10. MapReduce

1. 맵 리듀스 개념
2. Hadoop의 HDFS, MapReduce
3. MongoDB의 MapReduce
4. Word Count MapReduce 구현
5. Inverted Search Index 구현
6. 통계함수 MapReduce 구현

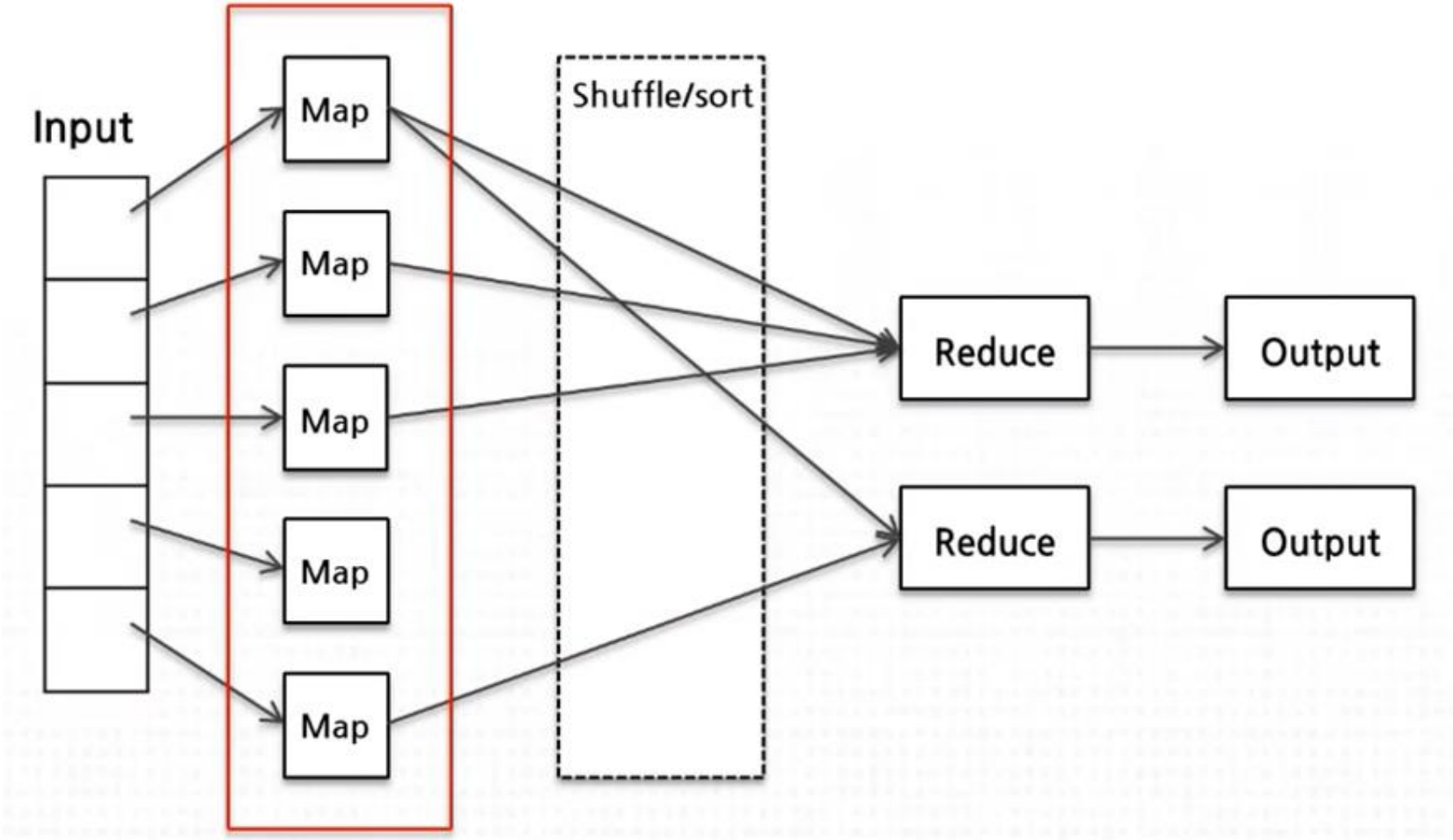
1. 맵 리듀스 개념

❖ MapReduce

- 대용량의 데이터를 안전하고 빠르게 처리하기 위한 방법
- 한 대 이상의 하드웨어를 활용하는 분산 프로그래밍 모델
- 2004년 OSDI컨퍼런스에서 "MapReduce : Simplified Data Processing on Large Clusters"란 논문을 통해 발표
- 루씬(Lucene)의 개발자였던 Doug Cutting이 2006년 Hadoop 이라는 오픈소스 프로젝트를 진행
  - ✓Hadoop은 HDFS(Hadoop File System)이라는 대규모 분산 파일시스템을 구축하여 탁월한 성능과 안정성을 보여줌
  - ✓맵리듀스는 대용량 파일에 대한 로그 분석, 색인 구축, 검색에 탁월한 능력을 보여줌

1. 맵 리듀스 개념

- 맵 리듀스 동작원리



1. 맵 리듀스 개념

❖ 맵리듀스 특징

- 맵 리듀스는 데이터를 분산하여 연산하고 다시 합치는 기술
- 맵과 리듀스 단계로 나누고 맵 단계는 입력과 출력으로 Key-value의 형태를 가지고 있음
- 데이터를 섞어서 병합하고 리듀스 함수를 통해 최종적으로 결과를 제공함
- 맵과 리듀스는 사용자가 임의로 코딩이 가능한 형태로 제공
- 분산을 통해 분할된 조각으로 처리한뒤 다시 모아 훨씬 짧은 시간에 계산을 완료함
- 분할된 조각이 작으면 작을수록 부하 분산에 더 좋은 효과를 냄
- 너무 과하게 데이터를 분할할 경우 맵을 생성하기 위한 태스크의 오버헤드가 커지기 때문에 역효과가 날 수도 있음

