**System Programming Project 2**

담당 교수 :

이름 : 변성진

학번 : 20180546

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)**
* 리눅스의 socket interface를 활용해서 client process의 요청에 응답할 수 있는 server (process)를 설계한다
* 설계는 synchronous 방식 Event-Driven, async 방식인 Thread-Based 두가지로 설계한다
* 성능 측정을 위해 benchmark 어플리케이션을 통해 두 가지 방식의 성능 비교평가 / Thread-based의 최적 쓰레드 수 / 쿼리 유형에 따른 성능 측정 /   client process 수 변화에 의한 성능 평가 로 parameter를 변경하며 어떤 환경에서 최적화가 이루어지는지 평가한다

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**
* **1)Task 1: Event-driven Approach**
* event-driven 방식은 모든 client 와의 통신을 한가지 프로세스에서 순차적으로 수행한다.
* 동작 (1) server booting
* server를 ./stockserver {port} PRODUCTION으로 배포 버전으로 구동한다. 단 ./stockserver {port}만으로 구동해도 default는 PRODUCTION으로 되어 실행된다.(BENCHMARK version은 아래에서 설명 예정)  Server running on {port} 를 STDOUT에 출력하며 connection을 기다린다
* 동작 (2) client connection
* client와 연결을 성공하면 Connected to ({clienthost}:{clientport})를 출력한다
* 동작 (3) client interactive
* client가 데이터를 전송하면 입력받은 데이터의 바이트 수를 서버의 stdout에 Server received n bytes 포맷으로 출력하고, 데이터를 파싱해 알맞은 동작을 수행한다
* 동작 (3-1) show
* 클라이언트가 "show\n" 를 전송하면 현재 서버에 적재된 주식 정보를
* "{ID} {COUNT} {PRICE}\n" 으로 모두 출력한다
* 동작 (3-2) buy
* 클라이언트가 "buy {ID} {COUNT}\n" 을 전송하면 해당 ID를 가진 주식이 존재하는지 파악한다. 없으면 NO SUCH STOCK을 전송한다
* 존재한다면 남은 주식 수량이 살 주식보다 부족하다면 NOT ENOUGH LEFT STOCK을 전송한다
* 해당 주식의 수량을 감소시키고 "[buy]success\n"를 전송한다
* 동작 (3-3) sell
* 클라이언트가 "sell {ID} {COUNT}\n" 을 전송하면 해당 ID를 가진 주식이 존재하는지 파악한다. 없으면 NO SUCH STOCK을 전송한다
* 해당 주식의 수량을 증가시키고 "[sell]success\n"를 전송한다
* 동작 (3-4) exit
* 클라이언트가 "exit\n"을 전송하면 Connection Closed를 서버의 stdout에 출력하고 해당 소켓을 닫는다
* 동작 (3-5) enter
* 클라이언트가 "\n"을 전송하면 "Please Type Command\n"를 전송한다
* 동작 (3-5) no cmd
* 클라이언트가 존재하지 않는 명령어를 전송하면 "No Such Command\n"를 전송한다
* 동작 (3-6) invalid argument
* 클라이언트가 존재하는 명령어를 전송했으나 잘못된 argument를 포함했을 경우 "Invalid Argument\n"를 전송한다
* 동작 (4) read stock.txt file
* 서버를 처음 부팅하면 디스크의 stock.txt 에 저장된 "{ID} {COUNT} {PRICE}\n" 포맷을 binary tree형태로 메모리에 적재한다
* 동작 (5) file synchronization
* 모든 클라이언트와의 연결이 종료되었거나 / 마지막 파일과 메모리 동기화 시간이 16초를 초과했을 경우 메모리 상의binary tree를 stock.txt에 동기화해준다
* **2)Task 2: Thread-based Approach**
* thread\_based approach는 두가지 유형의 thread가 각각의 context를 실행한다
* 하나의 parent thread는 client의 연결 요청을 받으면 global scope(sbuf)에 connfd,client ip,client 를 저장한다
* N개의 child thread(worker thread)는 parent thread가 global scope에 저장한 정보를 바탕으로 각자의 문맥을 실행한다
* 동작 (1) server booting
