이번 시간에는 data architecture 를 하겠습니다

근데 이제 Netflix를 곁들인.....

강사 : 윤성국

application system의 패러다임 변화

Compute-intensive

Data-intensive

standard building block

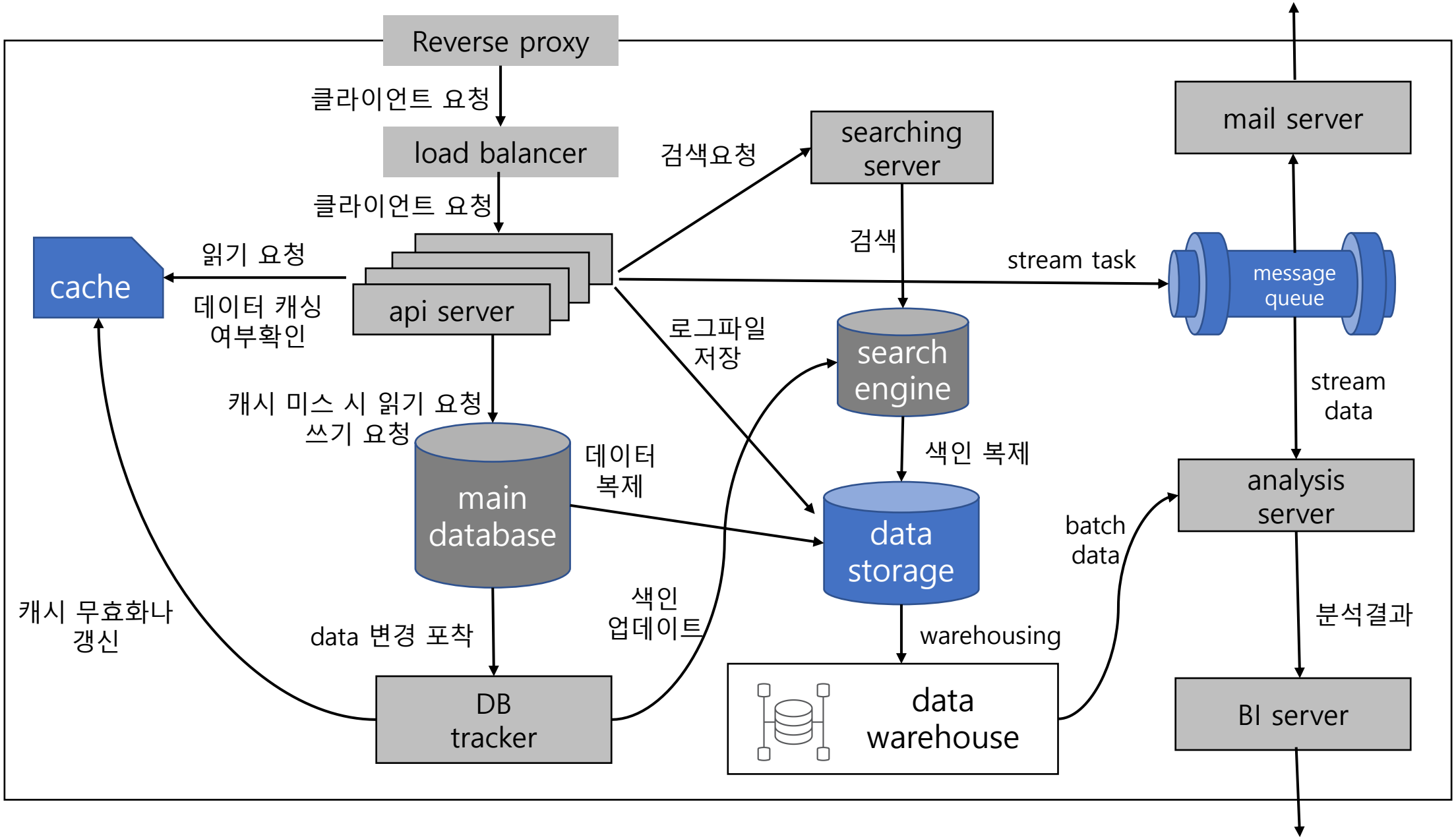
- application에서 데이터를 활용할 수 있도록 데이터저장( )
- 데이터 읽기 속도 향상을 위해 수행결과 기억( )
- 키워드로 데이터 검색하거나 필터링 가능하도록 제공( )
- 비동기 데이터 처리를 위해 프로세스간 메시지 송수신( )
- 주기적으로 대량의 누적된 데이터를 분석( )

data system

왜 각자 성격이 다른 tool들을 하나의 system으로 묶을까?

