

Neptune Programmer's Manual

2008-11-16, 한글판



김형준 (jindolk@nhncorp.com)

NHN

본 문서는 대용량 데이터 관리 시스템인 Neptune 의 사용자 매뉴얼로 Neptune 에서 제공하는 클라이언트 API 나 Shell 을 이용하여 테이블을 생성하거나 데이터를 관리하는 방법을 설명한다.

본 문서에는 Neptune 에서 제공하는 모든 API 에 대한 설명은 없으며 모든 API 에 대한 레퍼런스는 Neptune API java doc 문서를 참고한다.

목차

1. NEPTUNE 데이터 서비스	2
1.1. 실시간 데이터 서비스.....	2
1.2. 배치 데이터 서비스.....	3
2. 클라이언트 API 사용법	4
2.1. 테이블 관리.....	4
2.2. 실시간 데이터 관리.....	5
2.2.1. 데이터 저장.....	5
2.2.2. 데이터 조회.....	7
2.2.3. Parallel 데이터 조회.....	8
2.2.4. 데이터 삭제.....	9
2.3. MAP&REDUCE.....	9
3. SHELL 사용법	13

1. Neptune 데이터 서비스

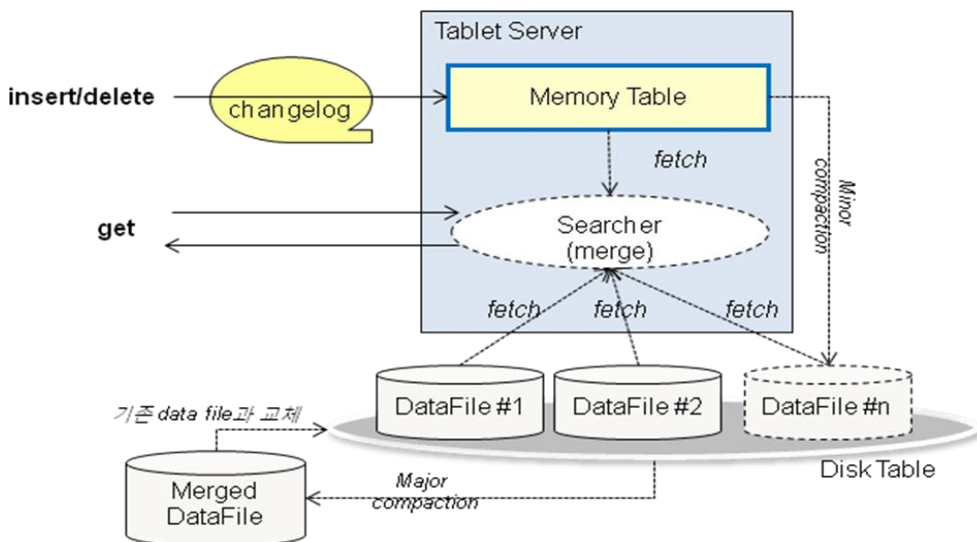
대부분의 관계형 데이터베이스는 실시간 데이터 서비스에 뛰어난 성능을 보장하지만 Neptune에서 제공하는 실시간 데이터 서비스는 적절한(reasonable) 성능만을 보장한다. 여기서 적절한 성능의 의미는 row-key를 이용하여 하나의 칼럼에 저장된 정보를 조회하는데 수 ms ~ 수십 ms 정도가 소요되며 row의 개수가 아무리 많아도 안정적인 응답속도를 보장한다는 의미이다. Neptune에서 배치 처리를 위한 데이터 서비스는 실시간 보다는 뛰어난 성능을 발휘하며 연속된 데이터에 대한 저장 및 조회에 적합하다.

따라서 Neptune을 이용하여 애플리케이션을 개발하는 사용자는 Neptune의 성능 수준을 고려하여야 하며 Neptune의 데이터 서비스 방법을 이해한 후 시스템에서 적절한 처리를 해야 한다.

1.1. 실시간 데이터 서비스

Neptune의 데이터 파일과 인덱스 파일은 한번 저장되면 더 이상 수정되지 않는 immutable 파일이다. 이런 immutable 파일을 이용하여 실시간 데이터 서비스를 하기 위해 Neptune은 메모리 기반 테이블과 디스크 기반 테이블 두 종류의 테이블을 이용한다. 각 TabletServer는 수백MB 정도의 메모리 테이블을 구성하고 insert, delete 등과 같은 데이터의 입력/수정 작업은 모두 메모리 테이블에 저장한다. 메모리 테이블은 실시간 처리를 위한 적절한 응답속도를 보장해주고 한번 저장되면 수정할 수 없는 데이터 파일의 제한에서 벗어 날수 있게 한다.

