탄력성, 고가용성, 고성능, 데이터 일관성을 고려한 Distributed Processing Solution 설계

요청하신 네 가지 핵심 요구사항(탄력성, 고가용성, 고성능, 데이터 일관성)을 모두 충족하는 분산 처리 솔루션 설계는 **각 요소 간의 트레이드오프(trade-off)를 이해하고 비즈니스 요구에 맞게 균형을 맞추는 것**이 핵심입니다.

이 네 가지 목표는 때로 상충관계에 있습니다. 예를 들어, 강력한 데이터 일관성(Strong Consistency)을 추구하면 성능과 가용성이 저하될 수 있으며(CAP 이론), 고성능을 위한 비동기 처리는 일관성 유지를 더 어렵게 만듭니다.

다음은 이 네 가지 요소를 고려한 현대적인 분산 처리 아키텍처 설계안입니다.

**분산 처리 솔루션 아키텍처 설계**

이 설계는 **느슨하게 결합된(Loosely Coupled)** 마이크로서비스와 \*\*이벤트 기반 아키텍처(Event-Driven Architecture)\*\*를 기반으로 합니다.

**1. 코어 아키텍처 원칙**

* **비동기 메시지 기반 통신:** 시스템 간의 직접적인 동기 호출(Synchronous Call)을 최소화합니다. 이는 **결합도를 낮추고** 한 서비스의 장애가 다른 서비스로 전파되는 것을 막아 **고가용성**과 **탄력성**을 확보하는 데 유리합니다.
* **상태 비저장(Stateless) 처리:** 처리 로직을 담당하는 컴포넌트(서버, 컨테이너)는 상태를 갖지 않습니다. 상태는 오직 외부의 신뢰할 수 있는 저장소(DB, 캐시, 메시지 큐)에만 저장합니다. 이는 \*\*탄력적 확장(Scaling)\*\*을 매우 용이하게 합니다.
* **분리를 통한 최적화:** 데이터 수집, 처리, 저장을 명확히 분리하여 각 컴포넌트가 자신의 역할에 맞는 최적의 기술을 사용하도록 합니다.

**2. 핵심 컴포넌트별 설계 전략**

각 컴포넌트가 네 가지 요구사항을 어떻게 충족하는지 설명합니다.

**가. 데이터 수집 및 큐잉 (Message Queue)**

대량의 데이터를 안정적으로 받아 처리 시스템에 전달하는 입구입니다.

* **기술 선택:** **Apache Kafka** 또는 **Pulsar**, 클라우드 기반 (AWS Kinesis, Google Pub/Sub)
* **탄력성:** 트래픽 증가 시 큐의 파티션(Partition)을 늘리고, 컨슈머(Consumer) 그룹의 인스턴스를 추가하여(Consumer Scale-out) 병렬 처리량을 동적으로 조절합니다.
* **고가용성:** 메시지 큐 자체를 클러스터로 구성하고, 데이터를 여러 브로커(Broker)에 \*\*복제(Replication)\*\*합니다. (e.g., Kafka의 3-way replication)
* **고성능:** 디스크 기반의 순차 I/O와 배치(Batch) 전송을 통해 초고속 데이터 처리가 가능합니다.
* **데이터 일관성:** 데이터가 큐에 **최소 1회 이상(At-least-once)** 저장됨을 보장합니다. (데이터 유실 방지)

**나. 데이터 처리 (Processing Layer)**

실제 비즈니스 로직을 수행하는 부분입니다.

* **기술 선택:** **Kubernetes(K8s) 기반 컨테이너** 또는 **서버리스(Serverless)** (AWS Lambda, Google Cloud Functions) + **스트림 처리 엔진** (Apache Flink, Spark Streaming)
* **탄력성 (E-):**
  + **K8s:** HPA(Horizontal Pod Autoscaler)를 사용하여 CPU/Memory 사용량이나 큐(Queue)의 메시지 수(Lag)에 따라 처리 인스턴스(Pod) 수를 \*\*자동으로 확장/축소(Auto-Scaling)\*\*합니다.
  + **서버리스:** 이벤트 발생 시에만 코드가 실행되므로 극단적인 탄력성을 제공합니다.
* **고가용성 (HA):**
  + **K8s:** 노드 장애 시 K8s가 자동으로 다른 노드에 Pod를 재시작시킵니다. **Liveness/Readiness Probe**를 통해 비정상 인스턴스를 자동으로 격리 및 복구합니다.
  + **지역 분산:** 여러 가용 영역(Multi-AZ) 또는 여러 리전(Multi-Region)에 걸쳐 워크로드를 배포합니다.
* **고성능 (P):**
  + **인메모리 처리:** Flink나 Spark 같은 엔진을 사용하여 데이터를 메모리상에서 고속으로 처리합니다.
  + **병렬 처리:** 큐의 파티션과 1:1로 매칭되는 컨슈머를 통해 데이터를 완벽하게 병렬 처리합니다.
* **데이터 일관성 (C):**
  + **정확히 1회 처리 (Exactly-once):** Flink와 Kafka는 \*\*2PC(Two-Phase Commit)\*\*와 유사한 메커니즘을 통해 End-to-End의 정확히 1회 처리를 지원합니다. 이는 일관성이 매우 중요할 때 사용합니다.
  + **멱등성(Idempotency) 설계:** 대부분의 경우(At-least-once 처리 시) 중복 처리가 발생해도 결과가 동일하도록 로직을 **멱등성 있게** 설계하는 것이 성능과 가용성을 유지하는 데 더 효율적입니다. (e.g., "UPDATE set A=5"는 멱등하지만, "UPDATE set A=A+1"은 멱등하지 않음)

