

명지대학교 인문캠퍼스 핵심 버스노선 교통흐름 분석

레포트 정보

학교 : 명지대학교

학기 : 2019 년 2 학기

과목명 : 빅데이터 프로그래밍

지도교수 : 권동섭 교수님

작성자 : 명지대학교 경영정보학과 60171918 이재준

[문제정의] 배경

명지대학교 (인문캠퍼스)의 교통 특성상 가장 많이 사용되는 대중교통은 버스이다. 하지만 버스가 이용하는 차선은 왕복 1 차선밖에 되지 않는다. 이는 자연스럽게 많은 교통체증을 약이 한다. 그만큼 교통 혼잡이 많이 일어나고 작성자 또한 학생 중 한 명으로서 다른 학생에게 빈번하게 들은 말 중 한 가지가 “지금 버스 오래 걸릴까?”이다. 이런 상황을 보안을 유지하기 위해 셔틀버스를 운행하나 한정적인 좌석과 고정된 배차 시간으로 인해 불편을 겪고 있다. 결과적으로 명지대학교 학생은 마을버스를 주 대중교통으로 이용하고 있다.

[문제정의] 목적

앞서 얘기한 배경을 사유로 버스 배차 간격을 시간대별로 분석하여 가장 교통이 혼잡한 시간대와 혼잡하지 않은 시간대를 나눠보려고 한다. 첫 번째로 명지대학교 정문 방향의 정류장[명지대(13195)]과 이삭 토스트 명지대점 앞의 정류장 [명지대(13194)] 2 가지의 변화하는 버스 배차 간격을 조사하고, 전체적인 버스 배차 간격의 평균을 함께 조사할 것이다. 본 레포트는 교통 흐름의 특성상 차가 많을수록 버스의 배차 간격은 점점 늘어난다는 사실을 전제로 조사하게 되었다.

[문제정의] 목표

버스가 운영하는 시간을 1 시간 단위로 나누어 배차 간격이 짧은 시간대부터 긴 시간대를 파악하고 특정 버스를 어떤 시간대에 피해야 하는지 파악한다. 결과값을 가시화 프로그램을 통해 전달하는 것이 최종 목표이다.

[문제정의] 사전조사

[명지대(13195)]와 [명지대(13194)] 정류장의 정보를 먼저 조사하였다.



그림 1_ 명지대학교 인문캠퍼스 지도

위 그림 1의 빨간색은 정류장 [명지대 13195], 파란색은 [명지대 13194] 정류장이며 녹색으로 표시된 부분이 명지대학교 인문캠퍼스의 행정동 건물이다.

정류장 [명지대 13195]는 버스 노선 7017, 7019, 7021, 7611, 7612, 7713, 7734가 운행 중이다. (2019.12.16 기준)

정류장 [명지대 13194]는 버스 노선 7017, 7019, 7021, 7611, 7612, 7713, 7734가 운행 중이다. (2019.12.16 기준)

시스템 아키텍처

전체적인 시스템의 흐름은 아래 그림 3 과 동일하다.

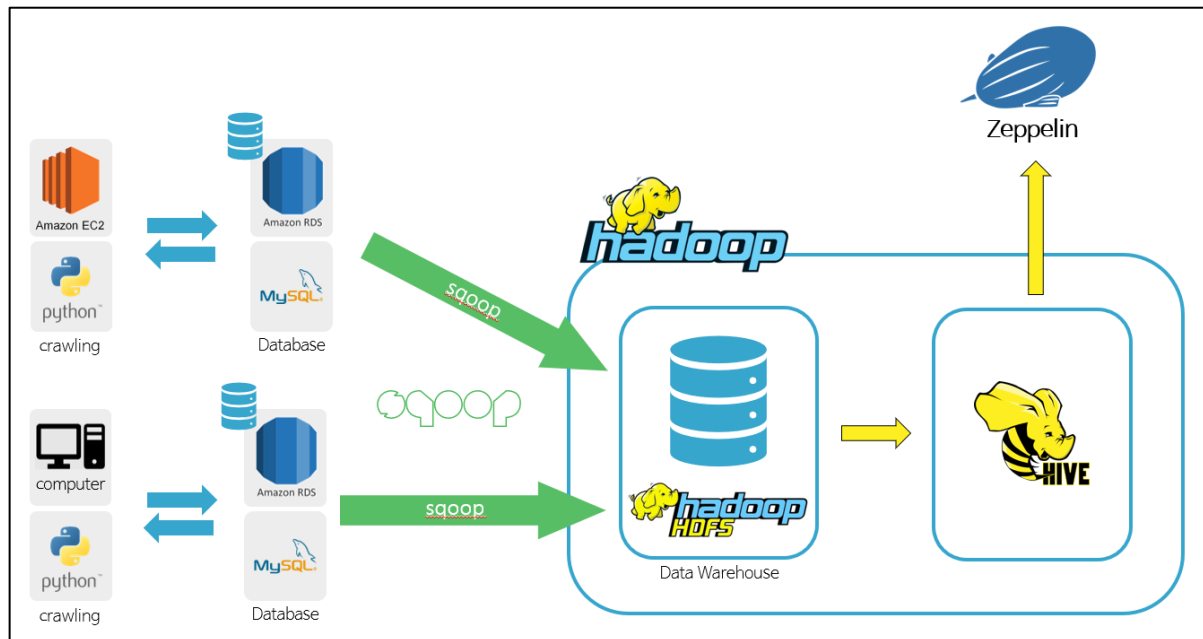


그림 2_시스템 아키텍처

1. m.bus.go.kr 을 크롤링할 웹 크롤러를 24 시간 가동하였다.
 - A. Amazon EC2 에 Ubuntu 를 사용하여 Python 으로 만들어 놓은 웹 크롤러를 Linux 의 Nohub(데몬)을 사용하여 가동하였다. (그림 4 참고)

```

ubuntu@ip-172-31-40-24: ~/bus-live
[2]- Killed                  nohup python3 main.py runserver 0.0.0.0:8000
ubuntu@ip-172-31-40-24:~/bus-live$ ps-i

Command 'ps-i' not found, did you mean:

  command 'psi' from deb psi

Try: sudo apt install <deb name>

[3]+  Killed                  nohup python3 main.py runserver 0.0.0.0:8000
[3]+  Killed                  nohup python3 main.py runserver 0.0.0.0:8000
ubuntu@ip-172-31-40-24:~/bus-live$ ps -l
F S  UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1745  1743  0  80   0 -  5814 wait  pts/0    00:00:00 bash
0 R  1000    19705  1745  0  80   0 -  7334 -    pts/0    00:00:00 ps
ubuntu@ip-172-31-40-24:~/bus-live$ vi main.py
ubuntu@ip-172-31-40-24:~/bus-live$ nohup python3 main.py runserver 0.0.0.0:8000
[1] 19707
nohup: ignoring input and appending output to 'nohup.out'
F S  UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1745  1743  0  80   0 -  5814 wait  pts/0    00:00:00 bash
0 R  1000    19707  1745  0  80   0 -  6903 -    pts/0    00:00:00 python3
0 R  1000    19708  1745  0  80   0 -  7334 -    pts/0    00:00:00 ps
ubuntu@ip-172-31-40-24:~/bus-live$
    
```

그림 3_Linux nohup

B. 일반 컴퓨터에 Windows 10 을 설치하고 Python 으로 제작한 웹 크롤러를 사용해 가동하였다. (그림 5 참고)

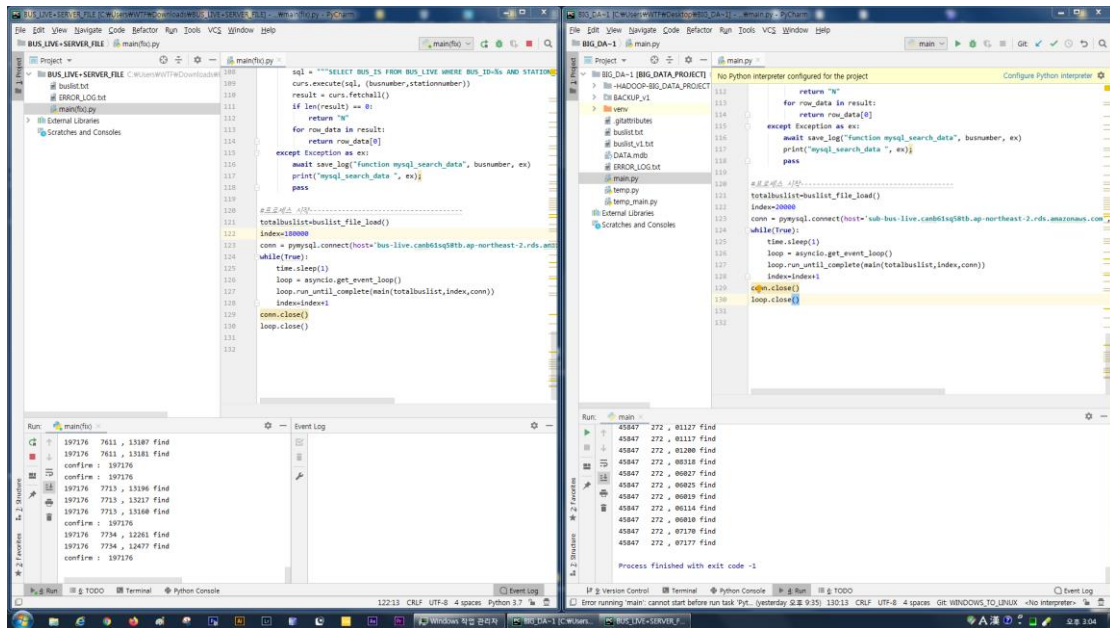


그림 4_Windows Python

2. 웹 크롤러를 통해 Amazon RDS 의 Mysql 에 접속하여 데이터 SELECT 와 INSERT 를 병행하였다.