그림과 같이 데이터 수정(insert/delete) 요청은 changelog 파일에 저장되고 메모리 테이블로 저장된다. Change log 파일은 분산 파일 시스템에 저장되고 TabletServer 장애 발생 시 다른 TabletServer가 해당 테블릿을 할당 받아 메모리 테이블을 재구성하는데 사용한다.



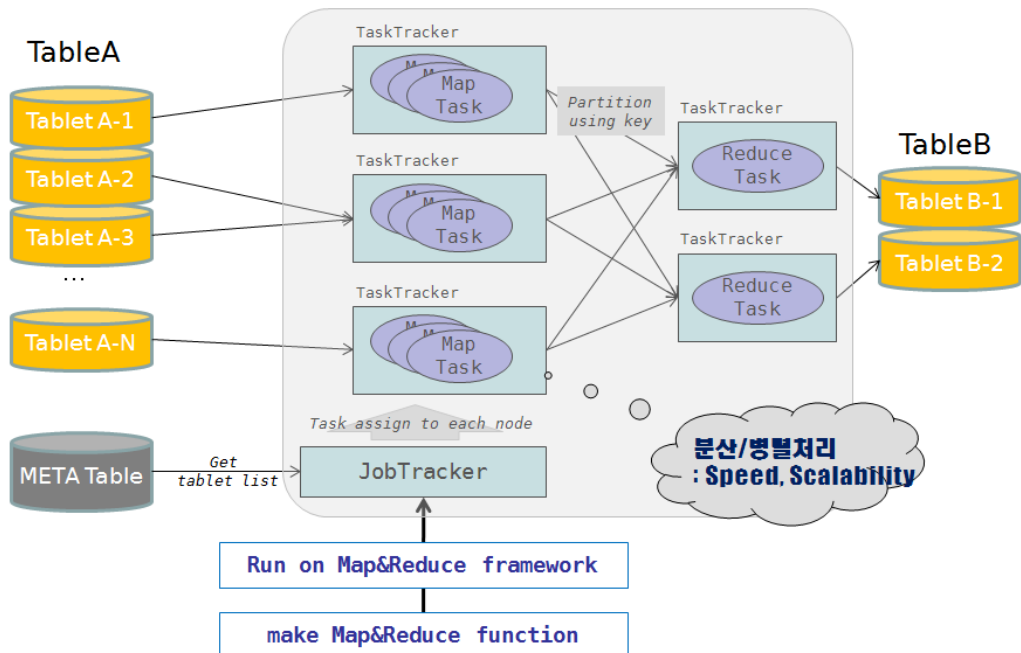
TabletServer는 메모리 테이블에 정해진 크기보다 더 많은 데이터가 저장되면 백그라운드 작업을 통해 메모리 테이블의 데이터를 분산 파일 시스템으로 저장한다. 이 과정을 "minor compaction"이라고 한다. Minor compaction 과정은 기존 데이터 파일에 추가하는 것이 아니라 그림10과 같이 새로운 데이터 파일과 인덱스 파일을 생성한다.

get 연산은 Neptune의 데이터를 실시간으로 조회하는 연산이다. get 요청을 받은 TabletServer는 메모리 테이블과 minor compaction으로 생성된 여러 개의 데이터 파일에서 조회된 데이터를 병합 (merge)한 후 사용자가 요청한 데이터를 전달한다. 데이터 파일의 개수가 get 연산 처리 성능에 많은 영향을 주기 때문에 백그라운드 작업을 통해 여러 개의 데이터 파일을 하나의 데이터 파일로 묶는 작업을 수행한다. 이 작업을 "major compaction"이라고 한다. Major compaction을 수행하면 삭제 요청된 레코드와 대한 삭제 작업과 오래된 버전 데이터를 삭제하는 가비지 콜렉션(Garbage Collection) 작업도 같이 수행한다.

1.2. 배치 데이터 서비스

Neptune에서는 관계형 데이터베이스에서 흔히 수행되는 like 연산이나 여러 row에 걸쳐서 데이터를 검색하는 연산 등은 실시간으로 수행할 수 없다. 데이터 모델에서 실시간으로 처리할 수 있는 범위를 하나의 row에서만 가능하도록 제한을 두었기 때문이다. 테이블에 저장된 전체 데이터 또는 일부 데이터를 이용하는 작업은 배치 작업으로 처리해야 하며 배치 작업은 하나의 프로세스로 작업하는 것이 아니라 대부분의 경우에 분산/병렬 처리한다.

