전 인덱스를 생성해 백그라운드로 재색인하고, **alias 전환**을 통해 무중단으로 반영했습니다. 이렇게 조회 전용 문서를 최소 단위로 설계하고, 라우팅·버전·보조 필드를 함께 적용하여 **색인 경로의 안정성과 조회 성능의 예측 가능성**을 동시에 확보할 수 있었습니다.

ElasticsearchProductSearchAdapter

이 어댑터는 **색인(upsert)** 과 **검색(query)** 을 한 클래스 안에서 책임지도록 구성하였습니다. 이벤트를 통해 유입된 변경 사항은 먼저 RDB에 반영한 뒤, 동일한 원천 데이터를 이용해 **ES 검색 전용 문서**로 업서트하도록 하였습니다. 이때 문서의 식별자를 `productCode`로 고정하여 중복/재처리 상황에서도 항상 동일 문서를 갱신하도록 하였고, 이벤트의 `version`을 문서에 함께 저장하여 **멱등성**을 확보하였습니다. 검색 경로에서는 다중 필드 질의, 최소한의 `_source` 반환, `track_total_hits` 회피, `search_after` 기반 딥 페이지네이션을 적용하여 검색 비용 효율을 높이고자 하였습니다. 모든 단계의 상태는 `MDCHelper`를 통해 **요청 단위 디버그 로그**로 남겨 원인 분석이 용이하도록 하였습니다.

단건 업서트: 문서 ID 고정 + 버전 비교로 멱등화



업서트 흐름은 매우 단순하게 가져갔습니다. 먼저 ES에 동일한 `productCode` 문서가 있는지 확인하고, 있다면 저장된 `version`과 이번 이벤트의 `eventVersion`을 비교하였습니다. 저장된 버전이 더 크거나 같으면 **역순/중복 이벤트**로 판단하여 곧바로 종료하였습니다. 그 외의 경우에는 RDB에서 상품과 화주 원천 데이터를 읽어와 검색 전용 문서를 만들고 저장하였습니다.

```
@Override
public void upsertByProductCode(Long productCode, Long eventVersion) {
    var existing = esRepository.findById(productCode).orElse(null);
    if (existing != null && existing.getVersion() != null
            && existing.getVersion() >= eventVersion) {
        debug("이미 최신 버전 존재 - skip (existingVersion=" + existing.getVersion() + ")");
        return;
    }

    ProductMaster p = productRepo.findById(productCode)
            .orElseThrow(() -> new IllegalStateException("Product not found: " + productCode));
    Long clientCode = Long.valueOf(p.getClientCode());
    ClientMaster c = clientRepo.findById(clientCode).orElse(null);

    ProductSearchDoc doc = ProductSearchDoc.builder()
```

```
            .productCode(p.getProductCode())   // 문서 ID → 업서트 기준
            .clientCode(clientCode)
            .clientName(c != null ? c.getClientName() : null)
            .productName(p.getProductName())
            .brand(p.getBrand())
            .style(p.getStyle())
            .color(p.getColor())
            .size(p.getSize())
            .productYear(p.getProductYear())
            .productSeason(p.getProductSeason())
            .retailPrice(p.getRetailPrice())
            .useYn(p.getUseYn())
            .keywords(buildKeywords(p))
            .lastEventAt(Instant.now())
            .version(eventVersion)           // 멱등 버전
            .build();

    esRepository.save(doc); // upsert
}
```

키워드 필드는 화면 검색에 주로 사용되는 텍스트를 공백으로 결합하여 구성하였습니다. 인덱스 템플릿에서 형태소/부분검색 분석기를 적용해 두면, 이 보조 필드는 **검색 품질과 내성**을 높이는 데 도움이 되었습니다.

```
private String buildKeywords(ProductMaster p) {
    StringBuilder sb = new StringBuilder();
    append(sb, p.getProductName());
    append(sb, p.getBrand());
    append(sb, p.getStyle());
    append(sb, p.getColor());
    append(sb, p.getSize());
    return sb.toString().trim();
}
```

부분 업데이트: 화주명 변경 시 필요한 필드만 갱신

화주 정보가 변경된 경우, 해당 화주에 소속된 상품들만 조회하여 ES 문서의 `clientName`과 `lastEventAt`을 갱신하였습니다. 기존 값과 동일하면 저장을 생략하여 **불필요한 쓰기**를 줄였습니다.

```
@Override
public void updateClientFields(Long clientCode) {
    var productCodes = productRepo.findProductCodesByClientCode(clientCode);
    var client = clientRepo.findById(clientCode).orElse(null);
    String newClientName = client != null ? client.getClientName() : null;

    for (Long pc : productCodes) {
        var existing = esRepository.findById(pc).orElse(null);
        if (existing == null) continue;
```

```
        if (!Objects.equals(existing.getClientName(), newClientName)) {
            existing.setClientName(newClientName);
            existing.setLastEventAt(Instant.now());
            esRepository.save(existing);
        }
    }
}
```

