(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2022-0089470
(43) 공개일자 2022년06월28일

(51) 국제특허분류(Int. Cl.)
G06F 12/02 (2018.01) *G06F 3/06* (2006.01)
(52) CPC특허분류
G06F 12/0253 (2013.01)
G06F 3/061 (2013.01)
(21) 출원번호 10-2020-0180138
(22) 출원일자 2020년12월21일
심사청구일자 없음

(71) 출원인
서강대학교산학협력단
서울특별시 마포구 백범로 35 (신수동, 서강대학교)
(72) 발명자
박성용
서울특별시 서대문구 가재울미래로 2 DMC파크뷰자이 107동 1001호
정연우
경기도 파주시 가온로 205 해솔마을7단지롯데캐슬 720동 1503호
(뒷면에 계속)
(74) 대리인
이지연

전체 청구항 수 : 총 1 항

(54) 발명의 명칭 **스트리밍 환경에서 적응적 상태 기반 객체 메모리 해제를 위한 GC 마킹 방법**

(57) 요 약

본 발명은 스트리밍 환경에서 적응적 상태 기반 객체 메모리 해제를 위한 GC 마킹 방법에 관한 것이다. 상기 방법은, 스트리밍 메모리 테이블과 Join Manager 사이에 State Manager를 구비하고, 상기 State Manager를 이용하여 데이터 연산이 완료된 RDD가 가진 Stateful 정보를 주기적으로 확인하고, 만약 데이터 연산이 완료되어 기존의 데이터 연산에 사용된 RDD 객체들이 사용되지 않는다면 Watermark에 관계없이 해당 RDD에 연결된 Stateful 정보를 GC가 메모리에서 해제할 수 있도록 마킹 태그를 부여하는 것을 특징으로 한다.

대 표 도 - 도5

Algorithm 1 스트리밍 상황에서 적응적 상태 기반 객체 메모리 해제를 위한 마킹 알고리즘

1: ST_{RDD} : RDD pairs required for streaming query
2: ST_i : Each RDD pair
3: $State_{buffer}$: Buffer for checking the completed state query
4: ST_{queue} : Streaming job queue
5:
6: **while** ST_{RDD} exists in ST_{queue} **do**
7: Sort ST_{RDD} by timestamp in recent order
8: $ST_i \leftarrow ST_{RDD}$
9: Join ST_i and insert ST_i to $State_{buffer}$
10: Update *STATE Store* by ST_i
11: **end while**
12:
13: **while** $State^i_{buffer}$ exists in $State_{buffer}$ **do**
14: **if** $State^i_{buffer}$ not referenced **then**
15: Remove the all state in *State Store*
16: Mark the deletion tag to $State^i_{buffer}$
17: **end if**
18: **end while**

(52) CPC특허분류
G06F 3/0662 (2013.01)
G06F 2212/1016 (2013.01)
G06F 2212/7205 (2013.01)
(72) 발명자
김민우
서울특별시 동작구 사당로14길 83 1층

이수연
경기도 구리시 건원대로 92 인창1단지주공아파트 102동 1303호

이 발명을 지원한 국가연구개발사업

과제고유번호	1711060044
과제번호	2017M3C4A7080245
부처명	원천기술개발사업
과제관리(전문)기관명	과학기술정보통신부
연구사업명	차세대정보컴퓨팅기술개발사업
연구과제명	HSA 클라우드에서 SLA 준수 및 에너지-최적 지능형 시스템SW 개발
기 여 율	1/1
과제수행기관명	성균관대학교
연구기간	2017.11.01 ~ 2020.12.31

명 세 서

청구범위

청구항 1

스트리밍 메모리 테이블과 Join Manager 사이에 State Manager를 구비하고, 상기 State Manager를 이용하여 데이터 연산이 완료된 RDD가 가진 Stateful 정보를 주기적으로 확인하고,

만약 데이터 연산이 완료되어 기존의 데이터 연산에 사용된 RDD 객체들이 사용되지 않는다면 Watermark에 관계없이 해당 RDD에 연결된 Stateful 정보를 GC가 메모리에서 해제할 수 있도록 마킹 태그를 부여하는 것을 특징으로 하는 스트리밍 환경에서 적응적 상태 기반 객체 메모리 해제를 위한 GC 마킹 방법.