* server를 ./stockserver {port} PRODUCTION {thread\_n}으로 실행할 수 있다. 단 ./stockserver {port}만으로 실행해도 default는 PRODUCTION, child 쓰레드의 갯수는 4개다.
* Server running on {port} 를 STDOUT에 출력하며 parent thread는 connection을 기다리고, worker thread들은 parent thread가 전역공간에 connection에 대한 정보를 저장하기를 기다린다
* 동작 (2) client connection
* client와 연결을 성공하면 Connected to ({clienthost}:{clientport})를 출력한다
* 동작 (3) client interactive
* client가 데이터를 전송하면 입력받은 데이터의 바이트 수를 서버의 stdout에 Server received n bytes 포맷으로 출력하고, 데이터를 파싱해 알맞은 동작을 수행한다
* 동작 (3-1) show
* 클라이언트가 "show\n" 를 전송하면 현재 서버에 적재된 주식 정보를
* "[show]success\n"와 함께
* "{ID} {COUNT} {PRICE}\n" 으로 모두 출력한다
* 동작 (3-2) buy
* 클라이언트가 "buy {ID} {COUNT}\n" 을 전송하면 해당 ID를 가진 주식이 존재하는지 파악한다. 없으면 NO SUCH STOCK을 전송한다
* 존재한다면 남은 주식 수량이 살 주식보다 부족하다면 NOT ENOUGH LEFT STOCK을 전송한다
* 해당 주식의 수량을 감소시키고 "[buy]success\n"를 전송한다
* 동작 (3-3) sell
* 클라이언트가 "sell {ID} {COUNT}\n" 을 전송하면 해당 ID를 가진 주식이 존재하는지 파악한다. 없으면 NO SUCH STOCK을 전송한다
* 해당 주식의 수량을 증가시키고 "[sell]success\n"를 전송한다
* 동작 (3-4) exit
* 클라이언트가 "exit\n"을 전송하면 Connection Closed ({clienthost}:{clientport})를 서버의 stdout에 출력하고 해당 소켓을 닫는다
* 동작 (3-5) enter
* 클라이언트가 "\n"을 전송하면 "Please Type Command\n"를 전송한다
* 동작 (3-5) no cmd
* 클라이언트가 존재하지 않는 명령어를 전송하면 "No Such Command\n"를 전송한다
* 동작 (3-6) invalid argument
* 클라이언트가 존재하는 명령어를 전송했으나 잘못된 argument를 포함했을 경우 "Invalid Argument\n"를 전송한다
* 동작 (4) read stock.txt file
* 서버를 처음 부팅하면 디스크의 stock.txt 에 저장된 "{ID} {COUNT} {PRICE}\n" 포맷을 binary tree형태로 메모리에 적재한다
* 동작 (5) file synchronization
* **모든 클라이언트와의 연결이 종료되어 모든 worker thread가 context를 실행하지않고 잠들어있는 경우**/ 마지막 파일과 메모리 동기화 시간이 16초를 초과했을 경우 메모리 상의 binary tree를 stock.txt에 동기화해준다
* **3)Task 3: Performance Evaluation**
* ./benchmark <port> <client\_number> <total\_query> <-soption>
* port : 서버가 올라갈 port numberclient\_number : 연결을 시도할 client process의 총 갯수total\_query : 전체 client가 서버에 요청할 모든 쿼리의 갯수, 만약 total\_query=10000 client\_number=4라면 각 client에서 2500개의 쿼리를 서버에 보낸다-soption : -e 는 event\_driven 방식의 서버 benchmarking, -t는 thread\_based 방식의 서버 benchmakring-t뒤에 숫자를 붙여주면 총 worker thread의 갯수를 조정할 수 있다. default는 4개 worker thread다.
* example) ./benchmark 1024 64 600000 -t8 : 1024 포트에서 thread\_based 서버가 부팅된다. worker thread는 총 8개이다. 64개의 클라이언트가 총 600,000개의 쿼리를 나누어서 서버에 요청한다
* ./benchmark 1024 64 600000 -e 위와 동일한 조건에 event\_driven 서버가 부팅된다
* 총 4가지 척도를 측정한다
* 서버부팅시간 / 랜덤쿼리 / show 쿼리 / modify 쿼리
* 실행시간(execution time)과 , Query Throughput을 출력한다.
  1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Task1 (Event-driven Approach with select())**