Neptune의 데이터를 분석하기 위해서 Hadoop Map&Reduce 컴퓨팅 플랫폼을 사용한다. Map의 입력으로 테이블을 지정하는 경우 하나의 작업은 테블릿의 수만큼 쪼개어 지고 이렇게 쪼개진 작업을 태스크(Task)라고 부른다. 각 태스크의 입력은 하나의 테블릿이 되고 태스크가 각 노드에서 수행될 때 테블릿의 데이터 조회는 get 연산을 사용하지 않고 TableScanner 라고 하는 순차적 데이터 조회 처리 API를 이용하여 빠른 속도로 데이터를 조회한다.



앞의 그림에서 보는 것처럼 개발자는 map, reduce function만 개발한 다음 입력 테이블과 출력 테이블을 지정하면 Hadoop Map&Reduce와 Neptune에서 제공하는 AbstractTabletInputFormat에 의해 자동으로 여러 개의 태스크로 분배되고, 분산/병렬 처리되어 결과 테이블에 저장된다.

Neptune 사용자는 Neptune에 테이블을 생성하거나 생성된 테이블에 데이터를 저장, 조회하는 등

의 기능을 수행하기 위해 두 가지 방법을 이용할 수 있다.

첫 번째 방법은 사용자가 만든 프로그램 내에서 Neptune에서 제공하는 클라이언트 API 클래스를 이용하는 방법이다. 두 번째 방법은 Neptune Shell을 실행하여 shell에서 제공하는 명령을 이용하는 방법이다. 2장, 3장에서 각각의 방법에 대해 상세하게 설명한다.

2. 클라이언트 API 사용법

Neptune 사용자는 Neptune에서 제공하는 클라이언트 API를 이용하여 사용자가 개발하는 프로그램 내에서 Neptune에 테이블을 생성하거나, 데이터를 조회하는 등 다양한 기능을 수행한다. 클라이언트 API에서 제공하는 클래스는 다음과 같다.

클래스	기능
NTable	데이터의 추가, 삭제, 조회 등과 같은 테이블의 데이터를 관리하는 기능 수행
TableScanner	테이블 또는 Tablet의 전체 데이터를 대상으로 데이터를 검색할 때 사용 주로 Map&Reduce의 작업에 사용.
ScannerFactory	TableScanner를 오픈할 때 사용
DirectUploader	Row 단위로 데이터를 저장하는 것이 아니라 대량의 데이터를 한번에 테이블에 저장하는 데 사용.

2.1. 테이블 관리

Neptune에 데이터를 저장하기 위해서는 반드시 테이블을 생성해야 한다. 테이블은 데이터를 저장하는 기본 단위가 되며 테이블은 n개의 column을 가진다. Column의 수가 0인 테이블은 생성할 수 없으며 테이블을 생성하면 하나의 빈 Tablet이 생성된다.

테이블 이름은 영문자로 시작해야 하고 [a-z], [A-Z], [0-9], '_', 만 가능하며 최대 길이는 128이다. Column 이름도 테이블 이름과 동일한 규칙을 가지고 있다. 테이블 생성은 NTable 클래스를 이용한다. 다음 예제는 테이블을 생성(create), 삭제(drop)하는 예제이다. 실제 실행되는 코드는 Neptune 소스의 examples/first_apps에 있다.

```
private void createTable() throws IOException {
    String tableName = "SimpleTable";
    //Column
    String[] columns = new String[]{"Column1", "Column2"};
    //테이블 설명
    String description = "테스트";

    //TableSchema 객체를 생성한다.
    TableSchema tableSchema = new TableSchema(tableName, description, columns);

    NConfiguration nconf = new NConfiguration();
```

```

//기존에 테이블이 존재하는지 확인한다.
boolean exists = NTable.existsTable(nconf, tableName);
if(exists) {
    System.out.println("Already exists table");
    return;
}
//테이블을 생성한다.
NTable.createTable(nconf, tableSchema);

//테이블 drop
NTable.dropTable(nconf, tableName);
}

