



DIE
TI. UNI
NA

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

CORSO DI LAUREA MAGISTRALE
IN INFORMATICA

BASI DI DATI II MOD. B

Data warehouse “Fire Department Calls for Service”

Autori:

Emanuele Cioffi N97000277
Crispino Cicala N97000264

Professore:
Adriano Peron

22 maggio 2019

Indice

1	Introduzione	1
1.1	Traccia e scopo del progetto	1
1.2	Dataset	1
1.2.1	Tipo emergenze	3
1.2.2	Luoghi	4
1.2.3	Priority	4
2	Progettazione	5
2.1	Schema dei fatti	5
2.2	Query	7
2.3	ETL	8
2.3.1	Tuple fintizie	9
2.4	Piano di esecuzione	9
3	Implementazione	10
3.1	Query	10
3.2	Viste materializzate	17
3.2.1	Vista Giorno	17
3.2.2	Vista Notte	17
3.3	Piano operativo: lo script Python	18
4	Risultati	20
4.1	Tempi query	21
4.2	Aggiornamento Viste Materializzate	22
4.3	Aggiornamento Indici	23
4.4	Analisi dei tempi	24
4.4.1	ETL	24
4.4.2	Query 1A	25
4.4.3	Query 1B	26
4.4.4	Query 2	27
4.4.5	Query 3	28
4.4.6	Query 4	29
4.4.7	Query 5	30
4.4.8	Query 6	31
4.4.9	Query 7	32
4.4.10	Query 8	33
4.5	Conclusioni	34
5	Appendice	35

1 Introduzione

1.1 Traccia e scopo del progetto

Si vuole realizzare un Data Warehouse di tipo ROLAP (Relational On-Line Analytical Processing) per fornire gli strumenti atti ad analizzare la base di dati di partenza, valutandone le strategie offerte. Per lo svolgimento del progetto si è scelto di utilizzare il dataset “Fire Department Calls for Service” che raccoglie i dati relativi alle segnalazioni ricevute dal 911 e gestite dal dipartimento dei vigili del fuoco della città di San Francisco, descritto nel dettaglio nel paragrafo 1.2.

Il progetto si articola in due parti:

Progettazione ed implementazione di un’architettura per il Data Warehousing di tipo ROLAP che si basi sul dataset scelto e che fornisca una suite di analisi minimale.

Analisi e studio dei tempi ottenuti dall’esecuzioni di alcune operazioni di base e query esemplificative sul Data Warehouse, per confrontare le diverse scelte progettuali.

Per la realizzazione del progetto sono state utilizzate le seguenti tecnologie:

PostgreSQL 10, per la realizzazione del Data Warehouse;

Python, per la realizzazione del modulo ETL.

1.2 Dataset

Il dataset utilizzato è “Fire Department Calls for Service”, un dataset open data contenente tutte le chiamate e risposte delle unità del *Fire department* di San Francisco gestite dal servizio di emergenza 911. Ogni riga del dataset contiene il numero identificativo della chiamata, identificativo della o delle unità inviate in risposta, oltre che una serie di dati descrittivi dell'emergenza, quali l'indirizzo, il tipo di emergenza, la priorità assegnata ect. Tra i vari campi di particolare interesse figurano quelli di tipo data, ad esempio quella di arrivo al centralino, quella relativa al momento in cui sono state inviate le unità (con precisione al secondo) e all'arrivo delle stesse sul luogo dell'emergenza. Il dataset è ospitato oltre che sulla piattaforma di open data della città di San Francisco, anche dalla community di *Kaggle*, di proprietà di *Google*, ed è soggetto ad update quotidiani. I record raccolti vanno dall'anno 2000 ad oggi, ed ammontano a circa 5 milioni. Le colonne che invece descrivono i record sono 34 e sono descritte nel dettaglio nella tabella 1.2.

Nome campo	Tipo	Descrizione
call_number	Stringa	Numero univoco a 9 cifre assegnato dal <i>dispatch center</i> del 911 alla chiamata. Questo numero viene utilizzato sia dal <i>Fire department</i> che dal <i>Police department</i> .
unit_id	Stringa	Identificativo dell'unità inviata. Per esempio E01 viene utilizzato per <i>Engine01</i> oppure T01 per <i>Truck01</i>
incident_number	Stringa	Il numero univoco ad 8 cifre assegnato dal <i>dispatch center</i> del 911 all'incidente.
call_type	Stringa	Il tipo di incidente cui fa riferimento la chiamata. I tipi sono trattati in dettaglio nel paragrafo 1.2.1
call_date	Data e ora	Indica data e ora della chiamata ricevuta dal <i>dispatch center</i> del 911.
watch_date	Data	Indica la sola data della ricezione della chiamata. Questa data parte dalle 8 del mattino e termina alle 8 del giorno dopo.
received_dtTm	Data e ora	Data e ora in cui la chiamata viene effettivamente ricevuta dal <i>dispatch center</i> del 911.
entry_dtTm	Data e ora	Data e ora in cui l'operatore del 911 crea l'istanza della chiamata all'interno del sistema.
dispatch_dtTm	Data e ora	Data e ora in cui l'operatore del 911 invia la segnalazione all'unità del <i>fire department</i> .

response_dtTm	Data e ora	Data e ora in cui l'unità viene a conoscenza della segnalazione e registra il fatto che un'unità sia partita per raggiungere il luogo dell'incidente.
on_scene_dtTm	Data e ora	Data e ora in cui l'unità arriva sul luogo dell'incidente.
transport_dtTm	Data e ora	È un campo valido solo per le ambulanze, indica l'ora in cui l'ambulanza parte dal luogo dell'incidente per raggiungere l'ospedale.
hospital_dtTm	Data e ora	È un campo valido solo per le ambulanze, indica l'ora in cui l'ambulanza arriva all'ospedale.
call_final_disposition	Stringa	È un codice che descrive l'esito dell'intervento, per esempio TH2: <i>Transport to Hospital - Code 2</i> , FIR: <i>Resolved by Fire Department</i> .
available_dtTm	Data e ora	Data e ora in cui l'unità non è più impegnata e si rende disponibile per altri interventi.
address	Stringa	Parte dell'indirizzo del luogo dell'incidente, non è fornito l'indirizzo completo per tutela della privacy.
city	Stringa	Città dell'incidente (all'interno della contea di San Francisco).
zipcode	Stringa	Zipcode del luogo dell'incidente.
battalion	Stringa	La divisione a cui appartiene l'unità che effettua l'intervento. Esistono 10 diversi <i>battalion</i> , ad ognuno dei quali appartengono diverse <i>fire station</i> (https://sf-fire.org/fire-station-locations). ^{1.2.2}
station_area	Stringa	La fire station associata al luogo dell'incidente. ^{1.2.2}
box	Stringa	È la <i>fire box</i> associata al luogo dell'incidente. Un box è la più piccola unità utilizzata per la suddivisione amministrativa della città, che ne contiene circa 2400. ^{1.2.2}
origPriorityMapped	Stringa	È un codice che indica la priorità assegnata dal 911 all'incidente. Esistono diversi tipi di priorità, e sono descritti nel paragrafo 1.2.3.
callPriorityMapped	Stringa	Priorità assegnata dal <i>fire department</i> prima dell'intervento. ^{1.2.3}
finalPriorityMapped	Stringa	Priorità registrata al termine dell'intervento. ^{1.2.3}
als_unit	Booleano	Parametro booleano che indica se ad accompagnare l'unità c'è una risorsa ALS (Advance Life Support) o no, in altre parole se in quell'unità è presente o meno un paramedico.
call_type_group	Stringa	I tipi di chiamata possono appartenere a quattro diversi call type group: <i>Fire</i> , <i>Alarm</i> , <i>Potential Life Threatening</i> e <i>Non Life Threatening</i> . ^{1.2.1}
number_of_alarms	Numero	Numero di allarmi associati all'incidente, è un numero che va da 1 a 5.
unit_type	Stringa	Tipo dell'unità inviata (es: 'TRUCK').
unit_sequence_call_dispatch	Numero	Numero che indica l'ordine di chiamata dell'unità associata al record.
fire_prevention_district	Stringa	<i>Fire prevention district</i> associato al luogo dell'incidente.
supervisor_district	Stringa	<i>Supervisor district</i> associato al luogo dell'incidente.
neighborhood	Stringa	Quartiere della città in cui è avvenuto l'incidente.
lat_lon	Stringa	Latitudine e longitudine del luogo dell'incidente.
rowid	Stringa	Codice univoco per il record della segnalazione. Formato concatenando <i>Call_number</i> e <i>Unit_id</i> separati da un trattino.

Tabella 1: Colonne dataset

1.2.1 Tipo emergenze

Call_type insieme a **call_type_group** sono i campi che rappresentano il tipo di incidente cui fa riferimento la chiamata. I **call_type_group** sono quattro, e ognuno include una parte dei circa trenta **call_type**. La tabella 1.2.1 riporta tutti i tipi di chiamate, e relativo gruppo, registrate durante il periodo di acquisizione del dataset (2000-2019).

Call_type	Call_type_group
Aircraft Emergency	Alarm
Alarms	Alarm
Assist Police	Alarm
Citizen Assist / Service Call	Alarm
Electrical Hazard	Alarm
Elevator / Escalator Rescue	Alarm
Fuel Spill	Alarm
Gas Leak (Natural and LP Gases)	Alarm
HazMat	Alarm
Medical Incident	Alarm
Odor (Strange / Unknown)	Alarm
Oil Spill	Alarm
Other	Alarm
Outside Fire	Alarm
Smoke Investigation (Outside)	Alarm
Structure Fire	Alarm
Vehicle Fire	Alarm
Watercraft in Distress	Alarm
Administrative	Fire
Confined Space / Structure Collapse	Fire
Explosion	Fire
Extrication / Entrapped (Machinery Vehicle)	Fire
HazMat	Fire
High Angle Rescue	Fire
Industrial Accidents	Fire
Marine Fire	Fire
Mutual Aid / Assist Outside Agency	Fire
Odor (Strange / Unknown)	Fire
Outside Fire	Fire
Structure Fire	Fire
Suspicious Package	Fire
Train / Rail Fire	Fire
Train / Rail Incident	Fire
Vehicle Fire	Fire
Water Rescue	Fire
Watercraft in Distress	Fire
Medical Incident	Non Life-threatening
Other	Non Life-threatening
Traffic Collision	Non Life-threatening
Medical Incident	Potentially Life-Threatening
Other	Potentially Life-Threatening
Structure Fire	Potentially Life-Threatening
Traffic Collision	Potentially Life-Threatening
Water Rescue	Potentially Life-Threatening

Tabella 2: Tipo emergenze e relativo gruppo d'appartenenza.

La tabella, ottenuta mediante un'interrogazione sul database interamente popolato, mette in evidenza la possibilità che alcuni tipi di emergenza possano appartenere anche a gruppi di emergenza distinti, ne sono un esempio *Medical Incident*, che può appartenere sia al gruppo *Non Life-threatening* che *Alarm* o *Traffic Collision* che rientra sia in *Non Life-threatening* che *Potentially Life-threatening*. Questo tipo di suddivisione è subentrato negli anni successivi all'inizio della raccolta dei dati, per cui i primi record registrati non riportano alcun valore per il campo *call_type_group*. Per ovviare a questa carenza si è deciso di adottare una strategia “*oggi per ieri*”. In fase di ETL si è deciso di assegnare a questi record un valore per il campo *call_type_group*, determinato attraverso l'utilizzo di un dizionario ricavato dai valori di *call_type_group* assegnati ai record più recenti, piuttosto che considerare non valido l'intero record.

1.2.2 Luoghi

Tra i campi colonna che caratterizzano le tuple del dataset vi sono quelle che descrivono il luogo da cui proviene la segnalazione di emergenza. Per i fini di analisi preposti, terremo concettualmente separati i luoghi geografici da quelli amministrativi, come sarà possibile leggere nell'apposito paragrafo relativo alle trasformazioni in ETL 2.3. Sono stati raggruppati sotto il nome di *luogo geografico* quei campi che fanno riferimento alla posizione dove avviene l'emergenza, come ad esempio la città, il quartiere e la pseudo-via (ricordando come la via precisa venga omessa per privacy e sostituita da indicazioni quale l'isolato o la zona), mentre tutto ciò che concerne l'organizzazione amministrativa del corpo dei pompieri, come ad esempio le zone di pertinenza delle stazioni, le stazioni stesse e i battaglioni verranno indicati con il termine *luogo amministrativo* o *responsabilità amministrativa*.

1.2.3 Priority

Ad ogni tupla del dataset relativa ad una segnalazione d'emergenza sono assegnati tre valori di priorità: *Original Priority*, *Priority*, *Final Priority*. La documentazione allegata al dataset non è risultata esaustiva nello spiegare le differenze tra i tre campi, per cui si è deciso di considerare per l'analisi e le query solo il primo e il terzo campo, essendo più o meno facile intuirne il significato. I campi, quindi, sono stati trattati rispettivamente come la priorità dichiarata dal chiamante in fase di segnalazione, e la priorità confermata dall'unità inviata/centralino una volta risolta l'emergenza o non appena si è giunti sul posto. Data la vaghezza della documentazione in merito ai valori che tali campi possono assumere nel dataset, si è provveduto a ricercare in rete ulteriori chiarimenti rispetto a valori non numerici non presenti nelle brevi note allegate. Tale ricerca ci ha permesso di dare un significato a tutte i valori riscontrati nel dataset nei campi priorità e che non erano stati descritti negli allegati al dataset, e che sono risultati essere campi relativi alle emergenze di tipo medico.

Codice	Significato
2	Non Emergenza (guida senza sirene)
3	Emergenza (guida con sirene accese)
A,B,C	Emergenze mediche varie (tipicamente di tipo 2)
D,E	Emergenze mediche varie (tipicamente di tipo
I,1	Non specificato.

Tabella 3: Valori di priorità originali

Dal momento che le emergenze di tipo medico non ci interessano ai fini delle analisi, come sarà possibile leggere più approfonditamente nell'apposito paragrafo di ETL, le emergenze di tipo medico verranno mappate nelle rispettive emergenze a codice numerico 2 o 3.

Priorità	Valore Finale Assegnato
2	2
3	3
A, B, C	2
D, E	3
1, I	ignora

Tabella 4: Valori di priorità mappati

2 Progettazione

Per l'implementazione del data warehouse ROLAP si è scelto di adottare un'architettura a due livelli e schema a stella. La tabella originale è stata quindi processata in ETL per la generazione del nuovo schema ristrutturato. A partire dalla tupla originale si è scelto di estrarre delle tabelle di dimensione che saranno oggetto di query e che verosimilmente potrebbero interessare eventuali addetti all'analisi del dominio. È stata inoltre ricavata una dimensione a partire da attributi calcolati per facilitare raggruppamenti sulla durata dell'intervento, inteso come numero di minuti trascorsi dalla ricezione della chiamata al tempo di arrivo dell'unità sul punto ("Received-dttm" → "on_site_dttm").

Le dimensioni estratte sono quindi:

- Tipo di Emergenza - "dim_call_type"
- Durata intervento - "dim_duration"
- Località geografica - "dim_geo_place"
- Località amministrativa - "dim_responsibility"
- Data ricezione chiamata - "dim_received_date"

Si è poi provveduto ad implementare una **frammentazione orizzontale** del fatto generato, che separasse e raggruppasse i fatti per annata, generando così un numero di tabelle di fatto che vanno dall'anno 2000 fino all'ultimo anno incontrato in fase di loading dei dati. Si è deciso di non creare a priori queste tabelle, ma di delegare l'operazione allo script di ETL che ne crea una ogni volta che incontra un record relativo ad un anno non trattato in precedenza, in modo da mantenere nel data warehouse solo i frammenti relativi agli anni effettivamente incontrati nelle tuple del database originale.

Sono state formulate e create due viste **materializzate** che separano i fatti dello schema a stella in emergenze per cui la chiamata è stata registrata nelle ore del giorno, ed emergenze notturne. Le viste permettono di recuperare più velocemente le emergenze avvenute di notte o di giorno, e dovrebbero velocizzare un ipotetico processo di recupero di chiamate avvenute a specifiche ore. Le fasce orarie scelte simbolicamente per rappresentare giorno e notte sono rispettivamente rappresentate dagli intervalli che vanno dalle 5 alle 16 per il giorno, e dalle 17 alle 4 per la notte.

Come già accennato in precedenza il numero di record raccolti negli anni di acquisizione è pari a circa 5 milioni. Pur essendo un numero sufficientemente grande per ottenere un'analisi basilare della progettazione, si è deciso di aumentare questa dimensione inserendo altri 5 milioni di record fintizi, generati automaticamente tramite una procedura sviluppata per lo scopo. La principale differenza tra questi due insiemi di record è che quelli "originali" sono raccolti dall'anno 2000 al 2019, quelli fintizi fanno riferimento agli anni dal 2020 al 2040. Ulteriori dettagli riguardo il contenuto dei record fintizi sono riportati nel paragrafo 2.3.1.

2.1 Schema dei fatti

Si è deciso quindi di implementare uno schema a stella con tabella dei fatti singola, ed uno alternativo in cui la tabella dei fatti è stata frammentata per anno, in modo da poterne confrontare i tempi di utilizzo e valutarne l'utilità. Per alcune query si è previsto l'utilizzo delle viste materializzate, casi di cui si discuterà in seguito.

Le due implementazioni sono descritte tramite il formalismo grafico UML dalle figure 1 e 2.

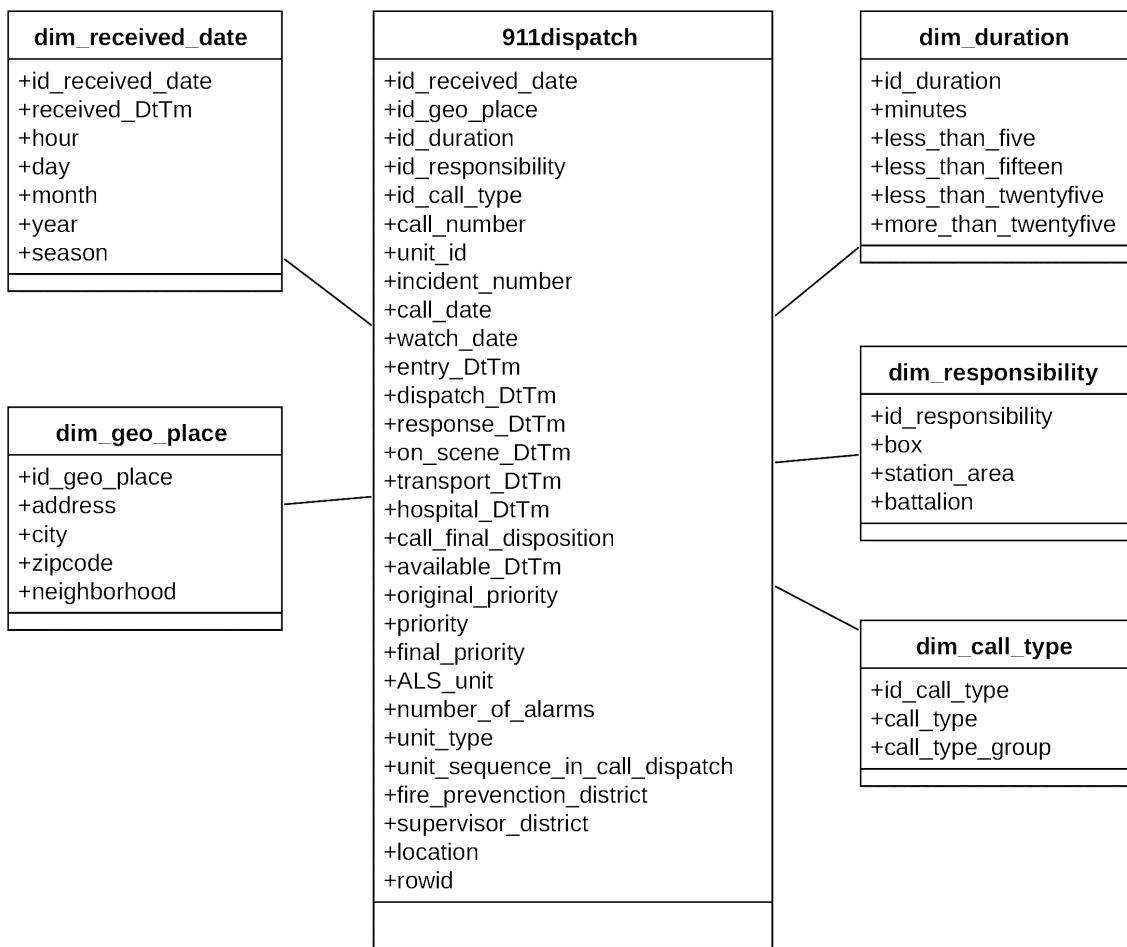


Figura 1: Schema a stella con tabella dei fatti singola.