발명의 설명

기 술 분 야

[0001] 본 발명은 스트리밍 환경에서의 인메모리 데이터 처리 알고리즘에 관한 것으로서, 더욱 구체적으로는 스트리밍 환경에서 인메모리 기반 고속 처리 방식의 이점을 극대화하면서 처리 지연 시간 감소, 성능 저하가 최소화되는 안정적인 메모리 관리를 위하여 적응적 상태 기반 객체 메모리 해제를 위한 GC 마킹 방법에 관한 것이다.

배 경 기 술

[0002] 스마트 팩토리, 실시간 영상 분석 등 대용량 스트리밍 분석을 필요로 하는 기술이 연구되고 있다. 이러한 기술들은 데이터 적재, 고속 병렬 처리, 정제된 데이터를 데이터베이스에 영구 저장하는 등 다양한 응용분야로 확대되고 있다. 오프라인에서 데이터를 처리 및 분석하는 배치 처리와는 달리 스트리밍 처리는 인-메모리 처리 방식을 통해 데이터 간 조인, 집계 등 쿼리를 통해 실시간으로 새로운 데이터를 가공한다. 실시간 스트리밍 연산을 위해 Spark, Flink 등 인메모리 스트리밍 데이터 처리 프레임워크가 등장하였다. 인메모리 처리 방식은 메모리로부터 처리할 데이터를 불러오고 처리 결과를 메모리에 쓰기 때문에 데이터 처리 방식에 비해 처리 속도가 빠른 장점이 있다. 하지만 스트리밍 쿼리와 같이 장시간 메모리에 데이터를 로드하는 작업의 경우, 워크로드에 따라 가용 메모리가 감소하여 잦은 메모리 페이지 교체, 메모리 부족 현상이 발생할 수 있다. 이는 지연 시간 증가와 시스템 중단 등 실시간 응용에 심각한 성능 저하와 서비스 신뢰 감소로 이어진다. 그러므로, 스트리밍 환경에서 인메모리 기반 고속 처리 방식의 이점을 극대화하면서 처리 지연 시간 감소, 성능 저하가 최소화되는 안정적인 메모리 관리 방식이 필요하다.

[0003] 인메모리 스트리밍 데이터 처리는 4가지로 구성된다. 도 1은 인메모리 데이터 처리 프로세스를 도시한 모식도이다. 도 1에 도시된 바와 같이, 데이터를 생산하는 데이터 생산자, 생산된 데이터를 안정적으로 버퍼링하는 데이터 적재, 적재된 데이터를 분석하는 데이터 처리, 처리된 데이터를 영구적으로 저장하는 컴포넌트로 구성된다. 인메모리 처리 방식은 스트리밍 쿼리 중 생성된 연산 정보를 저장하는 객체, 연속된 쿼리에 필요한 중간 데이터 객체를 디스크에 쓰지 않고 메모리에 쓰고 읽기 때문에 처리 성능이 증가된다. 하지만, 메모리에 저장되는 stateful object가 많아질수록 가용 메모리가 감소하여 새로 인입된 데이터를 처리하기 위하여 할당되는 메모리가 감소된다. 작업에 할당되는 메모리가 감소되면 데이터 처리가 느려지고 메모리 병목 현상이 발생하여 전체 시스템의 처리 성능이 감소하는 문제가 있다.

선행기술문헌

특허문헌

[0004] (특허문헌 0001) 한국등록특허공보 제 10-1827395호

발명의 내용

해결하려는 과제

[0005] 전술한 문제점을 해결하기 위한 본 발명의 목적은 스트리밍 쿼리시 발생한 중간 stateful 객체의 지속적인 메모리 저장으로 메모리 부족 현상이 심화되므로 적응적 상태 기반 객체 해제를 위한 GC 마킹 알고리즘을 제공하는 것이다.

과제의 해결 수단

[0006] 전술한 기술적 과제를 달성하기 위한 본 발명의 특징에 따른 스트리밍 환경에서 적응적 상태 기반 객체 메모리 해제를 위한 GC 마킹 방법은, 스트리밍 메모리 테이블과 Join Manager 사이에 State Manager를 구비하고, 상기 State Manager를 이용하여 데이터 연산이 완료된 RDD가 가진 Stateful 정보를 주기적으로 확인하고, 만약 데이터 연산이 완료되어 기존의 데이터 연산에 사용된 RDD 객체들이 사용되지 않는다면 Watermark에 관계없이 해당 RDD에 연결된 Stateful 정보를 GC가 메모리에서 해제할 수 있도록 마킹 태그를 부여하는 것을 특징으로 한다.