```

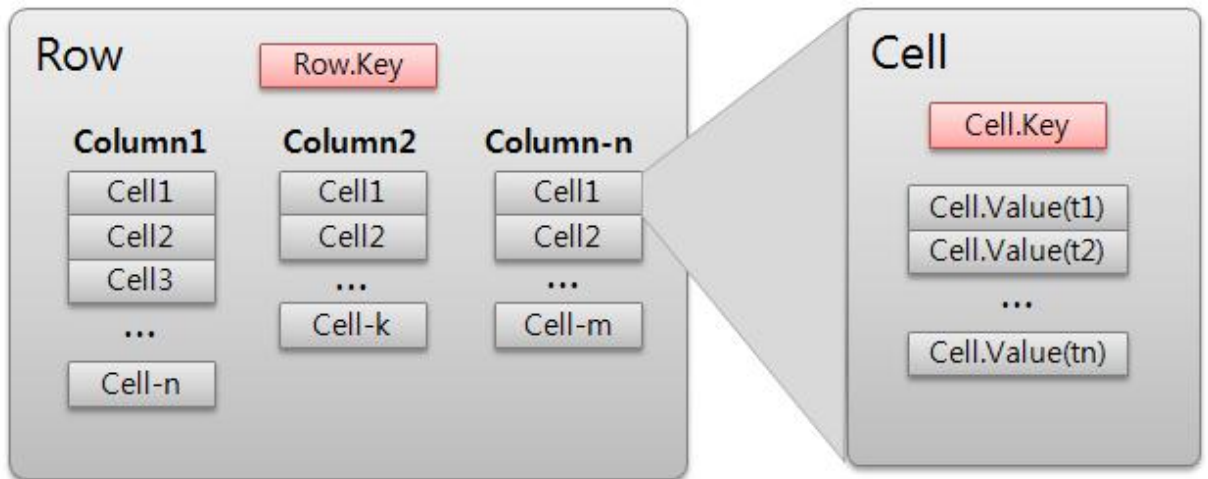
2.2. 실시간 데이터 관리

실시간 처리에서 테이블에 데이터를 저장하거나 삭제하기 위해서는 NTable 클래스를 이용한다. Neptune은 대용량 데이터 저장을 기본으로 하기 때문에 실시간 처리에서 다중 row에 대한 연산은 지원하지 않는다. 관계형 데이터베이스의 like, group by 등이 다중 row의 데이터에 대한 대표적인 사례이다. Neptune에 저장된 데이터를 실시간으로 처리하기 위해서는 반드시 rowkey를 이용해서 하나의 row에 대한 데이터만을 대상으로 해야 한다.

테이블의 전체 데이터 또는 부분 데이터에 대해 조회해야 하는 경우라면 TableScanner를 사용한다. TableScanner도 실시간 데이터 처리에 사용 가능하지만 동시에 open할 수 있는 TableScanner의 개수가 제한되어 있다. TableScanner에 대한 사용법은 2.3절에서 설명한다.

2.2.1. 데이터 저장

데이터 추가/수정/삭제는 NTable의 get, put 메소드를 이용하여 get 연산은 다양한 검색 조건을 지정할 수 있도록 되어 있다. 좀 더 복잡한 검색 조건 지정을 위해 RowFilter, CellFilter를 이용할 수 있다. get, put에 의해 조회된 결과 또는 저장할 정보는 Row, Cell이라는 객체를 이용하며 Row, Cell은 Neptune의 데이터 모델 개념을 구현한 클래스이다.



다음은 데이터 저장에 대한 실제 예제 코드이다.

```
private void putRow() throws IOException {
    //NTable 객체 오픈
    NTable ntable = NTable.openTable(nconf, tableName);

    //입력할 rowkey
    Row.Key rowKey = new Row.Key("RK1");

    //Row 객체 생성
    Row row = new Row(rowKey);

    //Cell 추가
    row.addCell("Column1", new Cell(new Cell.Key("CK1"), "TestData1".getBytes()));
    row.addCell("Column2", new Cell(Cell.Key.EMPTY_KEY, "TestData2".getBytes()));

    //테이블에 저장
    ntable.put(row);

    //새로운 Row 객체 저장
    row = new Row(new Row.Key("RK2"));
    row.addCell("Column1", new Cell(new Cell.Key("CK1"), "TestData3".getBytes()));

    //테이블에 저장
    ntable.put(row);
}
```

