Google Cloud Search

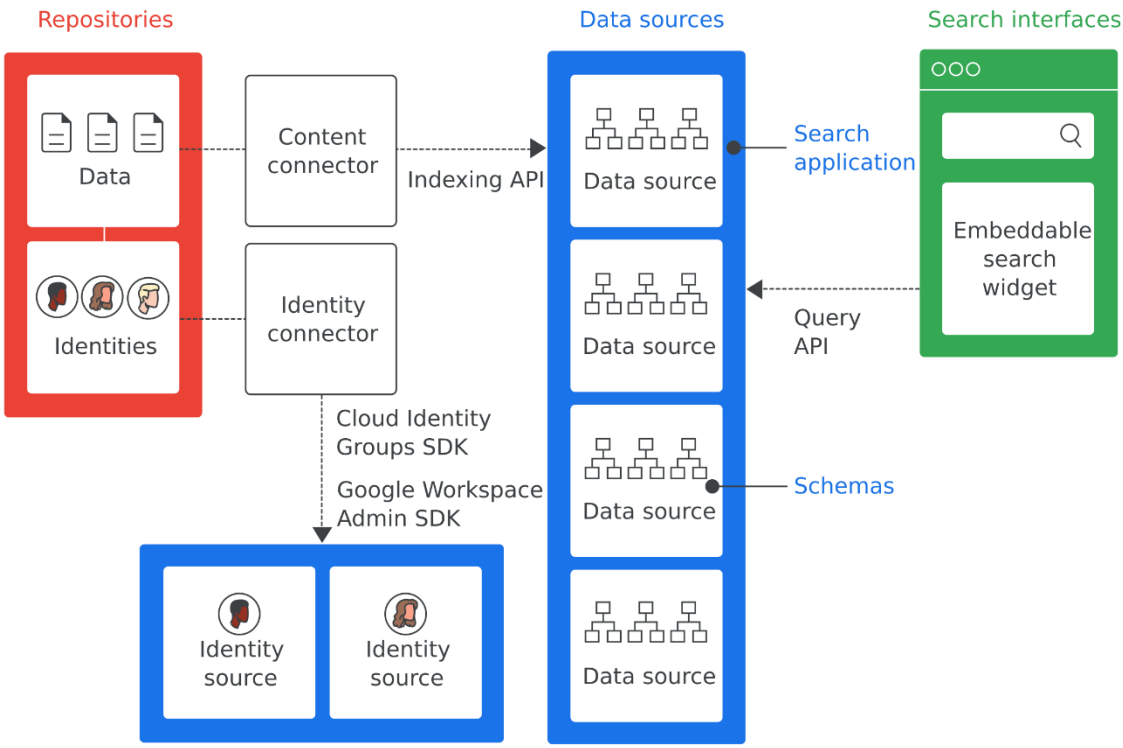
Index

Reference.