- 데이터 도구들의 use-case의 경계가 흐려지고 있다.
  (MQ로 쓰는 redis, 데이터 지속성 보장하는 kafka)

- work를 task로 세분화하여 관리 ( )


Reverse proxy
클라이언트 요청
load balancer
클라이언트 요청
검색요청
searching server
mail server
cache
읽기 요청
데이터 캐싱 여부확인
api server
검색
stream task
message queue
로그파일 저장
search engine
캐시 미스 시 읽기 요청
쓰기 요청
main database
데이터 복제
색인 복제
data storage
stream data
analysis server
batch data
캐시 무효화나 갱신
data 변경 포착
색인 업데이트
warehousing
분석결과
DB tracker
data warehouse
BI server

data system에서 신경써야할 점

- Reliability
- Scalability
- Maintainability

Reliability

- 신뢰성에 대한 일반적인 기대치
  - application은 user가 기대한 기능을 수행한다
  - system은 사용자가 범한 실수나 예상치 못한 소프트웨어 사용법을 허용
  - system 성능은 예상된 부하와 데이터 양에서 필수적인 사용성 만족
  - system은 허가되지 않는 접근과 오남용 방지
- fault-tolerant(내결함성)
  - fault(결함)는 failure(장애)가 아니다 -> 엔드유저의 서비스 이용 가능 여부
  - 모든 fault를 견딜 순 없다. -> 빈번히 발생하는 특정 유형에 대처
  - 고의적 fault 발생으로 내결함성 확보 -> 어설픈 오류처리보단 확실한 에러
  - 결함해결보다 예방이 중요할 수 있다 -> 보안이슈

hardware fault

- hard disk의 평균 10~50년 ->10000개 RAID면 하루에 하나씩 박살
- redundancy 추가
  - disk RAID
  - 파워 이중화, CPU hot-swap
  - 예비전력 발전기 이중화
- 대용량 분산 데이터 시스템
  - 소프트웨어 fault-tolerant으로 해결
  - 단일 장비 신뢰성보다 유연성과 탄력성
  - 더 많은 하드웨어 redundancy 추가
  - 무 중단 업그레이드

software fault

- systematic error
  - 모든 인스턴스 노드가 한꺼번에 죽는 소프트웨어 버그(리눅스커널 윤초버그)
  - 공유 리소스를 과도하게 사용해서 리소스를 잠식해버리는 프로세스
  - 시스템 응답속도가 느려져서 반응이 없거나 잘못된 응답 반환
  - 시스템의 한 요소의 fault가 다른 요소의 fault로 전파되는 cascading failure
- 특정 상황에만 버그 재현이 발생하여 버그 재현율 떨어짐
- 테스트, 프로세스 격리, fail back, fail over, 모니터링 등으로 해결

human fault 사람이 문제다

- 실수를 줄이는 방법으로 설계, 추상화 layering, API, 인터페이스 제한
- 운영과 별개로 개발 및 실험 sandbox 제공
- 유닛테스트, 프로세스테스트, 시스템 통합 테스트, 운영테스트
- 설정 오류에 대한 rollback, rollout 제공
- 디테일한 모니터링 metric 설계
- 조작 교육 및 실습

Scalability

- 지금 안정적으로 잘 돌아가는 시스템이 앞으로도 잘 돌까?
- 1만명에서 10만명 100만명 1000만명으로 증가하고 데이터도 그만큼 계속 증가한다면?
- 확장성은 단순히 system에만 국한되는 특성이 아님
- 확장성을 논한다는 것은
  - X시스템은 확장 가능하다, Y시스템은 확장 가능성이 없다 -> 의미없음
  - 시스템이 특정방식으로 커질 때에 대처하기 위한 decision making
  - 추가되는 데이터 부하를 감당하기 위해 compute및 storage 투입 정책