❖ 맵 리듀스 장점/단점

장점	단점
▪ 분산모델을 감추어 대용량 처리를 단순하게 만듦 ▪ 특정 데이터 모델이나, 스키마에 의존적이지 않은 유연성 ▪ 저장구조의 독립성 ▪ 높은 확장성	▪ 기존 RDBMS보다 불편한 스키마와 질의 ▪ DBMS와 비교하여 낮은 성능 ▪ 개발 환경의 불편함과 운영 노하우 부족 ▪ 단순한 데이터 처리

❖ 맵 리듀스와 RDBMS의 비교

	RDBMS	맵리듀스
데이터 크기	기가바이트	페타바이트
액세스	대화형과 일괄처리	일괄처리
업데이트	여러 번 읽고 쓰기	한번 쓰면, 여러 번 읽기
구조	고정 스키마	동적 스키마
무결성	높음	낮음
확장성	비선형	선형

❖ 맵 리듀스와 RDBMS의 관계

- 맵 리듀스는 대용량의 처리를 위해 기존의 데이터베이스(RDBMS)를 보완할 수 있음
- 대화형으로 처리되는 문제나 업데이트에 적합한 RDBMS에 비해서 일괄처리 방식으로 전체 데이터 셋을 분석할 필요가 있는 문제에 적합
- 현재 클라우드 기반의 대부분 서비스들은 대용량 처리를 위해서 맵 리듀스 기반의 하둡과 같은 형태로 구축되어 운영되고 있음

2. Hadoop의 HDFS, 맵 리듀스

❖ HDFS(Hadoop Distribute File System) 개요

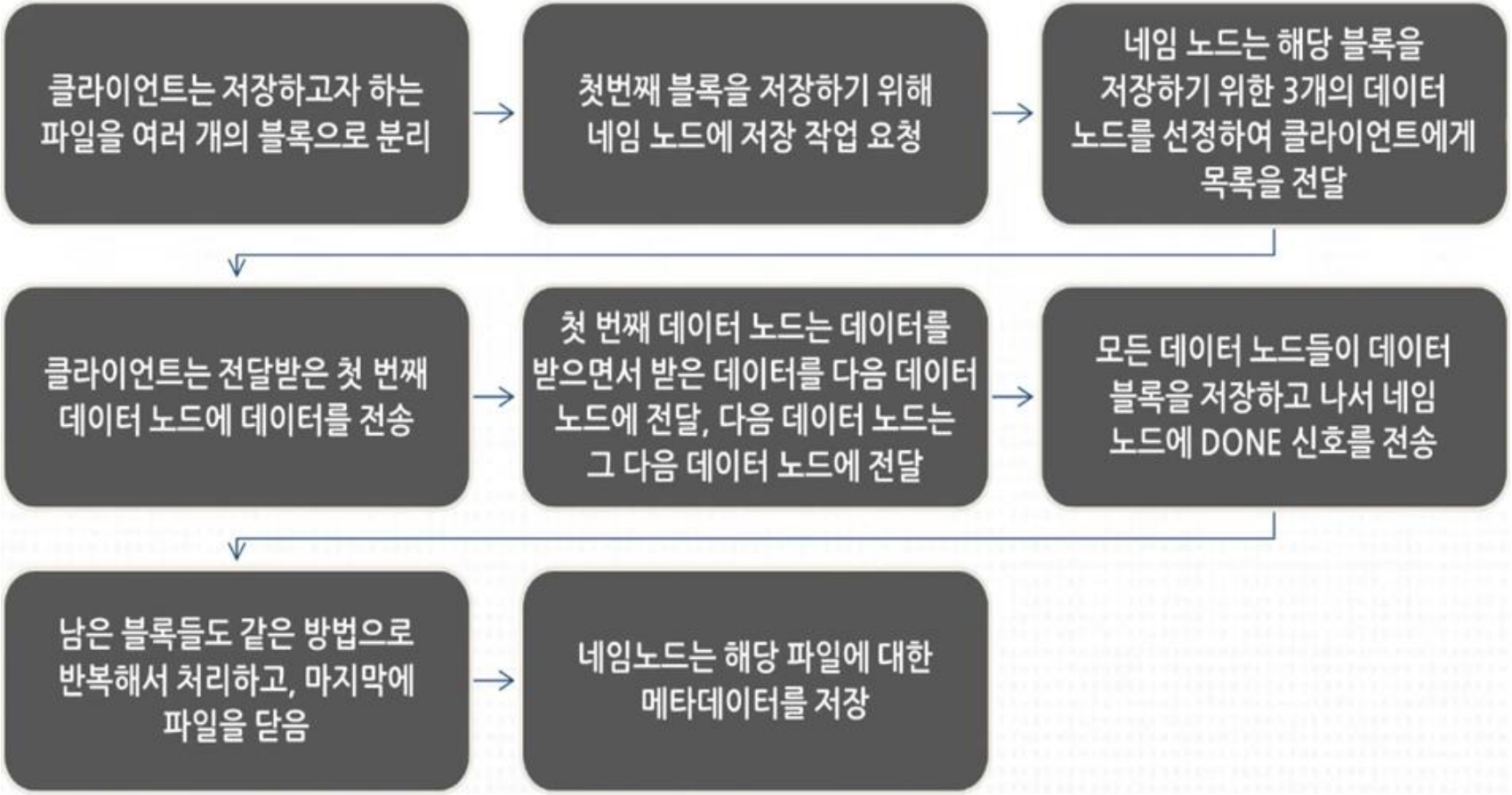
구성 요소

- 클라이언트 (Client) : 데이터 송수신 요청
- 네임 노드 (Namenode) : HDFS 전체 시스템 제어
- 데이터 노드 (Datanode) : 수천대의 서버로 구성, 데이터 저장 역할

