Open Source Summit Seoul 2026

From Clicks to Rankings: Building a Learning-to-Rank Pipeline with OpenSearch

AI Specialist SA, West Kim
Amazon Web Services

OpenSearch

Agenda

- OpenSearch
- 검색 이란?
- User Behavior Insight
- Search Relevance Workbench
- Learning to Rank

OpenSearch

OpenSearch

OpenSearch는 Community-driven의 Open Source 검색, 분석 및 벡터 데이터베이스 플랫폼으로 통합 가시성, 보안, 시각화 및 AI 기반 애플리케이션을 위한 도구가 통합되어 있습니다.

로드맵:

OpenSearch SOFTWARE FOUNDATION

THE LINUX FOUNDATION

aws Uber SAP IBM

aiven Aryn ATLASSIAN Canonical DataStax

DigitalOcean DTEX graylog NetApp Instaclustr

reSolve SIDECAR_portal26 CERN ByteDance

OpenSearch

정보 검색?

IR(Information Retrieval)

정보(Information) 검색(Retrieval) 이란?

커머스의 경우

43%

리테일 커머스 서비스 사용자의 43% 는 **가장 먼저 검색 기능**을 통해 상품 검색을 수행합니다.

21%

첫 검색 후 **20%**는 검색어를 수정 했으며, **21%는 웹 사이트를 이탈 했습니다.**

39%

상품 구매자의 **39%**는 **검색어 연관성**의 영향을 받았습니다.

61%

전체 이커머스의 **61% 가 사용자 기대 수준 이하의 검색 성능**을 제공합니다.

출처 : https://www.algolia.com/blog/ecommerce/e-commerce-search-and-kpis-statistics

OpenSearch

성공 적인 검색 시스템의 주요 목적

- 찾는 정보에 대한 **“검색이 실패 하지 않는것”**
- 검색자의 **“기대를 충족시킬 수 있는 검색 결과 노출”**
- 검색을 통한 정보를 검색자가 **“활용 하는것”**(예: 구매 등)
- 검색자가 미처 생각하지 못했던 **“새로운 정보 제시”**

사용자의 검색 의도는 무엇일까?

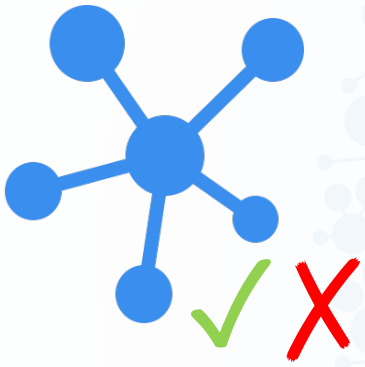
명확한 검색화 모호한 검색 사이의 의도 파악

정답형 검색자

명확한 답을 알고 해당 정보를
타겟팅 해서 검색

페르소나 예시

- 모델명 검색
- 정확한 상품명 검색
- 논문 제목
- 강력한 필터 조건



정보 검색(Information Retrieval)이
정확했다는 판단 근거는 무엇으로 할까?

탐색형 검색자

정답을 모르는 상태에서 멀티턴
검색을 통해 정보 추론

페르소나 예시

- 키워드 기반 검색
- 자연어 질의
- 정보 내의 특정 구문
- 추상적 표현

OpenSearch

검색 시스템이 찾은 정보가 정확한가?

RELEVANCE

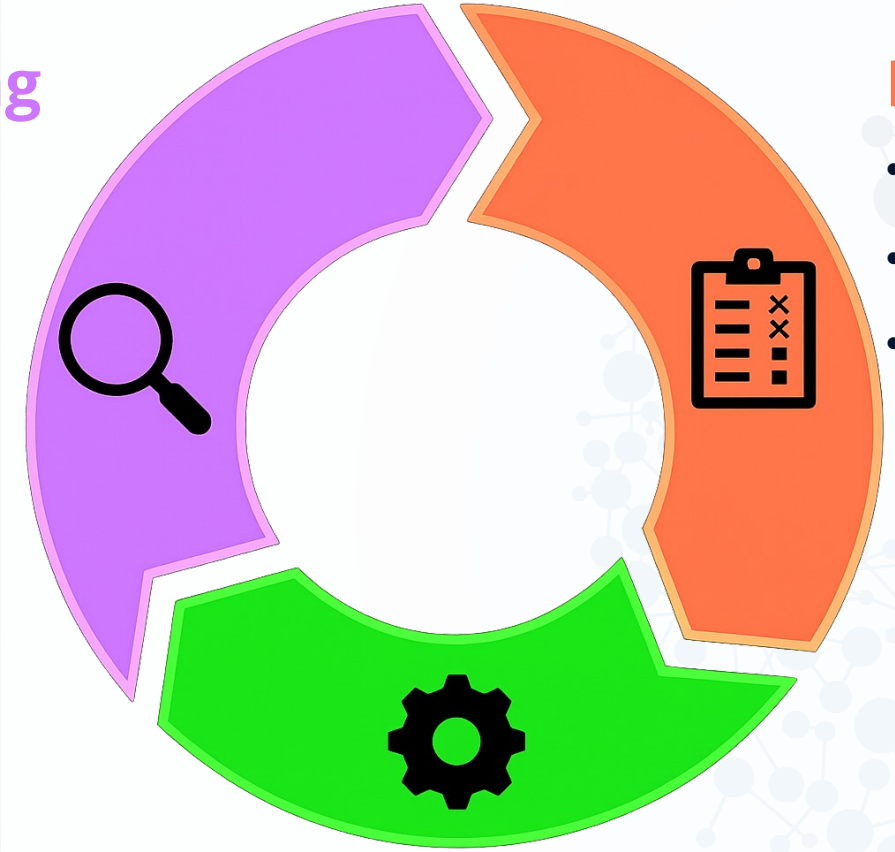
더 나은 검색 경험을 제공하기 위한 검색 핵심 지표

- **Precision@k -** 상위 k개 결과 중 관련 있는 문서의 비율
- **Recall@k -** 전체 관련 문서 중 상위 k개에 포함된 비율
- **MAP –** 평균 Precision의 평균
- **nDGC@k –** 순서와 관련성 정도를 반영한 정규화
- **MRR –** 첫 번째 문서 관련성 순위의 역수 평균
- **Coverage –** 상위 k개의 결과중 판정이 존재하는 문서 비율