load description

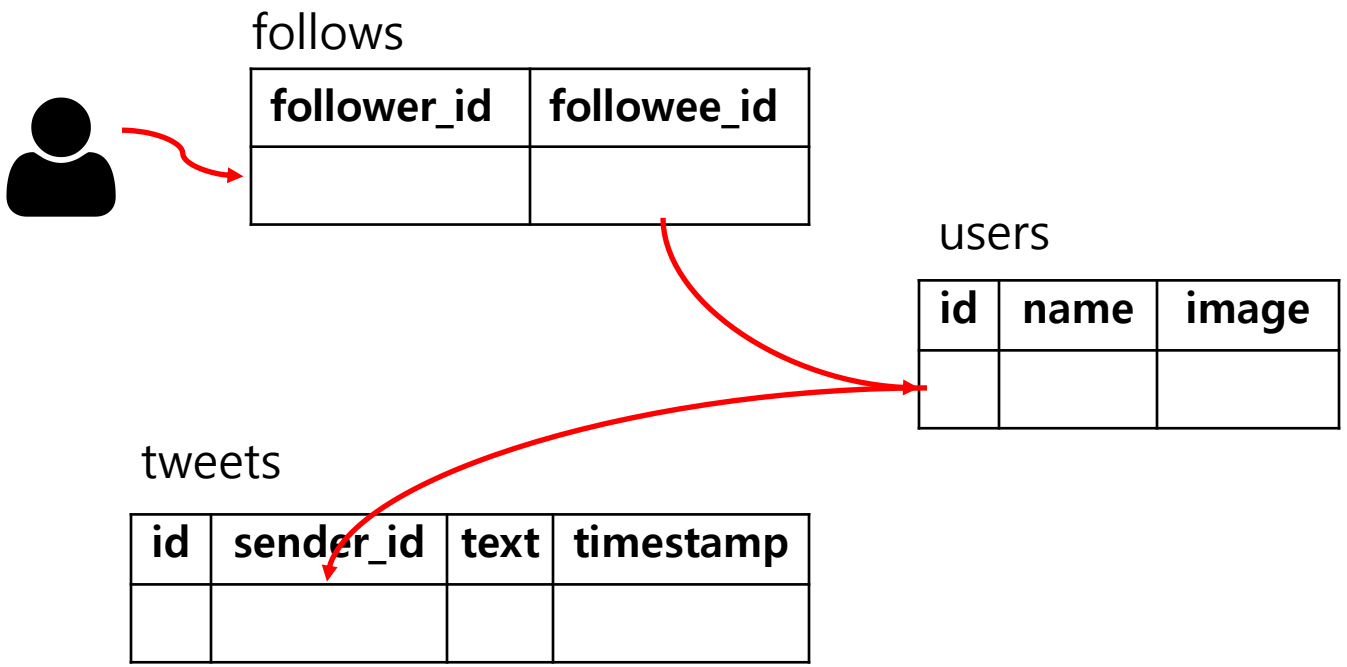
- 부하가 2배가 되면 어떻게 해야 할까? -> 2배라는 기준은?
- 부하를 기술하는 metric으로 삼는 것 -> load parameter
- load param : 초당 요청수, DB R/W rate, active user, cache 적중률
- 가장 적합한 parameter선택은 시스템 설계에 따라 달라짐
- 평균이 metric 기준이 될 수도 있고, 다른 것이 기준이 될 수도 있다

load description

- twitter의 경우
  - tweet 작성 : 새로운 메시지 게시(DB write연산, 평균 4k/s, 피크 12k/s)
  - timeline : 팔로우한 user tweet list(DB read연산, 평균 300k/s)
  - tweet 작성 처리는 쉬우나 follower가 N:M이기 때문에 read요청이 많아짐

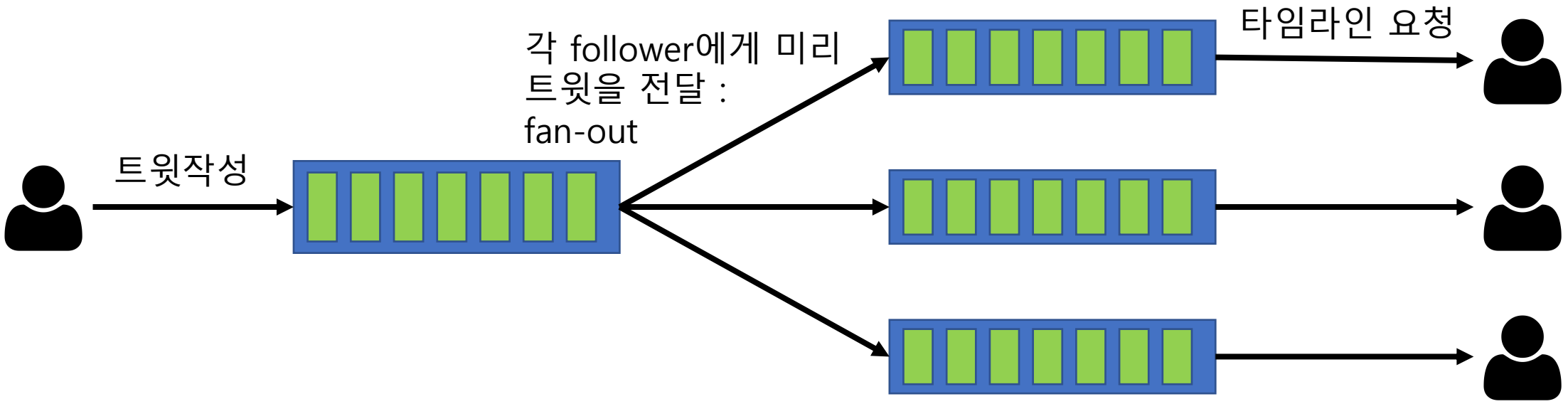
```
INSERT  INTO twwets
        VALUES (sender_id,tweet_text,timestamp)
```

```
SELECT tweets.*,users.* FROM tweets
        JOIN users ON tweets.sender_id = users.id
        JOIN follows ON followee_id =users.id
        WHERE follows.follower_id =current_id
```