샘플 코드 1 : https://github.com/google-cloudsearch/database-connector

샘플 코드 2 : https://github.com/google-cloudsearch/apache-nutch-indexer-plugin

00. Google Cloud Search



개요

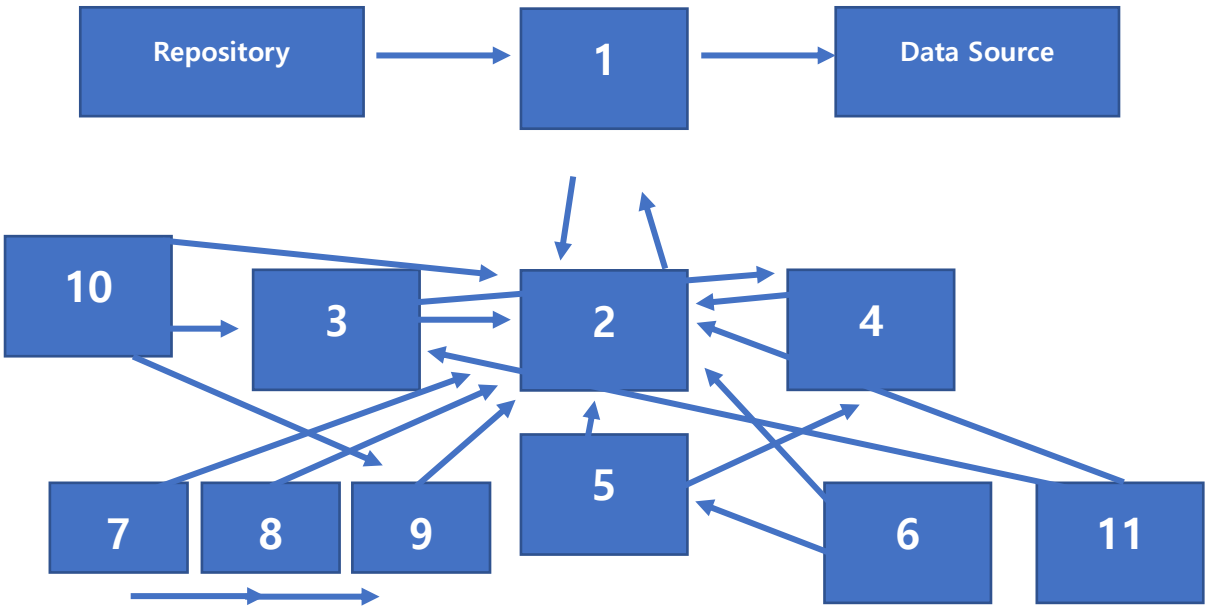
DB, Web 등의 다양한 Repository로부터 Connector를 통해 Data Sources로 Data를 적재하고, 사용자로 하여금 Search Interfaces를 통해 원하는 정보를 Search할 수 있도록 하는 시스템.

해당 문서에서는 주로 Content Connector에 관하여 다룬다.

01. database-connector의 Architecture

01-A. DB Connector 전체 Architecture

DB를 Repository로 Search의 Data Source에 업로드하는 Connector.
샘플 코드의 DB로는 H2, SqlServer가 사용되었다.



01. DataFullTraversalConnector.java
02. DatabaseRepository.java
03. ColumnManager.java
04, DatabaseAccess.java
05. ConnectionFactory.java
06. DatabaseConnectionFactory.java
07. Checkpoint.java
08. IncrementalCheckpoint.java
09. FullCheckpoint.java
10. Pagination.java
11. UniqueKey.java

01-B. Connector란?

Repositories의 데이터를 순회하고 모든 문서를 Data Source에 채우는데 사용하는 소프트웨어.
Repo의 종류로는 DB, Window File System, Office, E-mail, Crawling Data 등 수많은 종류가 있다.

01-C. 용어 정리

1. FullTraversal Connector – 해당 샘플 코드에서 활용하는 connector 종류

- 커넥터가 빠르게 업로드할 수 있는 상대적으로 정적이거나 작은 데이터 세트에 사용. 문서를 Cloud Search 대기열(queue)로 push하지 않고 모든 문서를 업로드.

그 외 Connector : Listing Connector

- 대규모 동적 및 계층적 데이터 Repository에 이 유형을 사용. List나 graph 순회 전략에 사용한다. 문서 ID를 Cloud Search 대기열로 push한 다음 index 생성을 위해 개별적으로 처리.

Repository가 문서 변경 검색을 지원하는 경우 Connector는 새로 수정된 문서만 읽고 다시 Indexing하는 Incremental 변경 순회를 수행할 수 있다.

2. Checkpoint란?

특정 시간의 동기화 이벤트. 일부 혹은 전체 dirty block 이미지가 DB에 쓰여지도록 한다. 그렇게 함으로서 특정 시간 이전의 dirty block이 쓰여지는 것을 보장.
dirty block : 변경이 발생하여 데이터 파일과 차이가 있는 buffer block을 뜻함.
- **Full Checkpoint** : 모든 instance의 모든 dirty block을 DB에 쓴다.
- **Incremental Checkpoint** : 일부 dirty buffer가 DB에 쓰여진다. 변경사항이 여러 번 존재한다면 순차적으로 Checkpoint 발생.