검색 향상을 위한 과정

Search Processing

- Lexical Search
- Semantic Search
- Hybrid Search



Measurement & Analysis

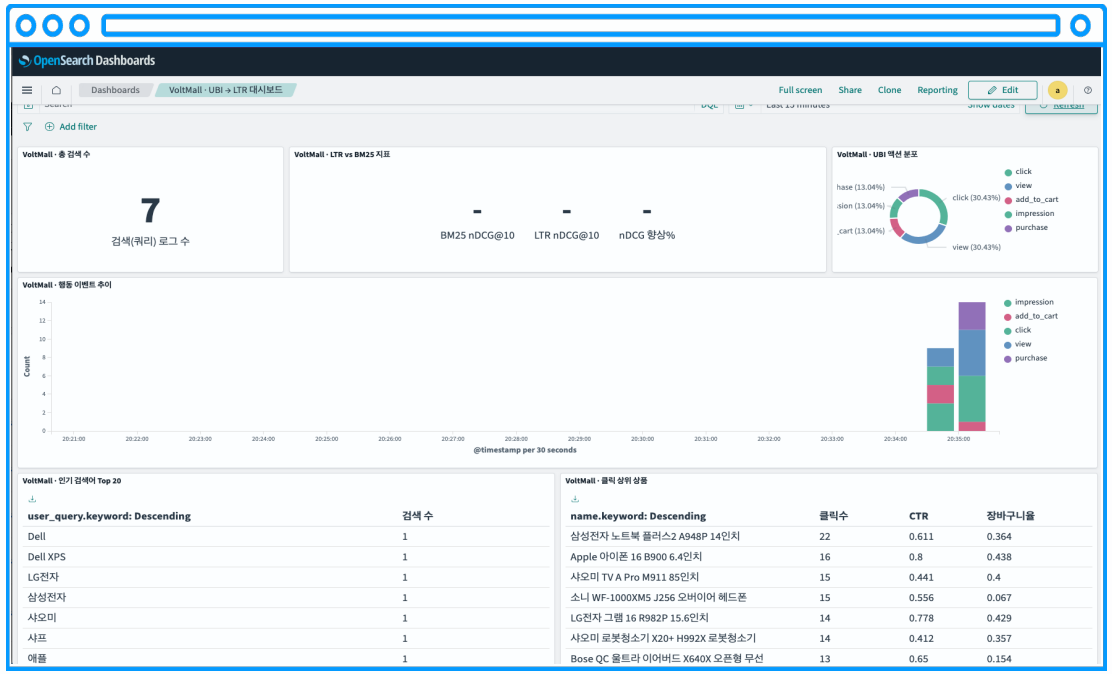
- Behavioral data
- Offline/Online Evaluation
- A/B or Regression Testing

Search Tuning

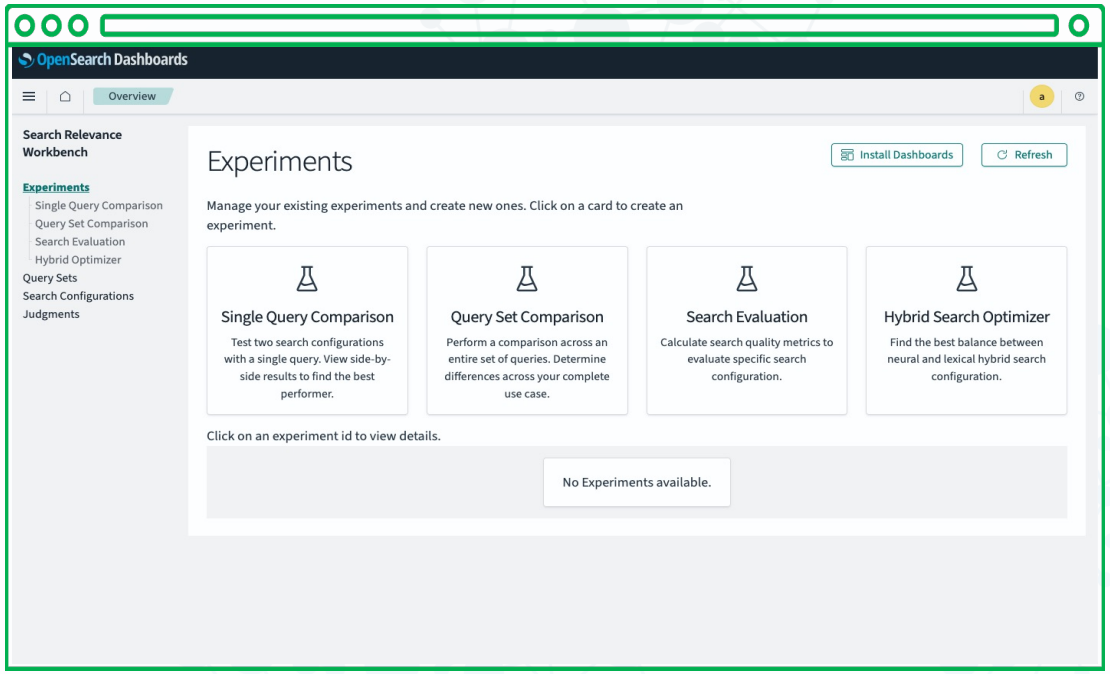
- Manual Tuning
- Learning To Rank
- Semantic Model Fine-Tuning

OpenSearch

방법과 도구



User Behavior Insights



Search Relevance Workbench

OpenSearch

OpenSearch

User Behavior Insights

User Behavior Insight – 사용자 행동 분석

- 사용자의 **검색 쿼리, 검색 결과, 그리고 결과에 대한 사용자의 액션을 체계적으로 수집하고 분석**하여 검색 품질을 향상시키는 표준 스키마
- 검색 행동은 사용자가 제출한 쿼리, 사용자에게 제시된 결과, 그리고 해당 결과에 대해 사용자가 취한 액션으로 구성
- UBI 스키마는 모든 사용자 상호작용(이벤트)을 수행된 검색 결과아 연결

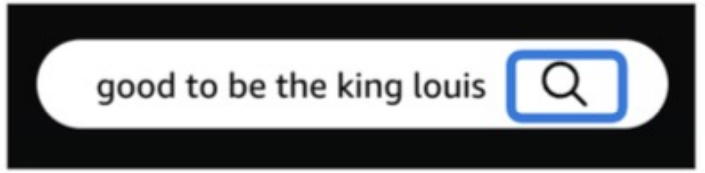
UBI의 핵심 구성 요소

- **표준 스키마:** 기계 판독 가능한 UBI 명세로 데이터 상호운용성을 보장
- 클라이언트 라이브러리 (ubi.js): 웹 페이지에서 사용자 검색과 이벤트를 캡처하는 JavaScript 라이브러리(선택 사항)
- OpenSearch 플러그인(Optional): 쿼리 데이터 기록을 자동화하는 서버 측 플러그인으로, 클라이언트 – 서버 왕복 없이 쿼리와 결과를 직접 UBI 엔드포인트에 전송하여 최적화