**다. 데이터 저장 (Storage Layer)**

처리된 결과나 중간 상태를 저장합니다.

* **기술 선택:** 목적에 따라 다름 (e.g., **NoSQL** (Cassandra, DynamoDB), **RDBMS** (PostgreSQL w/ Replication), **NewSQL** (Spanner, CockroachDB), **Cache** (Redis))
* **탄력성:** 샤딩(Sharding) 또는 파티셔닝(Partitioning)이 용이한 NoSQL DB를 사용하거나, 클라우드 기반의 관리형 DB(e.g., AWS Aurora, DynamoDB)를 사용하여 스토리지와 I/O를 동적으로 확장합니다.
* **고가용성:**
  + **데이터 복제:** 데이터를 여러 노드나 리전에 걸쳐 복제합니다.
  + **자동 장애 복구(Failover):** Primary 노드 장애 시 Secondary 노드가 즉시 Primary로 승격됩니다.
* **고성능:**
  + **인메모리 캐시 (Redis):** 자주 접근하는 데이터를 캐시에 저장하여 DB 부하를 줄이고 응답 속도를 극대화합니다.
  + **용도별 DB 선택:** 빠른 쓰기/읽기가 필요하면 NoSQL, 복잡한 분석이 필요하면 Columnar DB(BigQuery)를 사용합니다.
* **데이터 일관성:**
  + **강력한 일관성 (Strong Consistency):** 금융 거래 등 정합성이 매우 중요한 경우, **NewSQL** (Spanner, CockroachDB) 또는 **RDBMS**를 사용합니다. (단, 성능/가용성 희생 가능)
  + **최종 일관성 (Eventual Consistency):** 대부분의 고성능/고가용성 시스템(e.g., SNS 피드, 장바구니)은 이 모델을 채택합니다. 데이터가 결국에는 일관성을 맞추는 것을 보장합니다.

**라. 조정 및 모니터링 (Orchestration & Monitoring)**

* **기술 선택:** **Kubernetes**, **Prometheus/Grafana**, **Istio(Service Mesh)**
* **고가용성/탄력성 지원:** 시스템의 상태를 실시간으로 모니터링하고(Prometheus), 장애를 감지하며, 임계치 도달 시 스케일링(K8s HPA)을 트리거하는 역할을 합니다.
* **고성능:** Istio 같은 서비스 메시는 **서킷 브레이커(Circuit Breaker)** 패턴, **Retry**, **Timeout** 등을 적용하여 일부 서비스의 지연이 전체 시스템의 성능을 저해하지 않도록 제어합니다.

**3. 일관성과 성능/가용성의 균형 (Trade-off)**

이 설계에서 가장 어려운 부분은 **데이터 일관성**입니다.

