RocksDB

왜 RocksDB 를 만들었을까?
Google 이 개발한 LevelDB 기반으로 한 오픈소스

클라이언트-서버 모델에서
애플리케이션 서버 / 데이터베이스 서버 두 가지가 서로 분리되어 네트워크로 통신
네트워크 50ms 라면 디비 서버에서 스토리지(SSD)에 접근하는 시간은 100us, RAM 메모리는 100ns 정도이다.
이런 상황에서 네트워크 통신을 제거하고
데이터베이스 서버와 애플리케이션 서버로 옮기는 것이 좋다. 일종의 embedded storage database.
RocksDB 는 general purpose 에 잘 맞는 것은 아니고
애플리케이션 구현자는 어떤 형태의 앱에 이런 모델이 잘 어울릴지를 고민해야 한다.

그러나 ! ! !
이전에 Berkeley DB, SQLite, Kyoto TreeDB, LevelDB 등이 Open Source 로 존재했기에
위 중에 하나를 선택해서 발전시키기로 했다.
Facebook 이 이미 소유한 High Performant, No transaction log, Fixed Size keys 소프트웨어의 특성을 접목시킨다.

open source benchmarks
RANDOM READ 에서 [kyoto tree >= level >= sqlite3] 순서였고
RANDOM WRITE 에서 [leveldb >> kyoto tree >> sqlite3] 순서였다.
전체적으로 LevelDB 가 제일 훌륭한 performance(특히 random write 에서)를 보여준다고 판단해서
LevelDB 기반으로 RocksDB 를 선택하기로 했다.
또한 프레임워크 구조적인 차이로 많은 어려움이 있기는 했지만 HBase 의 특성도 참고하기도 함.

LevelDB 의 특성, 그리고 그것을 극복하는 방법들
Log Structured Merge Architecture(LSM)

read-write data in RAM ==> (after accumulate) ==> translation log or to flash storage file system
read only data in RAM on disk ==> periodic compaction(reduce and merge) ==> read-write data in RAM ==> scan request from Application

*** LSM explanation : https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/
*** The *compaction* operation is the way to reduce disk space usage by removing unused and old data from database or view index files.

LevelDB 는 사실 그다지 write rate 가 빠르지 않다. (밀려드는 거대한 데이터를 모두 처리하기에는 말이다)
머신 하나당 2MB/sec 이며 오직 1 CPU 만 사용한다.
ㄴ 이를 개선해서 multi-threaded compaction 을 구현했다.
그러면 write-rate 는 10 배 상승, cpu 는 거의 100% 사용할 수 있다.

LevelDB 는 stall 이 있다.
P99 latency 가 몇 십 초 단위이다.
모든 요청을 single-thread 로 처리하다보니 부하가 생겨서 compaction 도 제대로 하지 못한다.

*** p99 latency 는 request 가 도달할 때 99%가 처리되고 1%만이 delay 되는 latency 수치. 낮으면 낮을 수록 좋은 것.

ㄴ 이를 개선해서 thread-aware-compaction 을 구현했다.

3-level compaction 을 하는 방법으로 세 개의 thread-pool 을 두고
각각의 그룹에 해당하는 스레드는 정해진 태스크만 수행한다.
(1) dedicated thread(s) to flush memtable. (memtable 은 메모리 자료 구조.
스토리지의 SST(static stored table)에 통째로 쓰여지기 전에 read-write 가 일어나는 장소이다)
(2) piplined memtables
(3) p99 reduced to less than a second
memtable 을 여러개 두어서 하나가 꽉 찰 때 쯤 다음 memtable 을 사용하는 방식으로
latency 가 극적으로 줄일 수 있었다.

Level DB 는 아주 높은 Write Amplification 이 있다.
70%가 amplified size 였을 정도 → data defragmentation, throughput throttling 문제 등이 있음
예를 들어, 구글의 레벨 db 에서는

Level-0	5bytes			
Level-1	6bytes	→	11bytes	→
Level-2	10bytes		10bytes	10bytes
	[stage1]		[stage2]	[stage3]

ㄴ 이를 해결하기 위해서

Universal Style Compaction 을 만듦
start from “newest file”, include ((next file)) in ((candidate set))
if size_of_((candidate set)) >= size_of_((next file))

Level-0	5bytes		
Level-1	6bytes	→	
Level-2	10bytes	10bytes	

이 방법을 사용하면 항상 wirte amplification < 10 범위에 있는다.