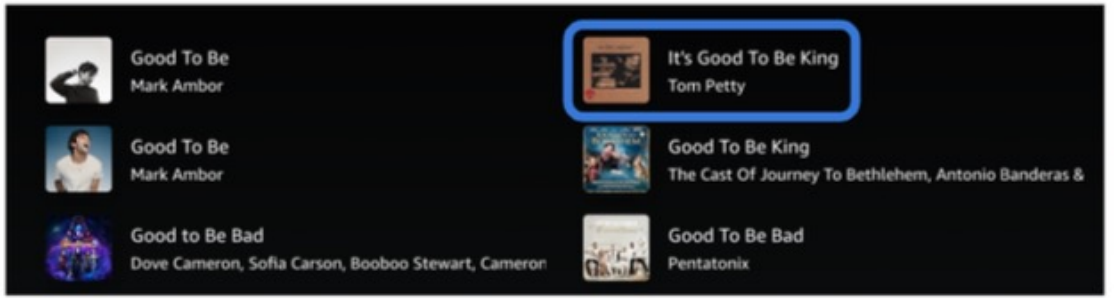
UBI 인덱스 스키마

- Query Index : 모든 검색 및 검색 결과를 저장
- Event Index : 검색어(query)와 관련된 모든 사용자의 행동을 저장

Search 14:23 SEARCH_CLICK
SearchID: 539 UserID:9696
Query: good to be the king louis



Results 14:24 RESULT_CLICK
SearchID: 539 UserID:9696
ObjectID: 9393
Position: 4



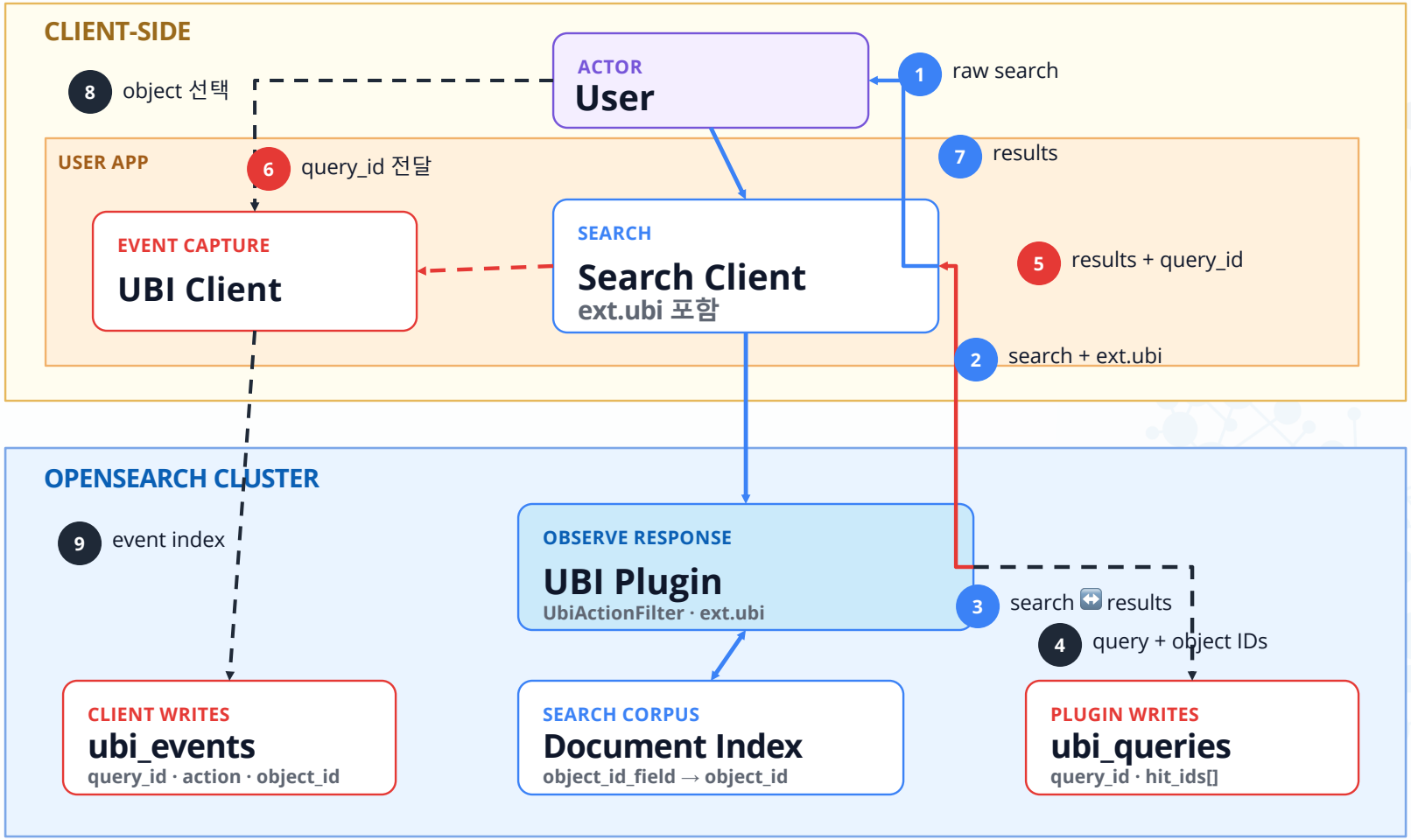
Conversion 14:25 PLAY_CLICK
SearchID: 539 UserID:9696
ObjectID: 9393

OpenSearch

핵심 식별자 – Key Identifier

- **object_id:** 검색 결과로 반환된 객체의 ID (예: ISBN, SKU)
- **query_id:** 쿼리의 고유 ID로, 모든 후속 이벤트를 해당 쿼리와 연결
- **client_id:** 사용자의 검색 히스토리를 추적하는 클라이언트 식별자
- **object_id_field:** 인덱스에서 object_id를 제공하는 필드 이름(예: isbn_code, sku)
- **action_name:** 사용자가 수행한 구체적인 액션(예: click, view, purchase)