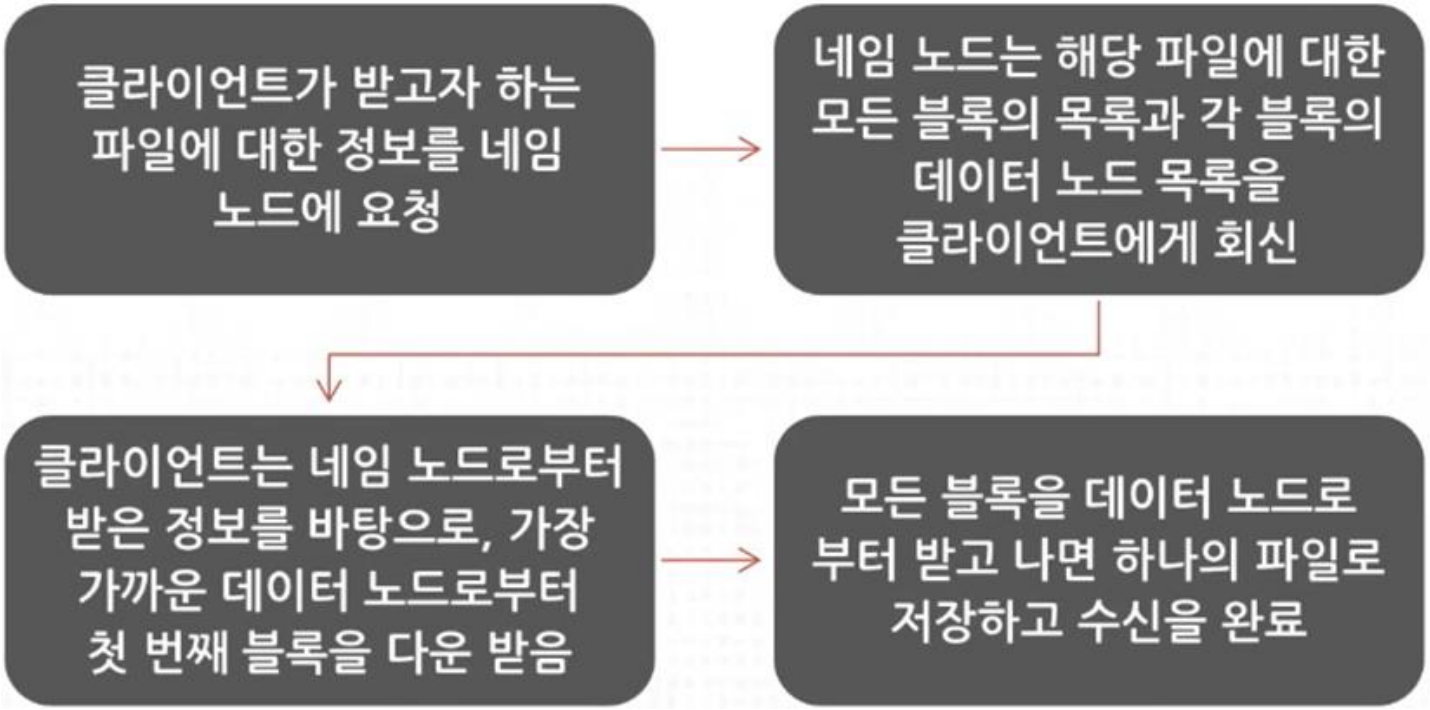
데이터 처리 원칙

- Block size : 파일을 저장하는 단위 (64MB or 128MB)
- Replication : 모든 블록은 여러 노드에 분산되어 저장 (기본 3개)

❖ HDFS 파일 전송 과정


클라이언트는 저장하고자 하는 파일을 여러 개의 블록으로 분리
첫번째 블록을 저장하기 위해 네임 노드에 저장 작업 요청
네임 노드는 해당 블록을 저장하기 위한 3개의 데이터 노드를 선정하여 클라이언트에게 목록을 전달
클라이언트는 전달받은 첫 번째 데이터 노드에 데이터를 전송
첫 번째 데이터 노드는 데이터를 받으면서 받은 데이터를 다음 데이터 노드에 전달, 다음 데이터 노드는 그 다음 데이터 노드에 전달
모든 데이터 노드들이 데이터 블록을 저장하고 나서 네임 노드에 DONE 신호를 전송
남은 블록들도 같은 방법으로 반복해서 처리하고, 마지막에 파일을 닫음
네임노드는 해당 파일에 대한 메타데이터를 저장

❖ HDFS 파일 수신 과정


클라이언트가 받고자 하는 파일에 대한 정보를 네임 노드에 요청
네임 노드는 해당 파일에 대한 모든 블록의 목록과 각 블록의 데이터 노드 목록을 클라이언트에게 회신
클라이언트는 네임 노드로부터 받은 정보를 바탕으로, 가장 가까운 데이터 노드로부터 첫 번째 블록을 다운 받음
모든 블록을 데이터 노드로부터 받고 나면 하나의 파일로 저장하고 수신을 완료

❖ HDFS의 오류 및 장애 대응

- 네임노드 오류: 모든 클러스터가 죽는 문제 발생(치명적)

데이터 노드 오류 처리

- 데이터 노드는 네임 노드에 3초 단위로 heart beat를 전송
- 네임 노드가 특정 데이터 노드의 heart beat를 10분 동안 받지 못하면 해당 데이터 노드가 죽었다고 판단함

데이터 송수신 시 오류 처리

- 클라이언트(송신자)가 데이터 노드에 데이터를 전송할 때마다 데이터 노드는 ACK 응답을 함
- ACK 응답이 오지 않으면 여러번 시도해보고 노드가 죽었거나 네트워크 오류로 판단함