* Multi-client 요청에 따른 I/O Multiplexing 설명
* Event driven 서버는 연결되어있는 connection fd를 저장하는 read set 자료구조를 커널 혹은 유저스페이스에 저장하고 커널로 꾸준히 request를 보내 read set 에 포함되는 connection의 event들을 관찰한다. 만약 event가 발생했을 경우 모든 명령어를 순차적으로 하나의 프로세스 내에서 처리한다. select 는 이 read set을 유저스페이스에서 관리한다.
* epoll은 connection에 관한 정보를 kernel에서 관리한다. select는 user space에서 관리한다. 다만 userspace에서 꾸준히 I/O에 입력이 들어왔는지 확인 request를 보내줘야한다는 점에서 overhead가 발생한다
* **Task2 (Thread-based Approach with pthread)**
* Thread-based Approach에는 race condition이 발생할 여지가 있는 부분 세가지가 있다
* 첫번째는 worker\_thread들이 connection을 획득하는 부분이고 두번째는 worker\_thread들이 메모리에 적재된 binary tree의 주식 갯수를 동시적으로 modify를 요청할때, 마지막으로는 reader(show)-writer(modify)간의 문제다
* ->Race Condition 1 : Master Thread의 Connection 관리
* Master thread(parent thread)는 client와의 connection에 성공하면 connection에 대한 정보인 struct sockaddr\_storage clientaddr와 int connfd를  sbuf의 wating\_connfd 배열에 저장한다.
* wainting\_connfd 배열은 런타임에 크기가 설정되고 , common.h 에 PENDING\_CONNECTION\_N 매크로로 4096개 만큼 할당된다. 즉 최대 4096개의 클라이언트 요청까지 대기상태로 둘 수 있다는 뜻이다
* ->Race Condition 2 : Worker Thread간 동시적 Modify
* 만약 두가지 쓰레드가 동시적으로 하나의 stock에 대해 semaphore 없이 접근하게 되면 bug가 발생할 수 있기때문에 각각의 노드 마다 semaphore를 만들어 두고 변경하려면 해당 세마포어를 획득한 뒤 수정, 수정후 세마포어 반납의 형태로 만들어야한다
* ->Race Condition 3 : reader-writer 간 문제
* binary tree에는 동시에 reader - writer 두가지 유형이 접근 할 수 없다. 즉, binary tree에 접근 중인 한 쓰레드의 command 유형이 show라면 나머지 접근 중인 쓰레드들도 모두 show이다
* 단, reader 보다 writer에 우선순위를 둔다. 만약 writer가 binary tree에 접근중이라면 reader는 들어갈 수 없고 스핀 루프를 돈다. **단 writer는 계속해서 binary tree에 들어갈 수 있다** ( 이 스핀루프가 bottleneck인데 개선 방안을 찾으면 좋을것 같다) writer 가 모두 종료된 후에야 reader가 들어갈 수 있는데 이는 reader의 starvation을 야기할 수 있다
* 역시 reader가 binary tree에 접근중이라면 writer는 들어갈 수 없다(스핀루프). 단 스핀루프로 pending 중인 writer가 존재한다면 reader는 는 binary tree에 들어갈 수 없다. read는 이내 수행을 끝내고, writer가 접근을 시작한다
* **Task3 (Performance Evaluation)**
* metric은 간단하게 서버가 클라이언트와 연결된 후부터 클라이언트와의 연결이 끊어질때까지의 걸리는 시간으로 정의한다
* total\_query/execution\_time으로 서버의 Query Throughput/Sec을 측정할 수 있다.
* total\_query가 같은 상황에서 클라이언트 수를 조정하여 **적은 수의 클라이언트의 많은 쿼리를 처리하는것과 많은 수의 클라이언트의 적은 쿼리를 처리하는 것 중 어떤것이 throughput이 우수한지** 확인한다.
* -> thread based 서버의 경우 각각의 worker thread가 병렬적으로 처리하기때문에 클라이언트 갯수와 큰 연관이 없겠지만, event driven 서버의 경우 select에 의한 오버헤드 때문에 클라이언트 갯수가 작아져
* 쓰레드 기반 서버의 경우에 **몇개의 worker thread를 생성했을때 최적인지**를 확인한다
* ->아마도 OS에서 추상화된 CPU와 갯수와 근접할수록 최적일 것이다.
* Event\_Driven 와 Thread\_based 중 Throughput 이 어떤 것이 더 높은지 측정한다
* 항상 Thread based가 Event Driven 서버를 압도할 것이다.
  1. **개발 방법**