load description

- twitter의 경우
  - 읽기 연산이 쓰기 연산보다 초당 요청이 훨씬 많다
  - 병목은 follower join에서 발생할 가능성이 크다
  - 쓰기 연산 효율을 좀 포기하더라도 읽기 효율을 올릴 수 있다면?
  - fan-out : 트윗을 작성하는 순간 모든 follower에게 송신



부하대응접근방식

- scale up / scale out
- automation elastic system / 수동 scale out
  -> 자동이라고 다 좋을까?
- read 양/write 양/data 양/ data 복잡도 / 응답시간 / 접근패턴에 따라 구성해야 할 아키텍처가 다 다르다.
- 실버블렛은 없다

Maintainability

- Operability
- simplicity
- evolvability

operability

- 모니터링 / 결함 발생 시 빠르게 복구
- 시스템 장애, 성능 저하 문제 tracking
- 보안 패치 등을 포함해 플랫폼 최신버전으로 유지
- 서로 시스템에 주는 영향을 파악해 문제 발생 가능성 있는 변경사항 사전 차단
- 발생 가능한 문제를 예측하여 사전 해결
- 배포, 설정 관리 등을 위한 사례와 도구 확보
- 마이그레이션 등 복잡한 유지보수 수행
- 설정 변경으로 생기는 시스템 보안 유지보수
- 예측 가능한 운영과 안정적 서비스 환경을 유지하기 위한 절차 정의
- 운영 문서화

simplicity

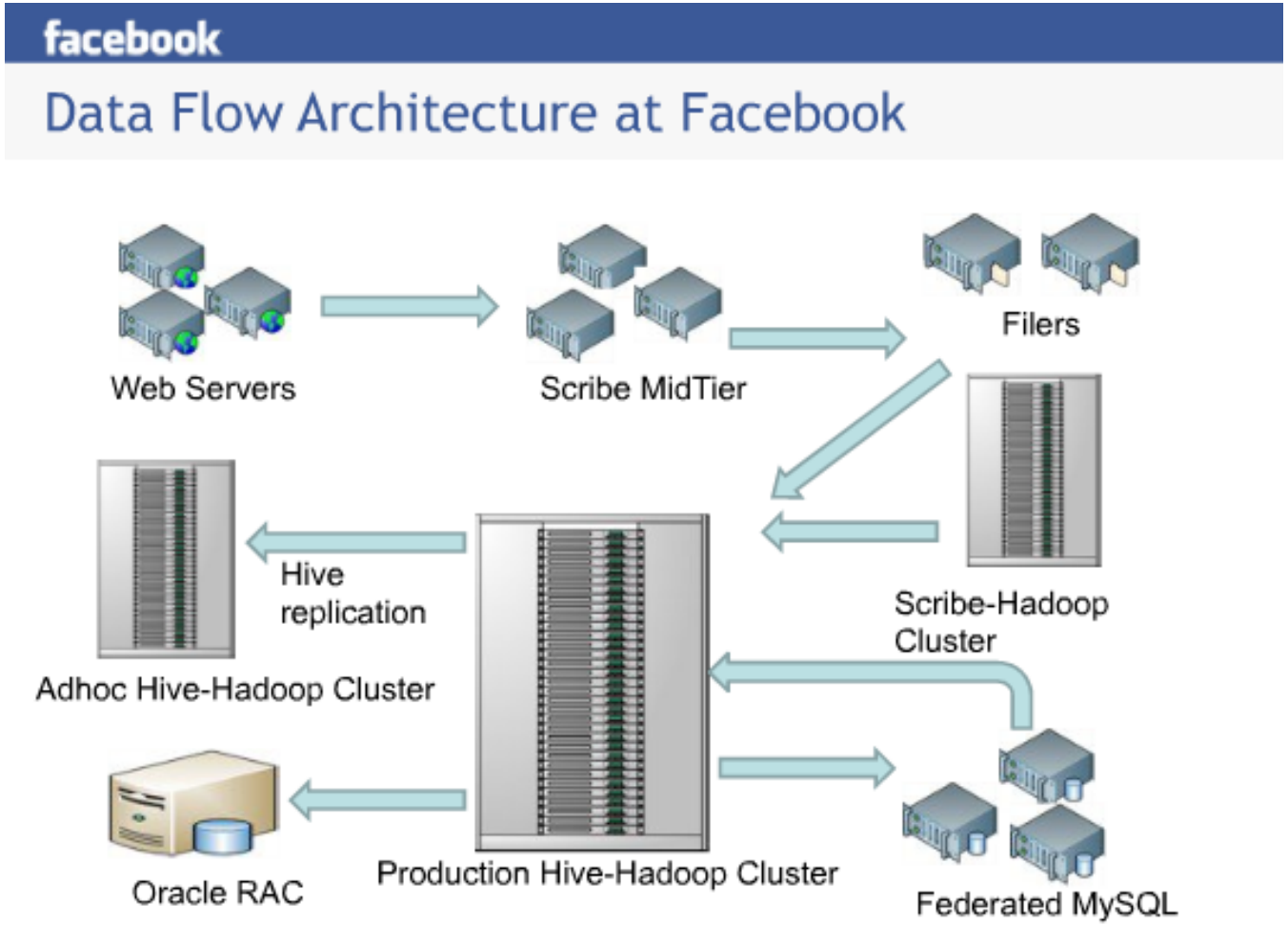
- 복잡하다고 좋은 아키텍처가 아니다
  -> 앞에 아파치 적힌 오픈소스 덕지덕지 펴 바를 필요가 없다