❖ HDFS의 오류 및 장애 대응

데이터 체크섬(checksum) 확인

- 데이터 전송시 해당 데이터에 대한 체크섬을 같이 보냄
- 데이터 노드에서 데이터를 하드디스크에 저장할때 체크섬도 같이 저장
- 데이터 노드는 주기적으로 네임 노드에 블록 리포트(Block Report)를 전송
- 데이터 노드가 블록 리포트를 보내기 전에 체크섬이 맞는지 확인하고, 손상된 블록은 제외하고 블록 목록을 작성하여 보냄
- 네임 노드는 블록 리포트를 통해 문제가 발생한 데이터 블록을 알아내고 조치할 수 있음

❖ 불완전 복제 방지

- 네임 노드는 “블록 목록”과 “데이터 노드 위치 목록”을 관리
- 2개의 목록을 지속적으로 업데이트하며 수시로 모니터링을 진행함
- 장애가 발생한 노드를 찾으면, 블록 목록과 데이터 노드 목록을 업데이트하고 오류가 발생한 데이터 노드와 블록을 삭제함
- 오류 발생으로 복제 개수가 완전하지 않은 블록을 “불완전 복제”라고 함
- 불완전 복제 블록을 제거하기 위해, 데이터 노드에 새로운 복제소로 복사할 것을 요청하여 복제 개수를 맞춤

❖ 복제 위치 선정 전략

- 클러스터는 여러 개의 데이터 노드를 가지는 랙으로 분리됨
- 첫번째 복제소는 클라이언트 데이터가 같은 렉의 노드에 있으면 첫번째 복제소로 선정하고, 그렇지 않으면 랜덤으로 복제 노드를 선택함
- 첫번째 노드와는 다른 렉에서 2개의 다른 데이터 노드를 선택함
- HDFS는 적어도 1개의 복제소에 대해 확실한 보장을 하기 위해, 최적의 복제소 선정이 중요함

❖ Hadoop 맵 리듀스

설계 특성

- 분산컴퓨팅에 적합한 함수형 프로그래밍
- 배치형 데이터 처리 시스템
- 어플리케이션 로직의 주요관심사를 파악, 많은 요소를 반영

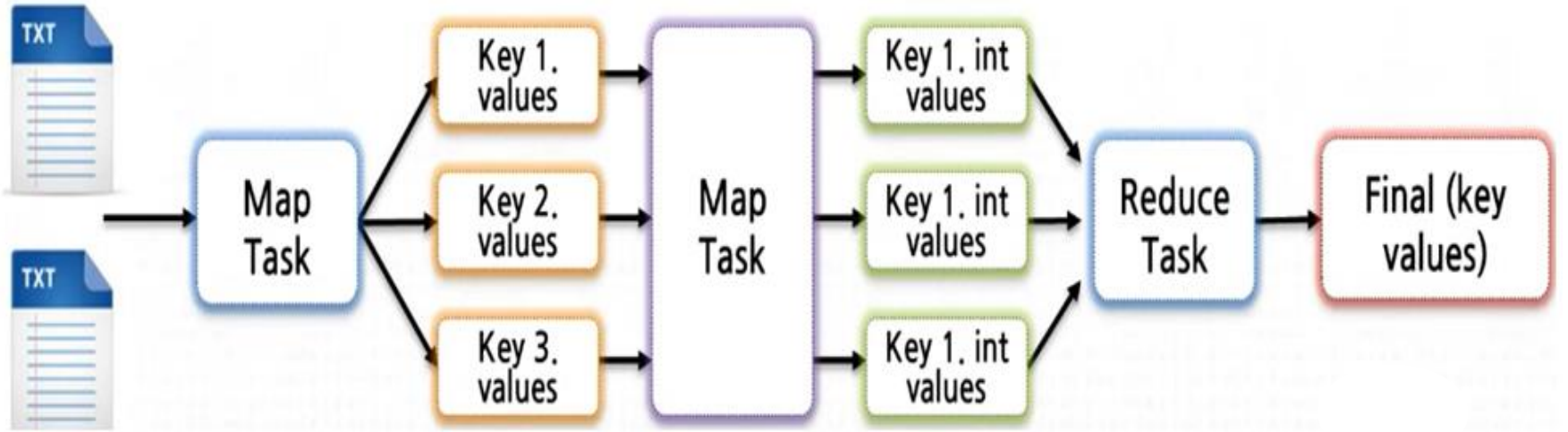
주요 기능

- 자동화된 병렬처리 및 분산처리
- Fault-tolerance(내고장성,결함허용)
- 상태 및 모니터링 툴들
- 프로그래머를 위한 추상클래스

❖ 맵 리듀스 과정

- 원본데이터(파일,DB레코드)는 map 함수에 의해서 <key, value> 쌍으로 전환됨
- map() 함수 :입력을 출력 key와 관련되는 1..N개의 <key,value>를 생성
- Map 단계 다음에서 출력 key의 중간 value 들은 하나의 리스트로 합쳐짐
- reduce() 함수 :같은 출력 key를 가지는 final value로 중간 value들을 통합

❖ 맵 리듀스 과정


TXT
TXT
Map Task
Key 1. values
Key 2. values
Key 3. values
Map Task
Key 1. int values
Key 1. int values
Key 1. int values
Reduce Task
Final (key values)

❖ 맵 리듀스 특징

- map() 함수들은 병렬(parallel)로 작동하며, 여러 입력 자료셋으로 부터 여러 중간 value 들을 생성
- reduce() 함수들도 역시 병렬로 작동하며, 출력 key를 기준으로 각각 작업을 수행
- 모든 value 들은 독립적으로 처리됨
- 병목 : reduce는 map 단계가 완료되지 않으면 시작할 수 없음
- MapReduce는 많은 영역에서 개발자에서 유용한 추상화 기능을 제공
- 대용량의 계산을 아주 심플하게 만들어 줌
- MapReduce의 함수화된 프로그래밍 패러다임은 대용량의 어플리케이션에도 적용될 수 있음