In figura 2 è descritto lo schema a stella con tabella dei fatti frammentata orizzontalmente per anno nel quale ogni singola tabella presenta la forma della tabella in figura 1 e contiene i record relativi all'anno da cui prende il nome. Nello schema sono riportate le tabelle dall'anno 2000 all'anno 2040. Le tabelle dall'anno 2000 all'anno 2018 contengono dati reali acquisiti dal dataset descritto nel capitolo 1.2, i restanti anni contengono elementi fintizi generati a scopo di analisi secondo i criteri descritti nel paragrafo 2.3.1.

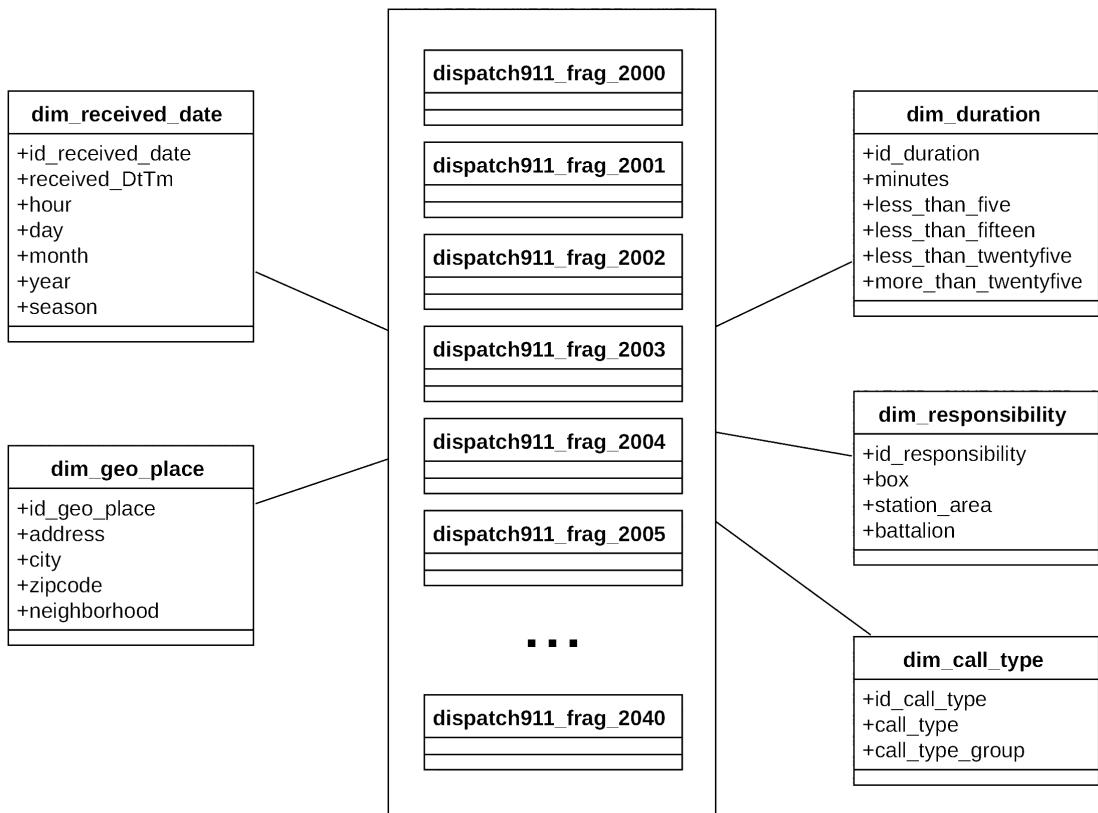


Figura 2: Schema a stella con tabella dei fatti frammentata per anno.

2.2 Query

Seguono le query proposte per valutare le diverse scelte implementative effettuate. Ogni query prevede il confronto tra almeno due scenari diversi, ed ogni query verrà confrontata tra le sue versioni con e senza indici, indici che sono stati posti sui campi ID del fatto, che fanno riferimento agli id delle dimensioni.

	Query	Descrizione
3	Per ogni stagione e per ogni quartiere, numero di segnalazioni per incendi.	Valuta l'efficacia dello schema a stella raggruppando sulle dimensioni di data, luogo e tipo di segnalazione.
4	Media numero unità inviate per città, emergenza e priorità dichiarata.	Valuta l'efficacia della dimensionalità geografica e del tipo di emergenza calcolando la media di unità inviate per ogni segnalazione.
7	Interventi del battaglione B01 per tipo di emergenza, con media durata di intervento e numero interventi.	Valuta i raggruppamenti fatti sulle dimensioni di durata, competenza amministrativa e tipo emergenza, indicando il numero di emergenze di competenza della battaglione B01 e calcolandone la media del tempo di arrivo delle unità sul posto.
1A	Numero di segnalazioni 'HazMat' per anno e per quartiere.	Dato un tipo particolare di emergenza ("HazMat"), riporta il numero di segnalazioni per anno e per distretto. Valuta la dimensionalità di data e tipo chiamata.

1B	Numero di segnalazioni 'HazMat' per quartiere negli anni compresi tra il 2010 e il 2015.	Valuta l'efficacia della frammentazione orizzontale proposta, raggruppando per quartiere e contando il numero di segnalazioni ricevute tra gli anni 2010-2015 per l'emergenza di tipo 'HazMat'.
6	Emergenze raggruppate per priorità con durata di intervento inferiore a 5 minuti.	Valuta i parametri generati nella dimensione relativa alla durata degli interventi per facilitare i raggruppamenti.
8	Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017.	Valuta l'efficacia della frammentazione su un insieme di anni, confrontando poi i risultati delle query con quelle effettuate sullo schema originale e su quello con dimensioni.
2	Tempo medio tra chiamata e arrivo raggruppati per tipologia di chiamata per giorno e notte.	Valuta l'efficacia delle viste materializzate progettate per selezionare emergenze avvenute di giorno o di notte, confrontando i risultati della query rispetto allo schema con dimensioni, restituendo per ogni tipo di emergenza, la media del tempo di arrivo delle unità sul luogo, separando emergenze diurne e notturne
5	Numero di falsi allarmi per tipo emergenza avvenuti di giorno e di notte.	È stata testata per valutare l'efficacia delle viste materializzate, raggruppando per tipo emergenza e calcolando la percentuale di falsi allarmi.

2.3 ETL

Le fasi che vanno a costituire il modulo ETL possono essere rappresentate dai seguenti passi:

- Lettura dei blocchi csv contenenti i record da inserire
- Controllo validità record
- Trasformazione record
- Generazione di file csv contenenti i record trasformati (separati in fatti e dimensioni)
- Caricamento dei file csv su tabelle del database

La fase di **validation** inizia con il metodo *rowValidation* che effettua controlli sulla validità dei campi della tupla che si sta esaminando. In questo metodo si cerca di acquisire righe dal database che presentino i campi minimi per il funzionamento delle query, e vengono scartate quelle righe che non presentano tali campi. Sono state implementate in questo metodo basilari operazione di *salvataggio* di righe che sarebbero altrimenti scartate, settando dei valori arbitrari o casuali, al fine di non scartare un numero troppo elevato di righe a causa di un singolo o pochi valori mancanti. Un esempio di operazione di questo tipo è rappresentata dal settaggio del parametro *received_dtTm* fondamentale per la generazione di dimensioni e utilizzato in molteplici query: in questo caso per tuple con data di ricezione mancante si è scelto di assegnare come data quella corrente. Discorso analogo è stato fatto per il campo *on_site_dtTm* che in caso sia non compilato nella tupla, viene settato a partire da quello di ricezione della chiamata, aggiungendo un offset casuale di minuti in un range fissato (tra 10 e 40).

La **transformation** segue poi nel metodo *rowManipulation* dove vengono effettuate operazioni di pulizia e adattamento ai formati adatti al database. Seguono le principali operazioni:

- Sostituzione del carattere “virgola” con il carattere “underscore” nella colonna *city*.
- Mappatura delle priorità nei valori noti “2” e “3”.
- Formattazione campo *received_dtTm* secondo il formato di Postgresl.

- Assegnamento del *call_type_group* alle righe dove questo risultava mancante.
- Rimozione delle virgolette nel campo *call_type*.
- Trasformazione del parametro *als_unit* da booleano ad intero.
- Trasformazione del parametro contenente latitudine e longitudine in una stringa composita.
- Uniformazione dei valori del campo *city* (es: SF, San Francisco → SAN FRANCISCO).

Ogni volta che una riga viene letta e manipolata come descritto, se considerata valida, allora si procede con la prima parte della fase di *loading*, nella quale l'istanza appena elaborata viene scritta sui diversi file csv di output. Per ogni riga, è prevista la scrittura su tre file separati, uno per il **fatto originale** che non prevede ulteriori trasformazioni; uno per l'implementazione dello schema a stella, visibile in figura 1, che prevede la separazione in tabelle relative alle dimensioni tramite identificativi numerici, ed infine in quelli relativi allo schema frammentato per anno.

Durante questa fase vengono gestite le tuple univoche delle dimensioni individuate e descritte nel paragrafo 2 mediante l'utilizzo di dizionari Python, che tengono traccia dei valori e degli identificativi associati, e che al termine della lettura del file di input vengono scritti sui relativi file csv. Al termine di quest'operazione ha inizio l'ultima parte della fase di **loading**: attraverso la chiamata a **COPY** di PostgreSQL tutti i file csv, quindi il fatto nelle sue diverse implementazioni e le dimensioni, vengono copiate nelle relative tabelle. Per garantire l'assenza di duplicati all'interno delle tabelle è necessario *ripulire* i file utilizzati da **COPY** al termine dell'esecuzione. Conservare righe relative ad iterazioni precedenti comporta la ripetizione della copia delle stesse righe e quindi la presenza di tabelle con elementi non univoci e di dimensioni superiori a quelle necessarie.

2.3.1 Tuple fittizie

Per permettere l'analisi delle query su un numero arbitrario di row del database si è sviluppata una procedura che permette di generare tuple internamente consistenti, che presentino tutti i campi utili alle query. Tale procedura utilizza un mix di dati reali (prelevati da altre tuple già esistenti), e campi generati in modo casuale per l'occasione. I parametri *call_number* e *unit_id* sono stati generati come stringhe casuali. I valori di priorità sono stati scelti in modo casuale tra quelli considerati legali (2 e 3). I campi relativi ai luoghi geografici, ossia quelli presenti nella dimensione *dim_geo_place* (es: *City, Neighbourhood*) sono assegnati in maniera casuale e consistente (all'interno della stessa tupla), prelevando combinazioni di tali parametri da tuple già esistenti. Analogamente, la scelta casuale per i campi della dimensione *dim_responsibility* inerenti le indicazioni di responsabilità amministrativa, sono stati prelevati da righe già presenti nel database. I campi che non sono interessati dalle query sviluppate sono stati lasciati nulli.

2.4 Piano di esecuzione

Per creazione del data warehouse progettato si è deciso di dividere il dataset originale, composto da circa 5 milioni di istanze, in diversi *blocchi*, ognuno dei quali composto da 250'000 elementi. L'inserimento graduale di questi blocchi serve a dare una buona panoramica del comportamento della struttura al crescere della dimensione dei dati contenuti al suo interno. Ulteriori 5 blocchi da un milione di dati fittizi ciascuno verranno inseriti dopo quelli estratti dal dataset, al fine di raggiungere un numero di istanze pari a circa dieci milioni.

Il piano d'esecuzione è quindi strutturato in questo modo:

- Opzionale: viene generato un csv contenente tuple fittizie.
- Viene aperto un file di input (che chiameremo *blocco*).
- Vengono eseguite le fasi di lettura e trasformazione dell'ETL.
- Terminato il file si procede con la fase di *loading* dell'ETL.
- Vengono aggiornate le viste.
- Vengono eseguite le query e per ognuna calcolata la media su 5 esecuzioni.

- Vengono creati gli indici.
- Vengono eseguite le query e per ognuna calcolata la media su 5 esecuzioni.
- Vengono eliminati gli indici.
- Si riparte dal primo punto fin quando non terminano i blocchi da inserire.

Un sistema di contatori settati in questa fase permettono la raccolta di tutti i tempi utili all'analisi, che verranno mostrati nei capitoli successivi..

3 Implementazione

È stato sviluppato uno script Python per simulare l'intera evoluzione del processo di acquisizione, aggiornamento e valutazione del data warehouse.

Prima di eseguire lo script, il dataset “Fire Department Calls for Service” è stato partizionato in diversi file csv, ognuno dei quali costituito da 250’000 record. Lo script si occupa della progressiva lettura dei file, dell’inserimento degli stessi nel database previa fase ETL, dell’esecuzione delle query di testing e dell’esportazione dei dati ottenuti.

Un’esecuzione dello script python prevede, dopo aver specificato i file csv (partizioni del dataset originale) ed un eventuale numero di tuple fittizie, che si vogliono importare nel data warehouse, la lettura progressiva di tali file, la pulizia delle tuple lette e l’esportazione di fatti e dimensioni così generate in file CSV appositi, con annessa copia nel database PostgreSQL tramite la funzione *copy_from* fornita dalla libreria *Psycopg2* di interfaccia per python. Il meccanismo che permette a fatti diversi di condividere riferimenti a medesime tuple delle tabelle dimensioni è stato implementato tramite dei dictionary (il cui funzionamento è analogo a quello delle map di Java) che associano ai valori delle tuple, i relativi ID già presenti nel warehouse.

Durante la fase di acquisizione e pulizia, essendo le partizioni scandite sequenzialmente, vengono fatte eseguire le query e raccolti i tempi di esecuzione. Raccogliere i tempi delle query dopo ogni troncone di dati inserito permette quindi di studiare il comportamento del warehouse al crescere delle righe importate, i cui risultati sono visibili nel paragrafo 4.4.

3.1 Query

In questo paragrafo viene mostrato il codice sorgente delle interrogazioni, si rimanda ai paragrafi 2.2 per una visione d’insieme ed ai paragrafi del capitolo 4 per i dettagli e le analisi relative a ciascuna query. Le query sono state eseguite dallo script Python attraverso la funzione *execute* di *Psycopg2*.

Per ogni query sono state previste diverse versioni legate alle esigenze di testing:

1A, Numero di segnalazioni ‘HazMat’ per anno e per quartiere. (Tabella fatto originale)

```
SELECT date_part('year', received_dttm), neighborhood_district, count(*)  
FROM dispatch911_original  
WHERE call_type='HazMat'  
GROUP BY date_part('year', received_dttm), neighborhood_district
```

1A, Numero di segnalazioni ‘HazMat’ per anno e per quartiere. (Tabella dimensioni)

```
SELECT dat.year_f, geo.neighborhoods, count(*)  
FROM dispatch911_dimensions AS dis INNER JOIN dim_geo_place AS geo ON  
    ↪ dis.id_geo_place=geo.id_geo_place  
INNER JOIN dim_call_type AS callt ON callt.id_call_type=dis.id_call_type  
INNER JOIN dim_received_date AS dat ON  
    ↪ dat.id_received_date=dis.id_received_date  
WHERE callt.call_type='HazMat'  
GROUP BY dat.year_f, geo.neighborhoods
```

1B, Numero di segnalazioni 'HazMat' per quartiere negli anni compresi tra il 2010 e il 2015.
(Tabella dimensioni)

```
SELECT geo.neighborhoods, COUNT( distinct call_number)  
FROM dispatch911_dimensions AS dis INNER JOIN dim_geo_place AS geo ON  
    ↪ dis.id_geo_place=geo.id_geo_place  
INNER JOIN dim_call_type AS callt ON callt.id_call_type=dis.id_call_type  
INNER JOIN dim_received_date AS dat ON  
    ↪ dat.id_received_date=dis.id_received_date  
WHERE callt.call_type='HazMat' AND dat.year_f IN  
    ↪ ('2010', '2011', '2012', '2013', '2014', '2015')  
GROUP BY geo.neighborhoods
```

1B, Numero di segnalazioni 'HazMat' per quartiere negli anni compresi tra il 2010 e il 2015.
(Tabella frammentazione)

```
SELECT neighborhoods, COUNT(DISTINCT q1.call_number)  
FROM(((SELECT * FROM dispatch911_frag_2010)  
    UNION ALL  
    (SELECT * FROM dispatch911_frag_2011)  
    UNION ALL  
    (SELECT * FROM dispatch911_frag_2012)  
    UNION ALL  
    (SELECT * FROM dispatch911_frag_2013)  
    UNION ALL  
    (SELECT * FROM dispatch911_frag_2014)  
    UNION ALL  
    (SELECT * FROM dispatch911_frag_2015) ) AS q1  
INNER JOIN  
(SELECT id_call_type,call_type  
FROM dim_call_type AS calltype  
WHERE calltype.call_type='HazMat'  
) AS q2  
ON q1.id_call_type = q2.id_call_type  
INNER JOIN  
(  
    SELECT id_geo_place,neighborhoods  
    FROM dim_geo_place  
) AS q3  
ON q1.id_geo_place = q3.id_geo_place)  
GROUP BY q3.neighborhoods
```