- 복잡도 유발 : 모듈간 강한 커플링, 복잡한 의존성, 일관성 없는 네이밍,
  성능 변태의 성능 삽질, 야매 해결코드

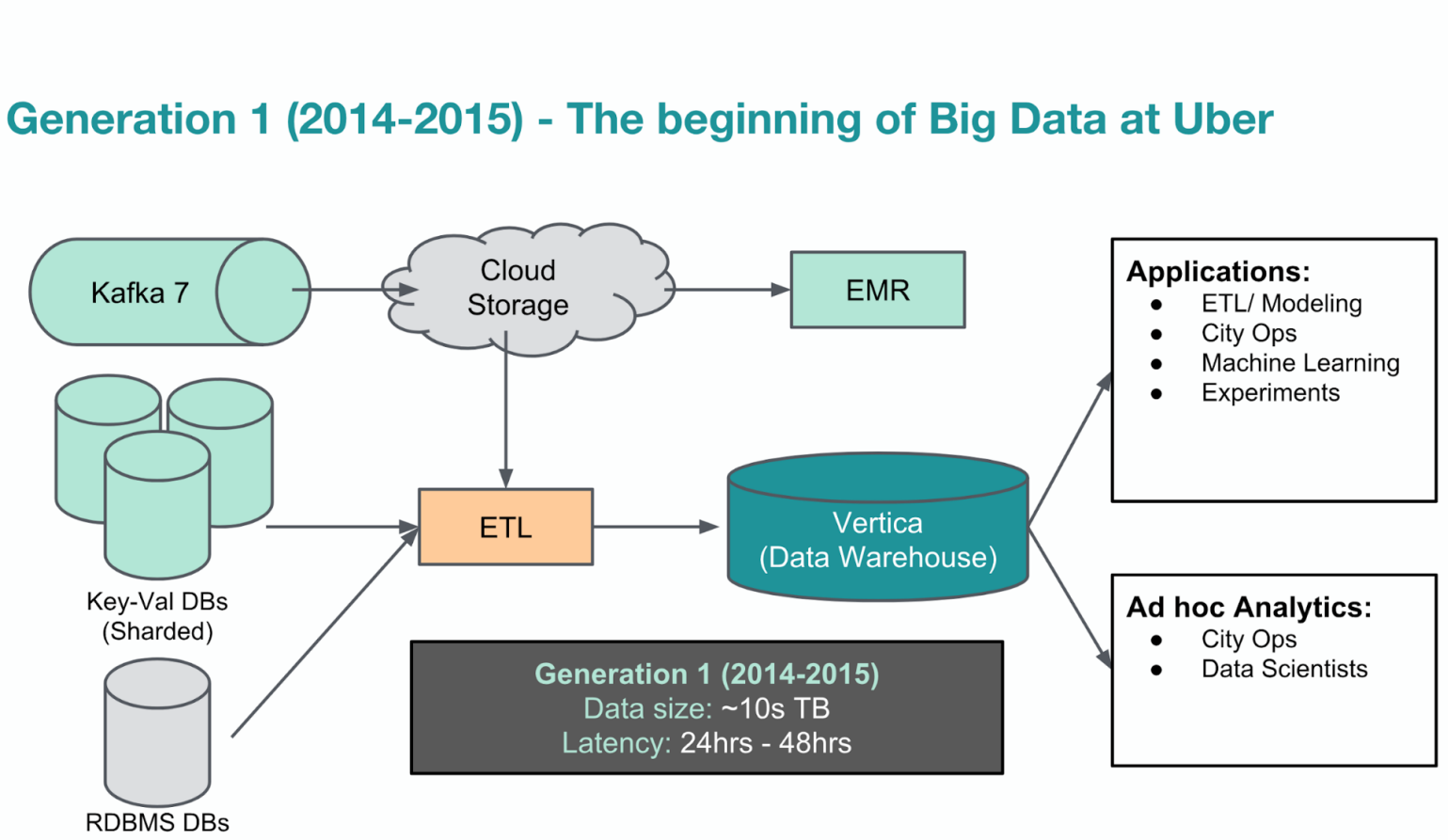
- 우발적 복잡도를 제거해야 한다 -> 추상화

- 지나친 추상화는 다시한번 복잡도를 높인다.
  (스파게티 피하려다 만난 라자냐)

사례 분석 : 페이스북


facebook
Data Flow Architecture at Facebook
Web Servers
Scribe MidTier
Filers
Scribe-Hadoop Cluster
Hive replication
Adhoc Hive-Hadoop Cluster
Oracle RAC
Production Hive-Hadoop Cluster
Federated MySQL