put() 메소드로 저장되는 데이터는 기본적으로는 시스템의 timestamp를 사용한다. 시스템 timestamp는 밀리세컨드 단위이며 동일한 데이터를 반복문을 통해 입력하는 경우 timestamp가 동일하게 적용될 수 있는데 이런 경우에는 다음과 같이 사용자 timestamp를 사용하게 해야 한다.

```
private void putRowTimestamp() throws IOException {
    //NTable 객체 오픈
    NTable ntable = NTable.openTable(nconf, tableName);

    //입력할 rowkey
    Row.Key rowKey = new Row.Key("RK10");

    //Row 객체 생성
    Row row = new Row(rowKey);

    //하나의 Row,Cell에 100개의 버전을 저장
    Cell.Key cellKey = new Cell.Key("CK1");
    long timestamp = System.currentTimeMillis();
    for(int i = 0; i < 100; i++) {
        //Cell 추가
        row.addCell("Column1", new Cell(cellKey, String.valueOf(i).getBytes(), timestamp));
        timestamp++;
    }

    //테이블에 저장
```

```

ntable.put(row);
}

```

2.2.2. 데이터 조회

Neptune에 저장된 데이터를 실시간으로 조회하기 위해서는 반드시 rowkey를 지정해야 한다. Rowkey를 알지 못하는 경우에는 TableScanner를 이용할 수 있지만 데이터가 많은 경우 실시간 응답이 가능한 수준의 성능은 보장하지 않는다.

데이터 조회를 위해 Neptune은 다양한 검색 조건을 갖는 get() 메소드를 지원한다. Get() 메소드에서 지원하지 않는 경우 RowFilter를 이용하여 자세한 검색 조건을 지정할 수 있다.

각각의 get() 메소드에 대한 세부 기능은 다음과 같다.

파라미터	리턴 값	설명
Row.Key	Row	모든 칼럼의 데이터를 조회한다. Row에는 모든 칼럼의 데이터가 저장되어 있다.
RowKey, columnNames	Row	지정된 칼럼의 데이터를 조회한다. Row에는 지정된 칼럼의 데이터만 저장되어 있다.
RowKey, columnName, Cell.Key	byte[]	특정 row, column, cellkey의 데이터를 조회한다.
RowFilter	Row	다양한 형태의 검색 조건을 지정하여 검색한다.(버전 개수, 버전 시간 등)
Collection<Row.Key> rowKeys, String[] columnNames, AsyncDataReceiver dataReceiver, int numOfThread, int timeout	void	여러 개의 Row에 대한 데이터를 multi-thread를 이용하여 get() 연산을 수행한다.

다음은 NTable 클래스의 get 메소드를 이용한 데이터 조회 예제이다.

```

private void getRow() throws IOException {
    NTable ntable = NTable.openTable(nconf, tableName);

    Row row = ntable.get(new Row.Key("RK1"));

    List<Cell> column1Cells = row.getCells("Column1");
    for(Cell eachCell: column1Cells) {
        //마지막 버전 값을 가져온다.
        System.out.println(eachCell.getValue().getValueAsString());
    }

    //여러 버전의 데이터를 조회하는 예제
    //putRow()에서 RK10에 대해 100개 버전의 데이터를 입력하였다.
    RowFilter rowFilter = new RowFilter(new Row.Key("RK10"));
    CellFilter cellFilter = new CellFilter("Column1");
    cellFilter.setStartTimestamp(Long.MIN_VALUE);
    cellFilter.setEndTimestamp(Long.MAX_VALUE);
    rowFilter.addCellFilter(cellFilter);
}

```

```

row = ntable.get(rowFilter);
Cell cell = row.getCells("Column1").get(0);
for(Cell.Value eachValue: cell.getValues()) {
    System.out.println(eachValue.getValueAsString());
}
}