기타 Checkpoint : Queue Checkpoint

전체 순회 중 삭제 감지에 사용되는 대기열 이름의 Checkpoint를 처리. Connector 파일 자체에는 없으나 FullTraversalConnector와 관련되는 **SDK 파일**에 존재함.

3. Pagination이란?

수백 개 이상의 record를 보유한 데이터 세트를 쿼리할 때 사용되는 전략. 대규모 데이터 세트를 청크(또는 페이지)로 분할 후 점진적으로 가져와서 사용자에게 표시하여 DB의 부하를 감소.

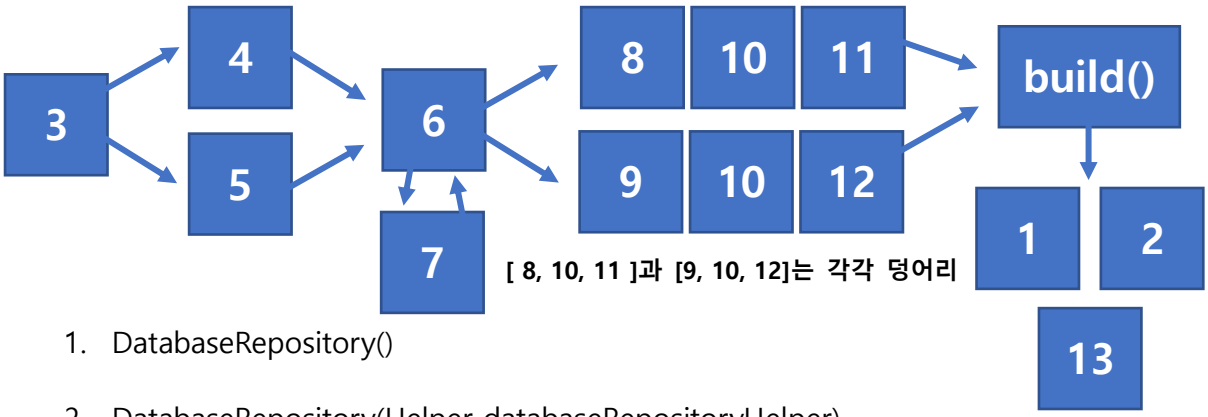
4. ColumnManager

DB의 Column과 UniqueKey, Pagiantion 등을 활용해 Template 생성, 반환.

5. ConnectionFactory, DatabaseConnectionFactory, DatabaseAccess

두 Factory의 DB Connection 정보, checkpoint 값을 받아 resultSet 객체의 값을 반환하고 CloumnTypeMap을 생성한다. SingleColumnValue, getAllColumnValues 등으로 value 반환.

01-D. DatabaseRepository.java에 대하여



1. DatabaseRepository()
2. DatabaseRepository(Helper databaseRepositoryHelper)
3. Public void init(RepositoryContext context)
4. Public CheckpointCloseableIterable<ApiOperation> getAllDocs(byte[] checkpoint)
5. Public CheckpointCloseableIterable<ApiOperation> getChanges(byte[] checkpoint)
6. ResultSetCloseableIterable<ApiOperation> getRepositoryDocIterable
   (DatabaseAccess databaseAccess, Checkpoint checkpoint)
7. private abstract static class ResultSetCloseableIterable<T>
   implements CheckpointCloseableIterable<T>
8. private class RepositoryDocIterable
   extends ResultSetCloseableIteralbe<ApiOperation>
9. private class RepositoryDocIterable
   extends ResultSetCloseableIterable<ApiOperation>
10. private Item createItem
    (Map<String, Object> allColumnValues, Checkpoint checkpoint)
11. private ByteArrayContent createContent
    (Map<String, Object> allColumnValues)
12. private ByteArrayContent createBlobContent
    (Map<String, Object> allColumnValues)
13. static class Helper

DatabaseRepository.java 코드 분석

01 / 02 DatabaseRepository() / DatabaseRepository (Helper helper)

기본생성자 : new Helper()

매개변수 Helper 생성자 : 아래 함수들이 만들어내는 죄종 반환값 + 현재 시각(startTimestamp)

---

13 static class Helper

ConnectionFactory, System.currentTimeMillis, ByteArrayConetent의 반환 값을 가진다.