발명의 효과

[0007] 본 발명은 실시간 대용량 처리에 쓰이는 인메모리 기반 데이터 처리 방식에서 메모리 충돌 현상을 해결하는 적응적 상태 기반 객체 메모리 해제를 위한 마킹 알고리즘이다. 본 발명에 따른 알고리즘 사용시 기존 방법 대비 메모리 사용량이 평균 30% 감소한다.

도면의 간단한 설명

[0008] 도 1은 인메모리 데이터 처리 프로세스를 도시한 모식도이다.

도 2는 GC 메모리 관리 방법을 설명하기 위하여 도시한 그림이다.

도 3은 Watermark별 JVM 메모리 비율과 시간별 메모리 사용량을 도시한 그래프이다.

도 4는 본 발명에 따른 시스템을 도시한 구성도이다.

도 5는 본 발명에 따른 스트리밍 상황에서 적응적 상태 기반 객체 메모리 해제를 위한 마킹 알고리즘에 대한 슈도코드를 도시한 것이다.

도 6은 발명에 따른 방법과 기존 방법의 성능을 비교하여 도시한 그래프들이다.

발명을 실시하기 위한 구체적인 내용

[0009] 본 발명에 따른 알고리즘은 Apache Spark가 사용되는 응용을 타겟 환경으로 정한다. Spark는 분산 환경에서 인메모리 방식으로 스트리밍 데이터를 병렬 처리하며, Spark 프레임워크는 데이터 처리, 연산, 저장 작업에 필요한 메모리 할당/해제를 JVM(Java Virtual Machine)의 GC를 통해 수행한다.

[0010] JVM 메모리 관리와 관련하여, Spark 실행을 위한 메모리는 JVM의 GC(가비지 콜렉션)에 의해 관리된다. Spark는 외부로부터 데이터를 가져온 후 스트리밍 처리에 사용될 수 있도록 Spark의 기본 연산 단위인 RDD를 생성하는데 이는 GC 메모리 관리 영역에 저장된다.

[0011] 도 2는 GC 메모리 관리 방법을 설명하기 위하여 도시한 그림이다. 도 2는 Spark 런타임에서 생성된 객체 생애 주기를 나타낸다. 처음 생성된 객체는 Young generation 중 Eden에 할당된다. 다음 주기에 생성된 객체가 Eden에 들어올 때 Eden이 가득차면 Minor GC를 통해 기존 객체들에게 참조되지 않는 Eden내 객체를 메모리에서 해제하고 나머지 객체들을 S0, S1으로 복사한다. Young generation의 메모리 영역이 일정 크기에 도달하면 GC는 Minor GC를 수행하여 기존 객체들에게 참조되는 객체들을 Old generation으로 복사한다. 마지막으로 Old generation 영역이 일정 임계치를 넘으면 Major GC를 수행하여 객체에 할당된 메모리를 해제한다. Major GC는 Minor GC 대비 처리 시간이 길고 GC를 수행하는 쓰레드를 제외한 모든 쓰레드가 중지되므로 상당한 처리 지연을 유발한다.

[0012] 스트리밍 환경에서 메모리에 Stateful 객체 누적 저장으로 인한 메모리 부족 현상을 알아보기 위하여 사전 실험을 진행하였다. 실험 구성을 위해 데이터를 생산하는 노드와 생산된 데이터를 적재, 처리 및 저장하는 프레임워크를 설치한 노드 총 2대로 구축하였다. 실험 구조는 도 4와 같다. 도 4는 실험을 위한 시스템 모델이다.

[0013] 사전 실험은 Watermark와 Stateful 객체 생애 주기에 따른 메모리 사용량 분석을 위해 각 RDD에 Watermark를 각

각 10분, 60분으로 설정하였고, GC 메모리 사용량을 측정하였다. 도 3은 Watermark별 JVM 메모리 비율과 시간별 메모리 사용량을 도시한 그래프이다. 도 3을 참조하면, 각 RDD에 Watermark가 10분으로 설정될 경우 RDD간 조인 연산에서 생성된 Stateful 객체는 RDD 생성 시점 10분후 GC 메모리 영역에서 해제된다. 이는 Young generation에서 충분히 객체들을 처리하고, Old generation으로의 객체 복사 작업이 많지 않기 때문에 메모리 비율이 거의 동등하게 나옴을 알 수 있다. 반면, 각 RDD에 Watermark가 60분으로 설정된 경우, 연산 중 생성된 Stateful 객체는 메모리에 60분동안 상주한다. 이때, Young generation은 긴 Watermark로 인해 stateful 객체들이 누적 저장되고 일정 임계치를 넘으면 Minor GC를 통해 Old generation으로 복사된다. 이는 Old generation의 메모리 영역을 증가시키고 JVM에서 Spark 실행시 할당되는 메모리를 고갈시킴으로써 메모리 부족 현상을 심화시키게 된다.