2, Tempo medio tra chiamata e arrivo raggruppati per tipologia di chiamata per giorno e notte. (Tabella dimensioni)

```

SELECT dayquery.call_type, (minutes_day), (minutes_night)
FROM
    (SELECT call_type, AVG(minutes) AS minutes_day
     FROM dispatch911_dimensions fact
      INNER JOIN dim_duration dur ON (fact.id_duration =
                                       ↪ dur.id_duration)
      INNER JOIN dim_call_type AS emergency ON
           ↪ fact.id_call_type = emergency.id_call_type
      INNER JOIN dim_received_date AS recdate ON
           ↪ fact.id_received_date =
           ↪ recdate.id_received_date
      WHERE recdate.hour_f IN
           ↪ (5,6,7,8,9,10,11,12,13,14,15,16)
      GROUP BY call_type
    ) AS dayquery
INNER JOIN
    (SELECT call_type, AVG(minutes) AS minutes_night
     FROM dispatch911_dimensions fact
      INNER JOIN dim_duration dur ON (fact.id_duration =
                                       ↪ dur.id_duration)
      INNER JOIN dim_call_type AS emergency ON fact.id_call_type
           ↪ = emergency.id_call_type
      INNER JOIN dim_received_date AS recdate ON
           ↪ fact.id_received_date = recdate.id_received_date
      WHERE recdate.hour_f IN (17,18,19,20,21,22,23,24,1,2,3,4)
      GROUP BY call_type
    ) AS nightquery
ON dayquery.call_type = nightquery.call_type

```

2, Tempo medio tra chiamata e arrivo raggruppati per tipologia di chiamata per giorno e notte. (Tabella dimensioni)

```

SELECT dayquery.call_type, dayquery.AVGminutes AS dayAVG,
       ↪ nightquery.AVGminutes AS nightAVG
FROM
    (SELECT call_type, AVG(minutes) AS AVGminutes
     FROM intervention_daytime
     GROUP BY call_type
    ) AS dayquery
INNER JOIN
    (SELECT call_type, AVG(minutes) AS AVGminutes
     FROM intervention_nighttime
     GROUP BY call_type
    ) AS nightquery
ON dayquery.call_type = nightquery.call_type

```

3, Per ogni stagione e per ogni quartiere, numero di segnalazioni per incendi. (Tabella fatto originale)

```
(SELECT '1' as season,neighborhood_district, count(*)
FROM dispatch911_original
```

```

WHERE date_part('month',received_dttm) IN (12,1,2) AND
    ↪ call_type_group='Fire'
GROUP BY neighborhood_district)
union
(SELECT '2' as season,neighborhood_district, count(*)
FROM dispatch911_original
WHERE date_part('month',received_dttm) IN (3,4,5) AND
    ↪ call_type_group='Fire'
GROUP BY neighborhood_district)
union
(SELECT '3' as season,neighborhood_district, count(*)
FROM dispatch911_original
WHERE date_part('month',received_dttm) IN (6,7,8) AND
    ↪ call_type_group='Fire'
GROUP BY neighborhood_district)
union
(SELECT '4' as season,neighborhood_district, count(*)
FROM dispatch911_original
WHERE date_part('month',received_dttm) IN (9,10,11) AND
    ↪ call_type_group='Fire'
GROUP BY neighborhood_district)
order by neighborhood_district, 1

```

3, Per ogni stagione e per ogni quartiere, numero di segnalazioni per incendi. (Tabella dimensioni)

```

SELECT dat.season, geo.neighborhoods, count(*)
FROM dispatch911_dimensions as dis INNER JOIN dim_geo_place as geo ON
    ↪ dis.id_geo_place=geo.id_geo_place
INNER JOIN dim_call_type as callt ON callt.id_call_type=dis.id_call_type
INNER JOIN dim_received_date as dat ON
    ↪ dat.id_received_date=dis.id_received_date
WHERE callt.call_type_group='Fire'
GROUP BY dat.season, geo.neighborhoods
order by 2,1

```

4, Media numero unità inviate per città, emergenza e priorità dichiarata. (Tabella fatto originale)

```

SELECT q2.city,q2.call_type,q2.original_priority,
    ↪ avg(number_of_unit_dispatched)
FROM
    (SELECT call_number, COUNT(*) as number_of_unit_dispatched
        FROM dispatch911_original
        GROUP BY call_number
    )as q1
INNER JOIN
    (SELECT distinct call_number, original_priority,
        ↪ call_type,city
        FROM dispatch911_original as fact
    )as q2
on q1.call_number = q2.call_number
GROUP BY q2.original_priority,q2.call_type,q2.city
ORDER BY 4

```

4, Media numero unità inviate per città, emergenza e priorità dichiarata. (Tabella dimensioni)

```

SELECT city,q2.call_type,q2.original_priority,
      ↪ avg(number_of_unit_dispatched)
FROM
      (SELECT call_number, COUNT(*) as number_of_unit_dispatched
       FROM dispatch911_dimensions
       GROUP BY call_number
      )as q1
INNER JOIN
      (SELECT distinct call_number, original_priority,
       ↪ call_type, city
       FROM dispatch911_dimensions as fact
       INNER JOIN
          dim_call_type as emer
          on fact.id_call_type = emer.id_call_type
       INNER JOIN
          dim_geo_place as geop
          on fact.id_geo_place = geop.id_geo_place
      )as q2
      on q1.call_number = q2.call_number
GROUP BY q2.city,q2.original_priority,q2.call_type
ORDER BY 4

```

5, Numero di falsi allarmi per tipo emergenza avvenuti di giorno e di notte. (Tabella dimensioni)

```

SELECT dayq.call_type, dayCOUNT, nightCOUNT
FROM
      (SELECT call_type, COUNT(*) as dayCOUNT
       FROM dispatch911_dimensions fact
       INNER JOIN dim_duration dur on (fact.id_duration =
           ↪ dur.id_duration)
       INNER JOIN dim_call_type as emergency on
           ↪ fact.id_call_type = emergency.id_call_type
       INNER JOIN dim_received_date as recdate on
           ↪ fact.id_received_date =
           ↪ recdate.id_received_date
       WHERE recdate.hour_f in
           ↪ (1,2,3,4,5,6,7,8,9,10,11,12) and
           ↪ final_priority < original_priority
       GROUP BY call_type) as dayq

      INNER JOIN

      (SELECT call_type, COUNT(*) as nightCOUNT
       FROM dispatch911_dimensions fact
       INNER JOIN dim_duration dur on (fact.id_duration =
           ↪ dur.id_duration)
       INNER JOIN dim_call_type as emergency on fact.id_call_type
           ↪ = emergency.id_call_type
       INNER JOIN dim_received_date as recdate on
           ↪ fact.id_received_date = recdate.id_received_date
       WHERE recdate.hour_f in
           ↪ (13,14,15,16,17,18,19,20,21,22,23,24) and
           ↪ final_priority < original_priority

```

```
        GROUP BY call_type) nightq  
  
    on dayq.call_type = nightq.call_type
```

5, Numero di falsi allarmi per tipo emergenza avvenuti di giorno e di notte. (Vista materializzata)

```
SELECT dayq.call_type, dayCOUNT, nightCOUNT  
FROM  
    (SELECT call_type, COUNT(*) as dayCOUNT  
     FROM intervention_daytime  
     WHERE final_priority < original_priority  
     GROUP BY call_type) as dayq  
  
    INNER JOIN  
  
    (SELECT call_type, COUNT(*) as nightCOUNT  
     FROM intervention_nighttime  
     WHERE final_priority < original_priority  
     GROUP BY call_type) nightq  
  
    on dayq.call_type = nightq.call_type
```

6, Emergenze raggruppate per priorità con durata di intervento inferiore a 5 minuti. (Tabella fatto originale)

```
SELECT fact.original_priority, count(*)  
FROM dispatch911_original as fact  
WHERE fact.durationminutes <= 5  
GROUP BY fact.original_priority
```

6, Emergenze raggruppate per priorità con durata di intervento inferiore a 5 minuti. (Tabella dimensioni)

```
SELECT fact.original_priority, count(*)  
FROM dispatch911_dimensions as fact  
INNER JOIN dim_call_type as emergency on fact.id_call_type =  
    ↪ emergency.id_call_type  
INNER JOIN dim_duration as dur on fact.id_duration = dur.id_duration  
WHERE dur.lessfive=TRUE  
GROUP BY fact.original_priority
```

7, Interventi del battaglione B01 per tipo di emergenza, con media durata di intervento e numero interventi. (Tabella fatto originale)

```
SELECT fact.call_type, count(*), avg(fact.durationminutes)  
FROM dispatch911_original as fact  
WHERE fact.battalion = 'B01'  
GROUP BY fact.call_type
```

7, Interventi del battaglione B01 per tipo di emergenza, con media durata di intervento e numero interventi. (Tabella dimensioni)

```

SELECT emergency.call_type, count(*), avg(dur.minutes)
FROM dispatch911_dimensions as fact
INNER JOIN dim_call_type as emergency on fact.id_call_type =
    ↪ emergency.id_call_type
INNER JOIN dim_duration as dur on fact.id_duration = dur.id_duration
INNER JOIN dim_responsibility as resp on fact.id_responsibility =
    ↪ resp.id_responsibility
WHERE resp.battalion = 'B01'
GROUP BY emergency.call_type

```

8, Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017. (Tabella dimensioni)

```

SELECT box, COUNT (DISTINCT call_number) as number_of_calls
FROM
(
    (
        SELECT *
            FROM dispatch911_dimensions as fact) as q1
        JOIN
        (
            SELECT id_geo_place
                FROM dim_geo_place as geo
                WHERE geo.city='SAN FRANCISCO') as q2
        on q1.id_geo_place = q2.id_geo_place
        JOIN
        (
            SELECT recdate.id_received_date
                FROM dim_received_date as recdate
                WHERE recdate.year_f in ('2015','2016','2017')
        )as q2bis
        on q1.id_received_date = q2bis.id_received_date
    )
    JOIN
    (
        SELECT id_responsibility, box
            FROM dim_responsibility as resp
    ) as q3
    on q1.id_responsibility = q3.id_responsibility
GROUP BY box
ORDER BY 2 DESC

```

8, Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017. (Tabella frammentazione)

```

SELECT box, COUNT (DISTINCT call_number) as number_of_calls
FROM
(
    (
        (
            SELECT * FROM dispatch911_frag_2015)
        UNION ALL
        (SELECT * FROM dispatch911_frag_2016)
        UNION ALL
        (SELECT * FROM dispatch911_frag_2017) ) as q1
        INNER JOIN
        (
            SELECT id_geo_place

```

```
        FROM dim_geo_place AS geo
        WHERE geo.city='SAN FRANCISCO') AS q2
    ON q1.id_geo_place = q2.id_geo_place
)
INNER JOIN
(
    SELECT id_responsibility, box
        FROM dim_responsibility AS resp
) AS q3
ON q1.id_responsibility = q3.id_responsibility
GROUP BY box
ORDER BY 2 DESC
```

8, Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017. (Tabella fatto originale)

```
SELECT box, COUNT(DISTINCT call_number) AS number_of_calls
FROM dispatch911_original AS fact
WHERE city='SAN FRANCISCO' AND date_part('year', received_dttm) IN
    ('2015', '2016', '2017')
GROUP BY box
ORDER BY 2 DESC
```

3.2 Viste materializzate

Seguono le definizioni delle viste materializzate create per testare le query che raggruppano emergenze avvenute di notte e di giorno.

3.2.1 Vista Giorno

Raccoglie le chiamate effettuate di giorno

```
SELECT fact.rowid,
    dur.minutes,
    emergency.call_type,
    fact.original_priority,
    fact.final_priority
FROM dispatch911_dimensions fact
JOIN dim_duration dur ON fact.id_duration = dur.id_duration
JOIN dim_call_type emergency ON fact.id_call_type =
    ↪ emergency.id_call_type
JOIN dim_received_date recdate ON fact.id_received_date =
    ↪ recdate.id_received_date
WHERE recdate.hour_f = ANY (ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    ↪ 12]);
```

3.2.2 Vista Notte

Raccoglie le chiamate effettuate di notte

```
SELECT fact.rowid,
    dur.minutes,
    emergency.call_type,
```

```
fact.original_priority,  
fact.final_priority  
FROM dispatch911_dimensions fact  
JOIN dim_duration dur ON fact.id_duration = dur.id_duration  
JOIN dim_call_type emergency ON fact.id_call_type =  
    ↪ emergency.id_call_type  
JOIN dim_received_date recdate ON fact.id_received_date =  
    ↪ recdate.id_received_date  
WHERE recdate.hour_f = ANY (ARRAY[13, 14, 15, 16, 17, 18, 19, 20, 21,  
    ↪ 22, 23, 24]);
```

3.3 Piano operativo: lo script Python

L'acquisizione dei tempi di esecuzione delle query, dell'aggiornamento delle viste materializzate e della creazione degli indici è delegata allo script Python.

Come già detto nel paragrafo 2.4 la prima parte del piano di testing riguarda la lettura dei blocchi da inserire. Per effettuare quest'operazione sono stati memorizzati in una lista i percorsi dei file su disco, quindi iterativamente viene letto il contenuto della lista ed aperto in lettura il file (righe 600-650, 704 del codice in appendice)⁵. Se il path non è relativo a uno dei blocchi estratti quelli dal dataset, sarà generato un file da un milione di tuple fittizie invocando il metodo `generateConsistentFakeRows` (riga 713)⁵.

Dopo aver aperto il scrittura i file di output del fatto originale e quello con dimensioni vengono aperti i file di output del fatto frammentato con la chiamata a `openFragmentationFiles`(riga 722)⁵. Per la gestione delle diverse tabelle legate alla frammentazione orizzontale del fatto si è deciso di utilizzare un dizionario (`fragTablesPath`), nel quale il campo chiave è un intero che indica l'anno di riferimento della tabella, e il valore è un'istanza della classe `FragFile`, che contiene per ogni anno incontrato durante l'elaborazione il nome della relativa tabella frammentata, il percorso e il file descriptor del file csv di output. Durante la creazione della classe viene anche eseguita la query per la creazione della relativa tabella in postgresSQL.

Il metodo per la creazione di queste istanze è `newFragFilePath`(riga 579)⁵.

A questo punto dell'esecuzione vengono caricate in diversi dizionari le istanze di dimensioni create fino a quel momento. Quest'operazione viene effettuata dalle funzioni `putDimensionTableInDictionary` dove *Dimension* varia in *duration*, *geoPlace*, *date*, *responsibility* e *callType*. Le funzioni a seconda della dimensione interessata interrogano il database chiedendo tutte le istanze presenti nella tabella delle diverse dimensioni, l'output ottenuto viene poi riscritto all'interno dei dizionari Python. In questa fase viene ritornato in output l'id dell'ultimo elemento inserito in tabella dimensione.

A questo punto una volta aperto il file csv di output in scrittura si procede con la lettura riga per riga del file di input, in questa fase vengono eseguite le operazioni di ETL che si articolano in diverse funzioni:

- `rowValidation`: ritorna una booleano che indica se la riga considerata è valida o meno a seconda dei criteri precedentemente descritti.^{2,3}
- `rowManipulation`: effettua le operazioni di *trasformazione* della procedura ETL.^{2,3}
- `getDimensionTableRow`: con *table* che prende il riferimento delle dimensioni *duration*, *geoPlace*, *date*, *responsibility* e *callType* provvede ad inserire all'interno dei dizionari Python i record delle dimensioni. Attraverso il metodo `get` dei dizionari Python viene effettuata una ricerca sui campi chiave sfruttando un'hash table, campi in cui si trovano i valori che caratterizzano univocamente il record dimensione. Se la ricerca di questo valore produce un risultato nullo allora viene creato un nuovo elemento del dizionario avente come chiave il valore logico della nuova dimensione e come valore l'id associato. Il campo valore dei dizionari contiene invece l'id che collega la dimensione al fatto.

Al termine di questa fase si hanno a disposizione tutti gli elementi per poter scrivere le istanze del fatto nelle sue diverse configurazioni nei relativi file csv di output.

Al termine della lettura del file di input, attraverso il metodo `exportDimensionToCsv`, viene scritto il contenuto dei dizionari delle dimensioni nei relativi file csv. Vengono scritte le sole tuple del dizionario il cui identificativo è maggiore di quello registrato come ultimo id all'inizio dell'esecuzione. Quest'operazione è necessaria per evitare la duplicazione dei record delle dimensioni. Una volta completata la scrittura di tutti i file csv è possibile

trasferire questi contenuti in PostgreSQL attraverso le chiamate al metodo `csvToPostgres`, che dato il percorso di un file e il nome di una tabella esegue il `COPY` del file nella tabella.

Terminata la fase di inserimento viene eseguito il piano di testing. La prima operazione riguarda l'aggiornamento delle viste materializzate (righe 819,820)⁵, seguita subito dopo dall'esecuzione delle query. In particolare per i tempi d'esecuzione delle query è stata sviluppata la classe `QueryTester`.

```
1  class QueryTester:
2      csvQueryResults=None
3      resultsCsvWriter=None
4      queryArray=[]
5      queryIterations = 6
6      csvQueryResultsPath = Path.cwd() / 'output/queryResults.csv'
7
8      def __init__(self):
9          self.csvQueryResults = open(self.csvQueryResultsPath, 'w',
10              ↪ newline='')
11         self.resultsCsvWriter = csv.writer(self.csvQueryResults,
12             ↪ lineterminator='\n', delimiter=';')
13
14         self.queryArray.append("...") # INSERIMENTO QUERY
15
16     def computeAndWriteAvgs(self, block, usingIndex):
17         queryIndex=1
18         for q in self.queryArray:
19             queryTime = 0
20             for i in range(0,self.queryIterations):
21                 query_start_time=time.time()
22                 cur.execute(q)
23                 if (i!=0):
24                     queryTime=queryTime+(time.time()-query_start_time)
25
26                     outResultRow=[block, queryIndex, usingIndex, queryTime /
27                         ↪ (self.queryIterations-1) * 1000]
28                     self.resultsCsvWriter.writerow(outResultRow)
29                     queryIndex=queryIndex+1
```

Gli attributi della classe sono il file descriptor e il percorso del file di output sul quale saranno esportati i dati, un array contenente le query da testare e un valore che indica il numero di volte in cui verranno eseguite le query e sul quale si calcolerà la media delle esecuzioni.

L'inizializzazione dell'istanza di `QueryTester` non necessita di parametri e si occupa dell'apertura del file di output e dell'inizializzazione dell'operatore di scrittura csv, ma soprattutto (*riga 12*) popola l'array delle query da testare. Per ragioni di leggibilità in questo caso non è stato riportato integralmente l'inserimento delle query, ma è possibile consultare la reale implementazione in appendice.⁵

Il metodo `computeAndWriteAvgs` si occupa dell'esecuzione delle query e della scrittura dei risultati sul file di output. Prende in input due parametri, il primo (*block*) indica quanti blocchi di dati sono stati inseriti in quel momento nel database, il secondo (*usingIndex*) è una stringa che indica se prima dell'esecuzione dei test sono stati creati o meno gli indici. Per ogni query contenuta nell'array della classe vengono eseguite *queryIterations* esecuzioni, ed ignorando la prima esecuzione vengono sommati i tempi calcolati attraverso il modulo `time` di Python ed infine divisi per il numero di esecuzioni. Al termine delle esecuzioni delle singole query vengono scritti su file csv i dati relativi al blocco di inserimento, alla query, all'utilizzo degli indici ed il tempo espresso in millisecondi. Prima di eseguire questo tipo di test è stata verificata la corrispondenza dei tempi raccolti attraverso questo metodo e quelli raccolti in *pgAdmin*.

Al termine di quest'operazione vengono creati gli indici attraverso la chiamata al metodo `createIndex` della classe `queryTester` e subito dopo rieseguito il test sulle query attraverso `computeAndWriteAvgs`.