- 각각 DB와의 연결, 현재 시각의 밀리세컨드 단위 변환, 실제 Index 생성이 가능한 항목을 만들기 위한 전처리 값을 뜻한다.

---

03 Public void init(RepositoryContext context)

설정 및 초기화 : DB connection, context의 build(), requestMode 등을 새로 가져온다.

requestMode : Item Index 및 삭제 요청에 대한 요청 모드.

---

04 getAllDocs(byte[] checkpoint) - Repo에서 모든 문서를 가져온다.

Input : checkpoint

1. Checkpoint가 null - Pagination 값을 가져와 FullCheckPoint 값 갱신 후 currentCheckpoint 삽입.
   null이 아니면 – FullCheckpoint의 checkpoint 값 가져옴.
2. AllReordSql을 가져오는 형태로 currentCheckpoint를 반영해서 databaseAccess를 빌드.

- Checkpoint 중 FullCheckpoint 사용

Output : databaseAccess, currentCheckpoint

---

05 getChanges(byte[] checkpoint) – 마지막 순회 이후 변경된 모든 문서를 가져온다.

Input : checkpoint

1. IncrementalCheckpoint가 업데이트 되었고 checkpoint가 null이라면 startTimestamp.
   null이 아니라면 IncrementalCheckpoint에서 checkpoint값 가져옴.
2. currentCheckpoint의 traversalStartTime을 CurrentTime으로 갱신.
3. IncUpdateSql을 가져오는 형태로 currentCheckpoint를 반영, IncUpdateTimezone도
   반영해서 databaseAccess를 빌드.

- Checkpoint 중 IncrementalCheckpoint 사용

Output : databaseAccess, currentCheckpoint

---

06 getRepositoryDocIterable(DatabaseAccess databaseAccess, Checkpoint checkpoint)

Input : **04** *or* **05** *Output*

1. blobColumn이 존재한다면 Output1.
2. 존재하지 않는다면 Output2.

- ColumnManager.java에서 blobCloumn 내역을 받아올 수 있다.

Output1 : RepositoryDocIterable(databaseAccess, checkpoint)

Output2 : RepositoryDocBlobIterable(databaseAccess, checkpoint)

---

07 private abstract static class 06 implements { 04 , 05 type }

Input : 없음. **06** *과 연동.*

저장소의 Column이 남아있는가 판단하고 남아있다면 resultRecordSet을 만들 수 있는 권한 부여.

Output : createResultSetRecord(access.getAllColumnValues()

---

08 + 10 + 11 / 09 + 10 + 12 RepositoryDocIterable extends 06

Input : 없음. **06** *과 연동.*

1. ColumnManger.java로부터 Sql을 받아오고 이것이 allColumnValues의 key에 포함되었다면 value 값을 가져와 ColumnMager.java를 통해 template을 생성한다. 이 값은 helper, 13 에도 저장.
   - allColumnValues의 값은 DatabaseAccess.java에서 가져온다.
   - blob이라면 String, byte[] 형변환의 과정이 한 차례 더 끼어있다.
2. allColumnValues의 값들을 IndexingItemBuilder를 통해 build를 진행.
3. 1(setContent), 2(setItem), requestMode의 값으로 RepositoryDoc.Builder().build()를 실행, 반환.
   - RepositoryDoc : SDK 파일. 단일 문서를 생성하는 내용... 인 것 같다.