[0014] 이하, 첨부된 도면을 참조하여 본 발명에 따른 알고리즘과 시스템 구조에 대하여 구체적으로 설명한다.

[0015] 도 4는 본 발명에 따른 시스템을 도시한 구성도이다. Spark는 데이터를 적재하고 있는 Kafka topic에 스트리밍 채널을 연결한다. 연결 후 Kafka topic에 데이터가 적재되는 즉시 Spark의 Object Controller로 JSON파일이 전송되며, 해당 파일들은 JVM의 메모리 할당 정책에 따라 RDD로 생성된다. 또한, 생성된 RDD는 각 스트리밍 메모리 테이블에 저장되며 연산 발생시 RDD 간 스트리밍 연산을 수행한다. RDD 연산에 따른 Stateful 객체는 State Store 해쉬맵에 저장된다. 이때 설정된 Watermark로 인해 RDD 연산에 사용된 메타데이터, Stateful 객체가 해쉬맵에 누적 저장되므로 Spark를 실행하는 JVM의 메모리 부족 현상이 심화된다.

[0016] 본 발명은 스트리밍 연산시 해쉬맵에 stateful 정보가 누적되어 메모리 부족 현상이 발생하는 문제를 해결하기 위하여 State Store를 모니터링하여 스트리밍 연산 결과에 따른 적응적 상태 기반 객체 메모리 해제 마킹 알고리즘을 제안한다. 본 발명은 기존의 스트리밍 메모리 테이블과 Join Manager 사이에 State Manager를 구현하여 데이터 연산이 완료된 RDD가 가진 Stateful 정보를 주기적으로 확인한다. 만약 데이터 연산이 완료되어 기존의 데이터 연산에 사용된 RDD 객체들이 사용되지 않는다면 Watermark에 관계없이 해당 RDD에 연결된 Stateful 정보를 GC가 메모리에서 해제할 수 있도록 마킹 태그를 부여한다. 도 5는 본 발명에 따른 스트리밍 상황에서 적응적 상태 기반 객체 메모리 해제를 위한 마킹 알고리즘에 대한 슈도코드를 도시한 것이다.

[0017] 이하, 본 발명에 따른 알고리즘을 적용하였을 때, 시간에 따른 JVM 메모리 영역 비율과 메모리 사용량을 평가한다. 본 실험은 12Core, 32GB 메모리, 500GB HDD, 1Gbps 네트워크 대역폭을 가진 2개의 작업 노드를 사용하였다. 데이터 생성 주기는 2초로 설정하며 3MB 크기의 JSON 파일을 전송한다. Spark 익스큐터는 4core, 8GB 메모리 할당후 총 2개 생성하였고, 10번의 실험 후 메모리 사용량을 평균 계산하였으며 모니터링 주기는 30초로 설정한다.

[0018] 알고리즘 별 JVM 메모리 비율을 비교하면 아래와 같다. 도 6은 발명에 따른 방법과 기존 방법의 성능을 비교하여 도시한 그래프들이다.