1. **CAP 이론:** 분산 시스템은 **일관성(C)**, **가용성(A)**, **분할 용인(P)** 중 최대 2가지만 동시에 만족할 수 있습니다. 현대적인 분산 시스템은 네트워크 단절(P)을 기본 전제로 하므로, \*\*일관성(CP)\*\*과 **가용성(AP)** 사이에서 선택해야 합니다.
   * **AP 시스템 (가용성 우선):** (e.g., NoSQL) 대부분의 시스템은 일시적인 데이터 불일치를 감수하더라도(최종 일관성) 항상 응답하는 것을 목표로 합니다.
   * **CP 시스템 (일관성 우선):** (e.g., RDBMS, NewSQL) 은행 시스템처럼 데이터가 절대 틀리면 안 되는 경우, 일시적 장애(응답 없음)를 감수하고 일관성을 지킵니다.
2. **성능 vs. 일관성:**
   * **강력한 일관성 (2PC, 분산 트랜잭션):** 여러 시스템의 데이터를 동시에 정확하게 맞춰야 할 때 사용합니다. 이는 모든 시스템의 응답을 기다려야 하므로 **성능이 크게 저하**됩니다.
   * **SAGA 패턴 (보상 트랜잭션):** 고가용성/고성능을 위해 일단 각 서비스를 비동기적으로 처리하고(성능/가용성 확보), 만약 중간에 실패가 발생하면 **보상(Compensation) 트랜잭션**을 통해 데이터를 되돌립니다. (e.g., 주문은 성공했으나 재고 차감 실패 시 -> 주문 취소 이벤트 발행) 이는 최종 일관성을 달성하는 효율적인 방법입니다.

**결론**

완벽하게 네 가지를 100% 만족시키는 단일 솔루션은 없습니다. 성공적인 설계는 다음과 같은 질문에 대한 답을 찾는 과정입니다.

* **"우리 비즈니스에서 허용 가능한 데이터 불일치 수준은 어느 정도인가?"** (e.g., 1초? 10분?)
* **"시스템이 응답하지 않는 것과, 오래된 데이터를 보여주는 것 중 무엇이 더 치명적인가?"**

이 질문에 대한 답을 바탕으로 **메시지 큐(Kafka)를 통한 비동기 처리**, **Kubernetes 기반의 탄력적 마이크로서비스**, **용도에 맞는 DB(NoSQL/RDBMS) 및 캐시(Redis)의 조합**, 그리고 **SAGA 패턴 또는 멱등성 설계**를 통해 일관성을 관리하는 것이 가장 현실적이고 강력한 분산 처리 솔루션 설계입니다.

Distributed Processing Solution의 탄력성 확보를 위한 적용 기술과 선정 이유

분산 처리 솔루션에서 \*\*탄력성(Elasticity)\*\*은 시스템이 **워크로드(Workload)의 변화에 따라 리소스를 자동으로 확장(Scale-out) 및 축소(Scale-in)하는 능력**을 의미합니다. 이는 비용을 최적화하고(사용한 만큼만 지불) 항상 일정한 성능을 유지하는 데 핵심적입니다.

탄력성 확보를 위한 주요 적용 기술과 선정 이유는 다음과 같습니다.

**1. 컨테이너 오케스트레이션 (Container Orchestration)**

* **적용 기술:** **쿠버네티스 (Kubernetes, K8s)**
* **선정 이유:**
  + **자동화된 수평 확장 (HPA):** \*\*HPA(Horizontal Pod Autoscaler)\*\*는 탄력성의 핵심입니다. CPU, 메모리 사용량 같은 표준 메트릭뿐만 아니라, 메시지 큐의 대기열 길이(Consumer Lag)와 같은 \*\*커스텀 메트릭(Custom Metric)\*\*을 기반으로 애플리케이션 인스턴스(Pod) 수를 실시간으로 자동 조절합니다.
  + **자동화된 인프라 확장 (Cluster Autoscaler):** HPA에 의해 Pod 수가 증가할 때, 클러스터(물리 서버/VM) 자체의 리소스가 부족하면 \*\*CA(Cluster Autoscaler)\*\*가 클라우드 제공업체와 통신하여 필요한 만큼의 \*\*노드(Node, VM)\*\*를 자동으로 추가합니다. 반대로 사용량이 줄면 노드를 반납하여 비용을 절감합니다.
  + **신속한 배포 및 복구:** 컨테이너 이미지를 사용하므로, 스케일 아웃 시 새로운 인스턴스를 VM보다 훨씬 빠르게(수초 내에) 시작할 수 있습니다.

**2. 서버리스 컴퓨팅 (Serverless Computing)**

* **적용 기술:** **AWS Lambda, Google Cloud Functions, Azure Functions**
* **선정 이유:**
  + **궁극적인 탄력성 (Scale-to-Zero):** 서버리스는 탄력성의 가장 극단적인 형태를 제공합니다. 평소에는 리소스를 전혀 점유하지 않다가(0개), 이벤트(e.g., API 호출, 메시지)가 발생하는 순간 필요한 만큼의 실행 환경을 즉시(수 밀리초 내에) 생성하여 처리합니다.
  + **관리 부담 제로:** 쿠버네티스의 HPA, CA 설정조차 필요 없이, 클라우드 제공업체가 모든 인프라의 스케일링을 100% 알아서 처리합니다. 개발자는 비즈니스 로직(코드)에만 집중할 수 있습니다.
  + **비용 효율성:** 코드가 실행된 시간(밀리초 단위)과 횟수만큼만 비용을 지불하므로, 워크로드가 불규칙하고 예측 불가능할 때 가장 비용 효율적입니다.