- A. 앞서 크롤링과 데이터 입력이 끝나면 쉼 틈없이 Python 을 통해 다시 크롤링 하길 무한 반복하였다.
- B. m.bus.go.kr 에서 크롤링을 성공하고 변동 여부가 파악되면 Python 에서는 Amazon RDS 의 Mysql 로 현재 크롤링한 버스과 정류장의 이전 시간 정보를 가져오고 현재 시간을 로드하여 차이 값을 계산한다. 이는 배차간격을 뜻한다.
- C. 웹 크롤러에 대한 세부 내용은 아래 “PYTHON 웹크롤러 아키텍처”를 확인하면 된다.

3. Aamazon RDS 에 저장 되어있는 데이터를 Sqoop 을 통해 Data Warehouse 인 Hadoop hdfs 에 저장한다.

- A. Sqoop 에서 Hdfs 로 데이터를 로드 하는 과정에서 Hive 를 함께 설정하여 Hdfs 로 데이터 전송이 완료되면 지정해 놓은 Hive 의 Database 로 값이 Insert 된다.

4. 전송된 Hive 값을 바탕으로 Zeppelin 을 통해 Visualization 한다.

왜 여러 개의 크롤러를 구동 시키는가?

시스템 아키텍처를 보면서 들 수 있는 가장 큰 의문이다. 한 개의 웹 크롤러를 이용하여 여러 개의 버스 노선 정보를 가져오면 되는 것이 아닌지 생각할 수 있다. 이번 프로젝트에서 웹 크롤러를 2 개 사용한 이유는 생각보다 단순하다. 1 초의 시간이 상당히 중요했기 때문이다. 다른 노선의 정류장 정보를 크롤링 하는 순간에도 시간은 흐른다. 그렇게 되면 해당하지 않는 노선의 버스 정류장은 버스의 유무가 바뀌더라도 신경 쓸 수 없게 된다. 사실, 5 개의 노선을 크롤링하는 과정은 문제가 없다. 하지만 초기 프로젝트 규모로 생각해보면 납득이 된다. 본래 현 프로젝트는 서울을 지나치는 모든 노선(약 650 개)의 정보를 크롤링하려고 하였다. 650 개의 프로세스를 한번에 크롤링 하는 것은 배차간격의 오차범위가 계속 늘어났다. 물론 “비동기”방식을 이용해서도 시간은 줄일 수 있지만, Amazon RDS 의 Mysql 을 사용하여 비동기까지 처리하는 방식은 리스크가 굉장히 컸다. 비동기에 여러 개의 Thread 까지 사용하는 방법은 최고의 해결 방법이었으나 실력의 한계로 1 초라도 정확도를 올려보고자 다른 컴퓨터를 사용하여 맡은 버스 노선을 크롤링하게 된 것이다. 어떤 Job 이 먼저 실행되고 처리될지 보장되지 않은 상황에 모든 방식을 비동기화 시키게 된다면 중복 데이터 값부터 불필요한 계산까지 요구하게 된다. 최악의 경우 알 수 없는 오류까지도 장담할 수 없는 것이다. 따라서 현 프로젝트에서 웹 크롤링은 비동기 방식으로 시도했으나 Mysql 의 데이터를 SELECT 하고 INSERT 하는 과정은 사실상 “동기”처리로 진행하였다.

참고 : 그림 4, 그림 5

왜 여러 개의 DATABASE 를 구동 시키는가?

크롤러는 그럴 수 있어도 데이터베이스는 함께 쓸 수 있지 않을까? 생각할 수 있다. 이는 현재의 프로젝트 보다 더 큰 규모를 생각하였을 때를 위해 구현하게 되었다. 서울시만 봐도 총 653개의 버스 노선이 있다. 서울시를 지나치는 버스 노선의 개수다. 이러한 노선이 지역별로 존재한다. 더 큰 규모로 보았을 때 나라별로 존재하게 된다. 막연하게 큰 규모지만 해당 규모를 생각하며 진행하게 되었다.

[데이터 수집 방법] M.BUS.GO.KR 정보

먼저, 정류장 [명지대(13195)]와 정류장 [명지대(13194)]를 “명지대 정류장”로 명시하며 버스정류장 정보는 <http://m.bus.go.kr/> 에서 크롤링하기로 하였다. 해당 사이트는 서울의 대중교통을 종합적으로 나타낸 페이지이다. 정확하고 간편한 웹 크롤링을 위해 모바일 웹사이트를 타겟으로 지정하였다. 사이트의 구체적인 모습을 보도록 하자.

두산위브아파트 13267	
연가초교삼성아파트앞 13232	
명지대 13194	
명지대삼거리 13196	
명지중고등학교 13219	
명지전문대.충암중고등학교 13221	
응암정보도서관 12415	
서울중앙교회앞 12417	
은가어린이공원구150번종점 12265	

그림 5_ m.bus.go.kr 버스노선 정보

위의 그림 2 는 실제 m.bus.go.kr 에서 제공하는 버스 노선별 상황이며 이를 예시로 설명하겠다. 현재 버스는 정류장 [서울중앙교회앞 12417] 노선 위에 표시되어있다. 이 의미는 버스는 정류장 [서울중앙교회앞 12417]에 이미 도착했거나 지나서 정류장 [은가어린이공원구 150 번종점]을 향하고 있다는 뜻이다. 결론적으로 노선 위의 버스 아이콘이 다음 정류장으로 넘어가면 동시에 값을 해당 시간을 체크하여 해당 정류장에 버스가 도착한 시간을 알 수 있다. 이와 같은 방법을 사용하여 웹 크롤러를 지속해서 반복시키고 해당하는 값이 변경되었을 때 버스 번호, 정류장 번호, 시간을 체크하였다. 해당 파일의 데이터를 2019 년 11 월 24 일부터 2019 년 12 월 15 일까지 총 22 일간 축적했고 시간대별로 배차 시간을 분석하였다.

[데이터 수집 방법] PYTHON 웹크롤러 아키텍처

아래 그림은 Python 으로 제작한 웹 크롤러의 전체적인 아키텍처이다.

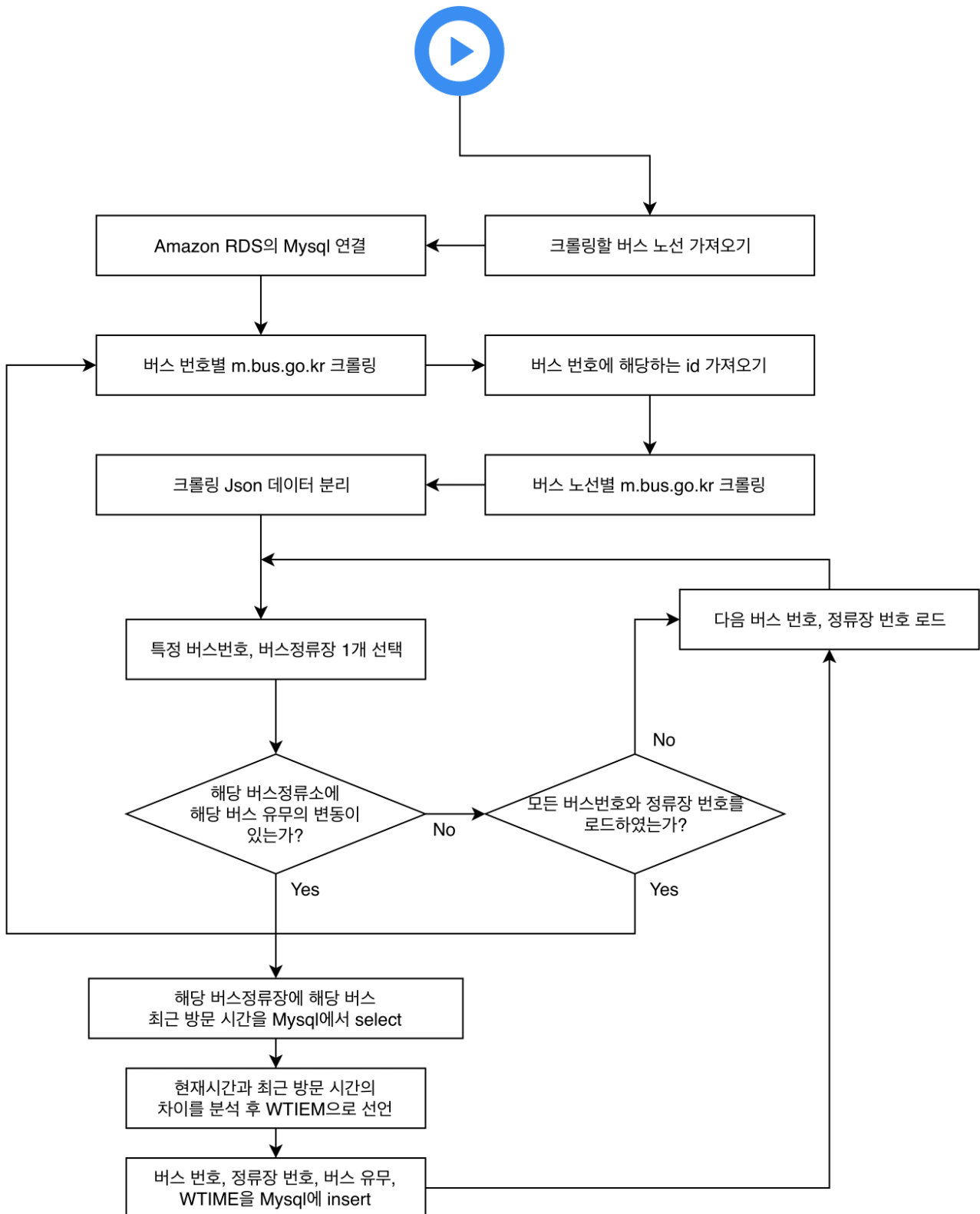


그림 6_ 웹크롤러 아키텍처

[데이터 수집 방법] PYTHON 웹 크롤러 코드

```
def buslist_file_load():
    f = open("buslist.txt", 'rt', encoding='UTF8')
    line = f.read().splitlines()
    f.close()
    return line
```