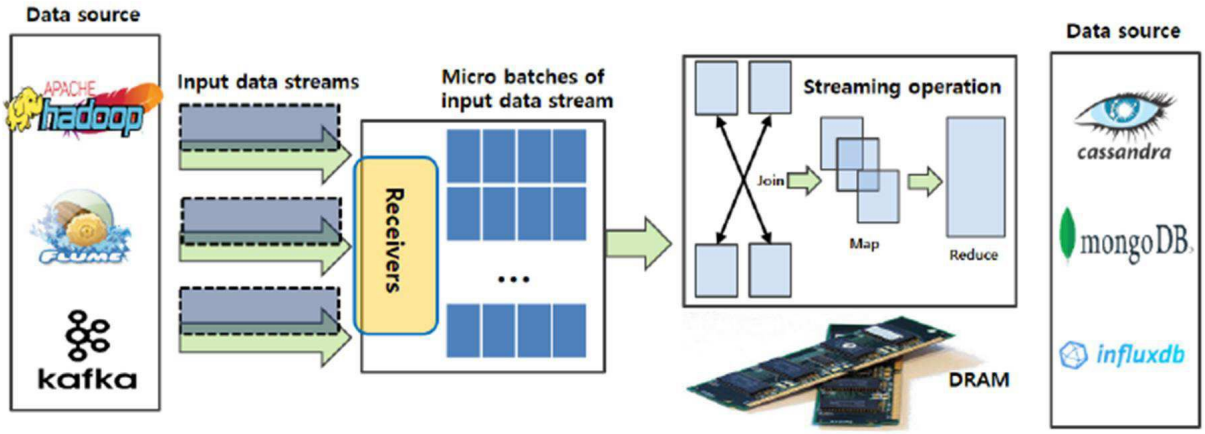
[0019] 도 6(a)는 Watermark를 60분으로 설정하였을 때 메모리 관련 알고리즘별 JVM 메모리 비율 차이를 나타낸다. 기존 알고리즘은 RDD간 데이터 연산이 완료되어도 RDD에 설정된 Watermark로 GC 메모리 영역에서 객체를 해제하지 않는다. Young generation은 Watermark가 설정된 RDD가 누적 저장되어 가용 메모리 공간이 적어지고 잦은 Minor GC로 RDD 객체를 Old generation으로 복사한다. 이로 인해 Old generation이 차지하는 비율이 급격하게 증가하여 메모리 충돌 현상을 발생시켰다. 반면, 본 발명에 따른 알고리즘은 주기적을 Sate Store를 모니터링하여 데이터 연산이 완료된 Stateful RDD 객체에 메모리 해제 마킹 태그를 부여한다. 이는 다음 GC에서 마킹된 RDD 객체에 연결된 모든 상태 정보를 메모리에서 해제시킨다. 그러므로, 본 발명에 따른 방법은 Young generation에서 Old generation로 RDD 객체 복사를 수행하는 Minor GC 횟수를 감소시키므로 기존 알고리즘 대비 GC 메모리 영역에서 Young generation 비율이 75% 증가된 것을 확인할 수 있다.

[0020] 도 6(b)는 알고리즘 별 메모리 사용량을 비교한 그래프이다. 본 발명에 따른 방법을 사용할 경우 14GB에서 saturation 되는 것을 확인할 수 있다. 이와 달리 기존 알고리즘을 사용하였을 경우 메모리 사용량이 특정 지점에서 saturation되지 않고 선형적으로 증가하여 메모리 충돌 현상을 발생시킨다.