La procedura termina quando tutti i file di input sono stati elaborati.

4 Risultati

In questi capitoli verranno riportati ed analizzati i risultati ottenuti dall'esecuzione del piano di esecuzione descritto nei capitoli precedenti. In particolare saranno studiati i dati relativi ai tempi di esecuzione delle query, alle operazioni di aggiornamento delle viste materializzate e quelli di creazione degli indici. In questa fase introduttiva verrà analizzata la bontà del dataset utilizzato.

In tabella 6 sono riportati i dati relativi all'inserimento delle tuple per ogni blocco di input. Come già detto sono stati inseriti in totale 23 blocchi di dati, i primi 18 da 250'000 elementi relativi alle tuple del dataset, i restanti da un milione, ossia le tuple fittizie generate. Secondo i criteri di validità descritti nel capitolo 2.3 una minima parte delle tuple vengono scartate, come leggibile nella tabella 6: nella quasi totalità dei casi è il 99% dei dati ad essere considerato valido, un valore indicativo della bontà del dataset studiato. Per i blocchi dal 19 al 23 la validità non è indice di qualità dato che si fa riferimento a dati generati appositamente e per i quali non si è ritenuto ragionevole inserire fattori che potessero determinarne la non validità.

Validità delle tuple per ogni blocco in ETL			
Blocchi inseriti	Non valide	Valide	Validità
1	377	249623	99,85%
2	483	249517	99,81%
3	427	249573	99,83%
4	381	249619	99,85%
5	451	249549	99,82%
6	1255	248745	99,50%
7	1522	248478	99,39%
8	1584	248416	99,37%
9	1498	248502	99,40%
10	1396	248604	99,44%
11	1441	248559	99,42%
12	950	249050	99,62%
13	1059	248941	99,58%
14	28242	221758	88,70%
15	1189	248811	99,52%
16	952	249048	99,62%
17	926	249074	99,63%
18	2734	247266	98,91%
19	0	1000000	100%
20	0	1000000	100%
21	0	1000000	100%
22	0	1000000	100%
23	0	1000000	100%

Tabella 6: Indice di bontà del dataset “Fire Department Calls for Service”.

4.1 Tempi query

Seguono i dati di tempo relativi al completamento delle query già descritte, effettuate sul dataset completamente acquisito. Nelle sezione a seguire saranno invece confrontati e commentati i risultati query per query, ad ogni acquisizione di blocco di dati.

<i>Query 1A: Numero di segnalazioni 'HazMat' per anno e per quartiere.</i>			
Tabella fatto originale		Tabella dimensioni	
Indice	No indice	Indice	No indice
1395 ms	1398 ms	2264 ms	2949 ms

<i>Query 1B: Numero di segnalazioni 'HazMat' per quartiere negli anni compresi tra il 2010 e il 2015.</i>			
Tabella dimensioni		Tabella frammentazione	
Indice	No indice	Indice	No indice
925 ms	950 ms	183 ms	316 ms

<i>Query 2: Tempo medio tra chiamata e arrivo raggruppati per tipologia di chiamata per giorno e notte.</i>			
Tabella dimensioni		Vista materializzata	
Indice	No indice	Indice	No indice
14826 ms	12839 ms	2224 ms	2252 ms

<i>Query 3: Per ogni stagione e per ogni quartiere, numero di segnalazioni per incendi.</i>			
Tabella fatto originale		Tabella dimensioni	
Indice	No indice	Indice	No indice
7091 ms	7331 ms	2687 ms	2945 ms

<i>Query 4: Media numero unità inviate per città, emergenza e priorità dichiarata.</i>			
Tabella fatto originale		Tabella dimensioni	
Indice	No indice	Indice	No indice
131848 ms	135349 ms	130341 ms	132787 ms

<i>Query 5: Numero di falsi allarmi per tipo emergenza avvenuti di giorno e di notte.</i>			
Tabella dimensioni		Vista materializzata	
Indice	No indice	Indice	No indice
6668 ms	6695 ms	2381 ms	2398 ms

<i>Query 8: Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017.</i>			
Tabella fatto originale		Tabella dimensioni	
Indice	No indice	Indice	No indice
1989 ms	2009 ms	1299 ms	1296 ms

<i>Query 7: Interventi del battaglione B01 per tipo di emergenza, con media durata di intervento e numero interventi.</i>			
Tabella fatto originale		Tabella dimensioni	
Indice	No indice	Indice	No indice
1787 ms	1800 ms	1320 ms	1341 ms

<i>Query 6: Emergenze raggruppate per priorità con durata di intervento inferiore a 5 minuti.</i>					
Tabella fatto originale		Tabella dimensioni		Tabella frammentazione	
Indice	No indice	Indice	No indice	Indice	No indice
4400 ms	4418 ms	6280 ms	5085 ms	3314 ms	3260 ms

I tempi riportati in tabella rappresentano la media delle esecuzioni delle query registrate in seguito all'inserimento dell'ultimo blocco di dati. I dati inseriti sono circa 10 milioni e per ottenere l'output dalle query si registrano tempi in generale nell'ordine dei pochi secondi. È possibile notare come in alcuni casi risulti essere più conveniente l'uso degli indici, e in molti delle viste materializzate, o di tabelle frutto di una frammentazione, ma un'analisi più precisa delle singole query verrà affrontata nei paragrafi successivi.

4.2 Aggiornamento Viste Materializzate

La tabella riporta i tempi relativi ai secondi richiesti per l'aggiornamento delle due viste materializzate riportate nei capitoli 3.2.2 e 3.2.1. Come ci si aspettava il tempo necessario per quest'operazione cresce all'aumentare del numero di elementi inseriti, partendo da un secondo richiesto quando gli elementi inseriti sono 250'000, fino ad arrivare agli 87 secondi a fronte dei 10 milioni di elementi correntemente presenti nel data warehouse.

Le query direttamente coinvolte dall'utilizzo delle viste materializzate sono la 2 e la 5. All'ultimo inserimento, utilizzando le viste materializzate, le due query richiedevano nel migliore dei casi rispettivamente 2,2 secondi e 2,4 secondi.^{4.1} Questi tempi sono decisamente inferiori ai tempi richiesti per l'aggiornamento delle viste, ma va ovviamente considerato il fatto che l'operazione di aggiornamento viene effettuata una sola volta ad inserimento e per queste ragioni il suo costo in termini di tempo va distribuito sul numero di query eseguite. Tenendo conto dei tempi richiesti dalle due query se non supportate da viste materializzate (13 secondi e 6,6 secondi) per la query 2 all'ottava esecuzione risulta essere più conveniente l'uso della vista materializzata, per la query 5 la quindicesima.

Refresh Viste Materializzate	
Blocchi Inseriti	Secondi
1	1,09
2	1,95
3	3,04
4	4,05
5	4,43
6	7,56
7	7,30
8	9,32
9	10,45
10	11,51
11	12,53
12	14,39
13	14,06
14	14,99
15	15,38
16	18,54
17	20,08
18	22,10
19	40,38
20	48,25
21	62,99
22	87,28

4.3 Aggiornamento Indici

La tabella riporta i dati relativi alla creazione degli indici, al termine dell'esecuzione delle query gli indici vengono eliminati per consentire alla successiva iterazione di misurare i tempi senza l'ausilio degli stessi. Gli indici sono stati creati in corrispondenza dei campi id presenti nel fatto relativi alle dimensioni, la scelta degli indici è stata fatta in seguito ad uno studio delle analisi da eseguire. Tra le diverse tipologie di indici offerte da PostgreSQL si è scelto di utilizzare i B-tree.

Osservando i tempi notiamo che nell'ultima misura rilevata sono necessari 64 secondi per creare gli indici, un tempo che va però ammortizzato per tutte le query che intendono farne uso, pertanto visti i risultati mostrati in precedenza risulta essere conveniente l'uso degli indici, si rimanda tuttavia ai paragrafi successivi per ulteriori considerazioni.

Refresh indici	
Blocchi inseriti	Secondi
1	1,55
2	2,63
3	5,95
4	6,46
5	6,82
6	8,94
7	11,11
8	12,22
9	13,19
10	13,32
11	15,39
12	17,69
13	18,46
14	19,96
15	21,65
16	23,08
17	25,42
18	28,45
19	31,29
20	39,12
21	49,83
22	54,31
23	64,57

4.4 Analisi dei tempi

Seguono grafici e considerazioni fatte sui tempi raccolti eseguendo le query su porzioni di grandezza incrementale del database. Per ogni grafico vale la notazione in cui sulle ascisse viene indicato il numero di blocchi di dati inseriti e correntemente nel data warehouse, mentre sulle ordinate il tempo impiegato dall'operazione (query o ETL) misurato in millisecondi. I blocchi rappresentano partizioni del dataset originale. I blocchi da 1 a 18 sono blocchi in numero di tuple uniforme (250'000 elementi), mentre i blocchi da 19 a 23 rappresentano blocchi contenenti tuple fittizie generate per l'occasione e contengono ciascuno un milione di elementi.

4.4.1 ETL

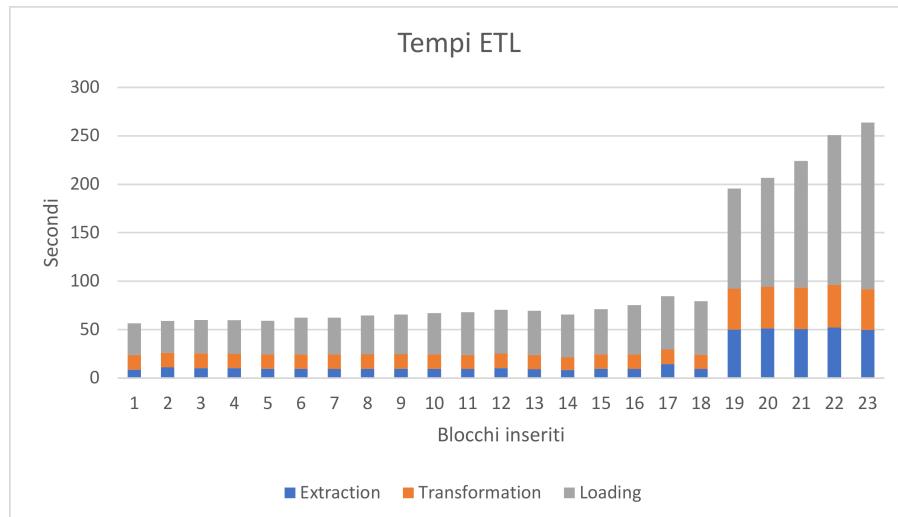


Figura 3: Tempi ETL

In figura 3 sono visionabili i dati relativi ai tempi della fase di pulizia e acquisizione dei dati dal dataset originale. Sono stati misurati i tempi specifici di ognuna delle tre fasi nel processamento di ognuna delle partizioni del dataset inserito. Una volta individuate le parti dell'esecuzione legate a ciascuna fase sono stati utilizzati diversi timer che calcolano il tempo impiegato per le operazioni di estrazione, e quindi apertura e lettura del file di origine dei dati, di trasformazione, descritta del dettaglio nel paragrafo 2.3 e infine caricamento, che unisce le operazioni di scrittura su file csv e di caricamento su database.

Dal grafico è possibile notare la regolarità delle fasi di estrazione e trasformazione che per i blocchi da 250'000 elementi è intorno ai 30 secondi mentre per i blocchi da un milione circa 100 secondi. Il discorso è diverso per quanto riguarda le fasi di caricamento, soprattutto per i blocchi da un milione di nota come al crescere degli elementi inseriti il tempo necessario alle operazioni di inserimento sia sempre maggiore.

4.4.2 Query 1A



Figura 4: Query 1A, Numero di segnalazioni 'HazMat' per anno e per quartiere.

Nella query 1A sono state contate le segnalazioni del tipo emergenza 'HazMat', raggruppate per anno in cui è avvenuta la chiamata e per quartiere. All'interno del singolo grafico vengono comparati gli andamenti dei tempi ottenuti sulla query nella versione "No Dimensioni", ossia con lo schema relazionale originale, e "Dimensioni", ossia lo schema a stella implementato. L'andamento della query man mano che cresce la dimensione del data warehouse come visibile nei grafici, mostra un comportamento piuttosto lineare, con una certa similarità nei tempi fin quando non si raggiungono gli inserimenti finali, dove l'esecuzione che sfrutta gli indici mostra un vantaggio rispetto a quella senza, quando si parla di esecuzione con dimensioni. A tali risultati vanno però aggiunti i tempi di attivazione degli indici, che per un data warehouse al 19esimo inserimento ammontano a circa 30 secondi. Tale overhead risulta quindi essere eccessivo se comparato al guadagno ottenuto attivando gli indici e se si intende utilizzare la query sporadicamente.

Il momento in cui la query con dimensioni risulta non essere più conveniente coincide con l'inserimento del 19esimo blocco, il primo da un milione di dati.

4.4.3 Query 1B

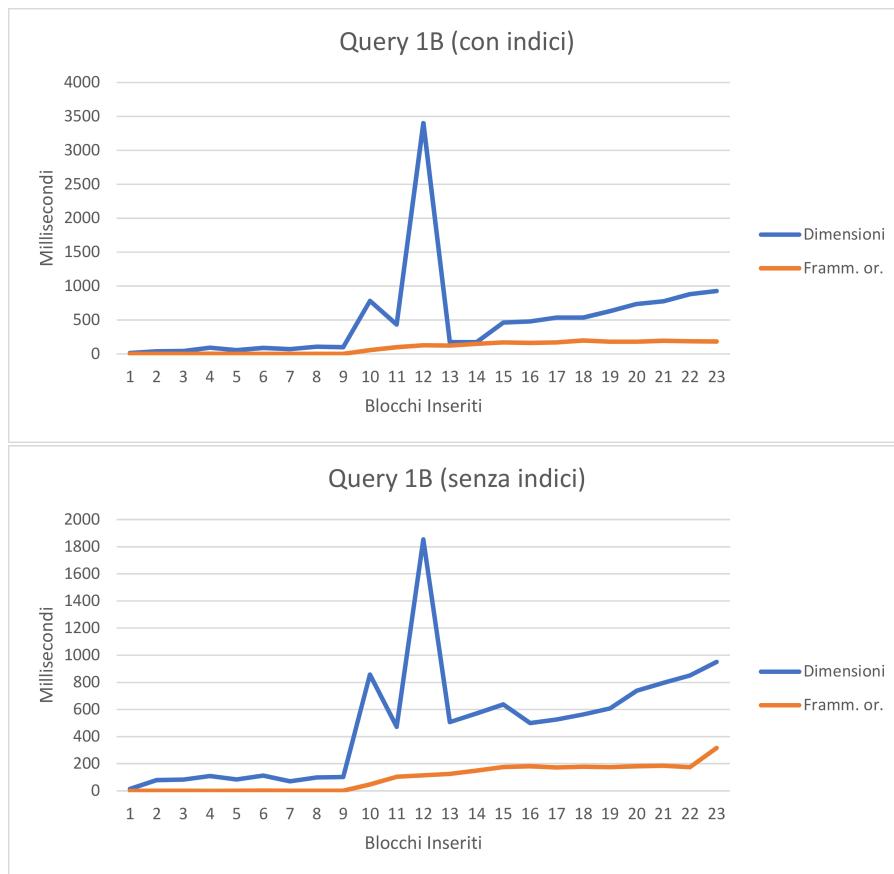


Figura 5: Query 1B, Numero di segnalazioni 'HazMat' per quartiere negli anni compresi tra il 2010 e il 2015.

Nella query 1B si è voluto comparare l'esecuzione di una query variante della 1A che filtri le chiamate in un intervallo di anni, nella versione schema a stella con unica tabella dei fatti, e nella versione a stella con frammentazione orizzontale sugli anni. L'innalzamento dei valori di tempo che avviene in corrispondenza del 12esimo inserimento corrisponde al momento in cui le chiamate che interessano gli anni 2010-2015 vengono effettivamente trovate nel data warehouse (i blocchi sono ordinati temporalmente). Nei grafici si può vedere come l'utilizzo di indici non comporti nessun vantaggio sensibile nel caso di tabella non frammentata e comporti invece un vantaggio apprezzabile nel caso di frammentazione orizzontale, ed inoltre di come l'utilizzo della frammentazione orizzontale incida in positivo in generale sull'esecuzione negli inserimenti finali.

4.4.4 Query 2

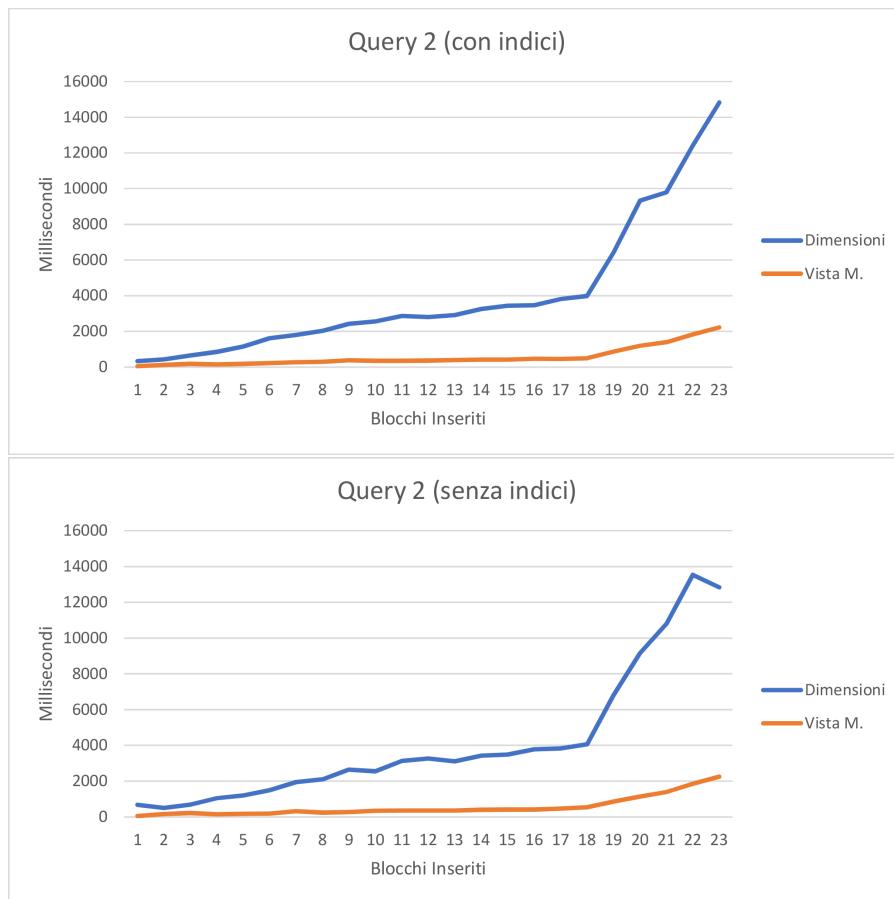


Figura 6: Query 2, Tempo medio tra chiamata e arrivo raggruppati per tipologia di chiamata per giorno e notte.