**3. 메시지 큐 & 스트리밍 플랫폼 (Message Queue & Streaming)**

* **적용 기술:** **Apache Kafka, RabbitMQ, Google Pub/Sub, AWS SQS/Kinesis**
* **선정 이유:**
  + **처리 계층의 디커플링(Decoupling):** 메시지 큐는 데이터 생산자(Producer)와 소비자(Consumer, 처리 로직)를 분리하는 **버퍼(Buffer)** 역할을 합니다.
  + **부하 평준화 (Load Leveling):** 데이터가 순간적으로 폭증(Spike)하더라도 큐에 안전하게 저장됩니다. 처리 시스템(컨슈머)은 자신이 처리할 수 있는 속도만큼 데이터를 가져와 처리합니다. 이로 인해 처리 시스템이 트래픽 폭증으로 인해 다운되는 것을 방지합니다.
  + **탄력적 스케일링의 기준 제공:** 큐에 쌓인 메시지의 수, 즉 \*\*'컨슈머 랙(Consumer Lag)'\*\*은 처리 시스템이 얼마나 바쁜지를 나타내는 가장 정확한 지표입니다. 쿠버네티스 HPA는 이 '랙'을 모니터링하여 "랙이 1000개 이상이면 컨슈머 Pod를 10개로 늘려라"와 같은 정교한 탄력성 정책을 구현할 수 있습니다.
  + **병렬 처리 최적화 (Kafka):** Kafka의 '파티션(Partition)' 모델은 컨슈머 그룹 내의 컨슈머 수에 따라 자동으로 데이터 처리를 병렬 분산(Rebalancing)시켜, 스케일 아웃 효과를 극대화합니다.

**4. 자동 확장형 데이터베이스 (Auto-scaling Database)**

* **적용 기술:** **AWS DynamoDB (On-demand), Google Spanner/Bigtable, AWS Aurora Serverless, MongoDB Atlas**
* **선정 이유:**
  + **병목 현상 방지:** 처리 계층(K8s, Lambda)이 아무리 빠르게 확장되어도, 데이터베이스가 부하를 견디지 못하면 전체 시스템이 마비됩니다.
  + **처리량(Throughput) 자동 확장:** DynamoDB나 Spanner 같은 NoSQL/NewSQL DB는 애플리케이션의 트래픽(읽기/쓰기 요청)에 맞춰 **I/O 용량(RCU/WCU)을 자동으로 확장**합니다.
  + **스토리지 자동 확장:** 데이터가 증가함에 따라 별도 작업 없이 스토리지가 자동으로 확장됩니다.
  + **컴퓨트 탄력성 (Aurora Serverless):** Aurora Serverless와 같은 서비스는 DB 쿼리가 없을 때는 컴퓨트 인스턴스를 0으로 축소했다가, 쿼리가 들어오면 다시 활성화하여 비용을 최적화합니다.

Distributed Processing Solution의 고가용성 확보를 위한 적용 기술과 선정 이유

분산 처리 솔루션에서 \*\*고가용성(High Availability, HA)\*\*은 시스템의 구성 요소 중 일부에 장애가 발생하더라도 **서비스가 중단 없이 지속적으로 제공될 수 있는 능력**을 의미합니다. 이는 사용자 경험과 비즈니스 연속성에 필수적입니다.

고가용성 확보를 위한 주요 적용 기술과 선정 이유는 다음과 같습니다.

**1. 다중화 및 복제 (Redundancy & Replication)**

* **적용 기술:**
  + **클러스터링 (Clustering):** 데이터베이스, 메시지 큐, 처리 엔진 등 핵심 컴포넌트를 여러 노드(인스턴스)로 구성된 클러스터 형태로 운영합니다.
  + **데이터 복제 (Data Replication):** 데이터베이스(RDBMS, NoSQL), 메시지 큐(Kafka, Pulsar), 분산 파일 시스템(HDFS) 등 데이터를 저장하는 모든 컴포넌트에서 데이터를 여러 노드나 저장소에 복사합니다. (e.g., 3-way replication)
  + **로드 밸런싱 (Load Balancing):** 여러 서비스 인스턴스에 트래픽을 분산하고, 비정상 인스턴스를 자동으로 트래픽 분배에서 제외합니다.
* **선정 이유:**
  + **단일 실패 지점 제거 (Single Point of Failure, SPOF):** 어떤 한 노드나 컴포넌트가 장애로 중단되더라도, 다른 노드들이 그 역할을 대신하여 서비스가 계속됩니다.
  + **자동 장애 복구 (Automatic Failover):** 클러스터 구성 시, Primary/Leader 노드에 장애가 발생하면 자동으로 Standby/Follower 노드가 Primary로 승격되어 서비스 중단을 최소화합니다.
  + **데이터 유실 방지:** 복제를 통해 한 노드의 디스크 손상이나 데이터 유실 시에도 다른 복제본을 통해 데이터를 복구할 수 있습니다.

