Ch9. SPARK SQL

아꿈사-박주희

Spark SQL

- 스파크 SQL은 구조화된 데이터(스키마를 가진 데이터)에 대해 데이터 불러오기 및 쿼리등을 통해 쉽게 다룰 수 있다.
- 주요 기능

  1. DataFrame 추상화 클래스를 통해 구조화된 데이터세트를 다루는 작업을 간편하게 만든다.
     - DataFrame은 관계형 DB의 테이블과 유사한 개념.
  2. 다양한 구조적 포맷의 데이터를 읽고 쓸 수 있다.
  3. JDBC/ODBC연결을 지원하므로 스파크SQL을 통해 SQL로 데이터를 질의할 수 있다.

- 데이터 프레임 이란: RDD의 확장 모델 (ROW객체(하나의 레코드)들의 RDD, 스키마 정보 포함)로 외부 데이터 소스, 쿼리 결과, 기존 RDD들로부터 생성할 수 있다.

Linking with Spark SQL

- 다른 스파크 라이브러리처럼 애플리케이션에 쓸 스파크 SQL도 추가적인 의존성을 요구함.
- 하이브를 지원하는 스파크 SQL은 하이브 테이블이나 UDF, SerDes, 하이브QL등에 접근이 가능하다. (필수 x)
- 일반적으로 위의 기능을 사용하기 위해 하이브 지원과 같이 스파크 SQL을 빌드하는 것이 좋다.

- 스파크 SQL 프로그래밍시 시작 포인트
  - 하이브 지원: HiveContext(하이브 설치본을 필요로 하지는 않음)
  - 하이브 미지원: SQLContext

- 스파크 SQL을 기존 하이브 설치본과 연결할 경우 hive-site.xml을 스파크의 설정 디렉토리($SPARK_HOME/conf)에 복사함.

Using Spark SQL in Application

- 스파크 SQL을 사용하는 가장 강력한 방식은 스파크 애플리케이션 내부에서 사용하는 것으로, 쉽게 SQL을 써서 데이터를 적재하고 쿼리를 날릴 수 있게 해주면서도 동시에 파이썬, 자바, 스칼라로 작성한 프로그램 코드와 같이 사용이 가능함.

- 이를 위해서는 SparkContext 기반 HiveContext 생성이 필요함.
  - (질의 기능 및 스파크 SQL 데이터와 연동하기 위한 추가기능 사용가능).

Spark SQL 초기화

- 필요한 타입 정보를 가진 RDD를 SparkSQL에 특화된 RDD로 변환해 질의를 요청하는 데 필요하므로 아래 모듈을 Import 해야 함.

- **Example 9-2 Scala SQL import**

```
//Spark SQL import
import org.apache.spark.sql.hive.HiveContext

//or 하이브 의존성을 쓰지 않는 경우
import org.apache.spark.sql.SQLContext

val sc = new SparkContext(...)
val hiveCtx = new HiveContext(sc)
```

기본 쿼리 예제

- **테이블에 대한 쿼리를 만들려면 HiveContext나 SQLContext에서 sql()을 호출하면 된다.**

- **Example 9-9 Loading and quering tweets in Scala**

```
//SQL 컨텍스트 생성
val input = hiveCtx.jsonFile(inputFile)

//입력 스키마 RDD 등록
input.registerTempTable("tweets")

//retweet 기준으로 트윗들을 가져온다.
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM tweets
ORDER BY retweetCount LIMIT 10")
```

DataFrame-1

- 데이터프레임은 전통적인 데이터베이스의 테이블과 비슷하다.
- 내부적으로 하나의 데이터프레임은 **Row**객체들과 각 컬럼에 대한 타입 정보로 구성된 **RDD**를 가진다.
  - **RDD** 호출이 가능하므로 **MAP(), FILTER()** 연산 가능.
  - 어떤 데이터 프레임이든 임시 테이블로 등록하여 **HiveContext.sql, SQLContext.sql**로 질의 가능.
  - 임시 테이블 등록이 **RDD** 조작없이 직접 연산을 제공하는 여러개의 트랜스포메이션을 가지고 있음.