```

2.2.3. Parallel 데이터 조회

Neptune은 데이터 모델에서 JOIN 연산을 지원하지 않기 때문에 서로 연관 있는 데이터를 조회하기 위해 여러 번의 get() 메소드를 호출해야 한다. 연관된 데이터가 많은 경우 실시간 서비스를 위한 데이터 조회 속도 요구사항을 만족시킬 수 없는 경우가 있다. 이 경우 여러 개의 Thread를 생성하여 동시에 데이터를 조회하여 데이터 조회 시간을 단축할 수 있다.

```

public class ParallelGet {
    public static void main(String[] args) throws IOException {
        //테이블 생성 및 데이터 입력
        String tableName = "SimpleTable2";
        String[] columns = new String[]{"Column1", "Column2"};
        String description = "테스트";
        TableSchema tableSchema = new TableSchema(tableName, description, columns);

        NConfiguration nconf = new NConfiguration();
        NTable.createTable(nconf, tableSchema);

        NTable ntable = NTable.openTable(nconf, tableName);

        List<Row.Key> rowKeys = new ArrayList<Row.Key>();
        for(int i = 0; i < 100; i++) {
            Row.Key rowKey = new Row.Key(String.valueOf(i));
            Row row = new Row(rowKey);

            row.addCell("Column1", new Cell(Cell.Key.EMPTY_KEY, String.valueOf(i).getBytes()));
        }

        List<CellFilter> cellFilters = new ArrayList<CellFilter>();
        cellFilters.add(new CellFilter("Column1"));
        ntable.get(rowKeys, cellFilters, new DataReceiver(), 10, 20);
    }

    static class DataReceiver implements AsyncDataReceiver {
        List<Row> results = new ArrayList<Row>();
        @Override
        public void error(Row.Key rowKey, Exception exception) {
            System.out.println("Error:" + rowKey);
        }

        @Override
        public void receive(Row.Key rowKey, Row row) throws IOException {
            results.add(row);
        }

        @Override

```

```

    public void timeout() {
    }
}

```

2.2.4. 데이터 삭제

Neptune에서 데이터 삭제는 실제로 데이터를 삭제하는 것이 아니라 삭제된 버전을 하나 추가하는 방식으로 처리된다. 삭제된 데이터가 실제로 삭제되는 시점은 minor, major compaction 되는 시점에 수행되면 테이블의 옵션에 설정된 버전의 개수에 따라 삭제된다. 예를 들어 버전의 개수가 무한대(0)로 설정된 경우에는 영원히 데이터 파일에 존재하게 되며, 버전 개수를 1로 설정한 경우 compaction 수행 시 바로 삭제된다. 버전이 존재하는 경우에는 삭제된 이전의 데이터도 앞에서 설명한 get() 기능을 이용하여 조회할 수 있다.

```

private void removeRow() throws IOException {
    NTable ntable = NTable.openTable(nconf, tableName);
    ntable.removeRow(new Row.Key("RK1"), System.currentTimeMillis());
}

```

2.3. Map&Reduce

Neptune 테이블에 저장된 데이터를 이용하여 분석 작업을 할 경우 분산/병렬 컴퓨팅 플랫폼을 이용한다. Neptune의 경우 default로 사용하는 컴퓨팅 플랫폼은 Hadoop으로 Hadoop Map&Reduce 연동을 위해 AbstractTabletInputFormat 클래스와 Map&Reduce 작업 시 JOIN 연산이 가능한 TableJoinInputFormat 클래스를 제공한다.

Map&Reduce 작업을 수행하는 방법은 Hadoop의 기본적인 Map&Reduce 작업과 동일하며 입력 데이터로 파일이 아니라 테이블을 사용하기 때문에 AbstractTabletInputFormat 또는 TableJoinInputFormat을 사용한다.

Table을 입력하는 MapReduce 작업의 경우 AbstractTabletInputFormat을 상속받은 클래스를 지정한다. AbstractTabletInputFormat는 세 개의 메소드 abstract 메소드를 가지고 있으며 이를 구현하면 된다.

```
public RowFilter getRowFilter(JobConf jobConf)
```

TableScanner에서 데이터를 scan하는 조건을 지정한다. 컬럼 명, 버전의 개수 등을 지정할 수 있다.

```
public String getTableName(JobConf jobConf)
```

입력으로 처리할 테이블 명을 반환한다.

```
public boolean isRowScan(JobConf jobConf)
```

map() 으로 전달되는 데이터를 Row 단위로 할 것인지 Cell 단위로 할 것인지를 지정한다. 하나의 Row에 많은 정보가 저장되어 있는 경우 Row 단위로 지정할 경우 메모리에 부하를 줄 수 있다.

AbstractTabletInputFormat를 구현한 DefaultTabletInputFormat을 제공하고 있으며 DefaultTabletInputFormat을 사용하기 위해서는 JobConf에 다음과 같은 파라미터를 설정하면 된다.