Output : RepositoryDoc.Builder().build()

RepositoryDoc.Builder().build()

.setContent : Content 및 ContentFormat을 설정

.setItem : Indexing할 항목을 설정

>> 즉 RepositoryDoc의 설정을 마치고 Instance를 생성한다, 는 뜻이다.

>> ApiOperation도 포함해 Repository 등이 SDK에서 indexing.template을 담당한다.

코드 상에 있으나 사용되지 않은 함수

getDoc(Item item) : Repo에서 단일 문서를 가져온다.

getIds(byte[] checkpoint) : Repo에서 모든 문서 ID를 가져온다.

Conclusion.

1. Repo(DB)와 관련된 설정을 초기화한다.

2. DB의 build()와 최근 Checkpoint를 가져온다.

3. DB의 Column이 더 남아있는가 판단하고 createResultSetRecord를 수행한다.

4. allColumnValues로부터 Content와 ContentFormat, 그리고 Indexing할 항목을 구한다.

5. 4를 바탕으로 RepositoryDoc.Builder를 수행하고 3의 create~ 로 반환한다.

6. FulltraversalConnector 등의 Connector SDK를 통한 업로드.

7. 완료.

Question.

Part 1. 샘플 코드의 전개에 대한 의문점

1. Helper 클래스가 최종 결과 값(01, 02함수)에 들어가는데 이 정도로 중요한 역할의 클래스가 왜 Helper라는 이름으로 작명되었는가.
2. Helper 클래스에는 08, 09함수에서 빌드한 resultSetRecord가 포함되지 않았다.
3. 04 – 12함수의 최종 산출값인 resultSetRecord는 어디에서도 사용되지 않는다.

Part 2. 샘플 코드의 내용에 대한 의문점

1. 해당 코드는 결국 따지고 보면 DB의 데이터를 받아와서 Checkpoint와 Blob 유무를 판단하고 적절하게 자르고 편집해서 전송할 뿐인 것 같다. 그럼 Search와 관련된 매커니즘은 어디에?

After Task.

src/it/java~ 경로의 DatabaseConnectorIT.java 와 src/main/test/java~ 경로의 *Test.java 분석 필요.
SDK에 indexing으로 분류된 structuredData.java 파일 이해 필요.
해당 샘플 코드는 H2, SQLServer가 Repo일 경우만을 전제로 삼았다. Connector에 들어가는 다양한 종류의 Repo 데이터를 정형화시키는 과정 숙지 필요.
Schema, Search Application, Search Interface, Query API 등에 대해서도 알아 두는 게 좋을 듯하다.
위 의문점대로, RepositoryDoc.build()의 값이나 Search와 관련된 매커니즘, IndexingAPI 해명.

01-E. 추가 분석

01. Structured Data

일반적으로 Connector는 Structured Data 개체 및 메서드에 직접 액세스하지 않는다. build()는 문서를 생성할 때 Connector가 제공한 데이터 값을 사용하여 Structured Data 생성을 자동화한다.

- Cloud Search Indexing API는 Cloud Search에서 데이터의 Index 생성 및 제공 방식을 설정하는데 사용할 수 있는 Schema Service를 제공. Local Repo Schema를 사용 중인 경우 Structured Data local Schema 이름을 지정해야 한다. >> 아마도 properties 파일에 정보 입력.

02. Index 생성이 지원되는 파일 형식

DOC, DOCX, XLS, XLSX, PPT, PPTX, PDF, RTF, TXT, HTML, XML

위 파일 형식 외에도 Cloud Search는 일반 텍스트 파일 내에 있는 콘텐츠의 Index 생성 지원.

03. com.google.enterprise.cloudsearch.sdk.indexing

Interface

- indexingService : Connector 개발자와 Indexing Service API BE 사이의 교차 지점.