UBI 역할 및 흐름



FLOW LEGEND

- 표준 검색 흐름
- UBI 수집·저장 흐름
- query_id 전달

핵심 역할

01 Search Client
검색과 결과 수신

02 UBI Plugin
query + 결과 ID 저장

03 UBI Client
클릭·전환 이벤트 저장

OpenSearch

ubi_queries와 ubi_events간 결합

PLUGIN AUTO-WRITES

ubi_queries

검색 요청과 노출 결과

timestamp	검색 수신 시각
query_id	검색 실행 ID
query_response_id	응답 실행 ID
query_response_hit_ids[]	노출 object_id 목록
client_id, application	사용자, 애플리케이션
user_query, query_attributes	질의와 추가 문맥

query_id

CORRELATE

JOIN

query_id

hit_ids[]

object_id

CLIENT WRITES

ubi_events

사용자의 후속 행동

timestamp	행동 발생 시각
query_id	검색 실행 ID
action_name	click, cart, purchase
event_attributes.position	결과 위치·좌표
event_attributes.object.object_id	행동 대상 object_id
object_id_field, client_id	ID 종류와 사용자

이 결합으로 답하는 질문

01 어떤 검색이 실행됐나?
query_id · user_query

02 무엇이 노출됐나?
query_response_hit_ids[]

03 무엇을 선택했나?
action_name · object_id

04 어디에서 선택했나?
position · client_id

분석 결과 어떤 쿼리가 어떤 객체를 노출했고, 사용자가 어떤 행동을 했는지 재구성

주의: ubi_events의 event_attributes는 동적 매핑이므로 사용자 정의 필드 증가를 관리해야 한다.

OpenSearch

검색은 6단계를 지나 ubi_queries 인덱스에 기록

검색 과정

REQUEST
① UBI 파싱
ext.ubi를 UbiParameters로 변환

FILTER
② 필터 진입
검색 요청만 식별하고 chain.proceed 호출

SEARCH
③ 실제 검색
OpenSearch가 점수, 정렬, hits를 계산

OBSERVE
④ 응답 스누핑
onResponse에서 완료된 SearchResponse를 관찰

BUILD
⑤ 레코드 조립
query_id, response_id, hit IDs를 묶음

PERSIST
⑥ 저장
기본 경로: ubi_queries에 비동기 색인

수집 경로

사전 1회 POST /_plugins/ubi/initialize → ubi_queries , ubi_events 생성

클래스, 주요 함수 호출 순서

1. **UbiParametersExtBuilder**
   parse(parser)
2. **UbiActionFilter**
   apply(..., chain)
3. **ActionFilterChain**
   proceed(...)
4. **ActionListener**
   onResponse(response)
5. **UbiActionFilter**
   handleSearchRequest(...)
6. **UbiActionFilter**
   indexQuery(queryRequest)
7. **UbiSearchResponse**
   toXContent(...)

OpenSearch

onResponse: 완료된 hits 스누핑

요청에서

REQUEST METADATA

UbiParameters

query_id
user_query
client_id, application
object_id_field, attributes

응답에서

RESULT OBSERVATION

SearchResponse.hits

각 SearchHit에서 결과 ID 추출

field 지정: _source[field]
미지정: hit.docId()

새로 생성

RESPONSE ID

UUID

query_response_id

동일 query의 각 응답을 구분

QueryResponse

query_id, response_id, hit_ids[]

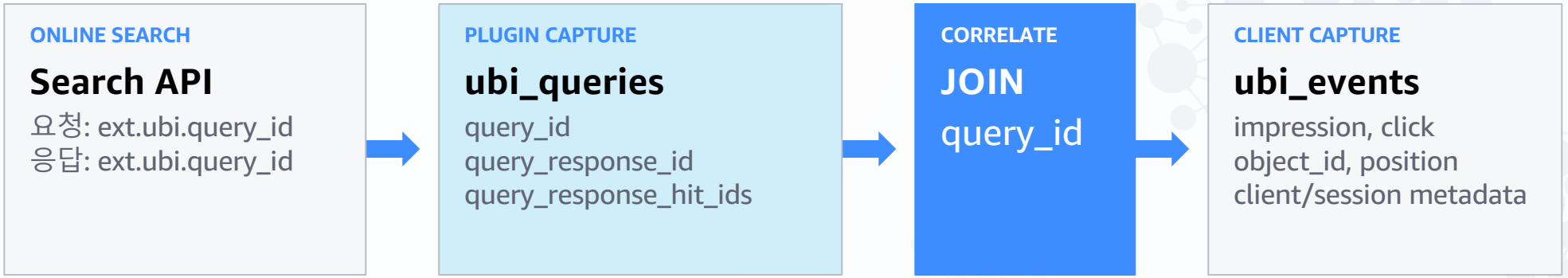
QueryRequest 요청 메타데이터 + raw query + QueryResponse

클래스, 주요 함수 호출 순서

1. **ActionListener**
   onResponse(response)
2. **UbiActionFilter**
   handleSearchRequest(...)
3. **UbiParameters**
   getQueryId()
4. **SearchResponse**
   getHits()
5. **SearchHit**
   getSourceAsMap() / docId()
6. **QueryResponse**
   new(...hitIds)
7. **QueryRequest**
   new(...queryResponse)