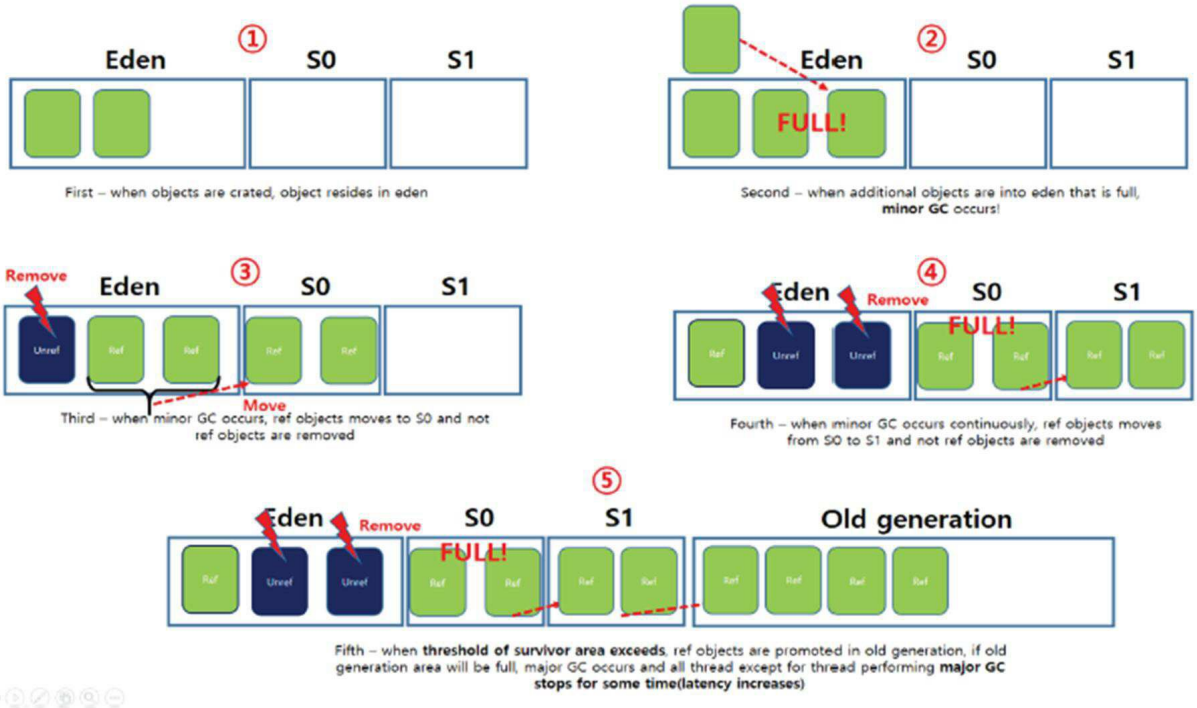
[0021] 이상에서 본 발명에 대하여 그 바람직한 실시예를 중심으로 설명하였으나, 이는 단지 예시일 뿐 본 발명을 한정하는 것이 아니며, 본 발명이 속하는 분야의 통상의 지식을 가진 자라면 본 발명의 본질적인 특성을 벗어나지 않는 범위에서 이상에 예시되지 않은 여러 가지의 변형과 응용이 가능함을 알 수 있을 것이다. 그리고, 이러한 변형과 응용에 관계된 차이점들은 첨부된 청구 범위에서 규정하는 본 발명의 범위에 포함되는 것으로 해석되어야 할 것이다.

도면

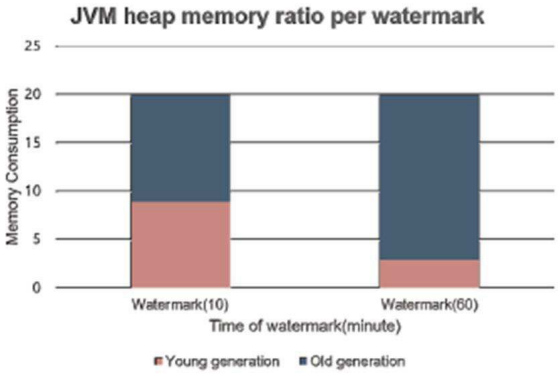
도면1


Data source
APACHE hadoop
FLUME
kafka
Input data streams
Micro batches of input data stream
Receivers
Streaming operation
Join
Map
Reduce
DRAM
Data source
cassandra
mongoDB.
influxdb

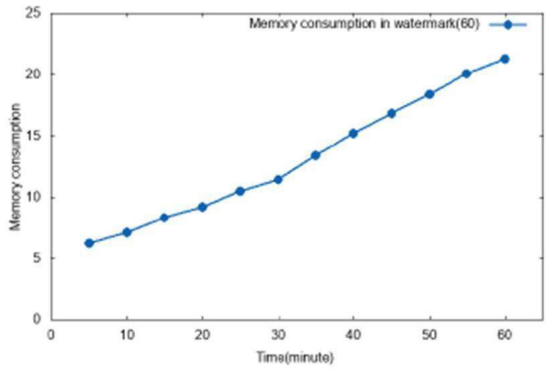
도면2


①
Eden
S0
S1
First – when objects are crated, object resides in eden
②
FULL!
Second – when additional objects are into eden that is full, minor GC occurs!
③
Remove
Unref
Ref
Move
Third – when minor GC occurs, ref objects moves to S0 and not ref objects are removed
④
Fourth – when minor GC occurs continuously, ref objects moves from S0 to S1 and not ref objects are removed
⑤
Old generation
Fifth – when threshold of survivor area exceeds, ref objects are promoted in old generation, if old generation area will be full, major GC occurs and all thread except for thread performing major GC stops for some time(latency increases)