Class

- ConnectorTraverser : Indexing Connector 순회 관련 작업의 예약 및 실행을 처리.
- ContentTemplate : Cloud Search에 업로드하기 위해 Repo 필드 데이터(DB, CSV, CRM 등)의 콘텐츠 형식을 지정하는 데 사용되는 HTML Template을 만드는 유틸리티.
- IndexingApplication : SDK의 기본 개체 및 액세스 지점.
- IndexingItemBuilder : Item의 빌드를 돕는 object.
- MockItem : IndexingItemBuilder을 사용하여 메타데이터에 대해 지정된 값 항목 생성.
- StructuredData : StructuredDataObject 생성을 돕는 utility.
- TestUtils : Indexing SDK의 통합 테스트를 위한 Utility 메서드.

04. Container

전체 Connector Template은 DB에서 삭제된 Record를 감지하기 위한 임시 Data Source Queue toggle 개념과 관련된 알고리즘을 사용한다. Traversal을 하면 Queue에서 가져온 새 Record가 이전 Traversal의 Queue에서 Indexing된 기존 Search Record를 대체하게 된다.

- traverse.queueTag=instance

Connector의 여러 Instance를 병렬로 실행하여 서로 간섭하지 않고 공통 Data Repo를 Indexing 하려면 각 Connector 실행에 고유한 Container 이름을 지정해야 한다.

- traverse.useQeueues=true/false

커넥터가 삭제 감지를 위해 Queue Toggle 논리를 사용하는지 여부를 나타낸다.

(Template을 구현하는 FullTraversalConnector에만 적용할 수 있다.)

02. database-connector 실행

02-A. 기본 프로세스 풀이 + 공식 문서 오류

1. GitHub에서 커넥터 저장소 복제.

```
git clone https://github.com/google-cloudsearch/database-connector.git
cd database-connector
```

2. Connector 버전 체크아웃. ZIP 파일 빌드.

```
git checkout tags/v1-0.0.5
mvn package
```

- "mvn package"를 수행해 나오는 zar 파일이 곧 db-connector. 추후 4 번에서 Connector 를 실행하게 되면 해당 zar 파일이 실행되는 것. mvn pacakage 작업을 진행한 폴더 상의 코드를 수정해도 zar 파일이 수정되지는 않으므로 Connector 에는 반영되지 않는다. 만일 Connector 의 코드를 수정했다면 다시 mvn package 를 통해 새로운 jar 파일을 만들어 새로 실행해야 한다.

3. Connector 구성 빌드.

"connector-config.properties" 파일 생성.
Data Source access API, Database Access parameters, SQL, columns 등 파라미터 조정.

4. Connector 실행.

```
java \
   -cp "google-cloudsearch-database-connector-v1-0.0.5.jar:mysql-connector-java-
5.1.41-bin.jar" \
   com.google.enterprise.cloudsearch.database.DatabaseFullTraversalConnector \
   -Dconfig=mysql.config
```

4-1. -cp "~~.jar:~~jar"

- -cp : classpath 를 뜻하는 명령어로 java 파일을 어디에서 실행할지 설정하는 경로.
- **중간의 ":" 표기가 잘못되었다. 여러 파일을 classpath 로 설정할 경우 각 파일의 구분자로 쓰이는 ";"가 올바른 표기. 해당 ":"로 실행할 경우 오류 발생.**

4-2. com.~~.DatabaseFullTraversalConnector

- classpath 로 해 둔 파일 내부를 탐색한다. 결코 -cp 설정 파일 바깥의 코드를 탐색하지 않음.

4-3. -Dconfig-mysql.config

- 3 에서 생성한 "connector-config.properties" 파일. 파일명을 정확하게 "connector-config.properties"로 생성했다면 해당 줄은 없어도 무방. 만약 mysql.config 등 다른 파일명으로 Connector 구성 빌드 파일을 생성했다면 예시처럼 다시 설정해 주어야 한다.

02-B. Log 분석

1. Default Log [샘플 코드에 기본적으로 깔려 있던 Log]

// 1. Connector 초기화
sdk.indexing.template.FullTraversalConnector – init

// 2. 작업 시작
sdk.ConnectorScheduler$ConnectorTraversal – connectTraversal Begin
sdk.indexing.template.FullTraversalConnector – doTraverse Begin

// 3. DB 연결, properties 설정에 따라 row 탐색
database.DatabaseAccess
sdk.BatchRequestService.flushIRequired – 50 개 row 단위로 Batch.