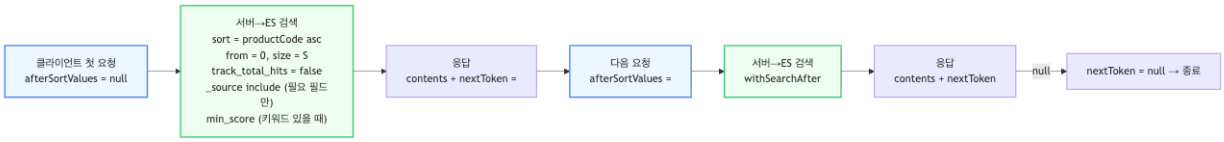
기본 검색: 다중 필드 가중치 + 경량 `_source` + 비용 절감 옵션을 적용

검색은 `multi_match`를 사용하여 `productName`, `brand`, `keywords`를 동시에 질의할 수 있게 하였고, 상품명과 브랜드에 상대 가중치를 부여하여 결과 순서를 조정하였습니다. 응답은 화면에 필요한 필드만 `_source include`로 제한하여 네트워크/역직렬화 비용을 줄였고, 목록 화면처럼 **전체 건수 집계가 중요하지 않은 경우**에는 `track_total_hits(false)` 를 사용해 불필요 카운트 비용을 줄였습니다.

```
@Override
public Page<ProductSearchResponse> search(Long clientCode, String keyword, int page, int size) {
    var bool = QueryBuilders.bool();
    if (clientCode != null) {
        bool.must(m -> m.term(t -> t.field("clientCode").value(clientCode)));
    }
    if (keyword != null && !keyword.isBlank()) {
        bool.must(m -> m.multiMatch(mm -> mm
                .query(keyword)
                .fields("productName^3","brand^2","keywords")));
    }

    String[] includes = {
            "productCode","clientCode","clientName",
            "productName","brand","style","color","size","retailPrice"
    };

    Query q = NativeQuery.builder()
            .withQuery(bool.build()._toQuery())
            .withPageable(PageRequest.of(Math.max(page, 0), Math.min(Math.max(size, 1), 100),
                    Sort.by(Sort.Order.asc("productCode"))))
            .withSourceFilter(new FetchSourceFilter(includes, null))
            .withTrackTotalHits(false)
            .apply(b -> {
                if (keyword != null && !keyword.isBlank()) b.withMinScore(0.01f);
            })
            .build();

    var hits = operations.search(q, ProductSearchDoc.class, index());
    var items = hits.getSearchHits().stream()
            .map(h -> ProductSearchResponse.builder()
```

```
                .productCode(h.getContent().getProductCode())
                .clientCode(h.getContent().getClientCode())
                .clientName(h.getContent().getClientName())
                .productName(h.getContent().getProductName())
                .brand(h.getContent().getBrand())
                .style(h.getContent().getStyle())
                .color(h.getContent().getColor())
                .size(h.getContent().getSize())
                .retailPrice(h.getContent().getRetailPrice())
                .build())
        .toList();

    return new PageImpl<>(items, q.getPageable(), hits.getTotalHits());
}
```

딥 페이지네이션(search_after) 구현 의도와 동작 설명



깊은 페이지로 내려갈수록 비용이 크게 증가하는 `from + size` 방식을 제거하고, 정렬 기준 이후로 계속 이어 탐색하는 `search_after`로 페이징을 전환했습니다. 초깊은 페이지에서도 일정한 지연 시간을 유지하기 위해 **정렬 키를 단일·안정 값**인 `productCode`로 고정했고, 초기에는 이 한 개의 키만으로도 순서를 보장하도록 단순화했습니다. 데이터 특성상 유일성이 완벽하지 않을 수 있다고 판단되면, 동일한 패턴으로 `createdAt` 또는 `_id`를 보조 정렬 키로 추가해 재현성을 높일 수 있도록 했습니다.

요청 단위 동작은 다음과 같습니다.