OpenSearch

query_id 를 통한 Query + Event 연결



플러그인이 자동으로 저장

클라이언트/미들웨어가 별도 저장

UBI의 결과

어떤 쿼리가 어떤 결과를 노출했고, 사용자가 무엇을 클릭했는지 분석 가능한 데이터셋

검색 순위를 바꾸는 기능이 아니라, 개선에 필요한 관찰 데이터를 만드는 계층

OpenSearch


OpenSearch
Search
Relevance
Workbench
bitly

Search Relevance Workbench

검색 결과 평가를 위한 도구

사용자 데이터와 쿼리에 맞춰 관련성을
평가하고 UBI로 수집된 데이터와 비교 분석

검색 결과에 대한 투명하고 공정한 평가

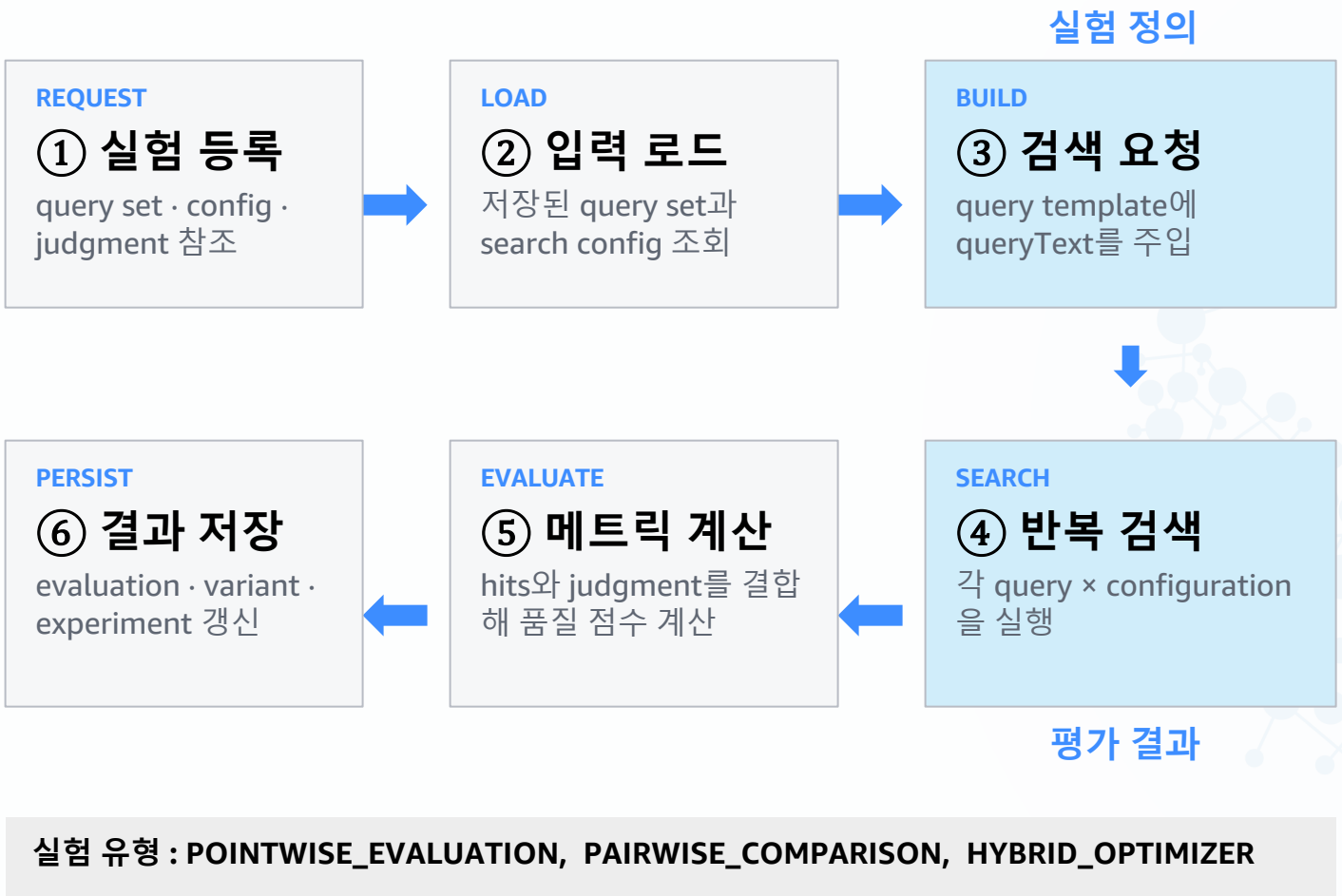
사용자 정의 가능한 가중치를 통하여
데이터와 함께 공정한 평가 수행

더 나은 검색 경험을 위하여

평가 경과에 따른 가중치 조정 및 쿼리 튜닝
루프를 통해 지속적인 개선

OpenSearch

실험별 평가 결과 기록



실험 유형 : POINTWISE_EVALUATION, PAIRWISE_COMPARISON, HYBRID_OPTIMIZER

클래스, 주요 함수 호출 순서

1. **RestPutExperimentAction**
   prepareRequest(...)
2. **PutExperimentTransportAction**
   doExecute(...)
3. **ExperimentDao**
   putExperiment(PROCESSING)
4. **ExperimentRunningManager**
   startExperimentRun(...)
5. **SearchRequestBuilder**
   buildSearchRequest(...)
6. **ExperimentTaskManager**
   executeVariantTask(...)
7. **SearchResponseProcessor**
   processSearchResponse(...)

OpenSearch

실험(experiment) 3가지 차이

정량적인 3가지 실험 타입

PAIRWISE_COMPARISON
두가지 결과를 비교 평가

지표
- Jaccard
- Frequency-weighted
- RBO50 / RBO90

사용 사례
- 두 결과 비교 차이점 평가
- A/B 테스트

POINTWISE_EVALUATION
Judgment에 대한 절대 평가

지표
- Coverage@k
- Precision@k
- MAP@k
- NDCG@k

사용 사례
- 실제 검색시 정량적 평가

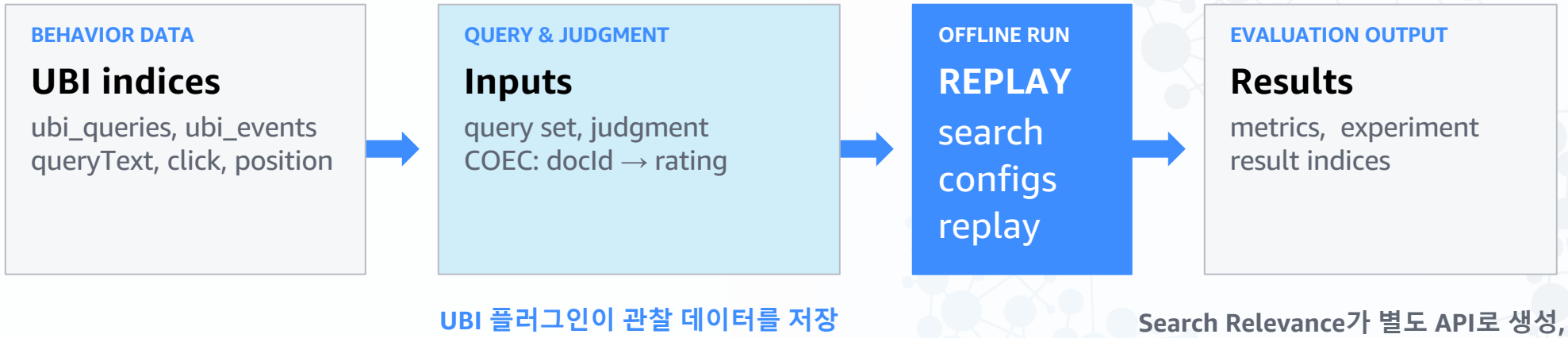
HYBRID_OPTIMIZER
Hybrid가중치 / 정규화된 무차별 대입