**2. 분산 배치 및 지역 분산 (Distributed Deployment & Geo-Redundancy)**

* **적용 기술:**
  + **다중 가용 영역(Multi-Availability Zone, Multi-AZ) 배포:** 클라우드 환경에서 물리적으로 독립된 여러 데이터센터(가용 영역)에 시스템 컴포넌트를 분산하여 배포합니다.
  + **다중 리전(Multi-Region) 배포:** 더 높은 수준의 재해 복구(Disaster Recovery, DR)를 위해 지리적으로 떨어진 여러 클라우드 리전에 시스템을 배포합니다.
  + **글로벌 로드 밸런서 (Global Load Balancer/DNS):** (e.g., AWS Route 53, Google Cloud DNS) 여러 리전/가용 영역에 분산된 서비스 엔드포인트 중 가장 가깝거나 정상적인 곳으로 트래픽을 라우팅합니다.
* **선정 이유:**
  + **데이터센터/지역 단위의 장애 대비:** 특정 가용 영역 전체가 다운되거나, 자연재해 등으로 특정 리전 전체에 장애가 발생하더라도, 다른 가용 영역/리전에서 서비스가 지속될 수 있도록 합니다.
  + **재해 복구 (DR):** RTO(Recovery Time Objective)와 RPO(Recovery Point Objective) 목표에 따라 Active-Active, Active-Passive 같은 다양한 DR 전략을 구현할 수 있습니다.

**3. 헬스 체크 및 자동 복구 (Health Checks & Self-Healing)**

* **적용 기술:**
  + **Liveness/Readiness Probe (Kubernetes):**
    - **Liveness Probe:** 컨테이너가 정상적으로 실행 중인지 확인하여, 비정상 상태 시 자동으로 컨테이너를 재시작합니다.
    - **Readiness Probe:** 컨테이너가 요청을 처리할 준비가 되었는지 확인하여, 준비되지 않은 컨테이너로는 트래픽을 보내지 않습니다.
  + **모니터링 및 알림:** Prometheus, Grafana, ELK Stack 등을 통해 시스템의 모든 지표를 실시간으로 모니터링하고, 임계치 초과 시 담당자에게 자동 알림을 보냅니다.
  + **자동 복구 스크립트/워크플로우:** 특정 유형의 장애 발생 시 미리 정의된 스크립트나 자동화된 워크플로우를 실행하여 복구를 시도합니다.
* **선정 이유:**
  + **사전 예방 및 신속한 감지:** 장애 발생 징후를 미리 감지하고, 실제 장애 발생 시 이를 즉시 인지하여 대응 시간을 단축합니다.
  + **서비스 지속성:** 비정상적인 컴포넌트를 빠르게 격리하고, 자동으로 재시작하거나 다른 정상적인 컴포넌트로 트래픽을 전환하여 서비스 중단을 방지합니다.

**4. 장애 격리 및 내결함성 패턴 (Fault Isolation & Tolerance Patterns)**

* **적용 기술:**
  + **서킷 브레이커 (Circuit Breaker):** 특정 서비스 호출이 계속 실패할 경우, 일시적으로 해당 서비스로의 모든 호출을 중단하고 즉시 실패 응답을 반환하여, 장애 전파를 막고 실패한 서비스가 복구될 시간을 벌어줍니다.
  + **벌크헤드 (Bulkhead):** 시스템 리소스를 격리된 풀(Pool)로 분할하여, 한 컴포넌트의 실패가 다른 컴포넌트의 리소스를 고갈시키지 않도록 합니다. (e.g., 특정 API에 대한 요청이 급증해도 다른 API가 영향을 받지 않도록 스레드 풀 분리)
  + **타임아웃 및 리트라이 (Timeout & Retry):** 외부 서비스 호출 시 무한정 기다리지 않고 특정 시간 후 타임아웃 처리하며, 일시적인 네트워크 문제 등에 대해서는 적절한 지연(Exponential Backoff) 후 재시도를 수행합니다.
  + **비동기 메시지 통신:** 서비스 간 직접적인 동기 호출을 최소화하고 메시지 큐를 통해 비동기적으로 통신하여 서비스 간의 결합도를 낮추고 장애 전파를 방지합니다.
* **선정 이유:**
  + **장애 전파 방지 (Cascading Failure):** 특정 컴포넌트의 작은 장애가 전체 시스템의 대규모 장애로 확산되는 것을 방지합니다.
  + **시스템 안정성 향상:** 예측 불가능한 외부 요인이나 일시적인 오류에도 시스템이 안정적으로 동작하도록 합니다.