클라이언트는 첫 요청에서 `afterSortValues` 없이 호출하여 첫 페이지를 가져옵니다. 어댑터는 항상 `from=0`으로 검색하고, 마지막 히트의 `sortValues`를 `nextToken`에 담아 응답합니다. 프런트는 이 `nextToken`을 그대로 다음 요청의 `afterSortValues`로 전달하여 이전 페이지의 **마지막 정렬 지점 바로 다음 레코드**부터 이어서 읽습니다. 더 가져올 데이터가 없으면 `nextToken`은 `null`이 되고, 이 시점이 종료 조건이 됩니다. 응답 페이로드는 `contents`와 `nextToken` 두 필드만 제공해 단순화했습니다.

다음은 핵심 코드 입니다.

```
// ===== 딥 페이지네이션 (search_after) =====
public static class SearchPage<T> {
    private final List<T> contents;
    private final List<Object> nextToken;
    public SearchPage(List<T> contents, List<Object> nextToken) {
        this.contents = contents; this.nextToken = nextToken;
    }
    public List<T> getContents() { return contents; }
```

```
    public List<Object> getNextToken() { return nextToken; }
}

public SearchPage<ProductSearchResponse> searchAfter(
        Long clientCode, String keyword, int size, @Nullable List<Object> afterSortValues) {

    int s = Math.min(Math.max(size, 1), 100);

    // 정렬은 안정·유일한 키 하나로 시작합니다. 필요 시 보조 키를 추가
    Sort sort = Sort.by(Sort.Order.asc("productCode"));

    var bool = QueryBuilders.bool();
    if (clientCode != null) {
        bool.must(m -> m.term(t -> t.field("clientCode").value(clientCode)));
    }
    if (keyword != null && !keyword.isBlank()) {
        bool.must(m -> m.multiMatch(mm -> mm
                .query(keyword)
                .fields("productName^3","brand^2"))); // keywords는 매핑 확정 후 재도입
    }

    String[] includes = {
            "productCode","clientCode","clientName",
            "productName","brand","style","color","size","retailPrice"
    };

    var builder = NativeQuery.builder()
            .withQuery(bool.build()._toQuery())
            .withSort(sort)
            .withPageable(PageRequest.of(0, s))       // search_after 사용 시 from=0 고정
            .withTrackTotalHits(false)                // 목록에서 총계 계산 비용 제거
            .withSourceFilter(new FetchSourceFilter(includes, null));

    if (keyword != null && !keyword.isBlank()) {
        builder.withMinScore(0.01f);                  // 낮은 점수 노이즈 컷
    }

    //  버전에 따라 List 또는 Object[]
    if (afterSortValues != null && !afterSortValues.isEmpty()) {
        builder.withSearchAfter(afterSortValues);
    }

    Query q = builder.build();

    try {
        var hits = operations.search(q, ProductSearchDoc.class, index());

        var items = hits.getSearchHits().stream().map(h -> {
```

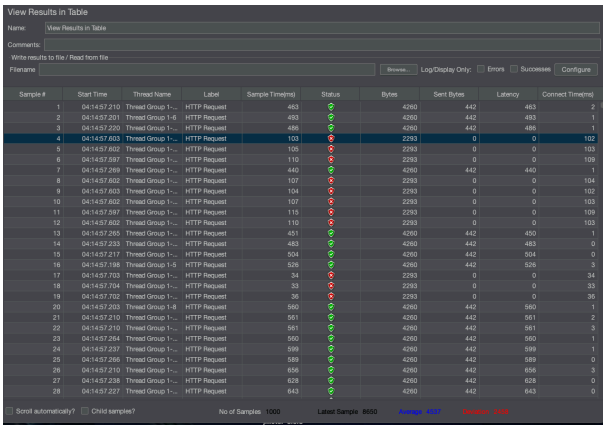
```
            var d = h.getContent();
            return ProductSearchResponse.builder()
                    .productCode(d.getProductCode())
                    .clientCode(d.getClientCode())
                    .clientName(d.getClientName())
                    .productName(d.getProductName())
                    .brand(d.getBrand())
                    .style(d.getStyle())
                    .color(d.getColor())
                    .size(d.getSize())
                    .retailPrice(d.getRetailPrice())
                    .build();
        }).collect(java.util.stream.Collectors.toList());

        List<Object> nextToken = null;
        var lastHit = hits.getSearchHits().isEmpty() ? null
                : hits.getSearchHits().get(hits.getSearchHits().size() - 1);
        if (lastHit != null && lastHit.getSortValues() != null && !lastHit.getSortValues().isEmpty()) {
            nextToken = lastHit.getSortValues();
        }

        return new SearchPage<>(items, nextToken);

    } catch (org.springframework.dao.DataAccessException e) {
        Throwable root = e.getCause();
        if (root instanceof co.elastic.clients.elasticsearch._types.ElasticsearchException ee) {
            log.error("[ES] type={}, reason={}", ee.error().type(), ee.error().reason(), ee);
        } else {
            log.error("[ES] search failed", e);
        }
        throw e;
    }
}
```