검색 대상
- Lexical / Semantic
- Normalization (min-max/L2)
- NDCG@k

사용 사례
- Hybrid 가중치 탐색 자동화
- 최적 구성을 자동 추출

OpenSearch

UBI 데이터 기반 평가



Search Relevance의 결과 동일한 query set을 여러 검색 구성으로 반복해 어떤 구성의 품질이 더 좋은지 비교

OpenSearch

Judgment List 생성 방식

Judgment 요청

```
{
  "type": "UBI_JUDGMENT",
  "clickModel": "coec",
  "maxRank": 20
}
or
{ "type": "LLM_JUDGMENT", ... }
```

자동 판정 방법
- **UBI/COEC**
- **LLM as a Judge**

PutJudgmentTransportAction

PROCESSING 먼저 저장

Judgment(id, type, metadata)
ratings 없음

JudgmentsProcessorFactory.getProcessor()

유형별 processor 선택

UBI → COEC
LLM as a Judge → 모델 판정

JudgmentDao.updateJudgment()

최종 Judgment List 저장

ratings 저장, COMPLETED / ERROR

클래스, 주요 함수 호출 순서

1. **RestPutJudgmentAction**
   prepareRequest(...)
2. **PutJudgmentTransportAction**
   doExecute(...)
3. **JudgmentDao**
   putJudgement(PROCESSING)
4. **PutJudgmentTransportAction**
   triggerAsyncProcessing(...)
5. **JudgmentsProcessorFactory**
   getProcessor(type)
6. **BaseJudgmentsProcessor**
   generateJudgmentRating(...)
7. **JudgmentDao**
   updateJudgment(COMPLETED)

OpenSearch

UBI/COEC – 위치 편향 보정후 클릭 Rating 변환

COEC 계산

POSITION CTR
위치별 기대 CTR
CTR(p) = clicks(p)
÷ impressions(p)

OBSERVED
문서 관측값
clicks(q,d)
impressions(q,d)
lowest rank = p

보정

COEC
최종 rating
clicks(q,d)
÷ [CTR(p) ×
impressions(q,d)]

1 초과 = 해당 위치의 기대보다 클릭이 많음

같은 쿼리, 문서 계산 예시

QUERY	gaming laptop
DOC A	rank 1, clicks(20) / impressions(100), CTR(1)=0.25
DOC B	rank 4, clicks(12) / impressions(100), CTR(4)=0.10
RATING A	20 ÷ (0.25 × 100) = 0.800
RATING B	12 ÷ (0.10 × 100) = 1.200
OUTPUT	query → [{A, 0.800}, {B, 1.200}]
MEANING	낮은 위치의 기대 클릭률을 기준으로 비교

클래스, 주요 함수 호출 순서

1. **UbiJudgmentsProcessor**
   generateJudgmentRating(...)
2. **CoecClickModel**
   calculateJudgments(...)
3. **CoecClickModel**
   getRankAggregated(...)
4. **CoecClickModel**
   getClickthroughRate(...)
5. **CoecClickModel**
   calculateCoecJudgments(...)
6. **UbiJudgmentsProcessor**
   format query, docId, rating
7. **ActionListener**
   onResponse(formattedRatings)

OpenSearch

LLM을 통한 문서 연관성 판정

LLM 요청

```
{
  "type": "LLM_JUDGMENT",
  "modelId": "model-01",
  "querySetId": "qs-01",
  "searchConfigurationList": ["sc-01"],
  "size": 10,
  "contextFields": ["title","body"]
}
```

modelId는 ML Commons 등록 모델 ID 등록
원격 모델은 connector를 연결됨

중요: LLM은 먼저 검색된 상위 문서만 판정함

LlmJudgmentsProcessor

후보 문서 검색

query set × search configs
상위 size개 hits 병합

getContextSource()

모델 Prompt 생성

query + 선택 필드 + prompt

MLAccessor.predict()

추론, rating 변환

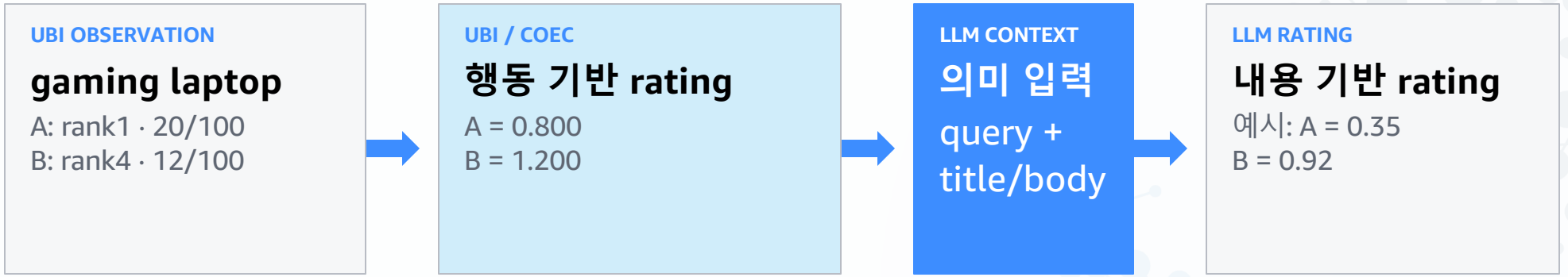
JSON → 0~1 / binary rating

클래스, 주요 함수 호출 순서

1. **LlmJudgmentsProcessor**
   generateJudgmentRating(...)
2. **QuerySetDao**
   getQuerySetSync(...)
3. **LlmJudgmentsProcessor**
   processQueryTextAsync(...)
4. **SearchRequestBuilder**
   buildSearchRequest(...)
5. **Client**
   search(request, listener)
6. **MLAccessor**
   predict(modelId, ...)
7. **RatingOutputProcessor**
   sanitize · convertRatingScore

OpenSearch

각 판단 방식에 따른 평가 기준 차이



방식 선택 기준

01 트래픽 충분 → UBI

02 신규,롱테일 → LLM

03 비용 최소화 → UBI

04 편향 점검 → 둘 다

05 외부 label → IMPORT

판단 기준

UBI는 '실제로 선택했는가', LLM은 '내용이 query와 맞는가'를 평가

결과가 다르면 노출 편향, 후보 recall, prompt/context를 함께 점검

OpenSearch


OpenSearch
Learning
to
Rank
bitly

Learing to Rank(LTR) : GBM 모델을 통한 리랭킹

2 단계 리랭킹 파이프라인

1단계 : 검색(Retrieve)