3. MongoDB 맵 리듀스

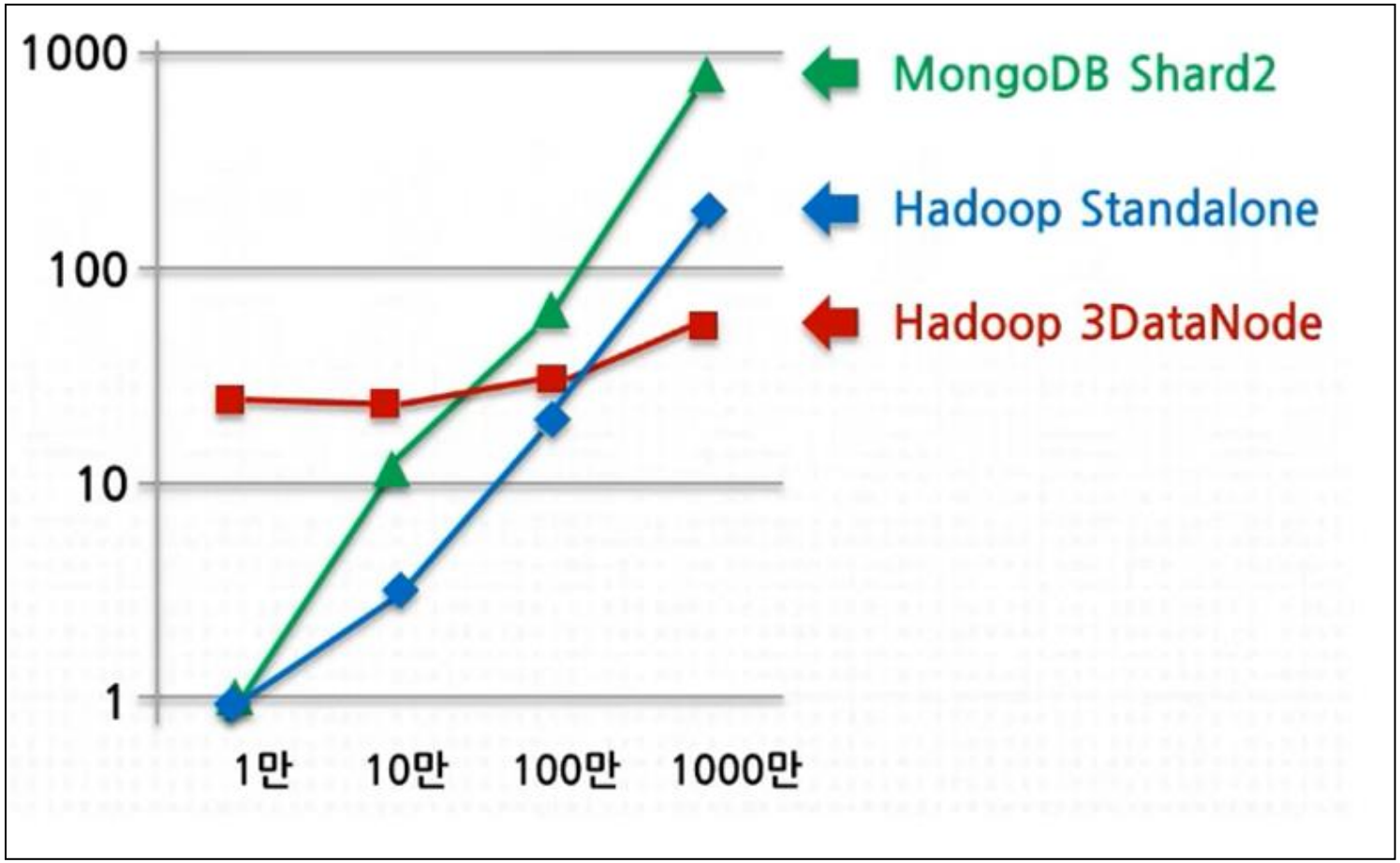
❖ MongoDB 맵 리듀스 특징

- MongoDB는 빅데이터 처리를 위한 다양한 기능들을 제공함
- MongoDB는 관계형 데이터베이스에서 흔히 제공하는 데이터 집계 함수들을 지원하고 있지 않음
  (예: 평균을 구하는 AVG나 RANK, DENSE_RANK등을 제공하지 않음)
- 기존 관계형 데이터 베이스 입장에서 보면 굉장히 불편함
- MongoDB에서는 집계 함수를 지원하지 않는 대신 mapreduce를 통해서만 집계 구현 가능
  (다른 솔루션의 MapReduce와는 다른 특징)

❖ Hadoop과 MongoDB 맵 리듀스 성능 비교

데이터 건 수	파일 크기	Hadoop		MongoDB Shard 2
		Standalone	Distributed	
1만 건	2MB	1초	23초	1초
10만 건	25MB	3초	22초	12초
100만 건	248MB	20초	29초	65초
1000만 건	2,480MB	183초	53초	805초
장비	-	1대	4대	5대

❖ Hadoop과 MongoDB 맵 리듀스 성능 비교



4. Word Count MapReduce 구현

❖ Word Count

- 입력 파일의 텍스트 내용에 포함된 단어의 수를 세는 프로그램
  - ✓입문용 MapReduce 프로그래밍 예제
  - ✓입력 파일의 사이즈가 작을 경우에는 일반 프로그램이 더 빠를 수 있지만, 입력 파일의 크기가 크면 클 수록 MapReduce로 구동 시키는 프로그램이 더 빠른 결과를 얻을 수 있음