Nella query 2 si vogliono confrontare l'implementazione dello schema a stella utilizzando o meno due viste materializzate, che semplificano il recupero di chiamate avvenute di giorno e chiamate avvenute di notte. Come si evince dai grafici, il guadagno di tempo offerto dall'utilizzo di viste materializzate è particolarmente apprezzabile quando il date warehouse risulta più corposo, come nell'ultimo inserimento dove possiamo notare un guadagno di circa 12 secondi, sia nella versione con indici che senza. Considerando i tempi necessari all'aggiornamento delle viste e quelli della query che non sfrutta le viste all'ultima iterazione registrata, risulta essere conveniente l'uso delle viste materializzate già dalla quinta iterazione. Non si segnala alcun guadagno nell'attivazione di indici.

4.4.5 Query 3

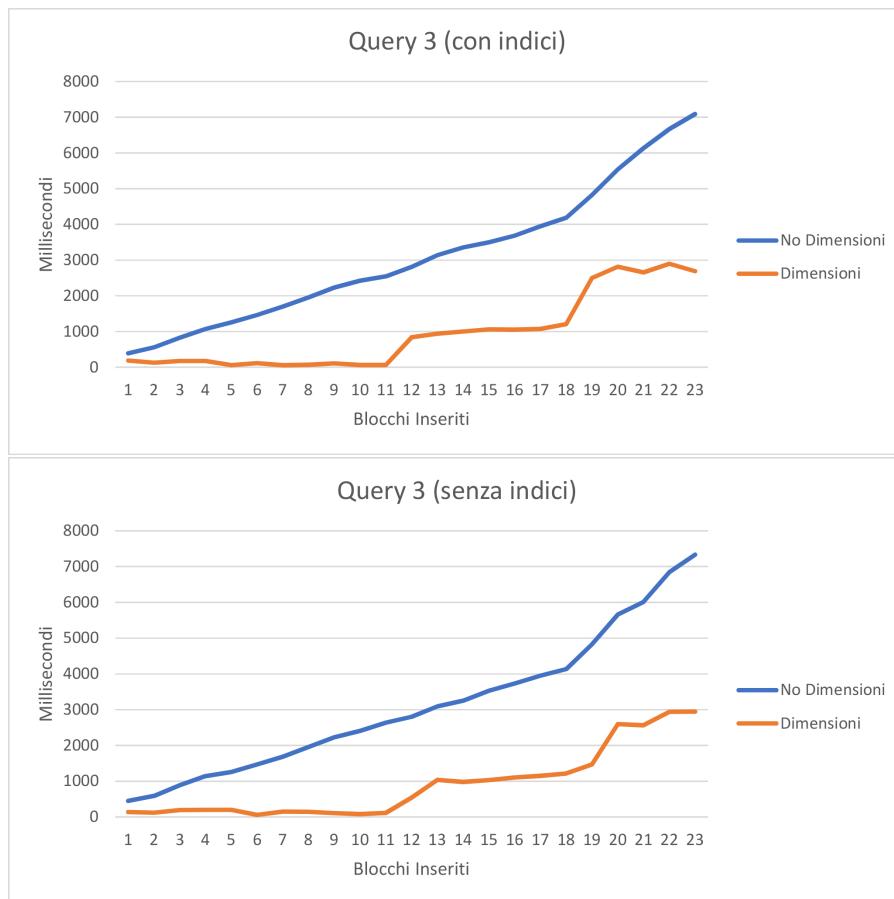


Figura 7: Query 3, Per ogni stagione e per ogni quartiere, numero di segnalazioni per incendi.

Nella query 3 vengono raggruppate le chiamate del tipo di emergenza "Incendio" per Stagione e Quartiere. Vengono quindi comparati lo schema a stella con fatto non frammentato, e lo schema originale senza separazione in fatti e dimensioni. In questa query è visibile il guadagno che si ottiene utilizzando le dimensioni ed annessi attributi gerarchici progettati, guadagno che arriva a toccare i 4 secondi in meno nel caso di non utilizzo di indici. L'attivazione comporta un guadagno in termini di millisecondi esiguo che richiederebbe un numero di esecuzione della query molto alto prima di poter garantire un vantaggio rispetto al costo di attivazione degli indici.

4.4.6 Query 4

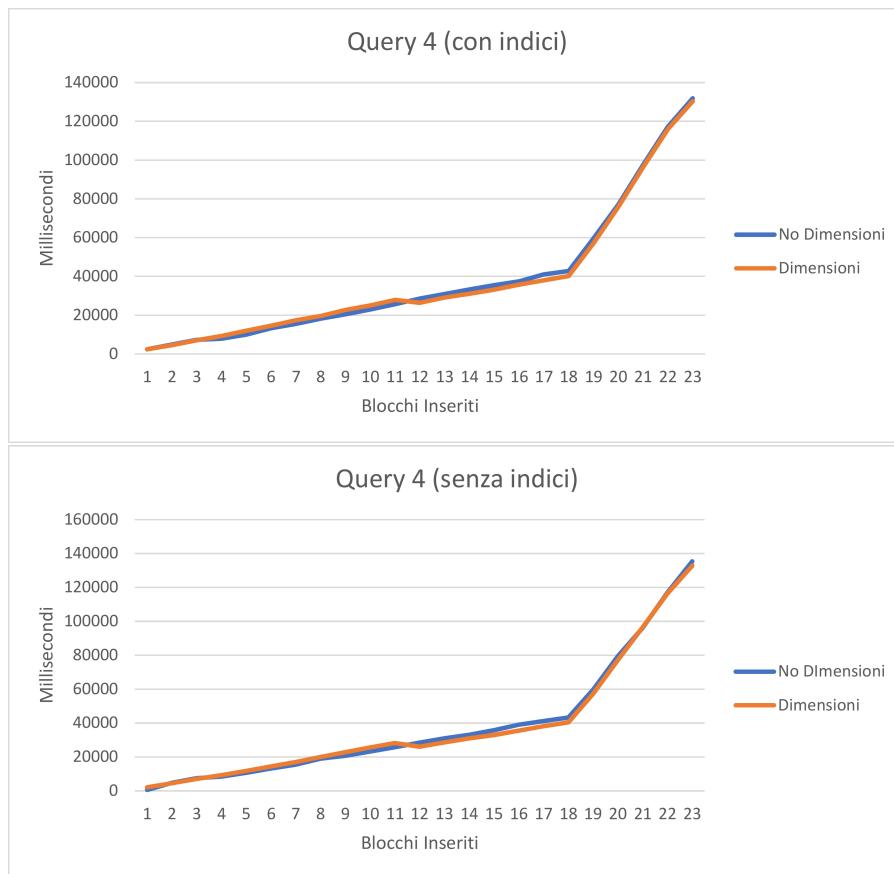


Figura 8: Query 4, Media numero unità inviate per città, emergenza e priorità dichiarata.

Nella query 4 viene effettuata una operazione di media del numero di unità che vengono inviate per ogni chiamata al 911, raggruppate per città da dove arriva la chiamata, tipo di emergenza e priorità dichiarata. Come si evince dai grafici, non si segnalano differenze degne di nota tra le varie strategie.

4.4.7 Query 5

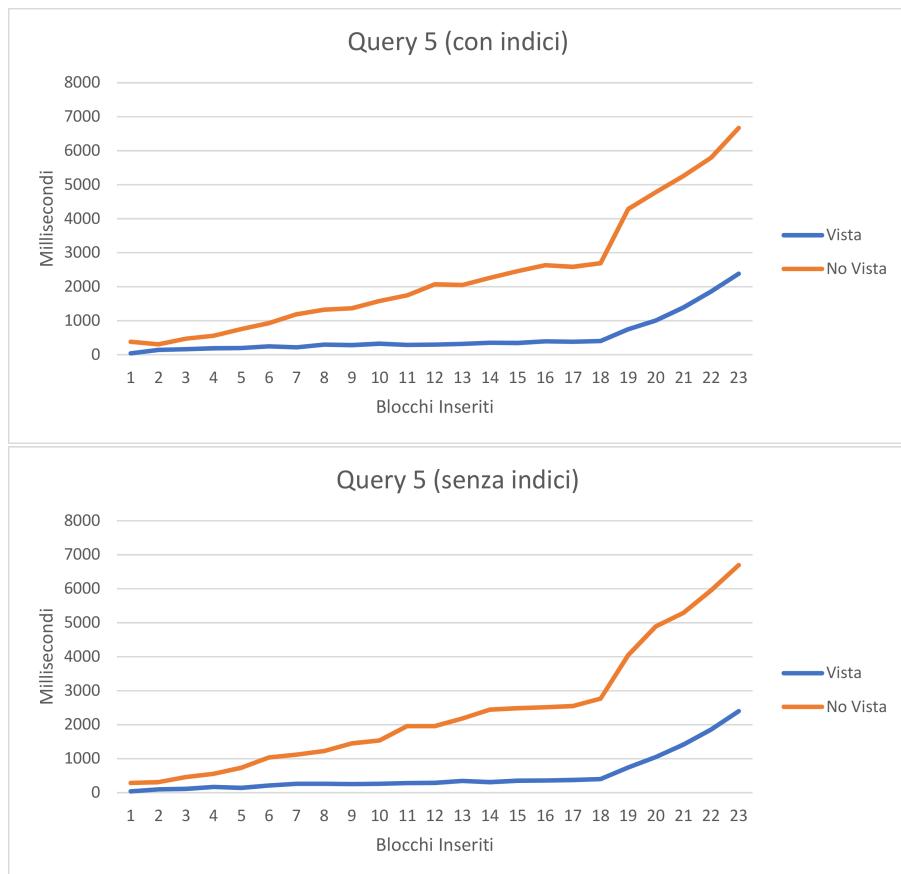


Figura 9: Query 5, Numero di falsi allarmi per tipo emergenza avvenuti di giorno e di notte.

Nella query 5, utilizzando lo schema a stella a fatto singolo, vengono contate le chiamate per cui la priorità finale risulta essere inferiore rispetto a quella dichiarata, raggruppate per tipo di emergenza e per frame temporale in cui questa è avvenuta (giorno o notte). Vengono quindi sfruttate di nuovo le viste materializzate che raggruppano chiamate per giorno e notte. Il guadagno nell'utilizzo di viste materializzate risulta essere tangibile, arrivando a toccare circa 4 secondi negli ultimi inserimenti nel warehouse. In questo caso, se si vuole tenere in considerazione il costo di aggiornamento della vista, il guadagno garantito dell'uso della variante con vista si registra orientativamente dopo 15 esecuzione della o delle query.

4.4.8 Query 6

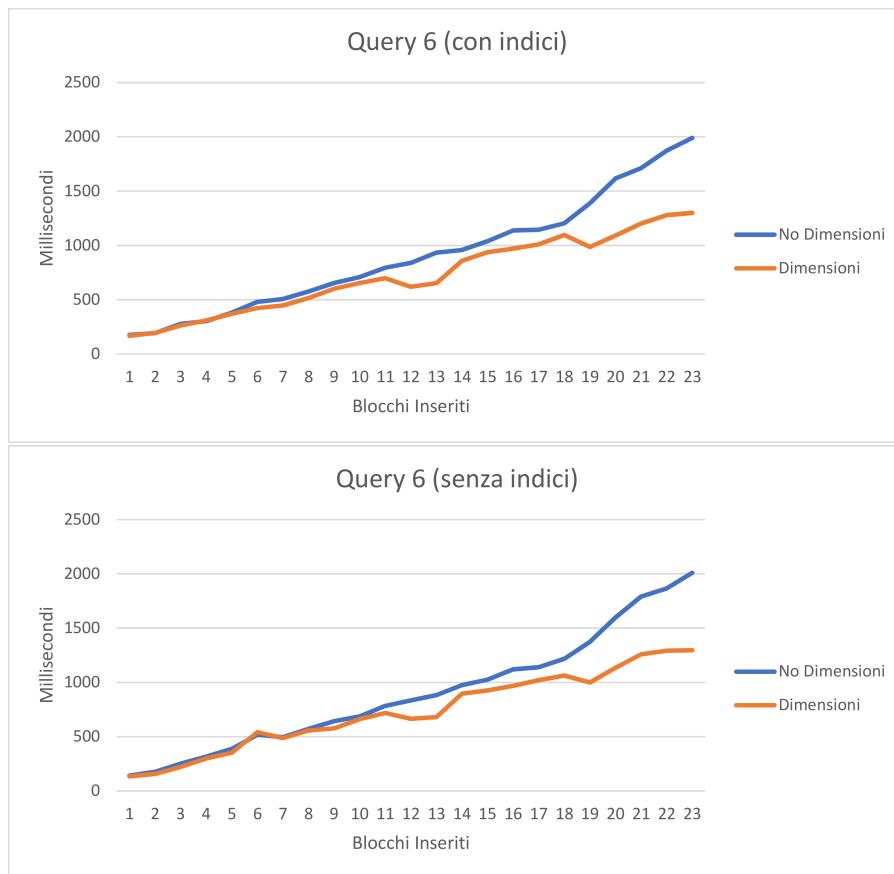


Figura 10: Query 6, Emergenze raggruppate per priorità con durata di intervento inferiore a 5 minuti.

Nella query 6 vengono raccolte le chiamate per cui la durata dell'intervento (intesa come differenza in minuti tra tempo d'arrivo della chiamata e tempo di arrivo dell'unità sul posto) sia inferiore a 5 minuti. Si vuole quindi testare l'efficacia dell'attributo gerarchico costruito sulla dimensione *duration* rispetto al semplice calcolo da effettuare su un campo di tipo DateTime del fatto originale (senza dimensioni). Il grafico mostra un comportamento leggermente migliore dello schema con dimensioni, che si palesa solo al raggiungimento di grande mole di dati. In conclusione per questa query risulta conveniente l'uso delle dimensioni.

4.4.9 Query 7

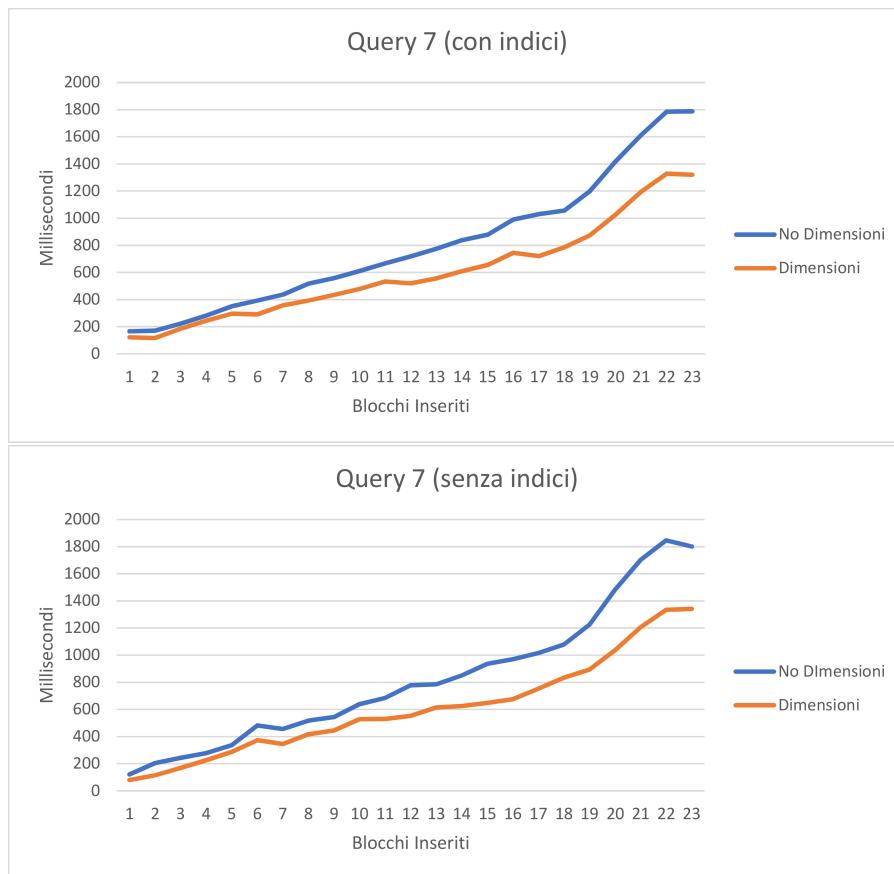


Figura 11: Query 7, Interventi del battaglione B01 per tipo di emergenza, con media durata di intervento e numero interventi.

Nella query 7 vengono testate contemporaneamente le dimensioni *Responsibility*, *call_type* e *duration* che tengono conto rispettivamente dell'area di responsabilità amministrativa, dei tipi d'emergenza, e della durata dell'intervento, ed utilizzando tali raggruppamenti, vengono contati gli interventi di un certo battaglione ('B01'). I grafici mostrano un guadagno nell'utilizzo di schema con dimensioni man mano che il data warehouse viene popolato. In questo caso particolare risulta essere più conveniente, seppur di poco, l'uso delle dimensioni già dai primi inserimenti.

4.4.10 Query 8



Figura 12: Query 8, Per ogni box della città di San Francisco, il numero di chiamate tra gli anni 2015-2017.

Nella query 8, dove vengono contate le chiamate in un intervallo limitato di anni, raggruppate per Box della città di San Francisco, vengono messe a confronto le tre strategie: schema originale, schema a stella con dimensioni e singolo fatto ed infine schema a stella frammentato orizzontalmente per anno. In corrispondenza del 18esimo inserimento è visibile uno spike nei tempi di risposta attribuibile all'entrata nel warehouse dei dati relativi agli anni interessati dalla query. Dai grafici si evince come la strategia migliore sia quella con dimensioni e frammentazione orizzontale, e che la presenza di indici attivi attutisca seppur di poco, i tempi nella strategia con dimensioni. Prima del 14esimo inserimento si registrano tempi molto bassi sia per le query con dimensioni che su tabelle frammentate, mentre all'aumentare dei dati crescono i tempi relativi alla query che non utilizza né dimensioni né frammentazione. In corrispondenza degli inserimenti 14-18 si registrano le prime variazioni dei tempi che in misura diversa coinvolgono tutte le implementazioni. Dopo il 18esimo blocco i tempi si assestano in quanto gli inserimenti successivi non toccano più gli anni interessati dalla query.

4.5 Conclusioni

Al netto delle osservazioni fatte nei paragrafi precedenti relativi alle specifiche query seguono considerazioni conclusive ricavate dall'osservazione dei risultati.

L'utilizzo degli **indici** B-tree così impostati non sembra garantire un vantaggio considerevole se non nella query 1A e 1B dove l'introduzione di questi potrebbe garantire un vantaggio sul lungo termine, ossia quando si prevede di effettuare un numero di query considerevole che vada a coprire i costi di creazione.

L'introduzione delle **dimensioni** nello schema a stella e gli attributi gerarchici così progettati hanno apportato un miglioramento ai tempi d'esecuzione nella maggior parte dei casi, in particolare query come la numero 3 dove viene effettuato un raggruppamento sull'attributo calcolato 'Month' presentano un netto vantaggio.

L'introduzione delle **viste materializzate** ha comportato un vantaggio apprezzabile nell'esecuzione delle query che ne fanno uso, e risulta quindi ragionevole confermarne l'utilità per query che operano su specifici periodi temporali, anche al netto dei tempi di creazione e refresh (a patto che il numero di esecuzioni delle query non sia eccessivamente scarso).