[크롤링할 버스노선 가져오기] Buslist.txt 에 있는 버스 노선의 정보를 Enter line 을 기준으로 읽어온다.

```
async def busnumberToid(busnum):
    try:
        async with aiohttp.ClientSession() as session:
            async with session.get('http://m.bus.go.kr/mBus/bus/getBusRouteList.bms?strSrch='
+busnum) as resp:
                jsondata=json.loads(await resp.text())
                return jsondata['resultList'][0]['busRouteId']
    except:
        print(busnum+"의 버스번호가 없는것 같거나, 오류가 발생했습니다.")
        pass
```

[버스 번호별 m.bus.go.kr 크롤링][버스 번호에 해당하는 id 가져오기] 버스 번호를 입력 받고 버스 번호에 해당하는 버스 id 값을 가져오는 부분이다. 해당 주소에 버스 번호로 데이터를 요청하면 json 형태로 값을 출력한다. 출력된 값의 resultList[0] → busRouteId 의 값을 출력한다.

```
async def busdetail(busRouteId):
    try:
        async with aiohttp.ClientSession() as session:
            async with session.get('http://m.bus.go.kr/mBus/bus/getRouteAndPos.bms?busRouteId='
+busRouteId) as resp:
                jsondata=json.loads(await resp.text())
                TEMP=[(i['stationNo'],i['existYn']) for i in jsondata['resultList']]
                return TEMP
    except:
        print(busRouteId+"의 데이터를 로드하지 못하였습니다.")
        pass
```

[버스 노선별 m.bus.go.kr 크롤링][크롤링 Json 데이터 분리][특정 버스번호, 버스정류장 선택] 해당하는 버스 번호의 id 값을 입력 받고 입력 받은 버스 id 의 전체 정류장과 정류장에 해당하는 버스의 유무를 파악한다. stationNo 는 정류장 번호이고 existYn 은 버스가 해당 정류장에 있으면 Y 를 출력하고 없으면 N 을 출력한다.


```

async def save_log(log,busnumber,ex):
    now = datetime.datetime.now()
    now_time = now.strftime("%y%m%d%H%M%S")
    file1 = open("ERROR_LOG.txt", "a")
    file1.write "[" + str(now_time) + "]" + log + "\n" + str(busnumber)
+ "정보를 DATABASE에 데이터를 저장하지 못하였습니다.\n" + str(ex) + "\n"
    file1.close()

```

[기타] 프로그램 실행 중에 발생하는 오류를 Log 파일에 저장해주는 함수이다. 오류가 발생한 시간과 오류명을 함께 ERROR_LOG.txt 파일에 저장한다.

```

async def main(totalbuslist,index,conn):
    try:
        for busnumber in totalbuslist:
            busnum=[busdetail(await busnumberToid(busnumber))]
            result=await asyncio.gather(*busnum)
            for bus in result:
                for stationnm, exist in bus:
                    str_exist=str(exist).strip().replace(" ","")
                    str_stationnm=str(stationnm).strip().replace(" ","")
                    str_before_exist = await mysql_search_data(busnumber, str_stationnm,conn)
                    if str_exist != str_before_exist or index==0:
                        await mysql_insert_data(busnumber, str_stationnm, str_exist,index,conn)
                print("confirm : ",str(index))
    except Exception as ex:
        await save_log("function main",busnumber,ex)
        print("main",ex);
        pass

```

[해당 버스정류소에 해당 버스 유무의 변동이 있는가?] [모든 버스번호와 정류장 번호를 로드하였는가?] [다음 버스 번호, 정류장 번호 로드] 프로세스가 시작되면 들어오는 main 함수이다. 먼저 메모장에서 읽어온 버스 번호 리스트를 busnumberToid 함수에 넣어 id 로 출력한다. Gather 은 result 에 busnum 을 전부 담을 때까지 기다려야함으로 사용한 함수이다.

result 에서 가져온 bus 값은 정류장번호와 버스 유무가 포함되어 있다. str_before_exist 는 mysql_search_data 함수와 연결 되어있는데, mysql_search_data 는 버스번호와 정류장 번호를 넘겨주어 이전의 BUS_IS 값이 Y 인지 N 인지 판단한다.

이전의 BUS_IS 값이 N 라고 가정하고 이번 값이 Y 가 들어오면 해당 정류장의 버스가 도착했으므로 값을 mysql_insert_data 함수를 통해 저장한다.

```

async def mysql_insert_data(busnumber,stationnumber,exist,index,conn):
    try:
        #시간시작-----
        str_before_time = await mysql_get_time_data(busnumber, stationnumber,conn)
        try:
            now = datetime.datetime.now()
            now_time = now.strftime("%y-%m-%d %H:%M:%S")
            wtime=str((now-str_before_time).seconds)
        except:
            wtime="0"
            pass
        #MYSQL insert-----
        curs = conn.cursor()
        sql = """INSERT INTO BUS_LIVE VALUES(%s,%s,%s,%s,%s,%s)"""
        curs.execute(sql, (index, busnumber, stationnumber, now_time, exist, wtime))
        conn.commit()
        print(index," ",busnumber," ",stationnumber,"find")
        # MYSQL insert-----
    except Exception as ex:
        await save_log("function mysql_insert_data",busnumber,ex)
        print("mysql_insert_data ", ex);
        pass

```

[현재시간과 최근 방문 시간의 차이를 분석 후 WTIME 으로 선언] [버스 번호, 정류장 번호, 버스 유무, WTIME 을 MYSQL 에 insert] mysql_insert_data 함수는 데이터 변동이 확인되었을 때 실행되는 함수이다. 해당 정류장의 버스가 도착했거나 떠났음으로 이를 알리기 위해 데이터를 추가하는 것이다.

#시간시작의 str_before_time 은 mysql_get_time_data 함수와 연결 되어있다. 해당 함수는 데이터를 변경할 정류장의 데이터를 변경할 버스 노선이 현재를 제외하고 최근에 언제 들어왔는지 시간을 파악해준다. 따라서 나온 날짜 값과 현재 날짜 값의 차이를 구해서 초 단위로 얼마의 시간이 흘렀는지 계산한다.

#MYSQL insert 는 Mysql 에 규격에 맞춰 데이터를 Insert 하게 된다.

```

async def mysql_get_time_data(busnumber,stationnumber,conn):
    try:
        curs = conn.cursor()
        sql = """SELECT TIME FROM BUS_LIVE WHERE BUS_ID=%s AND STATION_ID=%s AND
BUS_IS='Y' ORDER BY INDEX_NUM DESC LIMIT 1"""
        curs.execute(sql, (busnumber,stationnumber))
        result = curs.fetchall()
        if len(result) == 0:
            return datetime.datetime.strptime('2019-11-24 00:01:00', "%Y-%m-%d %H:%M:%S")
        for row_data in result:
            return row_data[0]
    except Exception as ex:
        await save_log("function mysql_get_time_data", busnumber, ex)
        print("mysql_get_time_data ", ex);
        pass

```

[기타] Mysql_get_time_data 의 사용방법은 앞의 mysql_insert_data 함수를 설명하는 과정에 간단한 아키텍처를 설명하였다.

```

async def mysql_search_data(busnumber,stationnumber,conn):
    try:
        curs = conn.cursor()
        sql = """SELECT BUS_IS FROM BUS_LIVE WHERE BUS_ID=%s AND STATION_ID=%s ORDER BY
INDEX_NUM DESC LIMIT 1"""
        curs.execute(sql, (busnumber,stationnumber))
        result = curs.fetchall()
        if len(result) == 0:
            return "N"
        for row_data in result:
            return row_data[0]
    except Exception as ex:
        await save_log("function mysql_search_data", busnumber, ex)
        print("mysql_search_data ", ex);
        pass

```

[해당 버스정류장에 해당 버스 최근 방문 시간을 Mysql 에서 select]
Mysql_search_data 는 가장 최근에 BUS_IS 의 값이 Y 이었는지 N 이었는지 확인하는 단계이다. 이를 확인해서 현재 불러온 값이 Y 이고 이전의 값이 N 였다면 데이터를 갱신해주기위 해서 mysql_insert_data 함수를 실행하게 되고 값이 같다면 변동이 없다는 뜻으로 별다른 코드를 실행시키지 않고 다시 웹크롤링 과정에 돌입하게 된다.