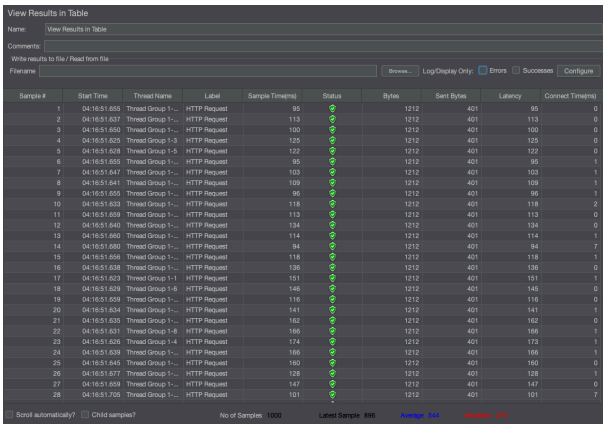
로깅과 관측성: MDC로 요청 단위 디버그를 기록

색인과 검색 전 과정에서 `MDCHelper.appendDebug`와 클래스 로거를 통해 **요청별 디버그 스트림**을 남기도록 하였습니다. “시작/종료, 경과 시간, 스킵 사유, 반환 건수” 같은 핵심 지표를 한 요청 컨텍스트에 종속시켜, Kibana 등에서 **한 번에 추적**할 수 있게 하였습니다. 이는 성능 회귀와 장애 구간을 찾는 데 유효하였습니다.

```
private void debug(String msg) {
    MDCHelper.appendDebug(ElasticsearchProductSearchAdapter.class, msg);
    log.info("[ES-ADAPTER] {}", msg);
}
```

마무리 — JMeter 결과로 본 효과





동일한 조건에서 **1,000건 샘플**을 연속 요청하여 비교하였습니다(동일한 스레드/러너 구성, 동일 API 조건). 아래 두 캡처는 각각 **RDB 조인 기반 조회**(첫 번째 이미지)와 **Elasticsearch 기반 조회**(두 번째 이미지)의 `View Results in Table` 결과를 보여줍니다.

요약

항목	RDB 조인 기반	Elasticsearch 기반
Average (ms)	**4,537**	**544**
Latest Sample (ms)	**8,650**	**896**
Deviation (표준편차, ms)	**2,468**	**219**
Status	**실패 다수 포함**(빨간표시)	**전량 성공**(녹색표시)
Bytes(대략, 단건)	4,260 / 2,293 혼재	1,212 고정

- **평균 응답 시간** 기준으로 약 **8.3배**(4,537 → 544ms) 개선
- **최대 지연(Latest Sample)** 기준으로 약 **9.6배**(8,650 → 896ms) 완화
- **분산(Deviation)** 이 2,468ms → 219ms로 크게 줄어 **지연 변동성**이 눈에 띄게 안정화
- RDB 조인 시 **실패 샘플이 다수** 관측되었고, ES 전환 후 **전량 성공**으로 확인
- 응답 **Bytes**가 ES에서 더 작게 관측되었는데, 이는 `_source include`로 **필요 필드만 반환**하도록 하여 **전송·역직렬화 비용**을 줄였기 때문입니다.

마치며...

이번 전환은 단순한 성능 개선이 아니라, **도메인에 맞는 아키텍처적 선택**이었습니다.

RDB 단일 경로로는 버거웠던 복잡 조회와 불안정한 지연 문제를, **CQRS + Elasticsearch**로 구조적으로 해소할 수 있었습니다.

- **읽기/쓰기 분리**: 트랜잭션과 검색 워크로드를 분리해야 예측 가능한 성능을 확보
- **검색 전용 문서 설계**: 단순 복제가 아니라, 실제 조회 요구에 맞춘 인덱스 모델

결과적으로 평균 응답 속도는 **8배 이상 개선**, 지연 변동성은 크게 줄었고, 실패 요청도 사라졌습니다.

이번 경험은 "데이터 정합성과 검색 성능을 동시에 만족시키는 방법"을 검증한 사례였으며, 앞으로도 **예측 가능한 운영 기반**을 마련했다는 점이 가장 큰 수확이었습니다.