*** Universal Style Compaction : ????? ?????

이렇게 구현하고 나자 read amplification 이 생긴다.
(왜 생길까?)
secondary index service
LevelDB not use blooms for scan
(물론 bloom scan 은 range scan 성능 좋음. 그러나 random read 시에는 오히려 성능 저하)
ㄴ 이를 개선해서 prefix scan 을 추가함

application 이 작동을 시작하고 나서 전체 데이터베이스를 스캔하지 않고
오직 same key prefix 인 범위의 데이터만 스캔
그리고 Bloom Filter 를 이 prefix 에 대해서 적용함

read – modify – write 과정이 느려짐 (2 X IOPs)
이것도 더욱 빠르게 만들수는 없을까 고민해봄
ㄴ MergeRecord 를 도입함

first class citizens of database
create “++” operation in MergeRecord
background compaction merges all

*** IOPs : input / output performance per second
*** first class citizen : In programming language design, a **first-class citizen** (also **type**, **object**, **entity**, or **value**) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable.

1. All items can be the actual parameters of functions.
2. All items can be returned as results of functions
3. All items can be the subject of assignment statements
4. All items can be tested for equality.

*** MergeRecored : this merge operation has these features

- Encapsulates the semantics for read-modify-write into a simple abstract interface.
- Allows user to avoid incurring extra cost from repeated Get() calls.
- Performs back-end optimizations in deciding when/how to combine the operands without changing the underlying semantics.
- Can, in some cases, amortize the cost over all incremental updates to provide asymptotic increases in efficiency.

LevelDB has a Rigit Design
: cannot tune system
: fixed file sizes

이를 넘어서기 위해 RocksDB 는 다음과 같은 원리를 적용한다.
::: pluggable compaction filter. e.g TimeToLive
::: pluggable memtable/sstable for RAM/FLASH
::: pluggable Compaction Algorithm

Level DB 로부터 가져온 것
→ LSM (Log Structured Merge)
→ gets(), puts(), scans() of keys
→ Forward and Reverse Iteration

RocksDB is born!

- Key-Value persistent store
- Embedded
- Optimized for fast storage
- Server workloads

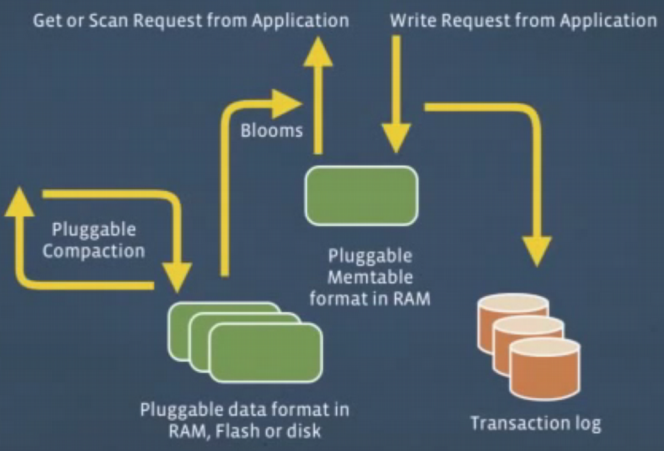
RocksDB

What is it not?

- Not distributed
- No failover
- Not highly-available, if machine dies you lose your data

*** failover : 하나의 메인 시스템에서 에러가 발생해서 정상적인 운영이 불가능할 경우에 stand-by 머신이 동일한 역할을 넘겨받아서 시스템 운영을 이어나가는 것.

RocksDB Architecture



Rocks DB 는 embedded store database 라는 말처럼 한 대의 로컬 머신에서 사용하는 것을 가정으로 만들어졌다. 클러스터 환경에서 사용하기 위해서는 Dynomite DB 를 선택한다.
http://www.dynomitedb.com/docs/dynomite/v0.5.8/introduction/