**event/thread 공통 :**

stockfile\_handle.c : 실행파일 stockserver와 동일한 경로에 위치한 stock.txt를 메모리에 binary tree형태로 적재시키고(read\_stockfile) 메모리의 binary tree를 stock.txt에 fsync하는 (fsync\_stockfile) 함수가 있다

util.c : csapp.c의 open\_listenfd, rio 패키지 를 커스텀한 함수가 있고 socket\_close는 close의 사용자 친화적인 래퍼함수다(연결 종료메시지, 에러메시지)

parser.c : 클라이언트의 command line을 파싱해서 command / argument를 찾거나 잘못된 요청의 경우 예외처리를 한다

binary\_tree.c :  삽입(insert), 찾기(find), 수정(modify), string에 binary tree를 stock.txt format으로 출력하기(print\_to\_buf) 가 있다.

impl.c : binary\_tree.c의 함수들을 client command에 대응하도록 만든 wrapper함수다. buy/sell의 경우 주식 노드를 찾고 (find) 수정(modify)을 한다

common.c : 전역에서 사용되는 변수 및 구조체 / 환경변수 / 매크로 함수 및 변수 / 사용하는 standard library 를 담고 있다

**event driven**

network.c FD POOL을 초기화하는 init\_pool, connection들을 관찰하고 있는 see\_pool, connection의 command를 파싱/실행/결과를 전송하는 write\_pool 함수가 있다

**thread based**

binary\_tree.c  
공통된 부분에서 binary\_tree를 modify하기 전 획득해야하는 semaphore가 추가된다(modify\_lock)

sem\_wait(&stock->modify\_mutex);

stock->count+=count;

sem\_post(&stock->modify\_mutex);

threading.c

typedef struct \_waiting\_connfd {

int connfd;

struct sockaddr\_storage clientaddr;

}waiting\_connfd;

typedef struct{

waiting\_connfd \*waiting\_connfd;

int n;

int front,rear;

sem\_t mutex;

sem\_t empty\_slots;

sem\_t items;

} sbuf\_t;

공통된 부분에서 connection에 관한 정보를 관리하기 위한 sbuf\_t가 추가되었다. 변수 N은 최대로 pending 가능한 connection의 수를 의마한다 worker thread가 connection을 획득하기 위해선 sem\_t mutex에 락을 획득해야한다.

network\_worker 함수에서는 conenction을 획득하고 놀고 있는 쓰레드의 갯수를 의미하는 변수인 idle\_threads를 감소시키고, service라는 함수를 호출한다. 이후 연결이 종료되어 service함수가 리턴되면 idle\_threads를 다시 증가시키고, 만약 모든 쓰레드가 idle 한 상태라면 sbuf에 락을 걸어놓고 자신의 thread group id, 즉 자기 자신에게 SIGSYNC 시그널을 보낸다 (SIGUSR1로 매크로처리되어있음).

SIGSYNC 핸들러는 fsync\_stockfile을 실행한뒤 다시 sbuf에 락을 해제하며 핸들러가 종료된다

network.c

service 함수는 무한루프를 돌며 쓰레드가 잡은 connection이 종료되기 전까지 커맨드를 파싱/실행/결과를 전송한다

impl.c

impl.c의 sell과 buy 에서는 현재 접속중인 reader(writer) 수 인 reader\_n과 writer\_n을 스핀루프로 확인함으로써 race condition을 처리하고있다

util.c

thread\_safe\_printf : printf는 thread\_unsafe한 함수이기때문에 thread\_safe한 vprintf로 래핑했다.