La **frammentazione orizzontale** ha garantito un miglioramento dei tempi in tutti i confronti effettuati e può quindi risultare utile nel caso le query interessino singoli momenti temporali (es: anni specifici), meno in quelle che invece toccano un numero elevato di anni differenti o che non interessino il dominio temporale del tutto, cosa che necessiterebbe di molteplici join tra i vari frammenti relativi agli anni.

Le query sono state eseguite su una macchina con le seguenti caratteristiche.

- Intel i7-8750H
- Ram: 16GB
- Hard Disk Meccanico

5 Appendice

```
1 import psycopg2
2 import csv
3 import codecs
4 import time
5 import datetime
6 import ast
7 from pathlib import Path
8 import random
9 import sys
10 import string
11
12 cntNotValidRows=0
13 cntValidRows=0
14
15 class FragFile:
16     def __init__(self,year,cur):
17         path = 'output/factFrag' + year + '.csv'
18         self.filePath=Path.cwd() / path
19         self.fileDesc=open(self.filePath, 'a', newline=' ')
20         self.postgresTableName='dispatch911_frag_'+year
21         query = 'CREATE TABLE IF NOT EXISTS ' + self.postgresTableName +
22             '(id_received_date integer,id_geo_place
23             integer,id_duration smallint,id_responsibility
24             integer,id_call_type smallint,call_number
25             varchar(20),unit_id varchar(10),incident_number
26             varchar(10),call_date timestamp without time zone,
27             watch_date timestamp without time zone,entry_DtTm
28             timestamp without time zone,dispatch_DtTm timestamp
29             without time zone,response_DtTm timestamp without time
30             zone,on_scene_DtTm timestamp without time
31             zone,transport_DtTm timestamp without time
32             zone,hospital_DtTm timestamp without time
33             zone,call_final_disposition varchar(30),available_DtTm
34             timestamp without time zone,original_priority
35             varchar(1),priority varchar(1),final_priority
36             varchar(1),ALS_unit bool,number_of_alarms
37             smallint,unit_type
38             varchar(20),unit_sequence_in_call_dispatch
39             smallint,fire_prevention_district
40             varchar(10),supervisor_district varchar(20),location_f
41             point,rowid varchar(50))'
42         cur.execute(query)
43
44 class QueryTester:
45     csvQueryResults=None
46     resultsCsvWriter=None
47     queryArray=[]
48     queryIterations = 6
49     csvQueryResultsPath = Path.cwd() / 'output/queryResults.csv'
50
51     def __init__(self):
52         self.csvQueryResults = open(self.csvQueryResultsPath, 'w',
53             newline='')
```

```

33     self.resultsCsvWriter = csv.writer(self.csvQueryResults,
34         ↪ lineterminator='\n', delimiter=';')
35
36     # Query 1A (Original) #1
37     self.queryArray.append("SELECT date_part('year', received_dttm),
38         ↪ neighborhood_district, count(*) FROM dispatch911_original
39         ↪ WHERE call_type='HazMat' GROUP BY date_part('year',
40         ↪ received_dttm), neighborhood_district")
41
42     # Query 1A (Dimensions) #2
43     self.queryArray.append("SELECT dat.year_f,
44         ↪ geo.neighborhoods, count(*) FROM dispatch911_dimensions as
45         ↪ dis INNER JOIN dim_geo_place as geo ON
46         ↪ dis.id_geo_place=geo.id_geo_place INNER JOIN dim_call_type
47         ↪ as callt ON callt.id_call_type=dis.id_call_type INNER JOIN
48         ↪ dim_received_date as dat ON
49         ↪ dat.id_received_date=dis.id_received_date WHERE
50         ↪ callt.call_type='HazMat' GROUP BY dat.year_f,
51         ↪ geo.neighborhoods")
52
53     # Query 1B (Dimensions) #3
54     self.queryArray.append("SELECT geo.neighborhoods, COUNT(
55         ↪ distinct call_number) FROM dispatch911_dimensions as dis
56         ↪ INNER JOIN dim_geo_place as geo ON
57         ↪ dis.id_geo_place=geo.id_geo_place INNER JOIN dim_call_type
58         ↪ as callt ON callt.id_call_type=dis.id_call_type INNER JOIN
59         ↪ dim_received_date as dat ON
60         ↪ dat.id_received_date=dis.id_received_date WHERE
61         ↪ callt.call_type='HazMat' AND dat.year_f in
62         ↪ ('2010','2011','2012','2013','2014','2015') GROUP BY
63         ↪ geo.neighborhoods")
64
65     # Query 1B (Frag) #4
66     self.queryArray.append("SELECT neighborhoods, COUNT(DISTINCT
67         ↪ q1.call_number) FROM(((SELECT * FROM
68         ↪ dispatch911_frag_2010) UNION ALL (SELECT * FROM
69         ↪ dispatch911_frag_2011) UNION ALL (SELECT * FROM
70         ↪ dispatch911_frag_2012) UNION ALL (SELECT * FROM
71         ↪ dispatch911_frag_2013) UNION ALL (SELECT * FROM
72         ↪ dispatch911_frag_2014) UNION ALL (SELECT * FROM
73         ↪ dispatch911_frag_2015) ) as q1 INNER JOIN (SELECT
74         ↪ id_call_type,call_type FROM dim_call_type as calltype
75         ↪ WHERE calltype.call_type='HazMat') as q2 on
76         ↪ q1.id_call_type = q2.id_call_type INNER JOIN(SELECT
77         ↪ id_geo_place,neighborhoods FROM dim_geo_place)as q3 on
78         ↪ q1.id_geo_place = q3.id_geo_place) GROUP BY
79         ↪ q3.neighborhoods ")
80
81     # Query 2 (No vista) #5
82     self.queryArray.append("select dayquery.call_type, (minutes_day),
83         ↪ (minutes_night) from (select call_type, avg(minutes) as
84         ↪ minutes_day from dispatch911_dimensions fact INNER JOIN
85         ↪ dim_duration dur on (fact.id_duration = dur.id_duration)
86         ↪ INNER JOIN dim_call_type as emergency on fact.id_call_type
87         ↪ = emergency.id_call_type INNER JOIN dim_received_date as
88         ↪ recdate on fact.id_received_date =
89         ↪ recdate.id_received_date where recdate.hour_f in
90         ↪ ")

```

```

→ (1,2,3,4,5,6,7,8,9,10,11,12) group by call_type ) AS
→ dayquery inner join (select call_type, avg(minutes) as
→ minutes_night from dispatch911_dimensions fact INNER JOIN
→ dim_duration dur on (fact.id_duration = dur.id_duration)
→ INNER JOIN dim_call_type as emergency on fact.id_call_type
→ = emergency.id_call_type INNER JOIN dim_received_date as
→ recdate on fact.id_received_date =
→ recdate.id_received_date where recdate.hour_f in
→ (13,14,15,16,17,18,19,20,21,22,23,24) group by call_type )
→ AS nightquery on dayquery.call_type =
→ nightquery.call_type")
47 # Query 2 (Vista) #6
48 self.queryArray.append("select dayquery.call_type,
→ dayquery.avgminutes as dayavg, nightquery.avgminutes as
→ nightavg from (select call_type, avg(minutes) as
→ avgminutes from intervention_daytime group by call_type
→ )AS dayquery inner join (select call_type, avg(minutes) as
→ avgminutes from intervention_nighttime group by call_type
→ )AS nightquery on dayquery.call_type =
→ nightquery.call_type")
49
50 # Query 3 (Original) #7
51 self.queryArray.append("(SELECT '1' as
→ season,neighborhood_district, count(*) FROM
→ dispatch911_original WHERE
→ date_part('month',received_dttm) IN (12,1,2) AND
→ call_type_group='Fire' GROUP BY neighborhood_district)
→ union (SELECT '2' as season,neighborhood_district,
→ count(*) FROM dispatch911_original WHERE
→ date_part('month',received_dttm) IN (3,4,5) AND
→ call_type_group='Fire' GROUP BY neighborhood_district)
→ union (SELECT '3' as season,neighborhood_district,
→ count(*) FROM dispatch911_original WHERE
→ date_part('month',received_dttm) IN (6,7,8) AND
→ call_type_group='Fire' GROUP BY neighborhood_district)
→ union (SELECT '4' as season,neighborhood_district,
→ count(*) FROM dispatch911_original WHERE
→ date_part('month',received_dttm) IN (9,10,11) AND
→ call_type_group='Fire' GROUP BY neighborhood_district)
→ order by neighborhood_district, 1")
52 # Query 3 (Dimensions) #8
53 self.queryArray.append("SELECT dat.season, geo.neighborhoods,
→ count(*) FROM dispatch911_dimensions as dis INNER JOIN
→ dim_geo_place as geo ON dis.id_geo_place=geo.id_geo_place
→ INNER JOIN dim_call_type as callt ON
→ callt.id_call_type=dis.id_call_type INNER JOIN
→ dim_received_date as dat ON
→ dat.id_received_date=dis.id_received_date WHERE
→ callt.call_type_group='Fire' GROUP BY dat.season,
→ geo.neighborhoods order by 2,1")
54
55 # Query 4 (Original) #9
56 self.queryArray.append("select
→ q2.city,q2.call_type,q2.original_priority,
→ avg(number_of_unit_dispatched) from (Select call_number,

```

```

    ↵ count(*) as number_of_unit_dispatched from
    ↵ dispatch911_original group by call_number)as q1 inner
    ↵ join (Select distinct call_number, original_priority,
    ↵ call_type,city from dispatch911_original as fact)as q2 on
    ↵ q1.call_number = q2.call_number group by
    ↵ q2.original_priority,q2.call_type,q2.city order by 4")
57   # Query 4 (Dimensions) #10
58   self.queryArray.append("select
    ↵ city,q2.call_type,q2.original_priority,
    ↵ avg(number_of_unit_dispatched) from (Select call_number,
    ↵ count(*) as number_of_unit_dispatched from
    ↵ dispatch911_dimensions group by call_number)as q1 inner
    ↵ join (Select distinct call_number, original_priority,
    ↵ call_type, city from dispatch911_dimensions as fact inner
    ↵ join dim_call_type as emer on fact.id_call_type =
    ↵ emer.id_call_type inner join dim_geo_place as geop on
    ↵ fact.id_geo_place = geop.id_geo_place)as q2 on
    ↵ q1.call_number = q2.call_number group by
    ↵ q2.city,q2.original_priority,q2.call_type order by 4 ")
59
60   # Query 5 (Vista) #11
61   self.queryArray.append("(select dayq.call_type, daycount,
    ↵ nightcount from (select call_type, count(*) as daycount
    ↵ from intervention_daytime where final_priority <
    ↵ original_priority group by call_type) as dayq inner join
    ↵ (select call_type, count(*) as nightcount from
    ↵ intervention_nighttime where final_priority <
    ↵ original_priority group by call_type) nightq on
    ↵ dayq.call_type = nightq.call_type)")
62   # Query 5 (NoVista) #12
63   self.queryArray.append("(select dayq.call_type, daycount,
    ↵ nightcount from (select call_type, count(*) as daycount
    ↵ from dispatch911_dimensions fact INNER JOIN dim_duration
    ↵ dur on (fact.id_duration = dur.id_duration) INNER JOIN
    ↵ dim_call_type as emergency on fact.id_call_type =
    ↵ emergency.id_call_type INNER JOIN dim_received_date as
    ↵ recdate on fact.id_received_date =
    ↵ recdate.id_received_date where recdate.hour_f in
    ↵ (1,2,3,4,5,6,7,8,9,10,11,12) and final_priority <
    ↵ original_priority group by call_type) as dayq inner join
    ↵ (select call_type, count(*) as nightcount from
    ↵ dispatch911_dimensions fact INNER JOIN dim_duration dur on
    ↵ (fact.id_duration = dur.id_duration) INNER JOIN
    ↵ dim_call_type as emergency on fact.id_call_type =
    ↵ emergency.id_call_type INNER JOIN dim_received_date as
    ↵ recdate on fact.id_received_date =
    ↵ recdate.id_received_date where recdate.hour_f in
    ↵ (13,14,15,16,17,18,19,20,21,22,23,24) and final_priority <
    ↵ original_priority group by call_type) nightq on
    ↵ dayq.call_type = nightq.call_type)")
64
65   # Query 6 (Original) #13
66   self.queryArray.append("select fact.original_priority, count(*)
    ↵ from dispatch911_original as fact where
    ↵ fact.durationminutes <= 5 group by fact.original_priority")

```

```

67     # Query 6 (Dimensions) #14
68     self.queryArray.append("select fact.original_priority, count(*)
69         ↪ from dispatch911_dimensions as fact inner join
70             ↪ dim_call_type as emergency on fact.id_call_type =
71                 ↪ emergency.id_call_type inner join dim_duration as dur on
72                     ↪ fact.id_duration = dur.id_duration where dur.lessfive=true
73                         ↪ group by fact.original_priority ")
74
75     # Query 7 (Original) #15
76     self.queryArray.append("select fact.call_type, count(*),
77         ↪ avg(fact.durationminutes) from dispatch911_original as
78             ↪ fact where fact.battalion = 'B01' group by fact.call_type")
79     # Query 7 (Dimension) #16
80     self.queryArray.append("select emergency.call_type, count(*),
81         ↪ avg(dur.minutes) from dispatch911_dimensions as fact inner
82             ↪ join dim_call_type as emergency on fact.id_call_type =
83                 ↪ emergency.id_call_type inner join dim_duration as dur on
84                     ↪ fact.id_duration = dur.id_duration inner join
85                         ↪ dim_responsibility as resp on fact.id_responsibility =
86                             ↪ resp.id_responsibility where resp.battalion = 'B01' group
87                                 ↪ by emergency.call_type")
88
89     # Query 8 (Original) #17
90     self.queryArray.append("SELECT box, COUNT (DISTINCT call_number)
91         ↪ as number_of_calls FROM dispatch911_original as fact WHERE
92             ↪ city='SAN FRANCISCO' and date_part('year', received_dttm)
93                 ↪ in ('2015','2016','2017') GROUP BY box ORDER BY 2 DESC")
94     # Query 8 (Dimension) #18
95     self.queryArray.append("SELECT box, COUNT (DISTINCT call_number)
96         ↪ as number_of_calls FROM ((SELECT * FROM
97             ↪ dispatch911_dimensions as fact) as q1 JOIN (SELECT
98                 ↪ id_geo_place  FROM dim_geo_place as geo WHERE
99                     ↪ geo.city='SAN FRANCISCO') as q2 on q1.id_geo_place =
100                         ↪ q2.id_geo_place JOIN (SELECT redate.id_received_date
101                             ↪ FROM dim_received_date as redate WHERE redate.year_f
102                               ↪ in ('2015','2016','2017') )as q2bis on
103                                 ↪ q1.id_received_date = q2bis.id_received_date ) JOIN (
104                                     ↪ SELECT id_responsibility, box  FROM dim_responsibility as
105                                         ↪ resp ) as q3 on q1.id_responsibility =
106                                             ↪ q3.id_responsibility GROUP BY box ORDER BY 2 DESC")
107     # Query 8 (Frag) #19
108     self.queryArray.append("SELECT box, COUNT (DISTINCT call_number)
109         ↪ as number_of_calls FROM ( ( (SELECT * FROM
110             ↪ dispatch911_frag_2015) UNION ALL (SELECT * FROM
111                 ↪ dispatch911_frag_2016) UNION ALL (SELECT * FROM
112                     ↪ dispatch911_frag_2017) ) as q1 INNER JOIN (SELECT
113                         ↪ id_geo_place  FROM dim_geo_place as geo  WHERE
114                             ↪ geo.city='SAN FRANCISCO') as q2 on q1.id_geo_place =
115                                 ↪ q2.id_geo_place ) INNER JOIN ( SELECT id_responsibility,
116                                     ↪ box  FROM dim_responsibility as resp ) as q3 on
117                                         ↪ q1.id_responsibility = q3.id_responsibility GROUP BY box
118                                             ↪ ORDER BY 2 DESC")
119
120     def computeAndWriteAvgs(self, block, usingIndex):
121         queryIndex=1

```

```

84     for q in self.queryArray:
85         queryTime = 0
86         for i in range(0,self.queryIterations):
87             query_start_time=time.time()
88             cur.execute(q)
89             if (i!=0):
90                 queryTime=queryTime+(time.time()-query_start_time)
91             #print(queryTime/(i+1))
92
93             outResultRow=[block, queryIndex, usingIndex, queryTime /
94                         ↪ (self.queryIterations-1) * 1000]
95             self.resultsCsvWriter.writerow(outResultRow)
96             queryIndex=queryIndex+1
97             print(outResultRow)
98
99     def createIndex(self):
100        cur.execute("CREATE INDEX dispatch_date_ind on
101                      ↪ dispatch911_dimensions (id_received_date);")
102        cur.execute("CREATE INDEX dispatch_date_geo on
103                      ↪ dispatch911_dimensions (id_geo_place);")
104        cur.execute("CREATE INDEX dispatch_date_dur on
105                      ↪ dispatch911_dimensions (id_duration);")
106        cur.execute("CREATE INDEX dispatch_date_resp on
107                      ↪ dispatch911_dimensions (id_responsibility);")
108        cur.execute("CREATE INDEX dispatch_date_calltype on
109                      ↪ dispatch911_dimensions (id_call_type);")
110
111    def dropIndex(self):
112        cur.execute("DROP INDEX dispatch_date_ind;")
113        cur.execute("DROP INDEX dispatch_date_geo;")
114        cur.execute("DROP INDEX dispatch_date_dur;")
115        cur.execute("DROP INDEX dispatch_date_resp;")
116        cur.execute("DROP INDEX dispatch_date_calltype;")
117
118    def createTables(cur,conn):
119        cur.execute("CREATE TABLE IF NOT EXISTS
120                      ↪ dim_received_date(id_received_date integer NOT
121                      ↪ NULL,received_DtM timestamp without time zone, hour_f
122                      ↪ smallint, day_f smallint, month_f smallint, year_f smallint,
123                      ↪ season smallint)")
124        cur.execute("CREATE TABLE IF NOT EXISTS dim_duration (id_duration
125                      ↪ smallint NOT NULL,minutes smallint NOT NULL,lessFive boolean
126                      ↪ NOT NULL DEFAULT '0',lessFifteen boolean NOT NULL DEFAULT
127                      ↪ '0',lessTwentyfive boolean NOT NULL DEFAULT '0',moreTwentyfive
128                      ↪ boolean NOT NULL DEFAULT '0')")
129        cur.execute("CREATE TABLE IF NOT EXISTS dim_geo_place(id_geo_place
130                      ↪ Integer NOT NULL,address varchar(100),city varchar(50),zipcode
131                      ↪ integer,Neighborhoods varchar(50))")
132        cur.execute("CREATE TABLE IF NOT EXISTS dim_responsibility
133                      ↪ (id_responsibility integer, box
134                      ↪ varchar,station_area varchar, battalion varchar(5))")
135        cur.execute("CREATE TABLE IF NOT EXISTS dim_call_type(id_call_type
136                      ↪ smallint, call_type enum_call_type, call_type_group
137                      ↪ enum_call_type_group)")
138        cur.execute("CREATE TABLE IF NOT EXISTS