AbstractTabletInputFormat.INPUT_TABLE: 입력 테이블명

AbstractTabletInputFormat.INPUT_COLUMN_LIST: 테이블 컬럼명, “,”로 구분

Map&Reduce를 이용하여 Row.Key-Cell.Key 관계를 Cell.Key가 Row.Key가 되고 Row.Key가 Cell.Key가 되는 inverted table을 구성하는 프로그램을 다음과 같이 만들 수 있다.

```
public class FirstMapReduce {
    public void main(String[] args) throws Exception {
        //Output 테이블 생성
        NConfiguration conf = new NConfiguration();
        String tableName = "InvertedTable";
        TableSchema tableSchema = new TableSchema();
        tableSchema.addColumn("Column1");
        if(!NTable.existsTable(conf, tableName)) {
            NTable.createTable(conf, tableSchema);
        }

        JobConf jobConf = new JobConf(FirstMapReduce.class);
        jobConf.setJobName("FirstMapReduce");

        //<Mapper>
        //Mapper 클래스 지정
        jobConf.setMapperClass(FirstMapReduceMapper.class);
        //InputFormat을 TabletInputFormat으로 지정
        jobConf.setInputFormat(FirstMapReduceInputFormat.class);
        jobConf.setMapOutputKeyClass(Text.class);
        jobConf.setMapOutputValueClass(Text.class);
        //</Mapper>

        //<Reducer>
        String outputPath = "temp/FirstMapReduce";
        jobConf.setOutputPath(new Path(outputPath));
        //Reducer 클래스 지정
        jobConf.setReducerClass(FirstMapReduceReducer.class);
        jobConf.setOutputKeyClass(Text.class);
        jobConf.setOutputValueClass(Text.class);
        //Map갯수를 Reduce 갯수와 동일하게 한다.
        NTable ntable = NTable.openTable(conf, "SampleTable1");
        TabletInfo[] tabletInfos = ntable.listTabletInfos();
        jobConf.setNumReduceTasks(tabletInfos.length);
        //Reduce의 경우 Tablet에 데이터를 저장하는 작업을 수행하기 때문에
        //Task 실패 시 반복하지 않도록 재시도 횟수를 0으로 설정한다.
        jobConf.setMaxReduceAttempts(0);
        //</Reducer>
    }
}
```

```

//Job 실행
JobClient.runJob(jobConf);

//Temp 삭제
NeptuneFileSystem fs = NeptuneFileSystem.get(new NConfiguration());
fs.delete(new GPath(outputPath), true);
}
}

```

먼저 Job에 대한 환경 구성을 위해 Job을 실행 시키는 클래스(FirstMapReduce)를 만든다. Job 환경 설정에는 Mapper, Reducer 클래스와 InputFormat 등을 지정한다. Job 환경 설정 시 주의할 사항은 Reduce Task의 개수와 Reduce 수행 시 오류 발생시 어떻게 할 것인가에 대한 설정이다. Reduce Task의 개수가 너무 작으면 전체 Job의 수행 속도가 떨어지게 된다. 예제에서는 입력 테이블의 Tablet의 개수와 동일하게 설정하였다. AbstractTabletInputFormat를 이용할 경우 Map의 개수는 테이블의 Tablet 수와 동일하다. 예제 프로그램의 경우 map() function을 수행해도 데이터의 크기는 입력 데이터의 크기와 동일하기 때문에 Reduce Task의 개수도 Map Task와 동일하게 하였다.

```

static class FirstMapReduceMapper
    implements Mapper<Row.Key, Row, Text, Text> {
    @Override
    public void map(Row.Key key, Row value, OutputCollector<Text, Text> output,
        Reporter reporter) throws IOException {
        for(Cell eachCell: value.getFirstColumnCells()) {
            output.collect(new Text(eachCell.getKey().toString()), new Text(key.toString()));
        }
    }

    @Override
    public void configure(JobConf job) {
    }

    @Override
    public void close() throws IOException {
    }
}

static class FirstMapReduceReducer
    implements Reducer<Text, Text, Text, Text> {
    NTable ntable;
    IOException exception;

    @Override
    public void reduce(Text key, Iterator<Text> values,
        OutputCollector<Text, Text> output, Reporter reporter)
        throws IOException {
        if(exception != null) {
            throw exception;
        }
        Row.Key rowKey = new Row.Key(key.toString());

        Row row = new Row(rowKey);
        while(values.hasNext()) {

```

```

        Text text = values.next();
        row.addCell("Column1",
            new Cell(new Cell.Key(text.toString()), null));
    }

    ntable.put(row);
}

@Override
public void configure(JobConf job) {
    try {
        NConfiguration conf = new NConfiguration();
        ntable = NTable.openTable(conf, "InvertedTable");
        if(ntable == null) {
            throw new IOException("No InvertedTable table");
        }
    } catch (IOException e) {
        exception = e;
    }
}

@Override
public void close() throws IOException {
}
}

```

Mapper의 map() 메소드로 전달되는 파라미터에서 key에는 Row.Key가 value에는 Row가 전달된다. 이것은 AbstractTabletInputFormat에서 Row Scan mode로 설정했기 때문이다. Row Scan mode를 false로 하는 경우에는 ScanCell이 전달된다.