함수형 프로그래밍

- 2가지 함수의 사용자 인터페이스 구현
- map (in_key, in_value) -> (inter_key, inter_value) list
- reduce (inter_key, inter_value list) -> (out_key,out_value) list

4. Word Count MapReduce 구현

❖ Split Mapper

- 데이타셋을 key,value의 리스트로 변경하는 map() 함수

```
let map( k,v ) =
   foreach word w in v : emit(w, 1 )

('text', 'read a book')    ->  ('read', 1), ('a', 1),  ('book', 1)
```

4. Word Count MapReduce 구현

❖ sum Reducer

```
let reduce( k,vals ) =
    sum = 0

foreach int v in vals :

    sum += v

  emit( k, sum )


('A', [42,100,312])   -> ('A', 454)

('B', [12,6   ,-2])    -> ('B', 16)
```

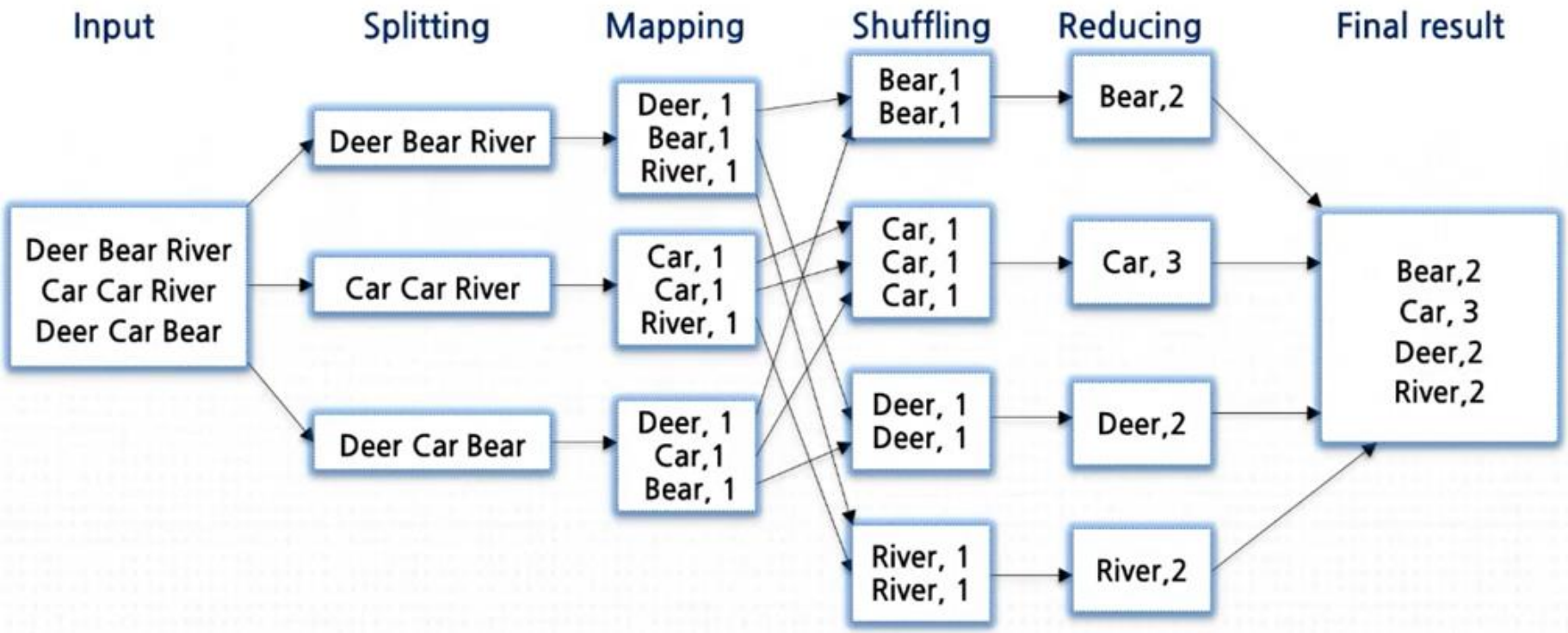
4. Word Count MapReduce 구현

❖ Word Count 프로그램 로직

```
map(String input_key, String  input_value):
// input_key : document name
// input_value : document contents
for each word w in input_value:
      emit(w, 1);

reduce(String output_key, Iterator<int> intermediate_values):
// output_key : a word
// output_values : a list of counts
int result =0;
for each v in intermediate_value:
      result += v;
emit(output_key, result);
```

4. Word Count MapReduce 구현

❖ Word count 수행 과정



4. Word Count MapReduce 구현

❖ 맵 리듀스 실행

- MapReduce 데이터 입력

```
➢ db.words.save({text : "read a book"});
➢ db.words.save({text :"write a book"});
```

- map() 함수 구현

```
map=function(){
    var res=this.text.split(" ");
    for(var i in res){
        key={word :res[i]};
        value={count :1};
        emit(key, value);
    }
}
```

- reduce() 함수 구현

```
reduce=function(key, values){
    var totalcount=0;
    for(var i in values){
        totalcount=values[i].count+totalcount;
    }
    return {count:totalcount};
}
```