BM25 / Hybrid로 top-200건 조회

2 단계 : LTR 리랭킹

GBM 모델로 특성 계산 -> 예측 점수로 리랭킹

top-10 반환

행동 데이터 / 인기도 반영
NDCG / Precision@10 향상

LTR 플러그인 3가지 구성요소

Feature(특성)

Mustache 템플릿으로 특성 정의
- BM25 (title / description)
- doc-only (popularity)
- neural / knn score
- Behavioral (UBI의 CTR)

모델 학습

OpenSearch 외부에서 학습 수행
- XGBoost (GBM 트리)
- RankLib (MART / LambdaMART)
- training = judgement + clickstream
- sample : hello-ltr

추론 (OpenSearch)

모델 업로드
- sltr 쿼리를 통해 rescore 호출
- window_size: 200 전후 recommand (1000 추가시 성능 저하 발생)

OpenSearch

LTR을 통한 학습 -> 리랭킹 파이프라인

핵심 경계 플러그인은 모델을 학습하지 않고 저장·실행한다

클래스. 주요 함수 호출 순서

1. **ExtractUbiData**
   handler(...)
2. **BuildFeatures**
   log_features(...)
3. **LoggingFetchSubPhase**
   process(hitContext)
4. **TrainModel**
   xgb.train(...)
5. **IndexFeatureStore**
   loadModel(...)
6. **StoredLtrQueryBuilder**
   doToQuery(...)
7. **RankerQuery**
   score()

OpenSearch

ltr_log를 사용해 hit별 응답 로그 생성

피처 로깅 요청

```
{
 "query":{"sltr":{
  "_name":"logged",
  "featureset":"ecom_features",
  "params":{"keywords":"이어폰"}
 }},
 "ext":{"ltr_log":{"log_specs":{
  "named_query":"logged",
  "missing_as_zero":true
 }}}
}
```

학습 시 계산한 피처 정의를 서빙 시에도 재사용해 train/serve skew를 절감

중요: feature logging은 벡터를 반환, 모델 학습 자체는 클러스터 외부에서 별도로 수행됨

`LoggingSearchExtBuilder.parse()`

로그 사양 파싱

named_query, missing_as_zero를 LogSpec으로 변환

`StoredLtrQueryBuilder.doToQuery()`

피처셋 로드

ecom_features + keywords로 RankerQuery 생성

`LoggingFetchSubPhase.process()`

각 hit의 피처 점수 수집

fields._ltrlog[].log_entry에 name/value 기록

클래스, 주요 함수 호출 순서

1. **LoggingSearchExtBuilder**
   parse(...)
2. **StoredLtrQueryBuilder**
   fromXContent(...)
3. **IndexFeatureStore**
   loadSet(...)
4. **RankerQuery**
   buildLogQuery(...)
5. **LoggingFetchSubPhase**
   getProcessor(...)
6. **LoggingFetchSubPhase**
   process(hitContext)
7. **HitLogConsumer**
   nextDoc(hit)

OpenSearch

GBM 모델을 사용한 학습

RankLib 행

```
3 qid:17 1:8.42 2:1.00 3:0.04 4:4.70 # p-100 이어폰
```

label query group feature:value ... document

각 query의 문서를 한 그룹으로 묶어 순서를 학습

RankNet의 pair loss → LambdaRank의 ΔNDCG gradient → MART tree ensemble

BuildFeatures

학습 파일 생성

judgment + _ltrlog 벡터 → training.ranklib.txt

TrainModel

query 그룹 구성

DMatrix.set_group(sizes)로 같은 query를 묶음

xgb.train

LambdaMART 학습

rank:ndcg, ΔNDCG pair, 100회

클래스, 주요 함수 호출 순서

1. **BuildFeatures**
   handler(...)
2. **TrainModel**
   parse_ranklib(...)
3. **TrainModel**
   group_rows(...)
4. **XGBoost.DMatrix**
   set_group(sizes)
5. **XGBoost**
   train(rank:ndcg)
6. **Booster**
   get_dump(json)
7. **TrainModel**
   upload model

OpenSearch

모델 업로드 및 추론

외부 모델 업로드

CreateModel API

feature set + model

POST _createmodel

→

STORE

Index store

.ltrstore*

feature / set / model

→

SEARCH

sltr rescore

BM25 후보 →
window_size
query_weight + model
score

저장된 모델을 로드,컴파일한 뒤 요청마다 상위 window의 문서만 재정렬

온라인 점수화에 필요한 요소

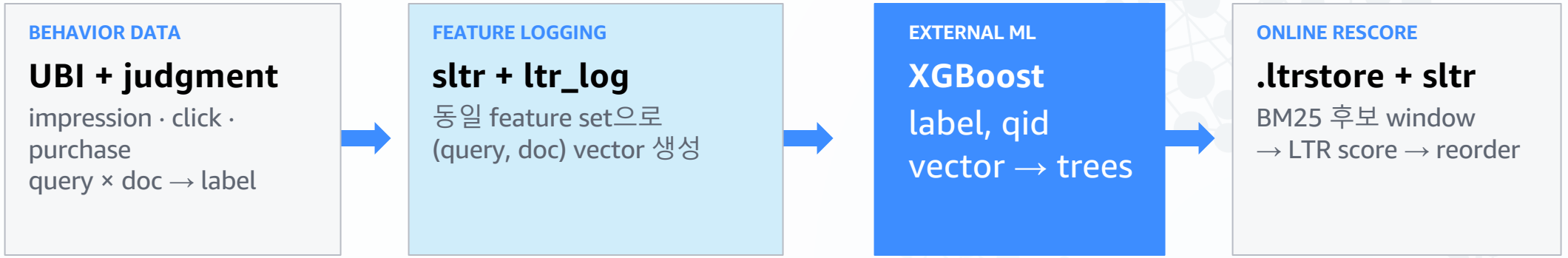
feature_set	BM25, 가격 , 평점, CTR 등 정의
model.type	model/xgboost+json, model/ranklib, model/linear
model.definition	트리 구조 또는 피처 가중치
params	keywords 등 query-time 값
window_size	BM25가 뽑은 상위 후보 범위
RankerQuery.score()	피처값을 벡터에 채워 ranker.score(fv)
final score	query_weight×BM25 + rescore_weight×LTR

클래스, 주요 함수 호출 순서

1. **CreateModel API**
   store model
2. **IndexFeatureStore**
   loadModel(...)
3. **StoredLtrModel**
   compile(...)
4. **StoredLtrQueryBuilder**
   doToQuery(...)
5. **RankerQuery**
   build(...)
6. **RankerScorer**
   score()
7. **LtrRanker**
   score(fv)