함수이름	목적	예
show()	데이터 프레임의 내용을 보여준다.	df.show()
select()	지정한 필드나 함수 결과를 가져온다.	df.select("name", df("age"+1)
filter()	조건에 맞는 레코드만을 가져온다.	df.filter(df("age") >19)
groupBy()	칼럼에 따라 그룹화한다. 이어서 min(), max(), mean(), agg()같은 집합 연산이 필요하다.	df.groupBy(df("name")).min()

DataFrame-2

Spark SQL/ HiveQL type	Scala type	Java type	Python
TINYINT	Byte	Byte/byte	int/long (in range of –128 to 127)
SMALLINT	Short	Short/short	int/long (in range of –32768 to 327 67)
INT	Int	Int/int	int or long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.TimeStamp	java.sql.TimeStamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple, or array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1: COL1_TYPE, … >	Row	Row	Row

데이터프레임과 RDD간의 변환

- **ROW 객체: 데이터 프레임 내부의 레코드, 필드들의 고정 길이 배열을 말함.**
- **접근 방법 :**

  - 자바, 스칼라 : 각 타입별 getter 함수 존재, 각 타입에는 해당 타입을 돌려주는 getType() 함수 존재

  자바:get(i), 스칼라:apply(i)

  - 파이썬 : row[i] 형태 접근, row.column_name 접근

- **Example 9-12**

```
val  topTweetText = topTweets.rdd().map(row => row.getString(0))
```

- **Example 9-14**

```
topTweetText = topTweets.rdd().map(lambda row: row.text)
```

캐싱

- **Spark SQL은 cacheTable("tableName")를 호출함으로써 메모리 상의 컬럼 지향 포맷으로 테이블을 캐쉬할 수 있다.**

- **캐시된 테이블은 메모리에 드라이버 프로그램이 실행되는 동안만 존재하므로, 프로그램이 끝나면 데이터를 새로 캐시해야 한다.**

- **HiveSQL이나 SQL 문장으로도 테이블 캐시 가능하며, Cache Table 'TableName' or UnCache Table 'TableName'을 사용한다.**

데이터 불러오고 저장하기.

- **Spark SQL은 여러가지 구조화된 데이터소스(하이브테이블, JSON, 파케이 파일)를 기본적으로 지원하며,복잡한 로딩절차 없이 데이터 소스로부터 ROW객체를 가져올 수 있다.**
  더불어 SQL로 쿼리를 날리고 필요한 필드만을 지정하면 그 필드의 데이터들만을 탐색한다.

- **데이터 소스 연동 API : 에이브로, HBASE, 일래스틱 서치, 카산드라 등**

- **일반 RDD에 스키마를 붙여 데이터프레임으로 변환하는 것도 가능하므로, 어떤 데이터 소스에 생성된 데이터 프레임이든 이런 RDD를 데이터프레임으로 변환하면 조인 역시 가능하다.**

아파치 하이브

- **데이터를 hive에서 읽어 올 때 SparkSQL은 텍스트,RC,ORC,파케이, 에이브로,프로토콜버퍼 등 하이브가 지원하는 어떤 저장 타입이든 모두 지원..**

- **SPARK with Yarn-Cluster설정**

  1. hive-site.xml을 spark의 conf 디렉토리에 복사.

  2. pyspark or spark-shell 실행할 경우 실행메모리 오류 발생시 yarn cluster의 가용실행메모리 설정 변경

  ex) - yarn-scheduler.minimum-allocation-mb 1.5G

  - yarn-scheduler.maximum-allocation-mb 1.5G

  3. spark-shell or pyspark 재실행.