// 4. 작업 정리
sdk.indexing.template.FullTraversalConnector – doTraverse End
sdk.ConnectorScheduler$ConnectorTraversal – connectTraversal Completed

// 5. 결과 출력
sdk.indexing.BatchingIndexingServiceImpl & IndexingService & IncrementalTraverser
- Registered Operation
- Registered Operation with responses
- Operation completed with success
- Operation completed with failure
- Response latency on operations : 각 row 의 latency 를 올림 후 지정 근사값으로 기록. 단위는 millisecond

2. Custom Log – DatabaseRepository.java 프로세스

- 01-D 에서 분석한 flow 대로 로그가 찍히는 것을 확인.
- 기타 특이사항

```
<html lang='en'>
<head>
<meta http-equiv='Content-Type' content='text/html; charset=utf-8'/>
<title>49</title>
</head>
<body>
<div id='actor_id'>
  <p>actor_id:</p>
  <h1>49</h1>
</div>
<div id='first_name'>
  <p>first_name:</p>
  <p><small>ANNE</small></p>
</div>
</body>
</html>
```

---

config.properties 에 입력한 Query 문으로 조회된 DB 의 각 row 마다 위와 같은 HTML 형태로 변환이 되고, 위 HTML 을 Binary 형태로 한 번 더 변환을 한 후에.

```
RepositoryDoc [item={itemType=CONTENT_ITEM, metadata={sourceRepositoryUrl=49},
name=49}, content=com.google.api.client.http.ByteArrayContent@7b690e9c,
contentFormat=HTML, contentHash=null, childIds={}, fragments={},
requestMode=UNSPECIFIED]
```

최종적으로 위와 같은 형태로 [Repository -> Connector -> Data Source] 프로세스를 통과한다.

02-C. config.properties와 ColumnManager.java 분석

- sdk.configuration 내부에서 Connector 의 Config 파일을 해시테이블로 삽입하는 코드 존재.
- properties 파일 각 줄의 등호 좌측을 Key 로 우측의 Value 를 가지고 나오는 형태.

1. required parameters

- db.allColumns

ex) actor_id, last_update

배열로 변환이 되어서 allRecordsSql 로 출력된 ResultMap 에서 해당 원소만을 추출.

- db.uniqueKeyColumns

ex) actor_id

- db.allRecordsSql : 전체 순회마다 실행되는 쿼리.

ex) select actor_id, first_name, last_name, last_update from actor

Access 한 DB 에 Query 문으로 삽입되어서 ResultMap<key, value>를 생성.

해당 ResultMap 으로부터 row 하나씩을 추출해서 02-B 에서 분석한 대로 HTML 로 변환, Binary 변환을 거쳐 Data Source 로 삽입.

- contentTemplate.db.title : index 생성 및 결과 우선순위 지정을 위한 최고 Column.

2. optional parameters

- "db.allRecordsSql.pagination" : none-페이지 사용 안 함 / offset-기준으로 페이지 나눔.
- "db.contentColumns" : DB 에서 Content 열을 지정. db.allColumns 에 없는 원소는 넣을 수 없다.
- "db.incrementalUpdateSql" : 증분 순회 예약 시 필수.
- "db.timestamp.timezone": DB 타임스탬프에 사용할 시간대를 지정.
- "db.blobColumn" : blobColumn 명을 적어야 Blob 으로 조회가 된다. 적어 두지 않으면 Blob Column 도 Blob 이 아닌 Column 으로 함께 조회가 된다.
- "readers_users" : ACL – 권한
- "readers_groups" : ACL – 권한
- "denied_users" : ACL – 권한
- "denied_groups" : ACL – 권한
- "timestamp_column" : 증분 순회 시 최종 Update 시간을 추적하려면 여기에 별칭 지정.
- "GMT" : Default TimeZone