[데이터 분석 방법] AMAZON RDS - MYSQL SCHEMA

```
CREATE TABLE `BUS_LIVE` (  
  `INDEX_NUM` BIGINT(20) NOT NULL,  
  `BUS_ID` VARCHAR(50) NOT NULL,  
  `STATION_ID` VARCHAR(50) NOT NULL,  
  `TIME` DATETIME NOT NULL,  
  `BUS_IS` VARCHAR(50) NOT NULL,  
  `WTIME` BIGINT(20) NULL DEFAULT NULL,  
  PRIMARY KEY (`INDEX_NUM`, `BUS_ID`, `STATION_ID`, `TIME`)  
)  
COMMENT='실시간 버스 정보'  
COLLATE='utf8_general_ci'  
ENGINE=InnoDB  
;
```

위의 SQL문은 BUS 정보를 데이터베이스인 BUS_LIVE의 실제 스키마 정보를 나타낸 SQL문이다.

INDEX_NUM : 가장 최근의 값을 가져오기 위해 설정하였다. 버스 번호, 버스 정류장 리스트를 한번 모두 반복하면 INDEX_NUM 의 값이 + 1 된다.

BUS_ID : 버스 번호의 정보를 담고 있다.

STATION_ID : 정류장 번호의 정보를 담고 있다.

TIME : 데이터가 입력된 시간 정보를 담고 있다.

BUS_IS : BUS_ID 의 버스가 STATION_ID 정류장에 있으면 “Y”를 입력, 없으면 “N”를 입력 받는다.

WTIME : 배차 간격을 뜻한다. BUS_ID 의 버스가 STATION_ID 정류장에 이전에 도착한 버스기록과 현 row 가 등록된 시간이 얼마만큼의 간격을 갖고 있는지 초 단위로 저장한다.

PRIMARY KEY 는 INDEX_NUM, BUS_ID, STATION_ID, TIME 을 묶었다. 검색속도가 빨라지는 경우 두 번 값이 입력될 수 있기에 INDEX_NUM 또한 PRIMARY KEY 로 묶었다.

[데이터 분석 방법] AMAZON RDS - MYSQL 전처리 예시

호스트: bus-live.canb61sq58... 데이터베이스: bus-live 테이블: BUS_LIVE 데이터 쿼리*

bus-live.BUS_LIVE: 2,571,399 행 (중) (대략적), 제한 수: 1,000

INDEX_NUM	BUS_ID	STATION_ID	TIME	BUS_IS	WTIME
205453	7011	13021	2019-12-16 05:41:32	N	120
205453	7612	19221	2019-12-16 05:41:54	N	126
205453	7612	19223	2019-12-16 05:41:54	Y	320
205453	7713	13160	2019-12-16 05:41:55	N	10
205453	7713	13214	2019-12-16 05:41:55	Y	19
205452	7017	02122	2019-12-16 05:41:38	N	106
205452	7017	02219	2019-12-16 05:41:38	Y	522
205452	7019	12418	2019-12-16 05:41:40	N	10
205452	7611	14004	2019-12-16 05:41:43	Y	840
205452	7611	14006	2019-12-16 05:41:43	N	39
205452	7611	19144	2019-12-16 05:41:43	Y	590
205452	7611	19148	2019-12-16 05:41:43	N	155
205452	7713	13003	2019-12-16 05:41:45	Y	647
205452	7713	13160	2019-12-16 05:41:45	Y	19
205452	7713	13214	2019-12-16 05:41:45	N	9
205452	7713	13330	2019-12-16 05:41:45	N	126
205452	7734	13218	2019-12-16 05:41:46	Y	879
205452	7734	13220	2019-12-16 05:41:46	N	78
205451	7017	12320	2019-12-16 05:41:29	N	49
205451	7017	12321	2019-12-16 05:41:29	Y	513
205451	7019	12418	2019-12-16 05:41:30	Y	10
205451	7612	13151	2019-12-16 05:41:34	Y	290
205451	7612	13193	2019-12-16 05:41:34	N	29
205451	7612	13222	2019-12-16 05:41:34	N	106
205451	7612	13230	2019-12-16 05:41:34	Y	368
205451	7713	13160	2019-12-16 05:41:36	N	10
205451	7713	13214	2019-12-16 05:41:36	Y	20
205450	7017	13155	2019-12-16 05:41:19	Y	513

그림 7_AmazonRDS 의 실제 MYSQL 데이터

위 그림 7 은 데이터가 담겨있는 실제 테이블이다. 왼쪽부터 INDEX_NUM, BUS_ID, STATION_ID, TIME, BUS_IS, WTIME 순이다. 컬럼별로 입력되는 데이터 값의 규칙은 아래 예시와 같다.

INDEX_NUM,BUS_ID,STATION_ID,TIME,BUS_IS,WTIME

200000,7019,13195,2019-12-16 07:24:12,Y,95

이를 해석하자면 200000 번째에 크롤링해본 결과이고, 7019 버스는 13195 정류장에 2019 년 12 월 16 일 07 시 24 분 12 초에 도착하였다(BUS_IS="Y"). 7019 버스는 13195 정류장에 이전 버스가 출발한지 95 초만에 다시 왔다.

[데이터 분석 방법] SQOOP 설정

Sqoop 을 설정하는 과정에서 크게 신경 써야 하는 부분은 총 2 가지였다. 첫번째는 Database 를 불러오면서 Hive 에도 업로드 되어야 하는 것. 두번째는 기존의 데이터에서 중복되는 데이터 처리 방법. 따라서 처음 실행할 때 Sqoop 코드와 이후 Sqoop 을 나눠서 항상 처리하였다. 두 코드가 별 차이 없지만 덮어쓰기 오류를 발생시키지 않는 큰 역할을 하였다.

[SQOOP 첫 실행]

```
sqoop import --connect jdbc:mysql://bus-live.canb61sq58tb.ap-northeast-2.rds.amazonaws.com:3306/bus-live \
--username admin \
--password wowns0034 \
--table BUS_LIVE \
--target-dir /user/maria_dev/main_bus \
--hive-import \
--hive-home /user/maria_dev/main_bus \
--hive-overwrite \
--create-hive-table \
--hive-table buslive.seoul
```

Amazon RDS 의 주소와 Mysql 의 기본 포트 3306 을 설정해 놓았다. /bus-live 는 Mysql 의 데이터베이스 명이다. -table 은 데이터베이스 bus-live 에 있는 테이블 명이며 -target-dir 은 해당 데이터베이스를 저장할 hdfs 의 경로를 고정으로 지정해 놓았다. 이후 Hive 에 데이터를 넣기 위해 -hive-import 를 사용하였으며 Hive 에 들어갈 데이터 또한 hdfs 와 같은 경로에 저장해 놓았다. 혹시 모를 경우를 대비하여 덮어쓰기를 허용해 놓았고 Hive 에 Database 만 설정해 놓은 후 새로운 Table 은 생성해 놓지 않은 상태이기 때문에 -create-hive-table 로 데이터를 생성해 놓았다. 마지막으로 -hive-table buslive.seoul 로 Hive 의 데이터베이스 명이 buslive 임을 밝히고 희망하는 table 명을 seoul 로 설정하였다.

[Sqoop 재실행 (데이터 덮어쓰기)]

```
sqoop import --connect jdbc:mysql://bus-live.canb61sq58tb.ap-northeast-2.rds.amazonaws.com:3306/bus-live \  
  
--username admin \  
  
--password wowns0034 \  
  
--table BUS_LIVE \  
  
--delete-target-dir \  
  
--target-dir /user/maria_dev/main_bus \  
  
--table BUS_LIVE \  
  
--hive-import \  
  
--hive-home /user/maria_dev/main_bus \  
  
--hive-overwrite \  
  
--hive-table buslive.seoul
```

위 Sqoop 처음 실행 코드와 큰 차이는 없다. 달라진 점은 target-dir 을 설정하기 앞서 데이터가 덮어쓰기 되지 않기 때문에 delete-target-dir 으로 target-dir 의 값을 제거하였다. 그리고 테이블이 이미 생성되어 있기 때문에 -create-hive-table 명령어를 삭제하고 -hive-overwrite 명령어만 유지시켰다. (기존에 -create-hive-table 을 제거하지 않아 오류가 지속적으로 발생하였다.)