- **Exmaple 9-15 파이썬에서 하이브 읽기**

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("select key,value FROM mytable")
keys = rows.map(lambda row: row[0])
```

데이터 소스/파케이

- **하이브에서 쓰는 범위 이상의 데이터 소스를 지원하기 위해 SparkSQL은 연결하기 쉬운 형태의 데이터 소스 연동 api를 제공함**
  - 다른 저장 플랫폼의 타입체계를 Spark SQL의 데이터 모델로 자동 매핑해준다
  - 파케이 : HDFS 자체가 컬럼 스토어를 구현하기 어렵고 속도에 한계가 있기 때문에 데이터를 컬럼 기반으로 저장해 속도를 높이려는 시도로 나온 하이브에 저장되는 새로운 파일포맷.

- **Example 9-18 파이썬에서 파케이 읽기**

```
rows = hiveCtx.load(parquetFile, "parquet")
names = rows.map(lambda row: row.name)
print "Everyone"
print names.collect()
```

- **Example 9-19 파이썬에서 파케이 질의**

```
tbl = rows.registerTempTable("people")
pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE favouriteAnimal = \"panda\"")
print "Panda friends"
print pandaFriends.map(lambda row: row.name).collect()
pandaFriends.save("hdfs://...", "parquet")
```

JSON

- 스칼라에서는 동일한 스키마로 맞춰진 레코드들의 **JSON** 파일을 갖고 있다면, 파일 스캔해 스키마를 추정하고 필드에 이름으로 접근할 수 있게 해준다.
- 단순히 **hiveCtx**에 **jsonFile()**을 호출하는 것만으로도 가능하다.

- **Example 9-22** 파이썬에서 **SparkSQL**로 **JSON** 불러오기

```
input = hiveCtx.jsonFile(inputFile)
```

- **Example 9.25** 스키마 보기

```
scala> rows.printSchema()
root
  |-- time: string (nullable = true)
  |-- ip: string (nullable = true)
  |-- country: string (nullable = true)
  |-- status: string (nullable = true)
```

RDD에서 가져오기

- 데이터를 읽어 들이는 것 이외에도 RDD로부터 데이터 프레임을 만들어 낼수 있다.
  - 스칼라: 케이스 클래스의 RDD가 데이터 프레임으로 변환
  - 파이썬: Row객체들의 RDD를 만들고 inferSchema를 부르면된다.

- **Example 9.28 파이썬에서 Row와 이름있는 튜플로 DataFrame 만들기**

```
happyPeopleRDD = sc.paralleize[Row(name="holden", favouriteBeverage="coffee")])
happyPeopleDataFrame = hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleDataFrame.registerTempTable("happ_people")
```

- **Example 9.29 스칼라에서 케이스 클래스로부터 DataFrame 만들기**

```
case class HappyPerson(handle: String, favouriteBeverage: String)
val happyPeopleRDD = sc.parallelize(List(HappPerson("holden", "coffee")))
happyPeopleRDD.registerTempTable("happy_people")
```

JDBC/ODBC 서버

- **Spark SQL은 JDBC 기능을 제공한다. JDBC 서버는 단독 Spark 드라이버 프로그램으로 실행되며 여러 클라이언트에 의해 공유될 수 있다.**
  - **JDBC 서버는 하이브의 HiveServer2와 연계되므로 하이브 지원을 포함해서 빌드되어야 함.**
  - **jdbc 서버는 Spark 디렉토리의 sbin/start-thriftserver.sh로 실행.**
  - **(spark-submit와 동일한 옵션을 다수 받아들임, 기본 포트 localhost:10000)**

- **example 9.31**

  ./start-thriftserver.sh --master sparkMaster

  starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /usr/lib/spark/logs/spark-hdfs-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-server02.hadoop.com.out

비라인으로 jdbc 접속