4. Word Count MapReduce 구현

❖ 맵 리듀스 실행

- MapReduce 명령 실행

```
db.words.mapReduce(map, reduce, "wordcount");
```

- MapReduce 실행 결과 확인

```
db.wordcount.find()
```

5. MapReduce 구현

❖ 예제: 1

```
db.stud.insert({name:"matt", score:95, grade:"A"});
db.stud.insert({name:"lara", score:83, grade:"B"});
db.stud.insert({name:"todd", score:89, grade:"B"});
db.stud.insert({name:"ammy", score:99, grade:"A"});
```

```
var mapper=function(){
        emit(this.grade, {score:this.score});

var reducer=function (key, values){
        var sum=0;
        values.forEach(function(doc){
                sum+=doc.score;
        });
        return {score:sum};
};

db.stud.MapReduce(mapper, reducer, {out:{inline:1}});
```

inline:1 = 컬렉션 없이 임시테이블에 바로 출력

4. MapReduce 구현

❖ 문법

```
db.collection_name.mapReduce(
    <map>  #map 함수 이름
    <reduce> #reduce 함수 이름
    {
        <out>  # 실행결과를 저장할 collection 명
        <query>     # 검색조건
        <sort> # sorting 조건
        <Limit>     # 데이터 검색 조건
        <finalize>   #실행결과를 집계하게 될 함수 이름
        <scope>
        <jsMode>
        <verbose> # 진행 메시지 표시})
```

reduce결과로 다시 한번 더 수행할 때

① map function : 해당 컬랙션에서 분석 대상 필드를 emit 함수를 이용하여 정의하는 함수
② reduce function: 컬렉션에서 데이터를 분석 및 통계 작업을 수행하는함수
③ Finalize: 처리된 결과를 집계하는 함수
④ mapReduce 함수 : map Function과 Reduce Function에 의해 리턴된 데이터를 이용해 결과(output)를 출력하는 함수

Filnalize 함수

- 맵리듀스 최종결과에 추가적인 처리를 더하거나 최종 결과를 변경하고자 할때 사용
- 형식

```
function(key, reducdValue) {
        …
        return modifiedObject;
}
```

```
var mapper=function(){
        emit(this.grade, {count : 1, score : this.score});

var reducer=function (key, values){
        var sum=0;
        var cnt=0;
        values.forEach(function(doc){
                cnt+=doc.count;
                sum+=doc.score;
        });
        return {count : cnt, score : sum};
};
var Finalize=function(key, value){
        value.avg=value.score/value.count;
        return value;
}
db.stud.mapReduce(mapper, reducer,
        {out : { inline : 1}, finalize:Finalize });
```

MapReduce

```
db.order.drop()
db.order.insert({cust_id:"A2012001",
                order_date:new Date("Oct 01,2012"),
                status: "A",
                price:250,
                items:[{itme_name:"Bunny Boot", gty: 5, price:2.5},
                        {itme_name:"Sky Ploe", gty: 5, price:2.5}]})

db.order.insert({cust_id:"A2012001",
                order_date:new Date("Sep 15, 2012"),
                status: "A",
                price:1125,
                items:[{itme_name:"Bunny Boot", gty: 15, price:2.5},
                        {itme_name:"Sky Ploe", gty: 5, price:2.5}]})
```

ex1) A212001 고객번호를 가진 고객의 주문 총금액 집계

```
var map_function=function() {emit(this.cust_id, this.price);}

var reduce_function=function(keyCustId, valuesPrices){
        return Array.sum(valuesPrices);};

db.order.mapReduce(
        map_function,
        reduce_function,
        {out:"order_cust_total"})

db.order_cust_total.find()
```

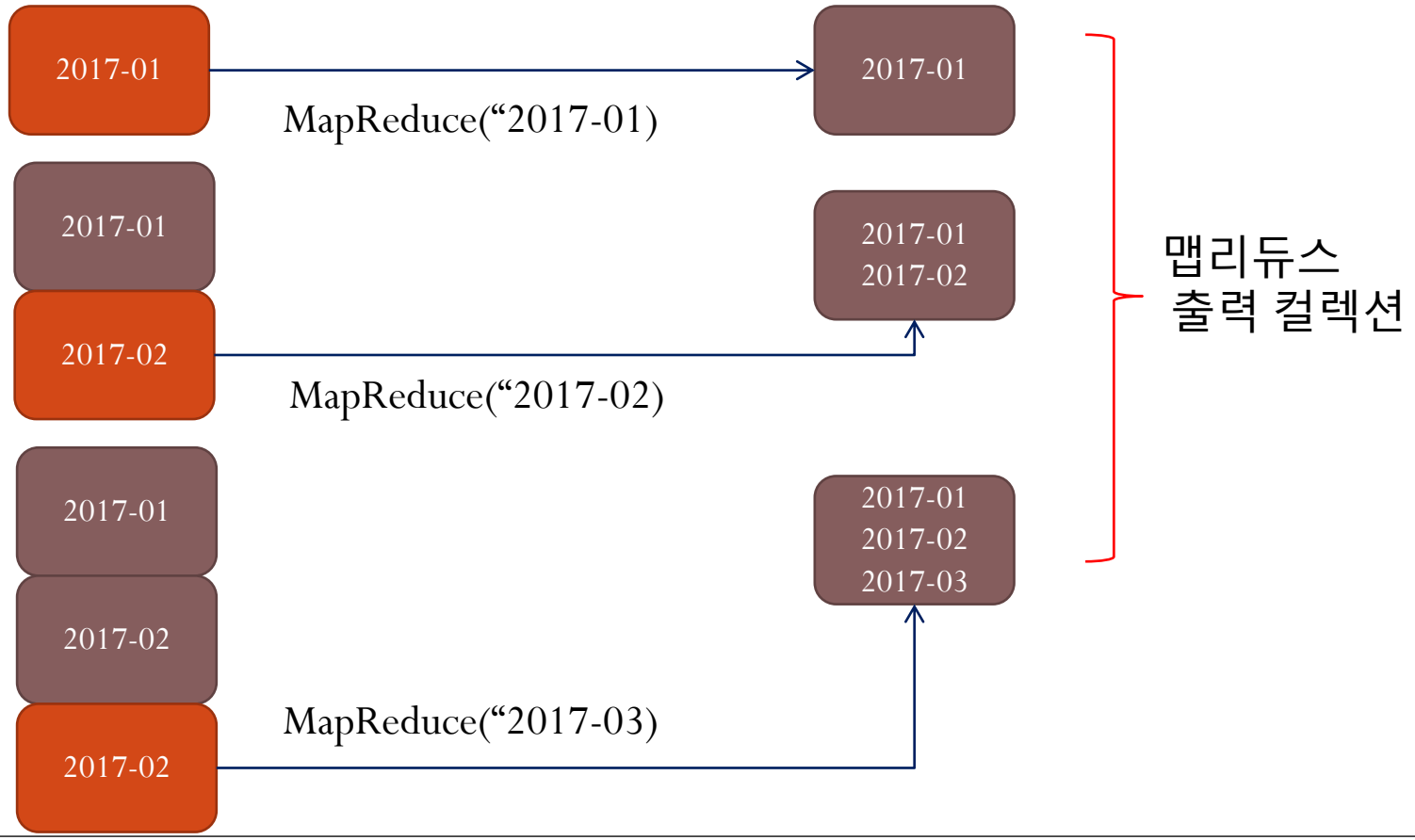
ex2) 제품명별 주문 수량(qty)평균을 집계

```
var map_function=function(){
        for(var idx=0;idx<this.items.length; idx++){
                var key=this.items[idx].item_name;
                var value={count :1, qty:this.items[idx].qty};
                emit(key, value);
        }
}

var reduce_function=function(keySKU, valuesCountObjects){
        reducedValue={count: 0, qty : 0}
        for(var idx=0; iex<valuesCountObjects.length; idx++){
                reducedValue.count+=valuesCountObjects[idx].count;
                reducedValue.qty+=valuesCountObjects[idx].qty;
                return reducedValue;
        }
}
```

```
var finalize_function=function(key, reducedValue){
        reduceValue.average=reducedValue.gty/reducedValue.count;
        return reduceValue;
}

db.order.mapReduce(
        map_function,
        reduce_function,
        {
                out:{merge:"map_reduce_example"},
                query:{order_date:{$gt: new Date('01/01/12012')}},
                finalize:finalize_function
        }
)
```