[데이터 분석 방법] OOZIE 설정

AMAZON RDS MYSQL > HDFS > HIVE 과정을 자동화 하기 위해 Oozie 의 Xml 파일을 만들고 Coordinate 를 제작하였다. Day 1 을 기준으로 하루에 한번 실행될 수 있도록 하였다. Xml 의 Command 는 위 Sqoop 에서 재실행 부분을 넣어 놓았다.

seoul_codi	RUNNING	maria_dev	11/30/2019 10:00 AM	11/12/2020 08:39 PM	0000012-191215135...	
------------	---------	-----------	---------------------	---------------------	----------------------	--

그림 8_ OOZIE DASH BOARD

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<coordinator-app xmlns="uri:oozie:coordinator:0.5" end="2020-11-12T11:39Z"
frequency="${coord:days(1)}" name="seoul_codi" start="2019-11-30T03:00Z"
timezone="GMT+09:00">
  <action>
    <workflow>
      <app-path>/user/maria_dev/seoul/shell/workflow.xml</app-path>
    </workflow>
  </action>
</coordinator-app>

```

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<workflow-app xmlns="uri:oozie:workflow:0.5" name="seoul_">
  <start to="shell_3"/>
  <action name="shell_3">
    <shell xmlns="uri:oozie:shell-action:0.3">
      <job-tracker>${resourceManager}</job-tracker>
      <name-node>${nameNode}</name-node>
      <exec>/user/maria_dev/seoul/workflow.xml</exec>
    </shell>
    <ok to="end"/>
    <error to="kill"/>
  </action>
  <kill name="kill">
    <message>${wf:errorMessage(wf:lastErrorNode())}</message>
  </kill>
  <end name="end"/>
</workflow-app>

```

그림 9_coordinator xml + exec shell xml

본 프로젝트에서 OOZIE 의 아키텍처는 Coordinator 가 이벤트를 실행시킬 기준을 갖고 일정 시간에 워크플로우를 실행시키며 시간이 되었을 때 Shell 을 통해 Sqoop import 가 담겨있는 xml 파일을 실행시키게 된다.


```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<workflow-app xmlns="uri:oozie:workflow:0.5" name="seoul_oozie">
  <start to="sqoop_1"/>
  <action name="sqoop_1">
    <sqoop xmlns="uri:oozie:sqoop-action:0.4">
      <job-tracker>${resourceManager}</job-tracker>
      <name-node>${nameNode}</name-node>
      <command>sqoop  import  --connect  jdbc:mysql://bus-live.canb61sq58tb.ap-
northeast-2.rds.amazonaws.com:3306/bus-live --username admin --password wowns0034 --table
BUS_LIVE --delete-target-dir --target-dir /user/maria_dev/main_bus --table BUS_LIVE --hive-import
--hive-home /user/maria_dev/main_bus --hive-overwrite --hive-table buslive.seoul</command>
    </sqoop>
    <ok to="end"/>
    <error to="kill"/>
  </action>
  <kill name="kill">
    <message>${wf:errorMessage(wf:lastErrorNode())}</message>
  </kill>
  <end name="end"/>
</workflow-app>

```

그림 10_sqoop_import_xml

[데이터 분석 방법] ZEPPELIN 설정

Visualization 을 위해서 Zeppelin 을 설정하였다. 위 SQOOP 을 통해 Hive 에 Database 를 Import 한 걸 참고하면 쉽게 이해할 수 있다.

1. 정류장 [명지대 (13195)]의 시간별 배차간격 결과를 출력하기 위해 아래와 같은 SQL 문을 zeppelin 에 추가하였다.

```

%jdbc(hive)
SELECT bus_id,AVG(wtime)/60 AS WAITTIME,HOUR(time) AS HOURTIME
FROM buslive.seoul WHERE station_id='13195' AND WTIME<5000 AND BUS_IS='Y'
GROUP BY BUS_ID,HOUR(time);

```

13195 정류장의 버스 노선, 배차간격, 시간대를 정렬하였으며 시간대를 그룹으로 설정하여 시간대별로 버스 노선과 평균 대기시간을 출력하였다.

참고로 bus_is='Y'는 버스가 해당 정류장에 도착했을 때 시간을 갖고 오기 위해 설정하였다. 또한 wtime < 5000 을 적용한 이유는 24 시간 돌아가는 크롤러 특성상, 막차에서 첫차까지 운영되지 않을 때의 간격이 배차간격으로 입력된다. 첫차-막차 간격이 평균적으로 5000 이 넘음으로 이를 조건에서 제외시키기 위해 5000 이라는 숫자를 입력하였다.

2. 정류장 [명지대 (13194)]의 시간별 배차간격 결과는 위와 동일한 방법이지만 station_id='13195' 항목을 station_id='13194'로 변경하였다

```
%jdbc(hive)
SELECT bus_id,AVG(wtime)/60 AS WAITTIME, HOUR(time) AS HOURTIME
FROM buslive.seoul WHERE station_id='13194' AND WTIME<5000 AND BUS_IS='Y'
GROUP BY BUS_ID,HOUR(time);
```

위의 SQL 문의 소요시간은 23 분 36 초이다.

3. 특정 정류장이 아닌 모든 정류장의 평균 배차간격 결과는 위와 동일한 방법이지만 station_id 의 조건을 제거하였다.


```
%jdbc(hive)
SELECT bus_id,AVG(wtime)/60 AS WAITTIME, HOUR(time) AS HOURTIME FROM
buslive.seoul WHERE WTIME<5000 AND BUS_IS='Y' GROUP BY BUS_ID,HOUR(time);
```

4. 출력된 그래프는 Keys 를 hourtime 으로, Groups 를 bus_id 로 그리고 Values 를 waittime 으로 설정하였다.

[데이터 분석 결과] 분석 데이터 정보

약 2,571,399 행만큼 정보를 저장하였으며, 해당 정보의 크기는 165.8MB 이다.

“ 2019-11-24 ~ 2019-12-15 ”

이름 ^	행	크기	생성됨	업데이트	엔진	코멘트	유형
 BUS_LIVE	2,571,399	165.8 M...	2019-12-01 21:1...	2019-12-15 20:5...	InnoDB	실시간 버스 ...	Table

세부 내용은 위 Amazon RDS Mysql Schema 를 참고하면 된다.

[데이터 분석 결과] ZEPPELIN VISUALIZATION

1. 정류장 [명지대(13195)]의 배차간격을 나타낸 표이다.