**benchmark**

benchmark.c의 경우 컴파일하여 실행파일에 위에서 언급한 parameter를 argv로 넣어주면 하나의 서버에서 많은 클라이언트들이 접속/쿼리를 요청하며 성능을 평가한다

다만 비교 평가를 하기에는 반복 작업이 많아지므로 macro.c 파일에 loop에서 파라미터를 바꿔가며 실험이 가능하도록 코드를 짜놓았다

다만 꼭 필요한 부분만 주석처리를 해서 사용하기를 추천하는데, 너무 시간이 많이 걸릴 수 있기 때문이다

1. **구현 결과**

Thread\_based에서는 connection에 관한 정보를 각 쓰레드가 가지고 있어서 connection이 종료될때 어떤 Connection이 종료됬는지 client ip와 port를 출력할 수 있었지만 event\_driven 방식에서는 구현하지 못했다. 아마 connfd를 key 값으로하는 배열을 만들어서 저장해놓고 -1을 수신하면 해당 connfd를 찾아서 client ip와 port를 출력하는 방식으로 구현하면 될것 같긴 하다

Thread\_based에서 reader / writer mutex부분에서 spin lock을 도는 부분이 있는데 이 부분이 정상적으로 작동하긴 하지만, bottle neck인 부분이다. 개선할 여지가 있다

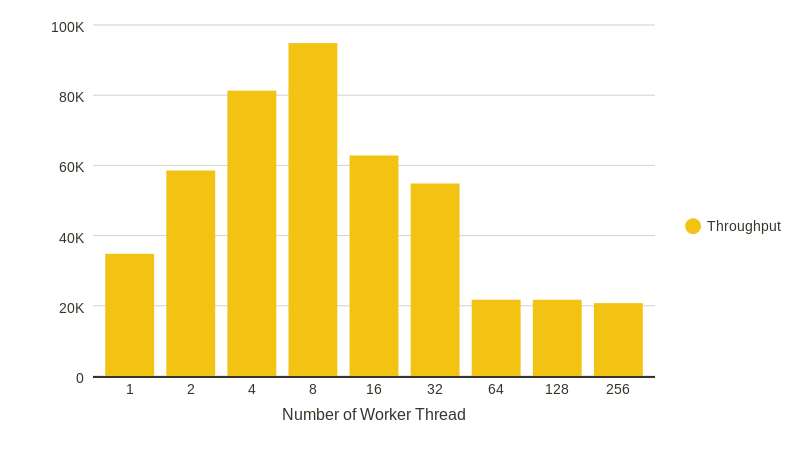
1. **성능 평가 결과 (Task 3)**

Benchmark environment

모두 OS : 64 bit Linux(Ubuntu 20.04) ,5.13.0-4 인텔® 코어™ i5-1035G4 1.10Ghz 프로세서, 16GB Memory 상에서 벤치마킹하였다

**1. 최적 쓰레드 수 평가**

total query = 2^22(4194304) , client\_n=64

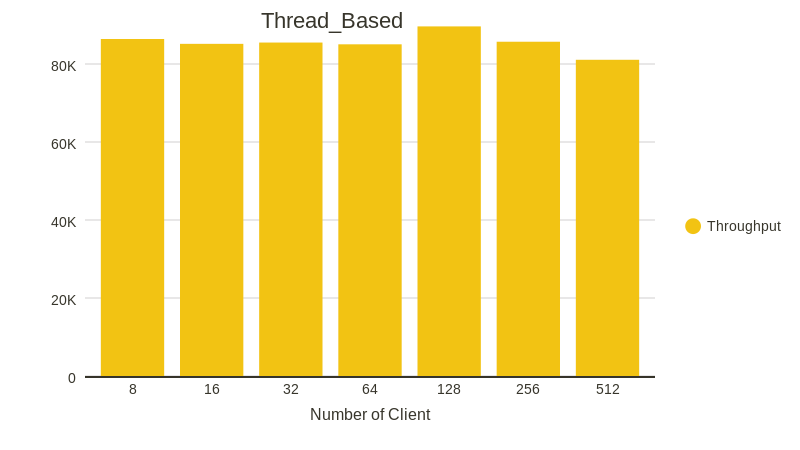


-> -t8일때 가장 throughput이 peak인 것을 확인할 수 있는데 , 이는 benchmark envirnoment의 하이퍼쓰레딩에 의해 CPU갯수가 8개이기 때문이다.