**5. 변경 관리 및 배포 자동화 (Change Management & Automated Deployment)**

* **적용 기술:**
  + **CI/CD 파이프라인 (Continuous Integration/Continuous Deployment):** 코드 변경사항을 자동으로 테스트하고 배포하는 프로세스입니다.
  + **롤링 업데이트 (Rolling Updates):** 새 버전의 애플리케이션을 배포할 때, 기존 버전을 한 번에 모두 중단시키지 않고 점진적으로 새 버전으로 교체합니다.
  + **카나리 배포 (Canary Deployment), 블루/그린 배포 (Blue/Green Deployment):** 새로운 버전을 소수의 사용자에게만 먼저 배포하거나(카나리), 기존 버전과 완전히 분리된 환경에 새 버전을 배포하여 테스트(블루/그린)한 후, 문제가 없을 시 전체 트래픽을 전환합니다.
* **선정 이유:**
  + **배포로 인한 장애 최소화:** 수동 배포 시 발생하는 인적 오류를 줄이고, 새로운 변경사항이 프로덕션 환경에 미치는 영향을 최소화하여 안정적인 서비스 제공을 가능하게 합니다.
  + **빠른 롤백 (Fast Rollback):** 새로운 버전 배포 후 문제가 발생하면, 즉시 이전 안정적인 버전으로 되돌릴 수 있는 기능을 제공하여 서비스 중단 시간을 단축합니다.

이러한 기술들을 복합적으로 적용함으로써, 분산 처리 솔루션은 예측 가능한 장애뿐만 아니라 예측 불가능한 장애 상황에서도 높은 수준의 가용성을 유지할 수 있게 됩니다.

Distributed Processing Solution의 고성능 확보를 위한 적용 기술과 선정 이유

분산 처리 솔루션에서 **고성능**은 \*\*처리량(Throughput)\*\*을 극대화하고 \*\*지연 시간(Latency)\*\*을 최소화하는 것을 목표로 합니다.

고성능 확보를 위한 핵심 적용 기술과 선정 이유는 다음과 같습니다.

**1. 병렬 및 분산 처리 (Parallel & Distributed Processing)**

* **적용 기술:**
  + **데이터 파티셔닝/샤딩 (Partitioning/Sharding):** 대규모 데이터를 여러 노드나 파티션에 분산 저장하고 처리합니다. (e.g., Kafka 파티션, DB 샤딩)
  + **분산 처리 프레임워크:** (e.g., **Apache Spark, Apache Flink, Dask**) 대규모 데이터셋을 여러 노드의 메모리에서 병렬로 계산합니다.
  + **수평 확장 (Scale-out):** 더 강력한 단일 서버(Scale-up)에 의존하는 대신, 더 많은 수의 범용 서버(노드)를 추가하여 처리 용량을 선형적으로 증가시킵니다.
* **선정 이유:**
  + **작업 부하 분산:** 단일 노드의 처리 한계를 극복하고 대규모 작업을 동시에 처리하여 전체 처리 시간을 획기적으로 단축합니다.
  + **병목 현상 제거:** 데이터 처리 및 저장 부하를 분산시켜 특정 지점(e.g., 단일 DB)에 병목이 발생하는 것을 방지합니다.

**2. 인메모리 컴퓨팅 (In-Memory Computing)**

* **적용 기술:**
  + **분산 캐시 (Distributed Cache):** (e.g., **Redis, Hazelcast, Memcached**) 자주 접근하거나 계산 비용이 비싼 데이터를 디스크가 아닌 메모리에 저장하여 매우 빠른 속도로 제공합니다.
  + **인메모리 데이터 그리드 (IMDG):** (e.g., Hazelcast, Apache Ignite) 데이터를 여러 노드의 RAM에 분산 저장하고 연산까지 메모리상에서 수행합니다.
  + **인메모리 스트림 처리:** (e.g., **Apache Flink, Spark Streaming**) 데이터를 디스크에 쓰지 않고 메모리상에서 실시간으로 처리합니다.
* **선정 이유:**
  + **I/O 병목 제거:** 디스크 I/O는 메모리 I/O보다 수천 배 느립니다. 데이터를 메모리에 상주시켜 이 병목을 제거함으로써 응답 속도를 극적으로 향상시킵니다.
  + **중복 계산 방지:** 캐시를 통해 동일한 연산이나 DB 조회를 반복 수행하지 않아 시스템 자원을 절약하고 응답 시간을 단축합니다.