사례 분석 : Uber

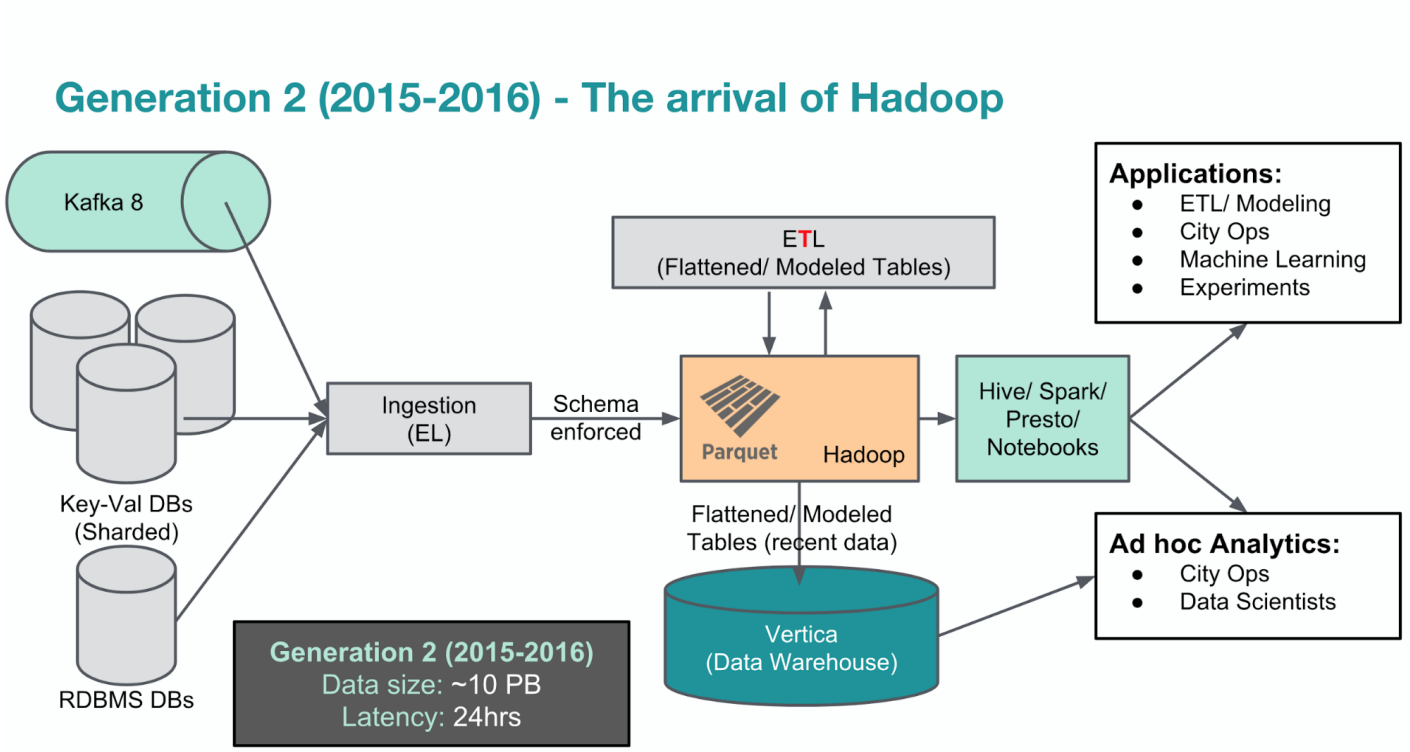


- data를 한곳으로 모으고 엑세스 간소화
- vertica를 datawarehouse로 사용
- 쿼리엔진 인터페이스로 SQL 표준화

한계