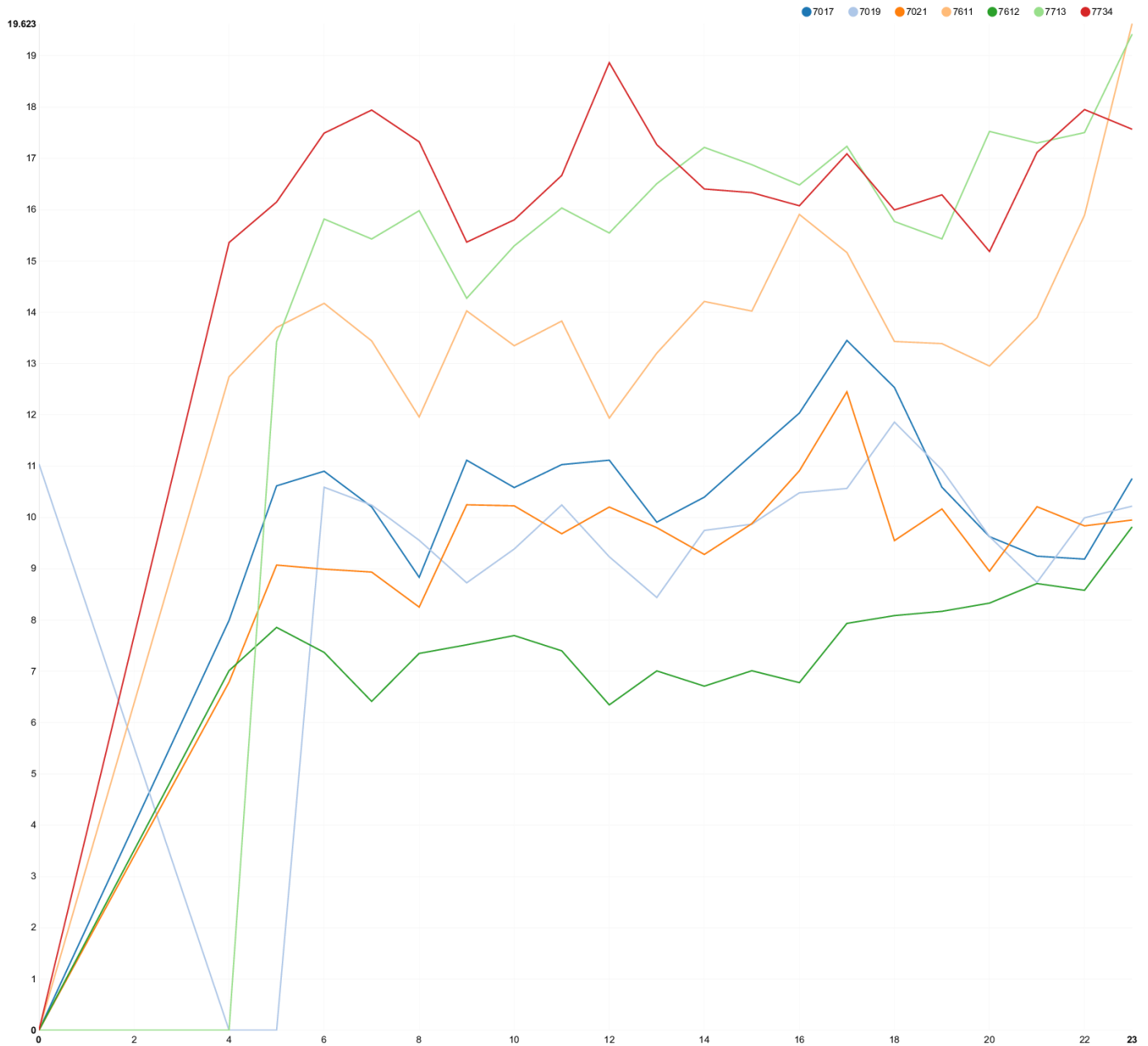


그림 11_정류장 13195 시간별 배차간격

● 7017 ● 7019 ● 7021 ● 7611 ● 7612 ● 7713 ● 7734

bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime
7017	4	7.991667	7612	4	7.010714	7611	4	12.7414	7734	4	15.35833
7017	5	10.61375	7612	5	7.854192	7611	5	13.70148	7734	5	16.14771
7017	6	10.89922	7612	6	7.367739	7611	6	14.17255	7734	6	17.49055
7017	7	10.20405	7612	7	6.411996	7611	7	13.44091	7734	7	17.94096
7017	8	8.831563	7612	8	7.347041	7611	8	11.95739	7734	8	17.32181
7017	9	11.11409	7612	9	7.514946	7611	9	14.02619	7734	9	15.36578
7017	10	10.58082	7612	10	7.695759	7611	10	13.34656	7734	10	15.80042
7017	11	11.0283	7612	11	7.397592	7611	11	13.82726	7734	11	16.66643
7017	12	11.11506	7612	12	6.344241	7611	12	11.93503	7734	12	18.86473
7017	13	9.905507	7612	13	7.00639	7611	13	13.20131	7734	13	17.26598
7017	14	10.3939	7612	14	6.709045	7611	14	14.20934	7734	14	16.40173
7017	15	11.21865	7612	15	7.009474	7611	15	14.02126	7734	15	16.33072
7017	16	12.03333	7612	16	6.777273	7611	16	15.9058	7734	16	16.07489
7017	17	13.44981	7612	17	7.932934	7611	17	15.1631	7734	17	17.08843
7017	18	12.53209	7612	18	8.084451	7611	18	13.42912	7734	18	15.99316
7017	19	10.5869	7612	19	8.167568	7611	19	13.38719	7734	19	16.28804
7017	20	9.624028	7612	20	8.327887	7611	20	12.95036	7734	20	15.18377
7017	21	9.242933	7612	21	8.709632	7611	21	13.89368	7734	21	17.1145
7017	22	9.185938	7612	22	8.578019	7611	22	15.89	7734	22	17.95046
7017	23	10.75537	7612	23	9.813485	7611	23	19.62281	7734	23	17.56741
7019	0	11.0402	7713	5	13.42458	7021	4	6.788095			
7019	6	10.58718	7713	6	15.81624	7021	5	9.069424			
7019	7	10.23846	7713	7	15.42729	7021	6	8.990204			
7019	8	9.553117	7713	8	15.97854	7021	7	8.933333			
7019	9	8.723744	7713	9	14.27133	7021	8	8.250357			
7019	10	9.382966	7713	10	15.29292	7021	9	10.24551			
7019	11	10.2418	7713	11	16.03171	7021	10	10.22469			
7019	12	9.231775	7713	12	15.5443	7021	11	9.680278			
7019	13	8.437919	7713	13	16.50571	7021	12	10.20093			
7019	14	9.745313	7713	14	17.21284	7021	13	9.799237			
7019	15	9.866667	7713	15	16.87583	7021	14	9.276764			
7019	16	10.47875	7713	16	16.48044	7021	15	9.877284			
7019	17	10.56391	7713	17	17.23405	7021	16	10.90765			
7019	18	11.85621	7713	18	15.76816	7021	17	12.44893			
7019	19	10.92664	7713	19	15.42917	7021	18	9.54677			
7019	20	9.625641	7713	20	17.52391	7021	19	10.16419			
7019	21	8.732734	7713	21	17.29758	7021	20	8.949746			
7019	22	9.991169	7713	22	17.50147	7021	21	10.20795			
7019	23	10.21772	7713	23	19.41875	7021	22	9.834121			
						7021	23	9.948039			

그림 11 의 그래프 세부사항

2. 정류장 [명지대(13194)]의 배차간격을 나타낸 표이다.



그림 12_ 정류장 13194 시간별 배차간격

● 7017 ● 7019 ● 7021 ● 7611 ● 7612 ● 7713 ● 7734

bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime
7017	0	9.039815	7612	0	9.3825	7611	0	15.02833	7734	0	19.01414
7017	1	7.8	7612	5	6.863596	7611	6	12.78681	7734	5	15.06111
7017	6	11.01791	7612	6	7.527778	7611	7	14.24007	7734	6	15.94831
7017	7	11.47796	7612	7	8.180159	7611	8	14.58136	7734	7	15.93359
7017	8	12.53727	7612	8	7.593671	7611	9	13.94897	7734	8	20.65909
7017	9	10.61139	7612	9	7.152188	7611	10	10.86964	7734	9	15.94145
7017	10	10.25977	7612	10	7.721717	7611	11	13.58152	7734	10	15.71857
7017	11	10.3	7612	11	7.550409	7611	12	12.3823	7734	11	15.61878
7017	12	11.67217	7612	12	7.610174	7611	13	13.13142	7734	12	18.24484
7017	13	10.47485	7612	13	7.027273	7611	14	12.04211	7734	13	16.63556
7017	14	12.28105	7612	14	6.656443	7611	15	12.10442	7734	14	15.09406
7017	15	10.99971	7612	15	7.572004	7611	16	14.14312	7734	15	15.8107
7017	16	11.09135	7612	16	7.107197	7611	17	14.40939	7734	16	16.11047
7017	17	12.11287	7612	17	8.440123	7611	18	16.29296	7734	17	16.91056
7017	18	11.63758	7612	18	9.204258	7611	19	14.09208	7734	18	16.73472
7017	19	11.9011	7612	19	7.833532	7611	20	11.49264	7734	19	14.2374
7017	20	10.81088	7612	20	7.717061	7611	21	13.05616	7734	20	15.70569
7017	21	9.478871	7612	21	8.707914	7611	22	13.99105	7734	21	15.3784
7017	22	9.94974	7612	22	9.507226	7611	23	14.25843	7734	22	16.99051
7017	23	9.201515	7612	23	8.362141	7021	0	8.618537	7734	23	17.24427
7019	4	6.133333	7713	0	16.31067	7021	6	9.590635			
7019	5	9.523669	7713	6	15.39259	7021	7	9.87507			
7019	6	8.411294	7713	7	16.32489	7021	8	11.2854			
7019	7	8.14702	7713	8	16.64249	7021	9	8.902273			
7019	8	9.052267	7713	9	15.63979	7021	10	8.824648			
7019	9	9.766667	7713	10	13.70019	7021	11	11.1942			
7019	10	9.341063	7713	11	14.3465	7021	12	9.819487			
7019	11	8.679	7713	12	16.18333	7021	13	9.89798			
7019	12	8.380111	7713	13	15.28963	7021	14	10.72319			
7019	13	9.273621	7713	14	15.48897	7021	15	9.667402			
7019	14	9.662281	7713	15	18.02169	7021	16	10.77866			
7019	15	10.00078	7713	16	16.43233	7021	17	9.584336			
7019	16	10.35637	7713	17	15.45	7021	18	12.11042			
7019	17	10.69093	7713	18	17.82111	7021	19	10.00517			
7019	18	10.92066	7713	19	15.30238	7021	20	9.151071			
7019	19	9.558273	7713	20	14.06118	7021	21	9.199879			
7019	20	9.059259	7713	21	16.67443	7021	22	9.491017			
7019	21	9.864938	7713	22	17.91407	7021	23	9.559681			
7019	22	10.87237	7713	23	17.6087						
7019	23	12.475									