```
[root@server02 /usr/lib/spark/bin]# beeline
Beeline version 1.1.0-cdh5.4.5 by Apache Hive
beeline> !connect jdbc:hive2://server03.hadoop.com:10000
scan complete in 3ms
Connecting to jdbc:hive2://server03.hadoop.com:10000
Enter username for jdbc:hive2://server03.hadoop.com:10000: admin
Enter password for jdbc:hive2://server03.hadoop.com:10000: *****
Connected to: Apache Hive (version 1.1.0-cdh5.4.5)
Driver: Hive JDBC (version 1.1.0-cdh5.4.5)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://server03.hadoop.com:10000>
0: jdbc:hive2://server03.hadoop.com:10000> show tables;
+----------------+--+
|    tab_name    |
+----------------+--+
| access_country |
| firewall_logs  |
| secu_info      |
| secu_mart_moum |
+----------------+--+
4 rows selected (0.679 seconds)
0: jdbc:hive2://server03.hadoop.com:10000>
```

비라인으로 작업하기

- **비라인을 쓰면 테이블을 만들고 목록을 보고 쿼리를 날려 보는 하이브 QL 명령어들을 실행해 볼 수 있다.**
- **HiveQL 메뉴얼 :**
  https://cwiki.apache.org/confluence/display/Hive/LanguageManual

- **1. Data 를 읽어서 Table 을 생성**
  Hive 는 ,(comma) 로 구분된 CSV 같은 파일로 된 데이터를 로딩할 수 있도록 지원한다.

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'learning-spark-examples/files/int_string.csv'
  INTO TABLE mytable;
```

- **만약 Table 을 Caching 하고자 한다면, CACHE TABLE tableName 을 사용하면 된다. 후에 Cache 에서 해제할 땐, UNCACHE TABLE tableName 을 사용하자.**

비라인으로 작업하기

```
0: jdbc:hive2://server03.hadoop.com:10000> select count(*) from access_country;
INFO  : Number of reduce tasks determined at compile time: 1
INFO  : In order to change the average load for a reducer (in bytes):
INFO  :   set hive.exec.reducers.bytes.per.reducer=<number>
INFO  : In order to limit the maximum number of reducers:
INFO  :   set hive.exec.reducers.max=<number>
INFO  : In order to set a constant number of reducers:
INFO  :   set mapred.reduce.tasks=<number>
INFO  : Starting Job = job_201511102000_0001, Tracking URL = http://server01.hadoop.com:50030/jobdetails.jsp?jobid=job_201511102000_0001
INFO  : Kill Command = /usr/lib/hadoop/bin/hadoop job  -kill job_201511102000_0001
INFO  : Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
INFO  : 2015-11-10 21:30:18,574 Stage-1 map = 0%,  reduce = 0%
INFO  : 2015-11-10 21:30:24,685 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 1.71 sec
INFO  : 2015-11-10 21:30:31,752 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 2.87 sec
INFO  : MapReduce Total cumulative CPU time: 2 seconds 870 msec
INFO  : Ended Job = job_201511102000_0001
+---------+--+
|   _c0   |
+---------+--+
| 141250  |
+---------+--+
1 row selected (25.508 seconds)
0: jdbc:hive2://server03.hadoop.com:10000>
```