**2. x=client number y=execution time**

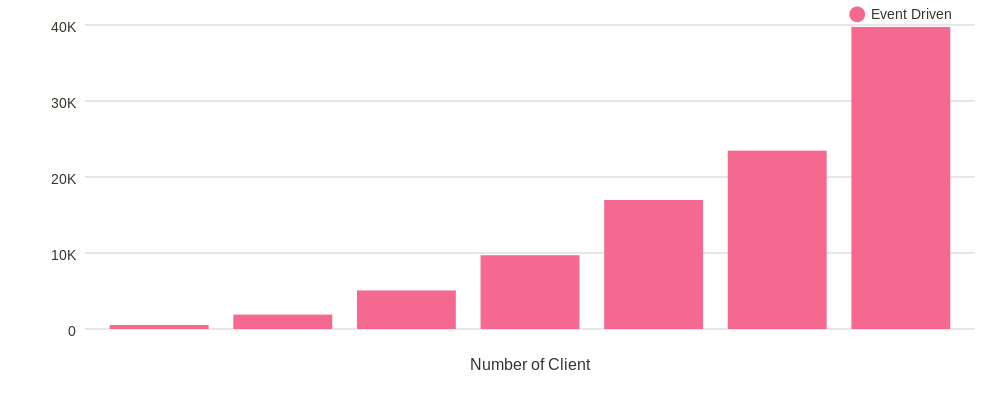
1)thread\_based

thread\_n=32 , total query= 2^20

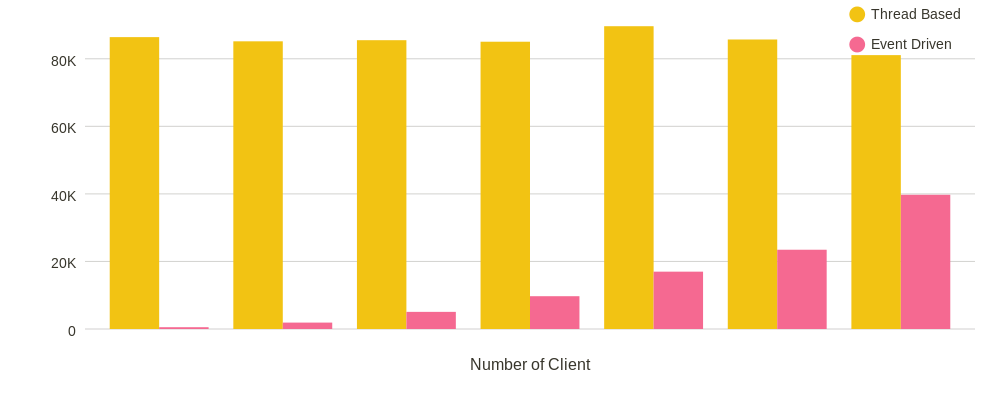


->thread based server의 경우 client의 갯수가 256~8개일때까지 큰 성능차이가 없음을 확인할 수 있다. 왜냐하면 최대 쓰레드가 동시에 8개까지만 돌아갈 수 있음으로 1번 실험과 같은 이유라고 추측한다.