그림 12 의 그래프 세부 사항

3. 정류소[명지대]를 지나는 버스 노선의 모든 정류장 평균 배차 간격

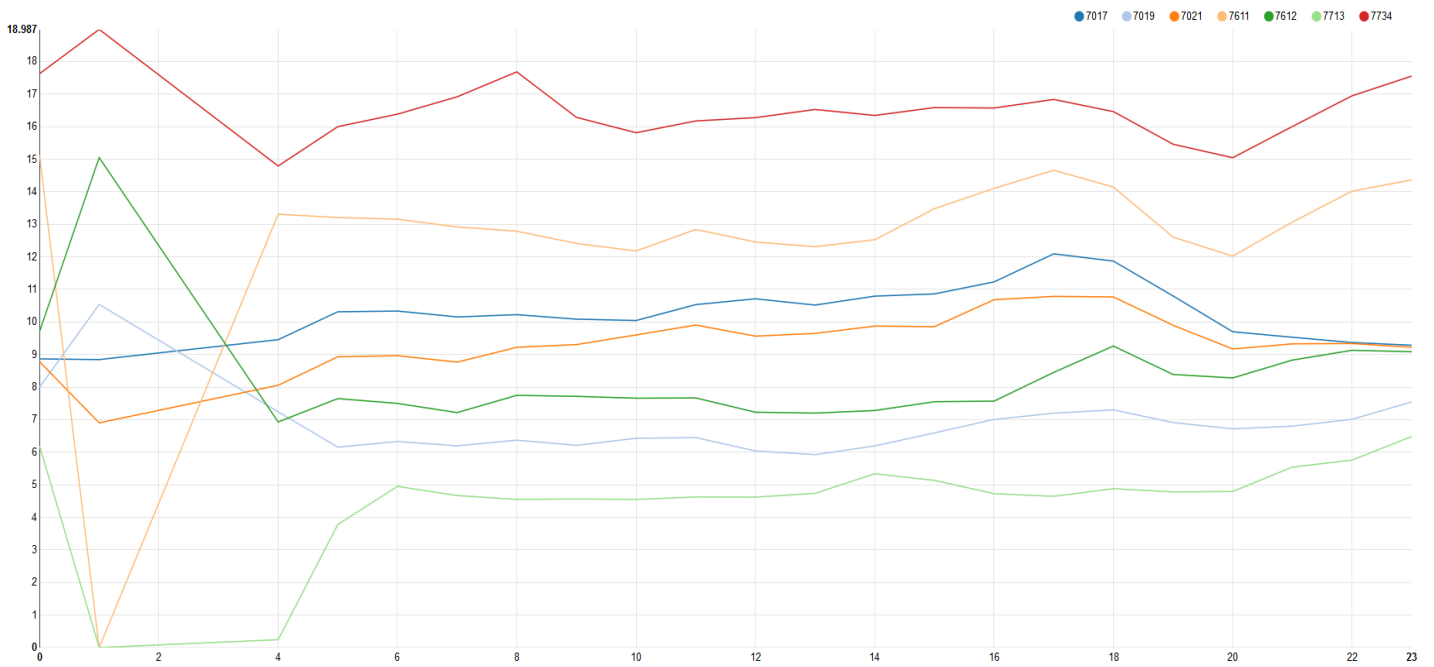


그림 13_ 해당 노선의 평균 배차간격

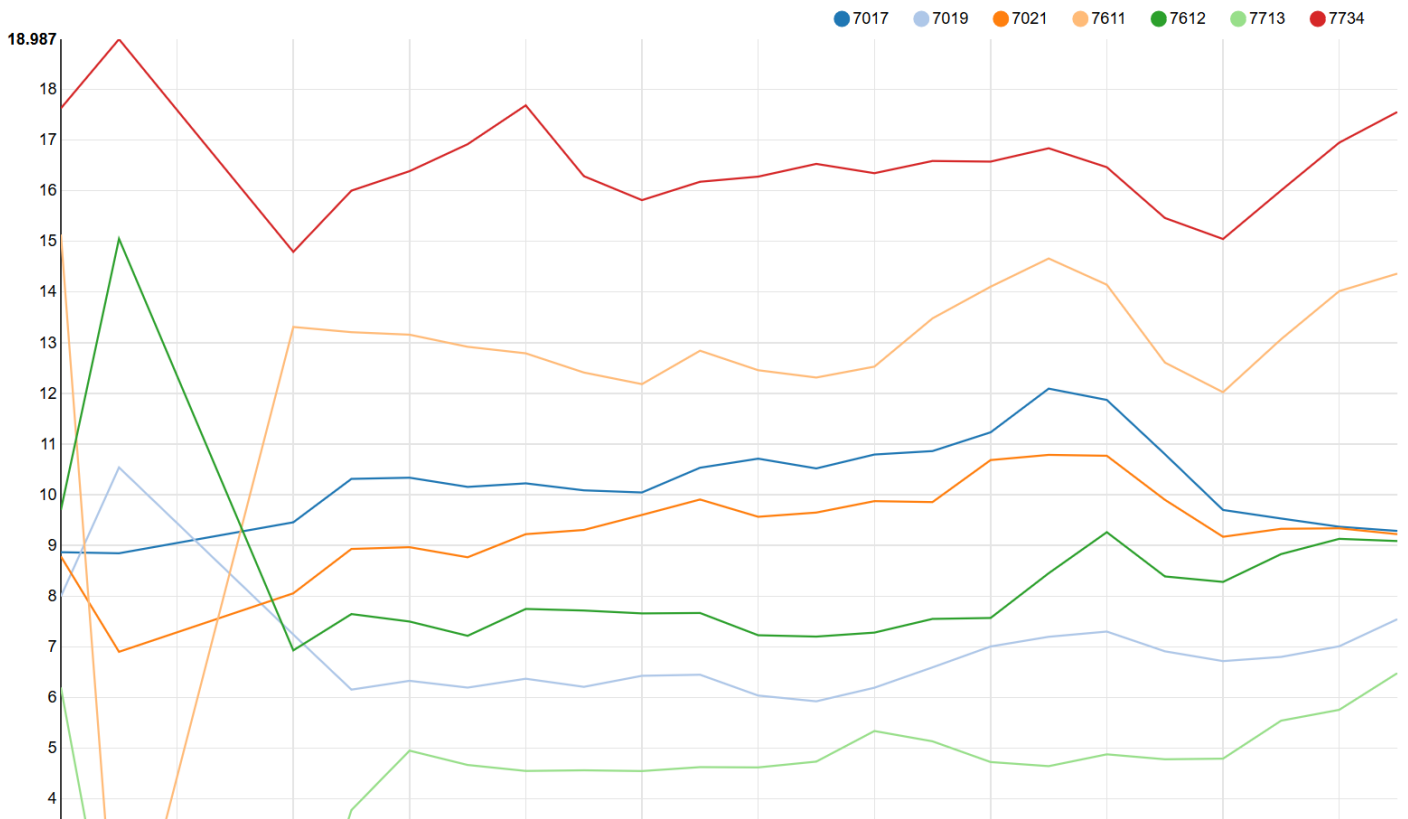


그림 14_ 해당 노선의 평균 배차간격 (핵심 부분 확대)

● 7017 ● 7019 ● 7021 ● 7611 ● 7612 ● 7713 ● 7734

bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime	bus_id	waittime	hourtime
7017	8.86648	0	7021	8.782792	0	7612	9.7012	0	7734	17.62454	0
7017	8.845565	1	7021	6.901667	1	7612	15.05	1	7734	18.98709	1
7017	9.456651	4	7021	8.056094	4	7612	6.930474	4	7734	14.7917	4
7017	10.31338	5	7021	8.931114	5	7612	7.646342	5	7734	15.99991	5
7017	10.33645	6	7021	8.965541	6	7612	7.498939	6	7734	16.38371	6
7017	10.15514	7	7021	8.766562	7	7612	7.217213	7	7734	16.91609	7
7017	10.22485	8	7021	9.222518	8	7612	7.747381	8	7734	17.68177	8
7017	10.08707	9	7021	9.305497	9	7612	7.71596	9	7734	16.28541	9
7017	10.04513	10	7021	9.602046	10	7612	7.659202	10	7734	15.81251	10
7017	10.53332	11	7021	9.906468	11	7612	7.667251	11	7734	16.17503	11
7017	10.71212	12	7021	9.56562	12	7612	7.228189	12	7734	16.27628	12
7017	10.51943	13	7021	9.648686	13	7612	7.201955	13	7734	16.52756	13
7017	10.7938	14	7021	9.874311	14	7612	7.28053	14	7734	16.34394	14
7017	10.86174	15	7021	9.855311	15	7612	7.550759	15	7734	16.58496	15
7017	11.23177	16	7021	10.68445	16	7612	7.569123	16	7734	16.57256	16
7017	12.09349	17	7021	10.78685	17	7612	8.452144	17	7734	16.83551	17
7017	11.87069	18	7021	10.76907	18	7612	9.2611	18	7734	16.46183	18
7017	10.79988	19	7021	9.900022	19	7612	8.388439	19	7734	15.46007	19
7017	9.699713	20	7021	9.171117	20	7612	8.280697	20	7734	15.04542	20
7017	9.531161	21	7021	9.328222	21	7612	8.829509	21	7734	16.0046	21
7017	9.368908	22	7021	9.33833	22	7612	9.130369	22	7734	16.94699	22
7017	9.286131	23	7021	9.22275	23	7612	9.087	23	7734	17.55207	23
7019	7.995409	0	7611	15.13426	0	7713	6.197702	0			
7019	10.53922	1	7611	13.31013	4	7713	0.241667	4			
7019	7.241684	4	7611	13.20731	5	7713	3.77929	5			
7019	6.155868	5	7611	13.15786	6	7713	4.951518	6			
7019	6.329326	6	7611	12.91918	7	7713	4.66903	7			
7019	6.195686	7	7611	12.79027	8	7713	4.551167	8			
7019	6.370371	8	7611	12.41159	9	7713	4.563958	9			
7019	6.210434	9	7611	12.18254	10	7713	4.54901	10			
7019	6.428369	10	7611	12.84163	11	7713	4.626635	11			
7019	6.449516	11	7611	12.45675	12	7713	4.620998	12			
7019	6.038854	12	7611	12.31263	13	7713	4.735553	13			
7019	5.924901	13	7611	12.52766	14	7713	5.339039	14			
7019	6.193081	14	7611	13.47979	15	7713	5.135691	15			
7019	6.594608	15	7611	14.10444	16	7713	4.726873	16			
7019	7.006942	16	7611	14.66105	17	7713	4.645706	17			
7019	7.198141	17	7611	14.1438	18	7713	4.879038	18			
7019	7.300991	18	7611	12.60757	19	7713	4.781111	19			
7019	6.911206	19	7611	12.02317	20	7713	4.793876	20			
7019	6.718179	20	7611	13.06929	21	7713	5.543188	21			
7019	6.801013	21	7611	14.01738	22	7713	5.757297	22			
7019	7.010558	22	7611	14.36277	23	7713	6.480713	23			
7019	7.544984	23									

그림 13, 그림 14 의 그래프 세부사항

[데이터 분석 결과] 결론

앞의 Visualization 된 결과를 통해 확인해보면 비교적 간격의 차이가 크지 않다. 평균 1 분, 2 분을 사이로 값이 변경되는데 평균적으로 1 정거장에 걸리는 시간이 2 분이라고 생각해보면 그렇다고 짧은 시간 또한 아니다.

결론을 도출하는 방법은 어렵지 않다. 값 중에서 평균 배차간격이 높은 시간을 교통 흐름이 혼잡한 것으로 전제를 잡았기 때문에, 배차간격이 길수록 해당 시간은 피하는 것이 맞다는 결론이다. 반대로 배차간격이 짧을수록 해당 시간은 이용하기 좋다는 결론을 내린다. 기준을 명확하게 내리기 위해 내림차순을 기준으로 상위 3 위안에 포함된다면 피해야하는 시간으로, 하위 3 위안에 포함된다면 좋은 시간으로 정의하겠다.