**3. 비동기/논블로킹 처리 (Asynchronous & Non-Blocking)**

* **적용 기술:**
  + **메시지 큐 (Message Queue):** (e.g., **Kafka, RabbitMQ, SQS**) 요청을 즉시 처리하는 대신 큐에 저장하고, 백그라운드에서 비동기적으로 처리합니다.
  + **이벤트 기반 아키텍처 (Event-Driven Architecture):** 작업 완료를 기다리지 않고(Non-Blocking), 이벤트가 발생할 때만 필요한 로직을 수행합니다.
  + **반응형 프로그래밍 (Reactive Programming):** (e.g., Project Reactor, RxJava) 데이터 스트림을 비동기적으로 처리하여 스레드 자원을 효율적으로 사용합니다.
* **선정 이유:**
  + **응답성 향상:** 사용자는 작업 완료를 기다릴 필요 없이 즉시 응답을 받으므로(e.g., "주문이 접수되었습니다"), 시스템의 체감 성능이 향상됩니다.
  + **자원 효율성 (스레드):** 기존의 '1요청 1스레드' 방식은 I/O 대기 시 스레드가 낭비됩니다. 논블로킹 방식은 적은 수의 스레드로도 대량의 동시 요청을 효율적으로 처리할 수 있습니다.

**4. 데이터 지역성 (Data Locality)**

* **적용 기술:**
  + **"코드를 데이터로 이동 (Moving Code to Data)":** (e.g., Hadoop MapReduce, Spark RDD) 대용량 데이터를 네트워크로 전송하는 대신, 데이터가 저장된 노드로 계산 로직(코드)을 전송하여 해당 노드에서 직접 처리합니다.
  + **CDN (Content Delivery Network):** 정적 콘텐츠(이미지, 영상)를 사용자와 가장 가까운 위치의 엣지 서버에 캐시하여 전송 속도를 높입니다.
  + **캐시 어피니티 (Cache Affinity):** 특정 데이터가 항상 동일한 캐시 노드나 파티션에 저장되도록 하여 캐시 히트율(Hit Rate)을 높입니다.
* **선정 이유:**
  + **네트워크 병목 최소화:** 대용량 데이터를 네트워크로 전송하는 비용은 매우 큽니다. 데이터가 있는 곳에서 직접 처리함으로써 네트워크 오버헤드를 최소화하고 성능을 향상시킵니다.

**5. 데이터 처리 및 전송 최적화**

* **적용 기술:**
  + **데이터 직렬화 (Data Serialization):** (e.g., **Protobuf, Avro**) JSON이나 XML보다 훨씬 작고 빠른 바이너리(Binary) 기반 포맷을 사용하여 데이터를 직렬화하고 네트워크로 전송합니다.
  + **배치 처리 (Batch Processing):** 데이터를 건건이(Record-by-record) 처리하는 대신, 여러 개의 데이터를 모아(Batch) 한 번에 처리하여 오버헤드를 줄입니다. (e.g., DB 배치 INSERT, Kafka 배치 전송)
  + **데이터 압축 (Data Compression):** (e.g., Snappy, LZ4) 전송되거나 저장되는 데이터를 압축하여 I/O 및 네트워크 사용량을 줄입니다. (빠른 압축/해제 속도가 중요)
* **선정 이유:**
  + **오버헤드 감소:** 데이터 직렬화 및 압축은 CPU를 조금 더 사용하는 대신, 훨씬 더 비싼 자원인 네트워크 대역폭과 디스크 I/O를 크게 절약하여 전체 성능을 높입니다.
  + **처리 효율 향상:** 배치 처리는 시스템 호출 및 네트워크 왕복 횟수를 줄여 건별 처리 대비 훨씬 높은 처리량을 달성하게 합니다.

<<Distributed Processing Solution의 데이터 일관성 확보를 위한 적용 기술과 선정 이유>>

분산 처리 솔루션에서 데이터 일관성을 확보하기 위해 적용되는 기술은 시스템의 요구사항에 따라 다양합니다

. 대표적인 기술과 그 선정 이유는 다음과 같습니다.