비라인으로 작업하기

```
0: jdbc:hive2://server03.hadoop.com:10000> explain select count(*) from access_country where region = 'UA';
+-----------------------------------------------------------------------------------------------------------+--+
|                                                Explain                                                    |
+-----------------------------------------------------------------------------------------------------------+--+
| STAGE DEPENDENCIES:                                                                                       |
|   Stage-1 is a root stage                                                                                 |
|   Stage-0 depends on stages: Stage-1                                                                      |
|                                                                                                           |
| STAGE PLANS:                                                                                              |
|   Stage: Stage-1                                                                                          |
|     Map Reduce                                                                                            |
|       Map Operator Tree:                                                                                  |
|           TableScan                                                                                       |
|             alias: access_country                                                                         |
|             Statistics: Num rows: 141250 Data size: 282500 Basic stats: COMPLETE Column stats: NONE       |
|             Filter Operator                                                                               |
|               predicate: (region = 'UA') (type: boolean)                                                  |
|               Statistics: Num rows: 70625 Data size: 141250 Basic stats: COMPLETE Column stats: NONE      |
|               Select Operator                                                                             |
|                 Statistics: Num rows: 70625 Data size: 141250 Basic stats: COMPLETE Column stats: NONE    |
|                 Group By Operator                                                                         |
|                   aggregations: count()                                                                   |
|                   mode: hash                                                                              |
|                   outputColumnNames: _col0                                                                |
|                   Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE           |
|                   Reduce Output Operator                                                                  |
|                     sort order:                                                                           |
|                     Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE         |
|                     value expressions: _col0 (type: bigint)                                               |
|       Reduce Operator Tree:                                                                               |
|         Group By Operator                                                                                 |
|           aggregations: count(VALUE._col0)                                                                |
|           mode: mergepartial                                                                              |
|           outputColumnNames: _col0                                                                        |
|           Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE                   |
|           Select Operator                                                                                 |
|             expressions: _col0 (type: bigint)                                                             |
|             outputColumnNames: _col0                                                                      |
|             Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE                 |
|             File Output Operator                                                                          |
|               compressed: false                                                                           |
|               Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE               |
|               table:                                                                                      |
|                   input format: org.apache.hadoop.mapred.TextInputFormat                                  |
|                   output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat               |
|                   serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe                               |
|                                                                                                           |
|   Stage: Stage-0                                                                                          |
|     Fetch Operator                                                                                        |
|       limit: -1                                                                                           |
|       Processor Tree:                                                                                     |
|         ListSink                                                                                          |
|                                                                                                           |
+-----------------------------------------------------------------------------------------------------------+--+
49 rows selected (0.315 seconds)
0: jdbc:hive2://server03.hadoop.com:10000>
```

장시간 지속 테이블과 쿼리

- **Spark SQL JDBC Server 를 써서 얻는 이점 중 하나는 여러 프로그램들 사이에서 캐시된 테이블을 공유 할수 있다는 것.**
- **테이블 캐시는 앞 절에서 본 것처럼 테이블을 등록하고 CACHE 명령어를 사용하기만 하면 된다.**

User-Defined Functions

- **SQL 에서 사용할 수 있는 사용자 정의 함수(이하 UDF)를 Python/Scala/Java 모두 지원한다.**
- **Spark SQL 은 자체 UDF 와 Hive UDF 모두 지원한다.**

Spark SQL UDFs

- **Spark SQL 은 UDF 를 쉽게 등록할 수 있도록 내장 함수를 제공한다. Scala 나 Python 에서는 자체적으로 함수를 넘기거나 lambda 문법을 지원하여 사용할 수 있고, Java 에서는 적절한 UDF class 를 상속해 쓰면 된다.**

- **Python 이나 Java 에서는 표9.2에 정의된 데이터 프레임 타입 중의 하나로 리턴 타입을 지정하는 것이 필요.**

- **Example 9.37**

```
hiveCtx.udf.register("strLenScala", ( _: String).length)
val tweetLength = hiveCtx.sql("SELECT strLenScala('tweet') FROM tweets LIMIT 10")
```

Hive UDFs

- **Spark SQL 은 표준 Hive UDF 를 사용할 수 있다. 만약 직접 작성한 UDF 가 있다면 UDF의 JAR 파일들이 애플리케이션에 포함되어 있는지가 중요하다. JDBC서버를 쓴다면 그 파일들을 --jars 커맨드 라인 플래그에 추가해야 한다.**
- **또한 Hive UDF 를 사용하기 위해서는 SQLContext 가 아니라 HiveContext 를 사용해야 한다.**

  hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS class.function")

Spark SQL Performance

- **Spark SQL은 데이터를 더 효율적으로 표현하기 위한 타입 정보를 활용할 수 있다. 데이터를 캐싱할 때 Spark SQL은메모리 기반 칼럼 지향 저장소를 사용한다. 이는 Caching 할 때 적은 공간을 사용할 뿐만 아니라 이후의 쿼리들이 데이터의 일부만 읽고자 할 때, SparkSQL의 데이터 읽기 연산을 최소화해 주기도 한다.**