```

```

→ dispatch911_original(call_number varchar(20),unit_id
→ varchar(10),incident_number varchar(10),call_type
→ varchar(50),call_date timestamp without time zone, watch_date
→ timestamp without time zone,received_DtTm timestamp without
→ time zone,entry_DtTm timestamp without time zone,dispatch_DtTm
→ timestamp without time zone,response_DtTm timestamp without
→ time zone,on_scene_DtTm timestamp without time
→ zone,transport_DtTm timestamp without time zone,hospital_DtTm
→ timestamp without time zone,call_final_disposition
→ varchar(30),available_DtTm timestamp without time zone,address
→ varchar(50),city varchar(30),zipcode_of_incident
→ varchar(10),battalion varchar(10),station_area varchar(20),box
→ varchar(10),original_priority varchar(1),priority
→ varchar(1),final_priority varchar(1),ALS_unit
→ bool,call_type_group varchar(35),number_of_alarms
→ smallint,unit_type varchar(20),unit_sequence_in_call_dispatch
→ smallint,fire_prevention_district
→ varchar(10),supervisor_district
→ varchar(20),neighborhood_district varchar(50),location_f
→ varchar(50),rowid varchar(50),durationMinutes smallint)")

119 cur.execute("CREATE TABLE IF NOT EXISTS
→ dispatch911_dimensions(id_received_date integer,id_geo_place
→ integer,id_duration smallint,id_responsibility
→ integer,id_call_type smallint,call_number varchar(20),unit_id
→ varchar(10),incident_number varchar(10),call_date timestamp
→ without time zone, watch_date timestamp without time
→ zone,entry_DtTm timestamp without time zone,dispatch_DtTm
→ timestamp without time zone,response_DtTm timestamp without
→ time zone,on_scene_DtTm timestamp without time
→ zone,transport_DtTm timestamp without time zone,hospital_DtTm
→ timestamp without time zone,call_final_disposition
→ varchar(30),available_DtTm timestamp without time
→ zone,original_priority varchar(1),priority
→ varchar(1),final_priority varchar(1),ALS_unit
→ bool,number_of_alarms smallint,unit_type
→ varchar(20),unit_sequence_in_call_dispatch
→ smallint,fire_prevention_district
→ varchar(10),supervisor_district varchar(20),location_f
→ point,rowid varchar(50))")

120 cur.execute("CREATE MATERIALIZED VIEW IF NOT EXISTS
→ intervention_daytime AS ((select rowid, minutes, call_type,
→ original_priority, final_priority from dispatch911_dimensions
→ fact INNER JOIN dim_duration dur on (fact.id_duration =
→ dur.id_duration)INNER JOIN dim_call_type as emergency on
→ fact.id_call_type = emergency.id_call_type INNER JOIN
→ dim_received_date as redate on fact.id_received_date =
→ redate.id_received_date where redate.hour_f in
→ (5,6,7,8,9,10,11,12,13,14,15,16)))")

121 cur.execute("CREATE MATERIALIZED VIEW IF NOT EXISTS
→ intervention_nighttime AS ((select rowid, minutes, call_type,
→ original_priority, final_priority from dispatch911_dimensions
→ fact INNER JOIN dim_duration dur on (fact.id_duration =
→ dur.id_duration) INNER JOIN dim_call_type as emergency on
→ fact.id_call_type = emergency.id_call_type INNER JOIN
→ dim_received_date as redate on fact.id_received_date =

```

```
122     ↪ redate.id_received_date where redate.hour_f in  
123     ↪ (17,18,19,20,21,22,23,24,1,2,3,4)))")  
124  
125     conn.commit()  
126  
127 def putDurationTableInDictionary(dict):  
128     cur.execute("SELECT dim_duration.id_duration, dim_duration.minutes  
129     ↪ FROM dim_duration")  
130     queryRes=cur.fetchall()  
131     for k in queryRes:  
132         dict[k[1]]=k[0]  
133     if len(queryRes)>0:  
134         return queryRes[len(queryRes)-1][0]  
135     else:  
136         return 0  
137  
138  
139 def putGeoPlaceTableInDictionary(dictLoc):  
140     cur.execute("SELECT * FROM dim_geo_place")  
141     queryRes=cur.fetchall()  
142  
143     for k in queryRes:  
144         geoPlaceString=k[1]+"@"+k[2]+"@"+str(k[3])+"@"+k[4]  
145         dictLoc[geoPlaceString]=k[0]  
146     if len(queryRes)>0:  
147         return queryRes[len(queryRes)-1][0]  
148     else:  
149         return 0  
150  
151 def putDateTableInDictionary(dict):  
152     cur.execute("SELECT id_received_date, received_DtTm FROM  
153     ↪ dim_received_date")  
154     queryRes=cur.fetchall()  
155  
156     for k,j in queryRes:  
157         dict[j.strftime("%Y-%m-%dT%H:%M:%S")]=k  
158  
159     if len(queryRes) > 0:  
160         return queryRes[len(queryRes) - 1][0]  
161     else:  
162         return 0  
163  
164 def putResponsibilityTableInDictionary(dictResp):  
165     cur.execute("SELECT * FROM dim_responsibility")  
166     queryRes=cur.fetchall()  
167     for k in queryRes:  
168         responsString=k[1]+"@"+k[2]+"@"+(k[3])  
169         dictResp[responsString]=k[0]  
170     if len(queryRes)>0:  
171         return queryRes[len(queryRes)-1][0]  
172     else:  
173         return 0  
174  
175 def putCallTypeTableInDictionary(dictCallType):  
176     cur.execute("SELECT * FROM dim_call_type")
```

```
173     queryRes=cur.fetchall()
174     for k in queryRes:
175         calltypeString=k[1]+"@"+k[2]
176         dictCallType[calltypeString]=k[0]
177     if len(queryRes)>0:
178         return queryRes[len(queryRes)-1][0]
179     else:
180         return 0
181
182 def getDimensionDurationRow(duration, tempTableDurata):
183     res=tempTableDurata.get(duration)
184     if res is None:
185         tempTableDurata[duration] = len(tempTableDurata)
186         return len(tempTableDurata)-1
187     else:
188         return res
189
190 def getDimensionDateRow(recDate,tempTableDate):
191     res=tempTableDate.get(recDate)
192     if res is None:
193         tempTableDate[recDate]=len(tempTableDate)
194         return len(tempTableDate)-1
195     else:
196         return res
197
198 def getDimensionGeoPlaceRow(address, city, zipcode, neigh,
199     ↪ tempTableGeoPlace):
200     dimensionString=address + "@" + city + "@" + zipcode + "@" + neigh
201     id=tempTableGeoPlace.get(dimensionString)
202     if (id is None):
203         tempTableGeoPlace[dimensionString] = len(tempTableGeoPlace)
204         return len(tempTableGeoPlace) - 1
205     else:
206         return id
207
208 def ↪ getDimensionResponsibilityRow(box,station_area,battalion,tempTableResponsibility):
209     responsibility=box+"@"+station_area+"@"+battalion
210     id = tempTableResponsibility.get(responsibility)
211     if (id is None):
212         tempTableResponsibility[responsibility] =
213             ↪ len(tempTableResponsibility)
214         return len(tempTableResponsibility)-1
215     else:
216         return id
217
218 def getDimensionCallTypeRow(call_type,call_type_group,tempTableCallType):
219     calltype=call_type+"@"+call_type_group
220     id = tempTableCallType.get(calltype)
221     if (id is None):
222         tempTableCallType[calltype]=len(tempTableCallType)
223         return len(tempTableCallType)-1
224     else:
225         return id
```

```
225 #Convert unknown priority values to known ones (2 non-emergency,3
226 # ↪ emergency)
227 def mapPriority(priority):
228     # Priority levels:
229     # A,B,C | 2 (driving without lights/sirens)
230     # D,E | 3 (driving with lights/sirens)
231     # I | 1
232     if (priority=="A") or (priority=="B" or (priority=="C") or
233         ↪ (priority=="2")):
234         return 2
235     elif (priority=="D") or (priority=="E") or (priority=="3"):
236         return 3
237     else: #(priority=="1") or (priority=="I"):
238         return 1
239
240 def rowManipulation(row,cur):
241     call_number=row[0]
242     unit_id=row[1]
243     incident_number=row[2]
244     call_type=row[3]
245     call_date=row[5]
246     watch_date=row[4]
247     received_dtTm=row[6]
248     entry_dtTm=row[7]
249     dispatch_dtTm=row[8]
250     response_dtTm=row[9]
251     on_scene_dtTm=row[10]
252     transport_dtTm=row[11]
253     hospital_dtTm=row[12]
254     call_final_disposition=row[13]
255     available_dtTm=row[14]
256     address=row[15].replace(' ','_')
257     city=row[16]
258     zipcode=row[17]
259     battalion=row[18]
260     station_area=row[19]
261     box=row[20]
262     origPriorityMapped=mapPriority(row[21])
263     callPriorityMapped=mapPriority(row[22])
264     finalPriorityMapped=mapPriority(row[23])
265     als_unit=row[24]
266     call_type_group=row[25]
267     number_of_alarms=row[26]
268     unit_type=row[27]
269     unit_sequence_call_dispatch=row[28]
270     fire_prevention_district=row[29]
271     supervisor_district=row[30]
272     neighborhood=row[31]
273     location=row[32]
274     rowid=row[33]
275
276     # Evaluate intervention duration
277     d1 = datetime.datetime.strptime(received_dtTm, "%Y-%m-%dT%H:%M:%S")
278     d2 = datetime.datetime.strptime(on_scene_dtTm, "%Y-%m-%dT%H:%M:%S")
279     durationInMinutes = d2-d1
```

```
278     durationInMinutes =(int(durationInMinutes.seconds/60))
279
280     call_type=call_type.replace(","," ")
281
282     # als_unit (bool) in postgres
283     if als_unit=='True':
284         als_unit=1
285     else:
286         als_unit=0
287
288     if(number_of_alarms is not 'None'):
289         number_of_alarms=int(number_of_alarms)
290     if (unit_sequence_call_dispatch is not 'None'):
291         unit_sequence_call_dispatch=int(unit_sequence_call_dispatch)
292
293     lat_lon='None'
294     dictTest=ast.literal_eval(location)
295     if(dictTest is not None):
296         longitude=dictTest.get('longitude')
297         latitude=dictTest.get('latitude')
298         lat_lon='('+latitude+","+longitude+')'
299
300     manRow=(call_number, #0
301             unit_id, #1
302             incident_number, #2
303             call_type, #3
304             call_date, #4
305             watch_date, #5
306             received_dtTm, #6
307             entry_dtTm, #7
308             dispatch_dtTm, #8
309             response_dtTm, #9
310             on_scene_dtTm, #10
311             transport_dtTm, #11
312             hospital_dtTm, #12
313             call_final_disposition, #13
314             available_dtTm, #14
315             address, #15
316             city, #16
317             zipcode, #17
318             battalion, #18
319             station_area, #19
320             box, #20
321             origPriorityMapped, #21
322             callPriorityMapped, #22
323             finalPriorityMapped, #23
324             als_unit, #24
325             call_type_group, #25
326             number_of_alarms, #26
327             unit_type, #27
328             unit_sequence_call_dispatch, #28
329             fire_prevention_district, #29
330             supervisor_district, #30
331             neighborhood, #31
332             lat_lon, #32
```

```
333         rowid, #33
334         durationInMinutes) #34
335
336         # ++++++
337         # +++ Not logically linked to row manipulation +++
338         dt = datetime.datetime.strptime(manRow[6], "%Y-%m-%dT%H:%M:%S")
339         if fragTablesPath.get(dt.year) is None:
340             newFragFilePath(dt.year, fragTablesPath, cur)
341         # -----
342         # ++++++
343
344     return manRow
345
345 def randomStr(size=6, chars=string.ascii_uppercase +
346     ↪ string.digits+string.ascii_lowercase):
346     return ''.join(random.choice(chars) for x in range(size))
347
348 def generateConsistentFakeRows(tableDurata, tableGeoPlace, tableDate,
349     ↪ tableResponsibility, tableCallType, numRows=100):
350     fakeStr = 'FAKE'
350     legalPriorities = [2, 3]
351
352     if (not tableGeoPlace or not tableResponsibility):
353         print("empty dimensions, fake rows will not be inserted")
354
355
356     outputFakeRows = Path.cwd() / 'datasource/fakeRows.csv'
357
358     f = codecs.open(outputFakeRows, 'w', encoding='utf-16-le')
359     writer = csv.writer(f, lineterminator='\n', delimiter=',')
360
361     for i in range(numRows):
362         fakeRow = [None] * 35
363         fakeRow[0] = fakeStr + randomStr(15) # call_number varchar(20)
364         fakeRow[1] = fakeStr + randomStr(5) # unit_id varchar(10)
365         fakeRow[21] = random.choice(legalPriorities)
366         fakeRow[22] = random.choice(legalPriorities)
367         fakeRow[23] = random.choice(legalPriorities)
368         fakeRow[25] = 'Alarm'
369         fakeRow[3] = 'Other'
370
371         geoTuple=random.choice(list(tableGeoPlace.keys()))
372         geoFields = geoTuple.split("@")
373         fakeRow[31]=geoFields[3]
374         fakeRow[15]=geoFields[0]
375         fakeRow[16] = geoFields[1]
376         fakeRow[17] = geoFields[2]
377
378         respTuple=random.choice(list(tableResponsibility.keys()))
379         respFields= respTuple.split("@")
380         fakeRow[19] = respFields[1]
381         fakeRow[18] = respFields[2]
382         fakeRow[20] = respFields[0]
383
384         writer.writerow(fakeRow)
385     f.close()
```

```
386
387 def createCallTypeDictionary():
388     dictCallType={ 'Administrative':'Fire',
389                   'Aircraft Emergency':'Alarm',
390                   'Alarms':'Alarm',
391                   'Assist Police':'Alarm',
392                   'Citizen Assist / Service Call':'Alarm',
393                   'Confined Space / Structure Collapse':'Fire',
394                   'Electrical Hazard':'Alarm',
395                   'Elevator / Escalator Rescue':'Alarm',
396                   'Explosion':'Fire',
397                   'Extrication / Entrapped (Machinery Vehicle)':'Fire',
398                   'Fuel Spill':'Alarm',
399                   'Gas Leak (Natural and LP Gases)':'Alarm',
400                   'HazMat':'Alarm',
401                   'HazMat':'Fire',
402                   'High Angle Rescue':'Fire',
403                   'Industrial Accidents':'Fire',
404                   'Marine Fire':'Fire',
405                   'Medical Incident':'Alarm',
406                   'Medical Incident':'Non Life-threatening',
407                   'Medical Incident':'Potentially Life-Threatening',
408                   'Mutual Aid / Assist Outside Agency':'Fire',
409                   'Odor (Strange / Unknown)':'Alarm',
410                   'Odor (Strange / Unknown)':'Fire',
411                   'Oil Spill':'Alarm','Other':'Alarm',
412                   'Other':'Non Life-threatening',
413                   'Other':'Potentially Life-Threatening',
414                   'Outside Fire':'Alarm',
415                   'Outside Fire':'Fire',
416                   'Smoke Investigation (Outside)':'Alarm',
417                   'Structure Fire':'Alarm',
418                   'Structure Fire':'Fire',
419                   'Structure Fire':'Potentially Life-Threatening',
420                   'Suspicious Package':'Fire',
421                   'Traffic Collision':'Non Life-threatening',
422                   'Traffic Collision':'Potentially Life-Threatening',
423                   'Train / Rail Fire':'Fire',
424                   'Train / Rail Incident':'Fire',
425                   'Vehicle Fire':'Alarm',
426                   'Vehicle Fire':'Fire',
427                   'Water Rescue':'Fire',
428                   'Water Rescue':'Potentially Life-Threatening',
429                   'Watercraft in Distress':'Alarm',
430                   'Watercraft in Distress':'Fire',
431                   'Extrication / Entrapped (Machinery, Vehicle)':'Alarm'}
432
433     return dictCallType;
434
435 def cityValidation(cityName):
436     cityName = cityName.upper()
437
438     if cityName == "TI":
439         return "TREASURE"
440     elif cityName == "BN":
```

```
441     return "BRISBANE"
442 elif cityName == "DC":
443     return "DALY CITY"
444 elif cityName == "FM":
445     return "FORT MASON"
446 elif cityName == "HP":
447     return "HUNTERS POINT"
448 elif cityName == "PR":
449     return "PRESIDIO"
450 elif cityName == "SF":
451     return "SAN FRANCISCO"
452 elif cityName == "YB":
453     return "YERBA BUENA"
454 elif cityName == "TI":
455     return "TREASURE ISLAND"
456 elif cityName == "TREASURE ISLA":
457     return "TREASURE ISLAND"
458 return cityName;
459
460 def rowValidation(row,dictCallType):
461
462     if row[25] == '' or row[25] == 'None' or row[25] is None:
463         if(dictCallType.get(row[3] is not None)):
464             row[25] = dictCallType.get(row[3])
465         else:
466             row[25]= 'NotAssigned'
467     if row[31] == '' or row[31] == 'None' or row[31] is None:
468         return False
469     if row[21]=="": #priority1
470         return False
471     if row[22]=="": #priority2
472         return False
473     if row[23]=="": #priority3
474         return False
475     if row[6]=="": #call_date
476         yearRandom=random.randint(2020,2040)
477         row[6]=
478             ↪ datetime.datetime.strftime(datetime.datetime.now().replace(year=yearRandom),
479             ↪ "%Y-%m-%dT%H:%M:%S")
480     if row[10]=="" or (row[10]==row[6] and row[6!=""] )or row[10]<row[6]:
481         row[10]=row[6]
482         onSiteDate=datetime.datetime.strptime(row[10],
483             ↪ "%Y-%m-%dT%H:%M:%S")
484         minutesOffset=random.randint(10,40)
485         onSiteDate=onSiteDate+datetime.timedelta(minutes=minutesOffset)
486         row[10]=datetime.datetime.strftime(onSiteDate,
487             ↪ "%Y-%m-%dT%H:%M:%S")
488     if row[15]== "": #address
489         return False
490     if row[16]=="": #city
491         return False
492     else:
493         row[16]=cityValidation(row[16])
494     if row[17]=="": #zipcode
495         return False
```

```

492     if row[19]=="": #station area
493         return False
494     if row[18] == "": #battalion
495         return False
496     if row[20] == "": #box
497         return False
498
499     for i in range(len(row)):
500         if row[i] == "":
501             row[i] = 'None'
502     return True
503
504 def exportDimensionDurataToCsv(dict, path, lastID):
505     with open(path, 'w',newline='\n') as fl:
506         for v,k in dict.items():
507             if k> lastID or lastID==0:
508                 dimRow = [k,v, 0, 0, 0, 0]
509                 if (v>=25):
510                     dimRow[5] = 1
511                 if (v<=5):
512                     dimRow[2] = 1
513                 if (v<=15):
514                     dimRow[3] = 1
515                 if (v<=25):
516                     dimRow[4] = 1
517
518                 fl.write(repr(dimRow[0]) + ";" + repr(dimRow[1]) + ";" +
519                         repr(dimRow[2]) + ";" + repr(dimRow[3]) + ";" +
520                         repr(dimRow[4]) + ";" + repr(dimRow[5]) +"\n")
521         fl.close()
522
523 def exportDimensionDateToCsv(dict,path,lastID):
524     with open(path,'w',newline='\n') as fl:
525         for k,v in dict.items():
526             if v> lastID or lastID==0:
527                 dt=datetime.datetime.strptime(k,"%Y-%m-%dT%H:%M:%S")
528                 if (dt.month==12) or (dt.month==1) or (dt.month==2):
529                     season=1
530                 elif (dt.month==3) or (dt.month==4) or (dt.month==5):
531                     season=2
532                 elif (dt.month==6) or (dt.month==7) or (dt.month==8):
533                     season=3
534                 elif (dt.month==9) or (dt.month==10) or (dt.month==11):
535                     season=4
536                 fl.write(repr(v) + ";" + k + ";" + repr(dt.hour)+ ";" +
537                         repr(dt.day)+ ";" + repr(dt.month)+ ";" +
538                         repr(dt.year)+ ";" + repr(season)+ "\n")
539         fl.close()
540
541 def exportDimensionGeoPlaceToCsv(dict, path, lastID):
542     with open(path, 'w', newline='\n') as fl:
543         for k, v in dict.items():
544             if v > lastID or lastID==0:
545                 fieldsList=k.split("@")
546                 fl.write(repr(v) + ";" + fieldsList[0] + ";" +