도면3

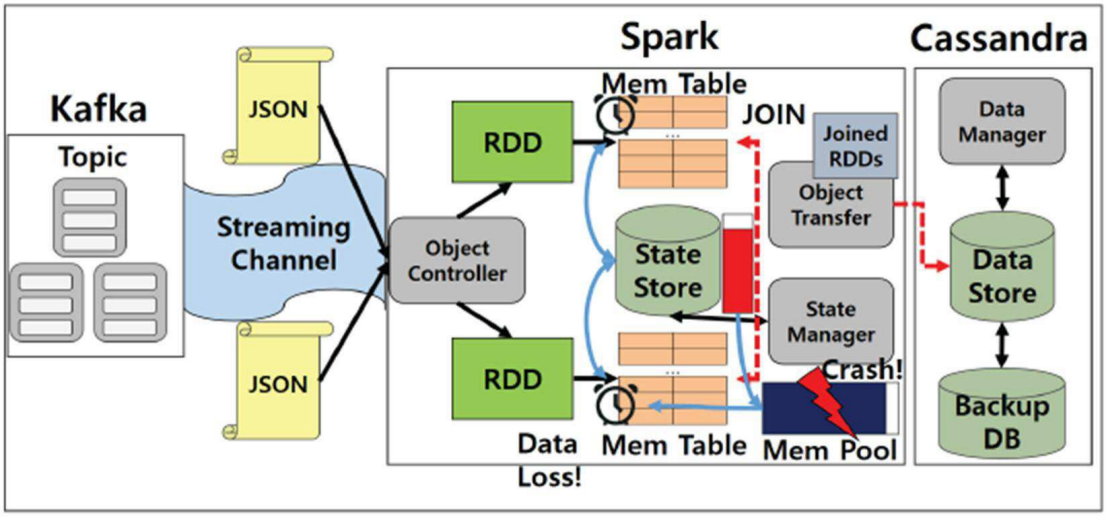


(a) JVM 메모리 비율



(b) 시간 별 메모리 사용량 측정

도면4


Spark
Cassandra
Kafka
Topic
JSON
JSON
Streaming Channel
Object Controller
RDD
RDD
Mem Table
JOIN
Joined RDDs
Object Transfer
State Store
State Manager
Crash!
Data Loss!
Mem Table
Mem Pool
Data Manager
Data Store
Backup DB

도면5

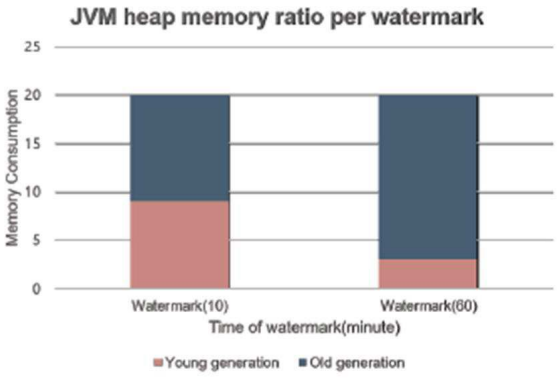
Algorithm 1 스트리밍 상황에서 적응적 상태 기반 객체 메모리 해제를 위한 마킹 알고리즘

ST_{RDD} : RDD pairs required for streaming query
ST_i : Each RDD pair
$State_{buffer}$: Buffer for checking the completed state query
ST_{queue} : Streaming job queue

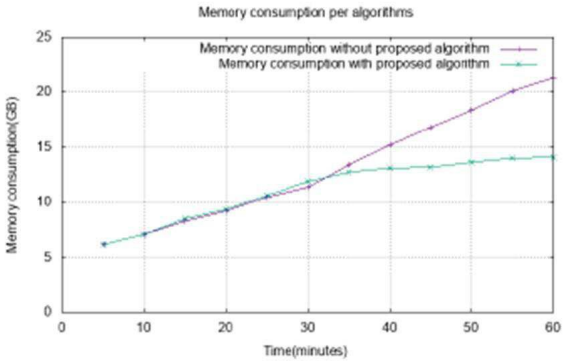
while ST_{RDD} exists in ST_{queue} **do**
    Sort ST_{RDD} by timestamp in recent order
    $ST_i \leftarrow ST_{RDD}$
    Join ST_i and insert ST_i to $State_{buffer}$
    Update *STATE Store* by ST_i
end while

while $State^i_{buffer}$ exists in $State_{buffer}$ **do**
    if $State^i_{buffer}$ not referenced **then**
        Remove the all state in *State Store*
        Mark the deletion tag to $State^i_{buffer}$
    end if
end while

도면6



(a) JVM 메모리 영역 비율



(b) 알고리즘 별 메모리 사용량 비교