OpenSearch

전체 파이프라인 recap



OpenSearch가 피처 벡터를 계산

외부 학습 모델을 클러스터 내 LTR플러그인이 실행

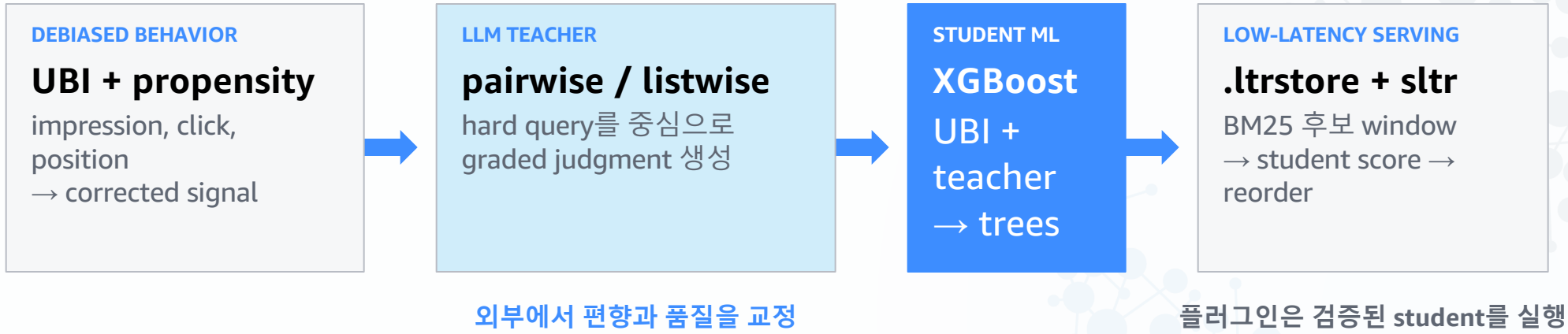
LTR의 결과 **행동, 판정과 여러 피처의 관계를 학습한 모델이 상위 후보의 순서를 다시 계산**

클릭 편향 보정, LLM judgment는 외부 학습에서, .ltrstore와 sltr는 클러스터 내부에서 수행됨

OpenSearch

RECOMMENDED EVOLUTION

LTR + LLM 판정 조합을 통한 평가



실행 우선순위

01 position 기록

02 propensity 검증

03 LLM hard-set 생성

04 student offline 평가

05 A/B 후 승격

권장 구조 **UBI 행동 + 편향 보정 → LLM teacher → LambdaMART student → sltr 서빙**

비용은 오프라인에, 지연은 student에 맡기고 nDCG, CTR, 전환을 A/B 테스트를 통해 검증

OpenSearch


OpenSearch
SUMMARY

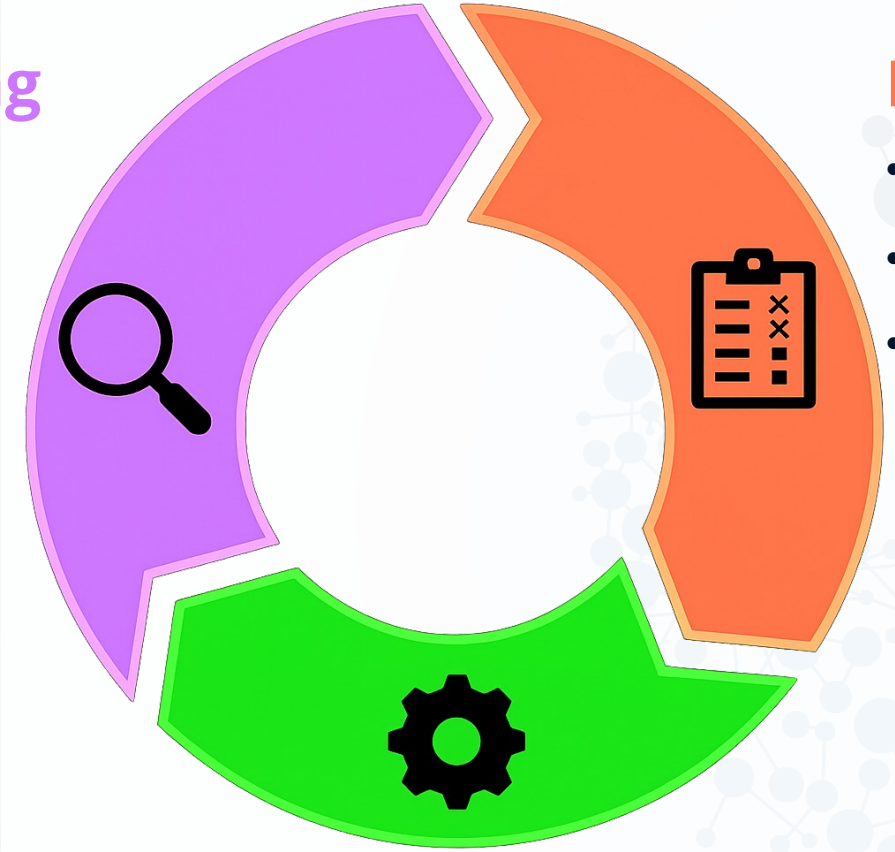
검색 서비스 운영 모델

1. 검색 서비스 지표 정의
2. 검색 질의 + 결과 수집
3. 결과에 따른 사용자의 행위 수집
4. 수집 결과를 통한 정량적 지표 분석
5. 개선 계획(LTR? Cross-encoder rerank model?)
6. 개선 반영 및 피드백

Loop

Search Processing

- Lexical Search
- Semantic Search
- Hybrid Search



Measurement & Analysis

- Behavioral data
- Offline/Online Evaluation
- A/B or Regression Testing

Search Tuning

- Manual Tuning
- Learning To Rank
- Semantic Model Fine-Tuning

OpenSearch

참고 자료

- Search relevance
  - https://opensearch.org/platform/search-relevance/
- User Behavior Insights
  - https://docs.opensearch.org/latest/search-plugins/ubi/index
- Search Relevance Workbench
  - https://docs.opensearch.org/latest/search-plugins/search-relevance/using-search-relevance-workbench
- Learn to Rank
  - https://docs.opensearch.org/latest/search-plugins/ltr/index
- OpenSearch Project (resources)
  - https://solo.to/opensearchproject
- XGBoost
  - https://xgboost.ai/

Thank you!

OpenSearch