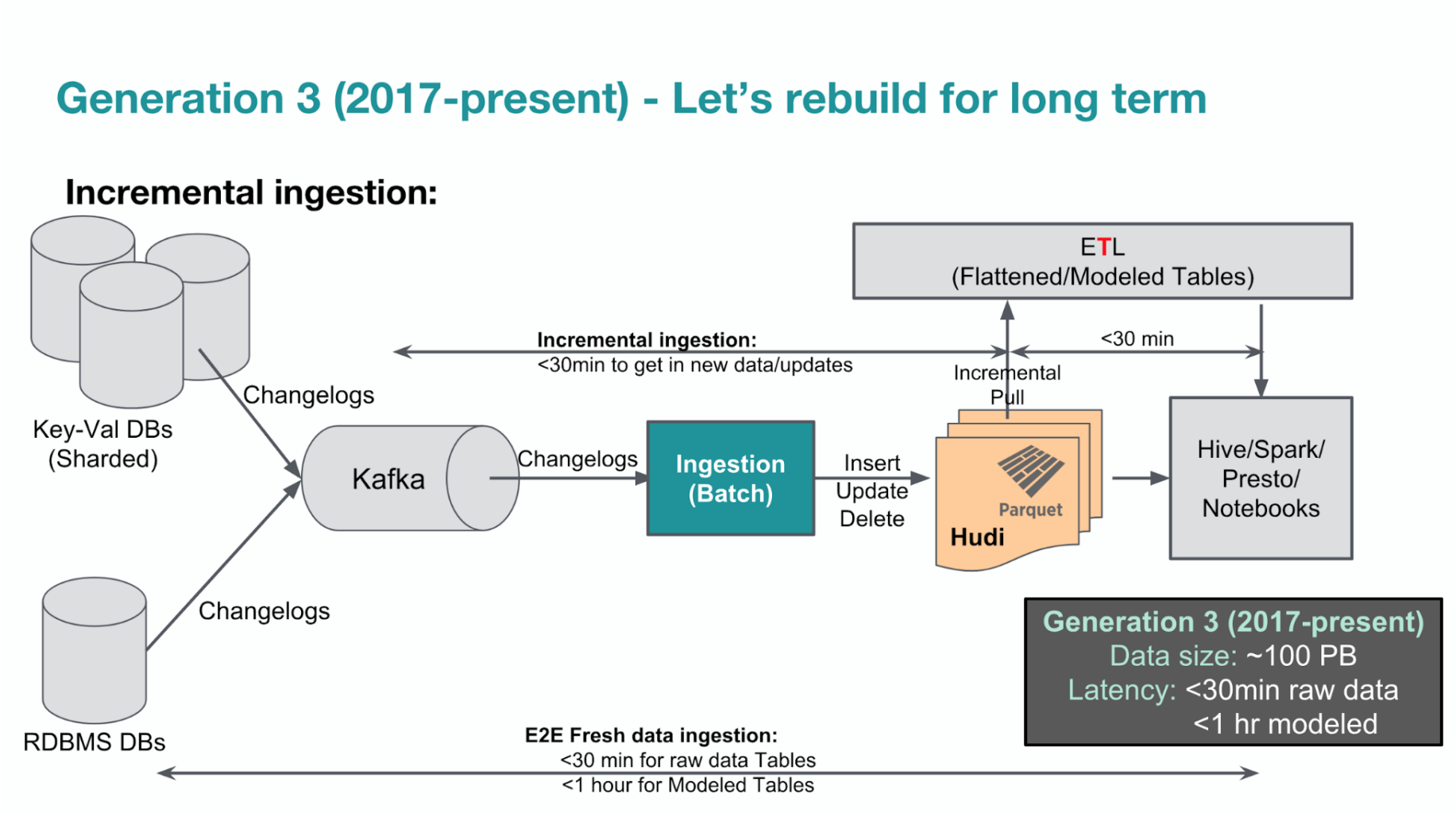
- formal한 스키마 통신 메커니즘이 없음
- source data의 변화에 대처하지 못함
- ingestion job들이 프로듀서 코드변화에 유연하지 못함
- 중앙집중식이었기 때문에 수평적 확장에 한계