**1)event\_driven**

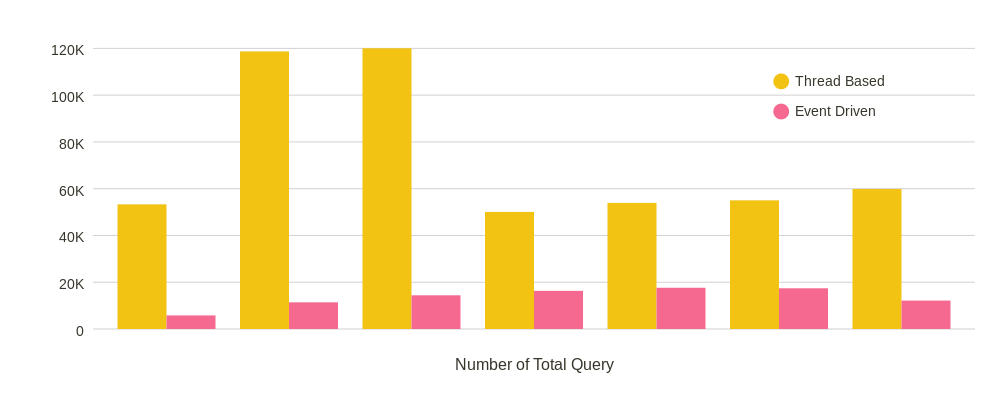


->event driven server의 경우 client\_n 이 감소할수록 급격하게 성능이 나빠지는 것을 볼 수 있다. 왜냐하면 한번의 select 루프에 처리할 수 있는 쿼리의 갯수가 작아지기때문에, client n이 감소할수록 select의 오버헤드가 급격히 증가하기 때문이다.



비교 결과 Thread based가 절대적으로 우수하며, client의 병렬적 처리에도 큰 영향을 받지 않는 모습을 확인할 수 있다

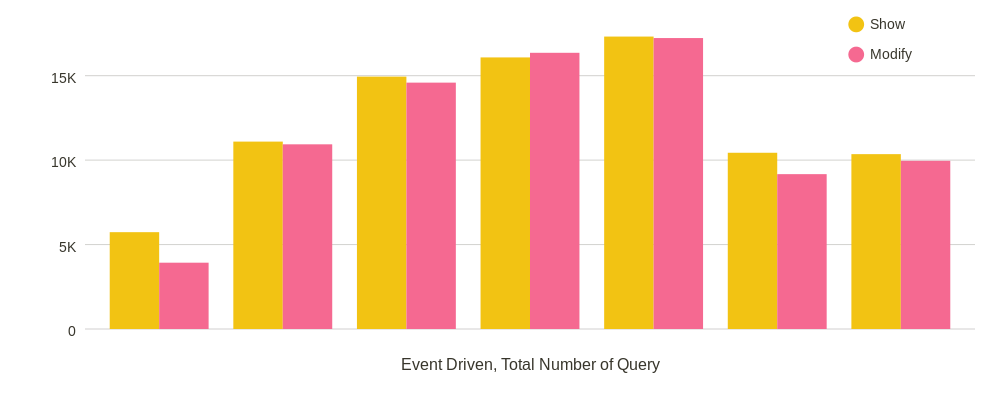
**3. thread vs event**



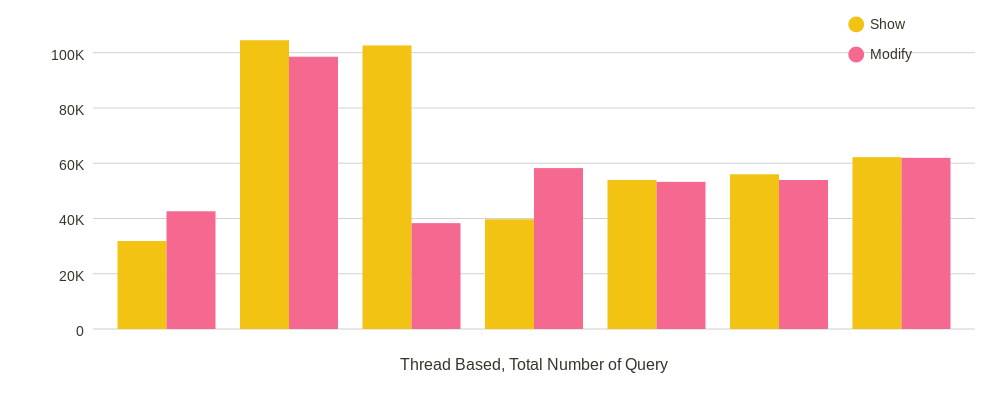
-> event based server의 경우 생성해야할 쓰레드가 없기 때문에 부팅속도는 thread based server보다 빠르다

-> 그러나 전체적인 Thoughput은 많게는 12배, 최소 3배 이상 thread\_Based server가 높은것을 확인할 수 있다.

4.query type



event driven server의 경우 쿼리 타입에 따른 로드워크 차이가 적음을 확인할 수 있다.



thread based server의 경우 lock에 관한 이슈와 thread 간 context switch에 관한 이슈가 있기때문에 때에 따라선 어느 한쪽이 더 우수하다.