증분 맵리듀스(incremental MapReduce)

- 증분 맵리듀스 작업
  - 특정 도규먼트가 계속 증가할때, 매일 맵리듀스를 수정하는 것이 아니라 증가한 만큼의 도규먼트만 맵리듀스를 수행해서 그 결과를 기존 맵리듀스 결과와 병합하는 방식으로 작업하는 것



실습

```
db.login_history({name : "matt", login_data : ISOData('2017-01-03 14:17:00')});
db.login_history({name : "lara", login_data : ISOData('2017-01-09 10:07:00')});
db.login_history({name : "todd", login_data : ISOData('2017-01-12 23:21:00')});
db.login_history({name : "matt", login_data : ISOData('2017-01-22 02:54:00')});
db.login_history({name : "todd", login_data : ISOData('2017-01-28 22:21:00')});
```

```
var mapper=finction(){
        emit(this.name, {login_count : 1});
};

var reducer=function(key, values){
        var count=0;
        values.forEach(finction(doc){
                count+=doc.login_count;
        });
        return {login_count : count};
};

db.login_history.mapReduce(mapper, reducer, {out : "login_count"});
db.login_count.find();
```

```
db.login_history({name : "matt", login_data : ISOData('2017-01-03 14:17:00')});
db.login_history({name : "lara", login_data : ISOData('2017-01-09 10:07:00')});
db.login_history({name : "todd", login_data : ISOData('2017-01-12 23:21:00')});
db.login_history({name : "matt", login_data : ISOData('2017-01-22 02:54:00')});
db.login_history({name : "todd", login_data : ISOData('2017-01-28 22:21:00')});
```

```
db.login_history({name : "matt", login_data : ISOData('2017-02-03 14:17:00')});
db.login_history({name : "lara", login_data : ISOData('2017-02-09 10:07:00')});
db.login_history({name : "todd", login_data : ISOData('2017-02-12 20:21:00')});
db.login_history({name : "matt", login_data : ISOData('2017-02-22 12:54:00')});
db.login_history({name : "todd", login_data : ISOData('2017-02-28 12:00:00')});

db.login_history.mapReduce(mapper, reducer, {
        query : login_data : {$gte : IOSDate('2017-02-01 00:00:00')}}.
        out : {reduce : "login_count"}} );
db.login_count.find();
```

맵리듀스 함수 개발시 주의 사항

- Map 함수에서 호출하는 emit() 함수의 두번째 인자와 Reduce 함수의 리턴값은 같은 포맷이여야 함
- Reduce 함수와 연산 작업은 멱등(Idempotent)이여야 함

실습1

- emit 두번째 인자와 reduce return의 포맷이 일치하지 않을때

```
db.stud.insert({name:"matt", score:95, grade:"A"});
db.stud.insert({name:"lara", score:83, grade:"B"});
db.stud.insert({name:"todd", score:89, grade:"B"});

var mapper=function(){
        emit(this.grade, {grade: this.grade, score:this.score}});

var reducer=function (key, values){
        var sum=0;
        values.forEach(function(doc){
                sum+=doc.score;
        });
        return {score:sum};
};

db,stud.MapReduce(mapper, reducer, {out:{inline:1}});
```

실습2

```
for(var idx=1; idx<=100000; idx++){
	db.user_scores.insert({_id:idx, name:"matt", score: Math.floor((Math.random()*99)+1)});
};

var mapper=function(){
	emit(this.name, {count :1, score :this.score});
}

var reducer=function(key, values){
	var sum=0;
	var vnd=0;
	values.forEach(function(doc){
		cnt+=1; // cnt+=doc.count;
		sum+=doc.score;
	});
	return {count: cnt; score: sum}
};

db.user_scores.mapReduce(mapper, reducer, {out : {inline:1}});
```

#전체 도큐먼트 건수 계산

```
var sum=0;
var cnt=0;
db.user_scores.find().forEach(function(d){
	cnt++;
	sum+=d.score;
});
```