- **Spark SQL 은 조건별 하부이동(Predicate Push-Down) 이 지원된다. 이 기능은 Spark SQL이 쿼리의 일부분을 쿼리를 수행하는 엔진의 아랫단으로 보내 준다.**

- **Spark의 일부 레코드만을 읽는다고 할 때, 일반적인 방법은 전체 데이터 셋을 읽은 다음 거기에 조건절을 실행해 걸러 내는 것이다. 그렇지만 SparkSQL은 하부의 저장 시스템이 키범위의 일부만 가져오거나 여타 다른 제한 기능을지원한다면, 제한 조건을 데이터 저장 시스템 레벨까지 내려 보내어 수행하게 함으로써 잠재적으로 필요 없는 데이터읽기를 줄여 준다.**

Spark SQL Performance

- **Spark SQL의 성능 옵션**

Option	Default	Usage
`spark.sql.codegen`	`false`	When `true`, Spark SQL will compile each query to Java bytecode on the fly. This can improve performance for large queries, but `codegen` can slow down very short queries.
`spark.sql.inMemoryColumnarStorage.compressed`	`false`	Compress the in-memory columnar storage automatically.
`spark.sql.inMemoryColumnarStorage.batchSize`	`1000`	The batch size for columnar caching. Larger values may cause out-of-memory problems
`spark.sql.parquet.compression.codec`	`snappy`	Which compression codec to use. Possible options include `uncompressed`, `snappy`, `gzip`, and `lzo`.

Spark SQL Performance

- **JDBC Connector 나 Beeline shell 을 사용할 때, set command 를 통해서 옵션들을 설정한다.**

- **Example 9.41 codegen 옵션을 활성화하는 비라인 명령**

```
beeline> set spark.sql.codegen=true;
SET spark.sql.codegen=true
spark.sql.codegen=true
Time taken: 1.196 seconds
```

- **전통적으로 SparkSQL 에서는 9.41 방식 대신 아래와 같이 Spark 설정객체에 지정한다.**
- **Example 9.42 codegen을 활성화하는 스칼라 코드**

```
conf.set("spark.sql.codegen", "true")
```

Spark SQL Performance

- 몇몇 옵션들은 설정할 때 신중해야 하는데, 그 중 하나는 **spark.sql.codegen**인데 이는 **Spark SQL**이 각 쿼리를 실행전에 자바 바이트 코드로 컴파일하게 한다.
- 쿼리가 아주 길거나 자주 수행되는 쿼리에 대해서는 상당히 빠르게 실행시키게 하지만, 매우 짧은 실시간성 쿼리(1-2초 이내)의 경우 매번 **SQL** 을 변환해야 하기 때문에 **overhead** 가 있을 수 있다.
- 따라서 큰 규모의 쿼리나 자주 실행되는 반복적인 쿼리에 대해서는 적용을 추천한다.

- **spark.sql.inMemoryColumnarStorage.batchSize**의 경우 데이터프레임을 캐시할 때 **Spark SQL**은 **RDD**의 레코드들을 옵션에 주어진 배치 사이즈만큼(기본값 **1000**)씩 묶어서 각 묶음을 압축한다.
- **Batch Size**를 너무 작게 잡은 경우는 압축효율이 떨어지지만, 반대로 너무 큰 사이즈에서는 메모리 상에서 압축을 하기에는 너무 클 수 있기 때문에 문제가 될 수 있다.
- 만약 사용하는 테이블의 레코드 사이즈가 크다면(**EX:** 수백개 필드를 보유하거나, 웹 페이지 콘텐츠처럼 매우 긴 문자열이있다면) 배치 사이즈를 줄이는 것이 좋다. 보통 기본 배치 사이즈가 적당하다.