AbstractTabletInputFormat을 상속받은 InputFormat는 다음과 같이 구성할 수 있다.

```

static class FirstMapReduceInputFormat
    extends AbstractTabletInputFormat {
    public FirstMapReduceInputFormat() throws IOException {
        super();
    }

    @Override
    public RowFilter getRowFilter(JobConf jobConf) {
        RowFilter rowFilter = new RowFilter();
        rowFilter.addCellFilter(new CellFilter("Column1"));
        return rowFilter;
    }

    @Override
    public String getTableName(JobConf jobConf) {
        return "SimpleTable";
    }

    @Override
    public boolean isRowScan(JobConf jobConf) {
        return true;
    }
}

```

이런 Scan 조건이 단순한 경우에는 다음과 같이 DefaultTabletInputFormat를 사용할 수도 있다.

```

jobConf.set(AbstractTabletInputFormat.INPUT_TABLE, "SampleTable1");
jobConf.set(AbstractTabletInputFormat.INPUT_COLUMN_LIST, "Column1");
jobConf.setInputFormat(DefaultTabletInputFormat.class);

```

TableJoinInputFormat의 경우 다음과 같은 파라미터 설정을 통해 두 개의 테이블의 데이터를 MapReduce에서 처리할 수 있다.

파라미터명	설명
TableJoinInputFormat.PIVOT_TABLE	Mapper의 입력 테이블 중 JOIN 기준이 되는 테이블
TableJoinInputFormat.PIVOT_TABLE_COLUMN_LIST	JOIN 기준이 되는 테이블의 칼럼 칼럼이 여러 개인 경우 “,”로 구분
TableJoinInputFormat.TARGET_TABLE	JOIN 대상이 되는 테이블 명
TableJoinInputFormat.TARGET_TABLE_COLUMN_LIST	JOIN 대상이 되는 테이블의 칼럼 명 칼럼이 여러 개인 경우 “,”로 구분

3. Shell 사용법

Neptune에 테이블을 생성하거나 데이터를 조회하기 위해 Neptune 클라이언트 API를 이용한 프로그램 이외에 shell을 제공함으로써 쉽게 Neptune의 데이터를 조작할 수 있다.

Neptune Shell은 다음과 같은 방법으로 실행한다.

```

[neptune]$ pwd
/home/neptune
[neptune]$ bin/neptune shell
Neptune Shell version 1.0
Type 'help;' for usage.

Neptune>

```

Neptune Shell에서 제공하는 명령은 다음과 같다.

명령어	설명	구문
SHOW TABLES	테이블 목록을 나타낸다.	SHOW TABLES
DESC	테이블의 칼럼 정보를 나타낸다.	DESC <table_name>
CREATE TABLE	테이블을 생성한다.	CREATE TABLE <table_name> <column_name>
DROP TABLE	테이블을 삭제한다.	DROP TABLE <table_name>
TRUNCATE TABLE	테이블의 모든 데이터를 삭제한다.	TRUNCATE TABLE <table_name>
REPORT TABLE	테이블의 상태 정보 및 테이블 정보를 나타낸다.	REPORT TABLE <table_name>

SELECT	테이블의 데이터를 조회한다.	<pre> SELECT <[* column_name1, ...> FROM <table_name> [WHERE rowkey='<row_key>'] [AND column='<column_key>'] [TABLET_ROWS=# rows each tablet] [TIMESTAMP <start(yyyyMMddHHmss)>, <end(yyyyMMddHHmss)>] </pre>
INSERT	테이블에 데이터를 저장한다.	<pre> INSERT INTO <table_name> (column_name1[, 'column_name2', ...]) VALUES (('key1', 'val1')[, ('key1', 'val1'), ...]) WHERE rowkey='row_key'; </pre>
DELETE	테이블 데이터를 삭제한다.	<pre> DELETE * ((column_name, 'column_key') [, (column_name, 'column_key') ...]) FROM <table_name> WHERE rowkey='row_key'; </pre>

각 명령어의 사용 방법은 Shell에서 "Help <명령어>"를 입력하면 확인할 수 있다.

Neptune shell은 neptune 자체 shell에서뿐만 아니라 명령행으로도 수행이 가능하다.

bin/Neptune shell -q "query"