사례 분석 : Uber

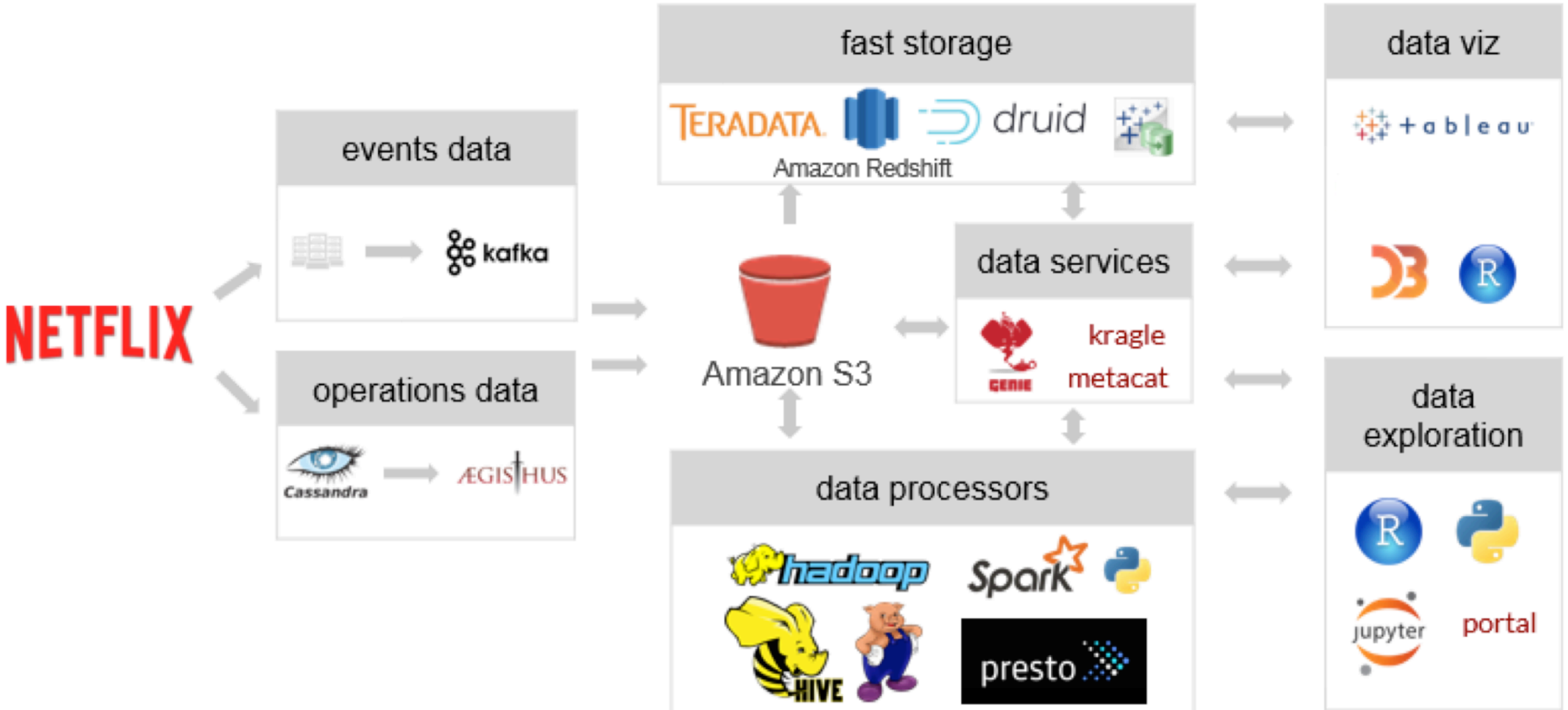


- hadoop 에코시스템으로 rebuilding
- hadoop에서만 데이터 모델링 및 변환이 발생하도록 하여 확장성 보장
- Apache Parquet의 표준 컬럼 파일 형식

사례 분석 : Uber


Generation 3 (2017-present) - Let's rebuild for long term
Incremental ingestion:
ETL
(Flattened/Modeled Tables)
Incremental ingestion:
<30min to get in new data/updates
<30 min
Incremental
Pull
Key-Val DBs
(Sharded)
Changelogs
Kafka
Changelogs
Ingestion
(Batch)
Insert
Update
Delete
Parquet
Hudi
Hive/Spark/
Presto/
Notebooks
Changelogs
RDBMS DBs
Generation 3 (2017-present)
Data size: ~100 PB
Latency: <30min raw data
<1 hr modeled
E2E Fresh data ingestion:
<30 min for raw data Tables
<1 hour for Modeled Tables

사례 분석 : 넷플릭스


NETFLIX
events data
kafka
operations data
Cassandra
ÆGISTHUS
fast storage
TERADATA
Amazon Redshift
druid
Amazon S3
data services
kragle
metacat
GENIE
data processors
hadoop
Spark
HIVE
presto
data viz
tableau
R
data exploration
R
jupyter
portal