```

```

543     ↪ fieldsList[1] + ";" + fieldsList[2] + ";" +
544     ↪ fieldsList[3] + "\n")
545 fl.close()
546
547 def exportDimensionResponsibilityToCsv(dict,path,lastID):
548     with open(path, 'w', newline=') as fl:
549         for k, v in dict.items():
550             if v > lastID or lastID==0:
551                 fieldsList = k.split("@")
552                 fl.write(repr(v) + ";" + fieldsList[0] + ";" +
553                         ↪ fieldsList[1] + ";" + fieldsList[2]+ "\n")
554 fl.close()
555
556 def exportDimensionCallTypeToCsv(dict,path,lastID):
557     with open(path, 'w', newline=') as fl:
558         for k, v in dict.items():
559             if v > lastID or lastID==0:
560                 fieldList=k.split("@")
561                 fl.write(repr(v)+";"+fieldList[0]+";"+fieldList[1]+"\n")
562 fl.close()
563
564
565 def exportFactOriginalToCsv(f, manRow):
566     writer = csv.writer(f,lineterminator='\n', delimiter=';')
567     writer.writerow(manRow)
568
569 def exportFactDimToCsv(f, manRow, idDuration, idDate, idGeoPlace,
570     ↪ idResponsibility, idCallType):
571     stw =
572         ↪ [(idDate),(idGeoPlace),(idDuration),(idResponsibility),(idCallType),manRow[0],(manRow[1]),(ma
573         ↪ (manRow[9]), (manRow[10]), (manRow[11]), (manRow[12]),
574         ↪ (manRow[13]), (manRow[14]), (manRow[21]), (manRow[22]),
575         ↪ (manRow[23]), (manRow[24]), (manRow[26]), (manRow[27]),
576         ↪ (manRow[28]), (manRow[29]), (manRow[30]), (manRow[32]),
577         ↪ (manRow[33])]
578
579     writer = csv.writer(f,lineterminator='\n', delimiter=';')
580     writer.writerow(stw)
581
582 def csvToPostgres(csvPath,tablename,cur,conn):
583     with open(csvPath, 'r') as f:
584         try:
585             cur.copy_from(f, tablename, sep=';',null='None')
586         except psycopg2.OperationalError as e:
587             print(e)
588
589 def newFragFilePath(year,fragTablesPath,cur):
590     fragFile=FragFile(repr(year),cur)
591     fragTablesPath[year]= fragFile
592
593 def exportFactToFragCSV(fragTablesPath ,manRow, idDuration, idDate,
594     ↪ idGeoPlace, idResponsibility, idCallType):
595     dt=datetime.datetime.strptime(manRow[6],"%Y-%m-%dT%H:%M:%S")
596     stw =
597         ↪ [(idDate),(idGeoPlace),(idDuration),(idResponsibility),(idCallType),manRow[0],(manRow[1]),(ma

```

```

586     ↪ (manRow[9]) , (manRow[10]) , (manRow[11]) , (manRow[12]) ,
587     ↪ (manRow[13]) , (manRow[14]) , (manRow[21]) , (manRow[22]) ,
588     ↪ (manRow[23]) , (manRow[24]) , (manRow[26]) , (manRow[27]) ,
589     ↪ (manRow[28]) , (manRow[29]) , (manRow[30]) , (manRow[32]) ,
590     ↪ (manRow[33]))]
591
592 writer = csv.writer(fragTablesPath.get(dt.year).fileDesc,
593                     ↪ lineterminator='\n', delimiter=';')
594 writer.writerow(stw)
595
596
597
598 def closeFragmentationFiles (fragTablesPath):
599     for year, fragFile in fragTablesPath.items():
600         fragFile.fileDesc.close()
601
602
603 def openFragmentationFiles (fragTablesPath):
604     for year, fragFile in fragTablesPath.items():
605         fragFile.fileDesc = open(fragFile.filePath, 'w', newline='')
606
607
608 postgresConnectionString = "dbname=test user=postgres password=1234
609     ↪ host=localhost"
610
611
612 inputCsvPath1 = Path.cwd() /
613     ↪ 'datasource/01-fire-department-calls-for-service.csv'
614 inputCsvPath2 = Path.cwd() /
615     ↪ 'datasource/02-fire-department-calls-for-service.csv'
616 inputCsvPath3 = Path.cwd() /
617     ↪ 'datasource/03-fire-department-calls-for-service.csv'
618 inputCsvPath4 = Path.cwd() /
619     ↪ 'datasource/04-fire-department-calls-for-service.csv'
620 inputCsvPath5 = Path.cwd() /
621     ↪ 'datasource/05-fire-department-calls-for-service.csv'
622
623
624 inputCsvPath6 = Path.cwd() /
625     ↪ 'datasource/06-fire-department-calls-for-service.csv'
626 inputCsvPath7 = Path.cwd() /
627     ↪ 'datasource/07-fire-department-calls-for-service.csv'
628 inputCsvPath8 = Path.cwd() /
629     ↪ 'datasource/08-fire-department-calls-for-service.csv'
630 inputCsvPath9 = Path.cwd() /
631     ↪ 'datasource/09-fire-department-calls-for-service.csv'
632 inputCsvPath10 = Path.cwd() /
633     ↪ 'datasource/10-fire-department-calls-for-service.csv'
634
635
636 inputCsvPath11 = Path.cwd() /
637     ↪ 'datasource/11-fire-department-calls-for-service.csv'
638 inputCsvPath12 = Path.cwd() /
639     ↪ 'datasource/12-fire-department-calls-for-service.csv'
640 inputCsvPath13 = Path.cwd() /
641     ↪ 'datasource/13-fire-department-calls-for-service.csv'
642 inputCsvPath14 = Path.cwd() /
643     ↪ 'datasource/14-fire-department-calls-for-service.csv'
644 inputCsvPath15 = Path.cwd() /
645     ↪ 'datasource/15-fire-department-calls-for-service.csv'
646
647
648 inputCsvPath16 = Path.cwd() /

```

```
    ↵ 'datasource/16-fire-department-calls-for-service.csv'
619 inputCsvPath17 = Path.cwd() /
    ↵ 'datasource/17-fire-department-calls-for-service.csv'
620 inputCsvPath18 = Path.cwd() /
    ↵ 'datasource/18-fire-department-calls-for-service.csv'
621 inputCsvPath19 = Path.cwd() /
    ↵ 'datasource/19-fire-department-calls-for-service.csv'

622
623 inputCsvPathFAKE = Path.cwd() / 'datasource/fakeRows.csv'
624 inputCsvPathTEST = Path.cwd() / 'datasource/testPython.csv'
625
626 inputList = []
627 inputList.append(inputCsvPath1)
628 inputList.append(inputCsvPath2)
629 inputList.append(inputCsvPath3)
630 inputList.append(inputCsvPath4)
631 inputList.append(inputCsvPath5)
632 inputList.append(inputCsvPath6)
633 inputList.append(inputCsvPath7)
634 inputList.append(inputCsvPath8)
635 inputList.append(inputCsvPath9)
636 inputList.append(inputCsvPath10)
637 inputList.append(inputCsvPath11)
638 inputList.append(inputCsvPath12)
639 inputList.append(inputCsvPath13)
640 inputList.append(inputCsvPath14)
641 inputList.append(inputCsvPath15)
642 inputList.append(inputCsvPath16)
643 inputList.append(inputCsvPath17)
644 inputList.append(inputCsvPath18)
645 inputList.append(inputCsvPathFAKE)
646 inputList.append(inputCsvPathFAKE)
647 inputList.append(inputCsvPathFAKE)
648 inputList.append(inputCsvPathFAKE)
649 inputList.append(inputCsvPathFAKE)

650
651 dimDurationCSVPath = Path.cwd() / 'output/dim_duration.csv'
652 dimDateCSVPath= Path.cwd() / 'output/dim_date.csv'
653 dimGeoPlaceCSVPath= Path.cwd() / 'output/dim_geo_place.csv'
654 dimResponsibilityCSVPath= Path.cwd() / 'output/dim_responsibility.csv'
655 dimCallTypeCSVPath= Path.cwd() / 'output/dim_call_type.csv'
656 factOriginal_csvPATH = Path.cwd() / 'output/factOriginal.csv'
657 factDimensions_csvPATH = Path.cwd() / 'output/factDimensions.csv'
658 csvTimeResultsPath = Path.cwd() / 'output/timeResults.csv'

659
660 clockTimeExtraction=0
661 clockTimeTransformation=0
662 clockTimeLoading=0
663 clockTimeMatView=0
664 clockTimeOther=0
665 clockTimeIndex=0

666
667 elapsedTimeExtraction=0
668 elapsedTimeTransformation=0
669 elapsedTimeLoading=0
```

```
670 elapsedTimeMatView=0
671 elapsedTimeOther=0
672 elapsedTimeIndex=0
673
674 callTypeGroupDictionary={
675     0:'Fire',
676     1:'Potentially Life-Threatening',
677     2:'Non Life-threatening',
678     3:'Alarm',
679     4:'NotAssigned'
680 }
681
682 tempTableDurata={}
683 tempTableGeoPlace={}
684 tempTableDate={}
685 tempTableResponsibility={}
686 tempTableCallType={}
687 fragTablesPath={}
688
689 conn = psycopg2.connect(postgresConnectionString)
690 cur = conn.cursor()
691
692 createTables(cur,conn)
693
694 csvTimeResults = open(csvTimeResultsPath, 'w', newline=' ')
695 timeCsvWriter = csv.writer(csvTimeResults, lineterminator='\n',
696     delimiter=';')
697
698 dictCallType=createCallTypeDictionary()
699
700 queryTester = QueryTester()
701 start_global_time = time.time()
702
703 csvIteration=0
704 for currentCSV in inputList:
705     csvIteration=csvIteration+1
706     # Fill dictionaries and fetch latest id
707     lastIDDuration = putDurationTableInDictionary(tempTableDurata)
708     lastIDGeoPlace = putGeoPlaceTableInDictionary(tempTableGeoPlace)
709     lastIDDate = putDateTableInDictionary(tempTableDate)
710     lastIDResponsibility =
711         putResponsibilityTableInDictionary(tempTableResponsibility)
712     lastIDCallType = putCallTypeTableInDictionary(tempTableCallType)
713
714 if currentCSV==inputCsvPathFAKE:
715     generateConsistentFakeRows(tempTableDurata, tempTableGeoPlace,
716         tempTableDate, tempTableResponsibility,
717         tempTableCallType, 1000000)
718
719 start_local_time=time.time()
720 clockTimeExtraction=time.time() # Start (Extraction phase)
721
722 f=open(factOriginal_csvPATH, 'w', newline=' ')
723 g=open(factDimensions_csvPATH, 'w', newline=' ')
```

```

722     openFragmentationFiles(fragTablesPath)
723     with codecs.open(currentCSV, 'rU', 'utf-16-le') as csv_file:
724         reader = csv.reader(csv_file)
725         cnt = 0
726         for row in reader:
727             if cnt != 0:
728                 valResult=rowValidation(row,dictCallType)
729                 if valResult:
730                     cntValidRows=cntValidRows+1
731
732                     elapsedTimeExtraction=elapsedTimeExtraction+(time.time()-clockTimeExtraction)
733                     # Pause (Extraction phase)
734
735                     # Start (Transformation phase)
736                     clockTimeTransformation=time.time()
737                     manRow = rowManipulation(row,cur)
738                     elapsedTimeTransformation=elapsedTimeTransformation+(time.time()-clockTimeTransformation)
739                     # End (Transformation phase)
740
741                     # Start (OTHER)
742                     clockTimeOther = time.time()
743
744                     idDuration = getDimensionDurationRow(manRow[34],
745                                         ↪ tempTableDurata)
746
747                     idDate = getDimensionDateRow(manRow[6], tempTableDate)
748
749                     idGeoPlace = getDimensionGeoPlaceRow(manRow[15],
750                                         ↪ manRow[16], manRow[17], manRow[31],
751                                         tempTableGeoPlace)
752
753                     idResponsibility =
754                         ↪ getDimensionResponsibilityRow(manRow[20],
755                                         ↪ manRow[19], manRow[18],
756                                         tempTableResponsibility)
757
758                     idCallType = getDimensionCallTypeRow(manRow[3],
759                                         ↪ manRow[25], tempTableCallType)
760
761                     elapsedTimeOther = elapsedTimeOther + (time.time() -
762                                         ↪ clockTimeOther)
763                     # End (Transformation phase)
764
765                     # Start (Loading)
766                     clockTimeLoading=time.time()
767
768                     exportFactOriginalToCsv(f, manRow)
769                     exportFactToFragCSV(fragTablesPath,manRow,idDuration,idDate,idGeoPlace,idResponsibility,
770                                         exportFactDimToCsv(g,manRow,idDuration,idDate,idGeoPlace,idResponsibility,idCallType)
771
772                     # Pause (Loading)
773                     elapsedTimeLoading=elapsedTimeLoading+(time.time()-clockTimeLoading)
774                 else:
775                     cntNotValidRows=cntNotValidRows+1
776                 # Re-Start (Extraction phase)

```

```

771         clockTimeExtraction = time.time()
772     else:
773         cnt = cnt + 1
774
775
776     # Re-Start (Loading)
777     clockTimeLoading = time.time()
778
779     exportDimensionDurataToCsv(tempTableDurata, dimDurationCSVPath,
780         ↪ lastIDDuration)
780     exportDimensionDateToCsv(tempTableDate, dimDateCSVPath, lastIDDate)
781     exportDimensionGeoPlaceToCsv(tempTableGeoPlace,
782         ↪ dimGeoPlaceCSVPath, lastIDGeoPlace)
783     exportDimensionResponsibilityToCsv(tempTableResponsibility, dimResponsibilityCSVPath, lastIDResponsibility)
784     exportDimensionCallTypeToCsv(tempTableCallType, dimCallTypeCSVPath, lastIDCallType)
785
786     # Pause (Loading)
787     elapsedTimeLoading = elapsedTimeLoading + (time.time() -
788         ↪ clockTimeLoading)
789
790     f.close()
791     g.close()
792     closeFragmentationFiles(fragTablesPath)
793
794     # Re-Start (Loading)
795     clockTimeLoading = time.time()
796
797     csvToPostgres(dimDurationCSVPath, 'dim_duration', cur, conn)
798     csvToPostgres(dimGeoPlaceCSVPath, 'dim_geo_place', cur, conn)
799     csvToPostgres(dimDateCSVPath, 'dim_received_date', cur, conn)
800     csvToPostgres(dimResponsibilityCSVPath, 'dim_responsibility', cur, conn)
801     csvToPostgres(dimCallTypeCSVPath, 'dim_call_type', cur, conn)
802     csvToPostgres(factOriginal_csvPATH, 'dispatch911_original', cur, conn)
803     csvToPostgres(factDimensions_csvPATH, 'dispatch911_dimensions', cur,
804         ↪ conn)
805     for y,fragFile in fragTablesPath.items():
806         csvToPostgres(fragFile.filePath,fragFile.postgresTableName,cur,conn)
807
808         open(dimDurationCSVPath, 'w').close()
809         open(dimDateCSVPath, 'w').close()
810         open(dimGeoPlaceCSVPath, 'w').close()
811         open(dimResponsibilityCSVPath, 'w').close()
812         open(dimCallTypeCSVPath, 'w').close()
813         open(factOriginal_csvPATH, 'w').close()
814         open(factDimensions_csvPATH, 'w').close()
815         open(inputCsvPathFAKE, 'w').close()
816         for year, fragFile in fragTablesPath.items():
817             open(fragFile.filePath, 'w').close()
818
819         # Start (Refresh materialized view)
820         clockTimeMatView = time.time()
821
822         cur.execute("REFRESH MATERIALIZED VIEW intervention_daytime")
823         cur.execute("REFRESH MATERIALIZED VIEW intervention_nighttime")

```

```
822 # End (Refresh materialized view)
823 elapsedTimeMatView=elapsedTimeMatView + (time.time() -
824     ↪ clockTimeMatView)
825 conn.commit()
826
827 # End (Loading)
828 elapsedTimeLoading = elapsedTimeLoading + (time.time() -
829     ↪ clockTimeLoading)
830 print("Fine ETL (sec): %s" % (time.time() - start_local_time))
831
832 queryTester.computeAndWriteAvgs(csvIteration,'NoIndex')
833
834 #Creating index (Start)
835 clockTimeIndex=time.time()
836 queryTester.createIndex()
837 elapsedTimeIndex=time.time() - clockTimeIndex
838
839 queryTester.computeAndWriteAvgs(csvIteration,'Index')
840 queryTester.dropIndex()
841
842 print("+ elapsedTimeExtraction: %s" % elapsedTimeExtraction)
843 print("+ elapsedTimeTransformation: %s" % elapsedTimeTransformation)
844 print("+ elapsedTimeLoading: %s" % elapsedTimeLoading)
845 print("+ elapsedTimeOther: %s" % elapsedTimeOther)
846 print("+ elapsedTimeMatView: %s" % elapsedTimeMatView)
847 print("+ elapsedTimeIndex: %s" % elapsedTimeIndex)
848
849 print("Righe non valide: %s" % (cntNotValidRows))
850 print("Righe valide: %s" % (cntValidRows))
851
852 outTimeRow=[csvIteration, elapsedTimeExtraction,
853             ↪ elapsedTimeTransformation, elapsedTimeLoading,
854             ↪ elapsedTimeMatView,elapsedTimeIndex,cntNotValidRows,cntValidRows]
855 timeCsvWriter.writerow(outTimeRow)
856
857
858 elapsedTimeExtraction = 0
859 elapsedTimeTransformation = 0
860 elapsedTimeLoading = 0
861 elapsedTimeOther = 0
862 elapsedTimeMatView = 0
863 cntNotValidRows = 0
864 cntValidRows = 0
865 elapsedTimeIndex = 0
866
867 print("+++")
868
869 queryTester.csvQueryResults.close()
870 csvTimeResults.close()
871
872 print("Tempo totale per tutti i file (sec): %s" % (time.time() -
873     ↪ start_global_time))
```