또한, 참고사항은 버스의 첫차 시간은 제각각 이기 때문에 첫차에 가까울수록 막차에 가까울수록 배차간격이 평균적으로 길 수 있다는 사실이다. 따라서 학생들이 점심 시간 이후부터 하교한다는 기준으로 학생들의 움직임이 있는 오후 1 시 ~ 오후 11 시를 평가 기준으로 지정하겠다.

1. 정류장[명지대(13195)]에 관한 결론이다.

[7017][13195]의 피해야하는 시간 : 17 시, 18 시, 16 시

[7017] [13195]의 좋은 시간 : 22 시, 21 시, 20 시

[7019] [13195]의 피해야하는 시간 : 18 시, 19 시, 17 시

[7019] [13195]의 좋은 시간 : 13 시, 21 시, 20 시

[7021] [13195]의 피해야하는 시간 : 17 시, 16 시, 21 시

[7021] [13195]의 좋은 시간 : 20 시, 14 시, 18 시

[7611] [13195]의 피해야하는 시간 : 23 시, 16 시, 22 시

[7611] [13195]의 좋은 시간 : 20 시, 13 시, 19 시

[7612] [13195]의 피해야하는 시간 : 23 시, 21 시, 22 시

[7612] [13195]의 좋은 시간 : 14 시, 16 시, 13 시

[7713] [13195]의 피해야하는 시간 : 23 시, 20 시, 22 시

[7713] [13195]의 좋은 시간 : 19 시, 18 시, 16 시

[7734] [13195]의 피해야하는 시간 : 22 시, 23 시, 13 시

[7734] [13195]의 좋은 시간 : 20 시, 18 시, 16 시

2. 정류장[명지대(13194)]에 관한 결론이다.

[7017][13194]의 피해야하는 시간 : 14 시, 17 시, 19 시.

[7017] [13194]의 좋은 시간 : 23 시, 21 시, 22 시

[7019] [13194]의 피해야하는 시간 : 23 시, 18 시, 22 시

[7019] [13194]의 좋은 시간 : 20 시, 13 시, 19 시

[7021] [13194]의 피해야하는 시간 : 18 시, 16 시, 14 시

[7021] [13194]의 좋은 시간 : 20 시, 21 시, 22 시

[7611] [13194]의 피해야하는 시간 : 18 시, 17 시, 23 시

[7611] [13194]의 좋은 시간 : 20 시, 14 시, 15 시

[7612] [13194]의 피해야하는 시간 : 22 시, 18 시, 21 시

[7612] [13194]의 좋은 시간 : 14 시, 13 시, 16 시

[7713] [13194]의 피해야하는 시간 : 15 시, 22 시, 18 시

[7713] [13194]의 좋은 시간 : 20 시, 13 시, 19 시

[7734] [13194]의 피해야하는 시간 : 23 시, 22 시, 17 시

[7734] [13194]의 좋은 시간 : 19 시, 14 시, 21 시

3. 전체적인 교통 흐름에 관한 결론이다. (밑줄은 13195 공통, 굵음은 13194 공통)

[7017]의 피해야하는 시간 : 17시, 18시, 16시.

[7017] 의 좋은 시간 : **23** 시, 22시, 21시

[7019] 의 피해야하는 시간 : **23** 시, 18시, 17시

[7019] 의 좋은 시간 : 13시, 12 시, 14 시

[7021] 의 피해야하는 시간 : 17시, **18**시, 16시

[7021] 의 좋은 시간 : 20시, 23 시, **21**시

[7611] 의 피해야하는 시간 : **17**시, 23시, **18**시

[7611] 의 좋은 시간 : 20시, 13시, **14**시

[7612] 의 피해야하는 시간 : **18**시, 22시, 23시

[7612] 의 좋은 시간 : 13 시, 14 시, 15 시

[7713] 의 피해야하는 시간 : 23 시, 22 시, 21 시

[7713] 의 좋은 시간 : 17 시, 16 시, 13 시

[7734] 의 피해야하는 시간 : 23 시, 22 시, 17 시

[7734] 의 좋은 시간 : 20 시, 19 시, 21 시

최종 결론

- 7019 의 18 시는 피하세요
- 7019 는 13 시에 이용하세요
- 7021 은 16 시는 피하세요
- 7021 은 20 시에 이용하세요
- 7611 은 23 시를 피하세요
- 7611 은 20 시에 이용하세요
- 7612 는 22 시를 피하세요
- 7612 는 13 시, 14 시를 이용하세요
- 7713 은 22 시를 피하세요
- 7734 는 23 시, 22 시를 피하세요

[추가적인 확장 가능성]

현재 데이터는 “버스” 노선을 기준으로 맞춰져 있다. 하지만 해당 내용을 타 대중교통에 적용해도 전혀 문제가 없다. 예를 들어, 지하철 정보를 수집한다면 지하철이 연착되는 시간대를 조사하여 해당 시간대를 피하는 방법이 있다.

이러한 방법으로 응용하거나 또 다른 용도는 본 프로젝트의 WTIME 과 BUS_IS="N"의 정보를 사용하는 것이다. 현재 데이터에도 BUS_IS="N"일 때

WTIME 이 입력되어 있다. 본 프로젝트에서는 사용되지 않았지만 BUS_IS 가 N 인 값은 해당 정류장에서 다음 정류장까지 버스가 가는데 걸리는 시간이다. 이 값이 생성되게 된 이유는 특정 버스가 특정 정류장에 도착하면 BUS_IS 의 값은 "Y"로 변경된다. 그렇게 Mysql 에 INSERT 하게 되고 INSERT 하는 조건 자체가 BUS_IS 의 값 변동이 확인되면 추가하는 것으로 되어있기 때문에 해당 버스가 다음 정류장에 도착하게 되면 BUS_IS 의 값이 Y 에서 N 으로 되었을 때 또한 데이터를 INSERT 하게 된다. 예를 들어보면

INDEX_NUM,BUS_ID,STATION_ID,TIME,BUS_IS,WTIME

300000,7019,13195,2019.12.16 13:00:00,N,120

값이 있다. 이 값을 해석해보자면 300000 번째 크롤링한 값이다. 7019 번 버스는 정류장 13195 부터 다음 정류장까지 걸린 시간이 120 초가 걸렸다는 것이다.

이처럼, 해당 정류장부터 다음 정류장까지 걸린 시간 정보를 알 수 있다. 이를 이용해서 서울에 있는 650 여개의 데이터를 모두 크롤링하고 데이터를 갖고 있다고 가정하면 두가지 방향으로 나뉘지는데 Real Time Data 분석을 통해 서울의 교통체증이 어느 정도인지 파악할 수 있다. 예를 들면, 특정 정류장부터 특정 정류장까지 평균보다 더 많은 시간이 소요된다면 해당 도로는 교통체증이 심한 상태인 것이다. 이 정보를 650 개의 노선과 비교해 본다면, 해당 도로의 상황을 쉽고 빠르게 파악할 수 있을 것이다. 요약하자면, 버스 배차간격을 통해 해당 도로의 교통상황을 정확하게 파악할 수 있다는 것이다.

Real Time Data 분석 외에 다른 방법은 해당 도로의 교통체증이 어떤 시간대에 얼마나 걸릴지 예측해볼 수 있다. 심지어 버스가 실제로 운영한 시간대를 가져오기 때문에 보다 정확하게 파악할 수 있다.

[참고사항] 배차간격의 의문점

데이터 값을 확인해보면 이상하다고 느낄 수 있는 값이 있다.



5000 이상의 값은 운행하지 않는 시간으로 인해 발생하는 오차 값이며 배차간격이 10 초거나 10 초후에 다음 버스가 도착하는 경우가 있는데 오류가 아닌 실제로 위 그림처럼 운행되는 일이 많음으로 그런 숫자가 발생하게 된 것이다.

[웹 크롤러 파이썬 코드]

https://github.com/Linho1150/HADOOP-BIG_DATA_PROJECT

[데이터 분석 결과 - ZEPL]

<https://www.zepl.com/viewer/notebooks/bm90ZTovL2hhaW5obzk1ODZAZ21haWwuY29tLzUzNzQ3OWY5ODIIMDRjN2ZiNTFhZjViNTI3MGZmNWQzL25vdGUuanNvbG>

[참고문헌]

Asyncio 모듈 [Asyncio 공식 레퍼런스]

<https://docs.python.org/ko/3/library/asyncio.html>

Asyncio 모듈 [asyncio : 단일 스레드 기반의 Nonblocking 비동기 코루틴 완전 정복]

<https://soooprmx.com/archives/6882>

Aiohttp 모듈 [Aiohttp 공식 레퍼런스]

<https://docs.python.org/ko/3/library/asyncio.html>

Pymysql 모듈 예제 [개발==삽질 tistory 블로그]

<https://sab-jil.tistory.com/6>

Sqoop user guide [sqoop 공식 레퍼런스]

<https://sqoop.apache.org/docs/1.4.7/SqoopUserGuide.html>

Sqoop use hive [Dz zone]

<https://dzone.com/articles/sqoop-import-data-from-mysql-to-hive>

Sqoop Overwrite [cloudera.com]

<https://community.cloudera.com/t5/Support-Questions/In-Sqoop-import-is-there-an-option-to-overwrite-or-delete/td-p/221270>

Zeppelin - hive 와 연동 [cloudera.com]

https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.4/using-zeppelin/content/using_the_jdbc_interpreter_to_access_hive.html

OOZIE [권동섭 교수님, 2018 OOZIE 교육자료]