**1. 분산 트랜잭션 (Distributed Transaction)**

**적용 기술**

* **2단계 커밋(2PC, Two-Phase Commit):** 여러 참여자(노드)가 모두 성공적으로 커밋할 때만 전체 트랜잭션을 확정하고, 하나라도 실패하면 모두 롤백하는 방식.
* **3단계 커밋(3PC, Three-Phase Commit):** 2PC의 문제점(단일 장애점, 블로킹)을 개선한 프로토콜로, 준비(Prepare) 단계를 세분화하여 참여자가 코디네이터 없이도 결정을 내릴 수 있게 함.
* **분산 락(Distributed Lock):** 여러 노드가 공유 자원에 동시에 접근하는 것을 제어하는 기술.

**선정 이유**

* **강력한 일관성 보장:** 여러 데이터베이스에 걸친 트랜잭션의 ACID(원자성, 일관성, 고립성, 지속성) 속성을 보장해야 하는 금융 시스템이나 재고 관리 시스템에 적합합니다.
* **안정적인 데이터 무결성:** 여러 단계의 엄격한 프로토콜을 통해 데이터의 정합성을 최우선으로 확보할 수 있습니다.

**2. 복제(Replication) 및 일관성 모델**

**적용 기술**

* **복제(Replication):** 데이터를 여러 노드에 복사하여 분산시키는 기술.
* **강력한 일관성(Strong Consistency):** 모든 노드에서 항상 동일한 최신 데이터를 읽을 수 있도록 보장.
* **최종 일관성(Eventual Consistency):** 데이터가 일시적으로 불일치할 수 있지만, 충분한 시간이 지나면 모든 노드에 동일한 데이터가 복제되는 모델.
* **일관성 수준(Consistency Level):** 복제된 데이터에 대해 읽기/쓰기 작업 시, 얼마나 많은 노드로부터 응답을 받아야 성공으로 간주할지 설정하는 방식 (예: Dynamo 스타일 복제).

**선정 이유**

* **CAP 이론에 따른 유연성:** CAP 이론(일관성, 가용성, 분할 내성 중 2가지만 선택 가능)에 따라, 비즈니스 요구사항에 맞춰 일관성 수준을 조절할 수 있습니다.
* **성능 및 확장성 고려:** 읽기 작업이 많은 시스템에서는 최종 일관성 모델을 적용하여 성능과 가용성을 높이고, 실시간으로 최신 데이터가 필요한 시스템에서는 강력한 일관성을 선택할 수 있습니다.

**3. 분산 트랜잭션 전략 (Microservice Architecture)**

**적용 기술**

* **SAGA 패턴:** 여러 개의 로컬 트랜잭션을 순차적으로 실행하여 전체 분산 트랜잭션을 완성하는 방식. 실패 시에는 보상 트랜잭션을 통해 이전 상태로 되돌립니다.
* **이벤트 소싱(Event Sourcing):** 시스템의 상태 변화를 일련의 이벤트로 저장하고, 이 이벤트를 통해 현재 상태를 재구성하는 패턴.
* **분산형 코디네이터(Distributed Coordinator):** ZooKeeper나 Consul과 같이 분산된 서비스들의 상태를 관리하여 일관된 정보를 제공하는 방식.

**선정 이유**

* **마이크로서비스 환경에 적합:** 마이크로서비스 아키텍처에서는 여러 서비스에 걸친 분산 트랜잭션을 단일 트랜잭션처럼 처리하기 어렵기 때문에, SAGA나 이벤트 소싱 패턴을 통해 데이터 일관성을 관리합니다.
* **비동기 처리와 높은 가용성:** 이벤트 기반의 비동기 방식을 사용하여 서비스 간의 결합도를 낮추고, 시스템의 가용성을 높일 수 있습니다.

**4. 기타 기술**

**적용 기술**

* **분산 캐시(Distributed Cache):** Redis나 Memcached와 같은 분산 캐시를 활용하여 데이터에 대한 동시 접근을 제어하거나, 데이터 일관성을 관리.
* **메시지 큐(Message Queue):** Kafka나 RabbitMQ와 같은 메시지 큐를 사용하여 비동기적으로 데이터 변경 이벤트를 처리함으로써, 느슨한 결합을 유지하며 일관성을 확보.

**선정 이유**

* **성능 향상 및 부하 분산:** 분산 캐시는 데이터베이스에 대한 접근을 줄여 성능을 향상하고, 메시지 큐는 작업의 비동기 처리를 통해 시스템 부하를 분산시킵니다.
* **결합도 완화:** 마이크로서비스 간에 직접적인 데이터 교환 대신 메시지 큐를 사용함으로써 서비스 간의 결합도를 낮추고, 유연성을 확보합니다.