



Presentation Report

Presentation Topic

Building a SQL-based data warehouse for a trillion
rows in Python

VII Sem. Database Warehouse & Data Mining

Academic Session 2020-21

Department of Computer Science and Engg.

Walchand College of Engineering, Sangli
(An Autonomous Institute)

Submitted By

2018BTECS00212

2018BTECS00211

2017BTECS00068

Table of Contents

Sr.no	Building a SQL-based data warehouse for a trillion rows in Python
1.1	Introduction
1.2	SQL for warehousing
1.3	Python for data analysis
1.4	Python for SQL based warehousing
2	Python ETL Tools
3	Aspects to be considered while building warehouse
3.1	storage
3.2	Query Optimization
3.2.1	Batching the load of the Fact table
3.2.2	Batched INSERT Statement
3.2.3	Normalization (Dimension) table
3.2.4	Purging old data
3.2.5	Master / slave
3.2.6	Sharding
3.3	Retrieval Efficiency
4	References

1.1 Introduction

Python is an elegant, versatile language with an ecosystem of powerful modules and code libraries. Writing Python for ETL starts with knowledge of the relevant frameworks and libraries, such as workflow management utilities, libraries for accessing and extracting data, and fully-featured ETL toolkits.

Extract, transform, load (ETL) is the main process through which enterprises gather information from data sources and replicate it to destinations like data warehouses for use with business intelligence (BI) tools.

Python has excellent support for synthetic data generation. Packages such as pydbgen, which is a wrapper around Faker, make it very easy to generate synthetic data that looks like real world data.

The ecosystem of tools and libraries in Python for data manipulation and analytics is truly impressive, and continues to grow. There are, however, gaps in their utility that can be filled by the capabilities of a data warehouse.

1.2 SQL for warehousing

Even after 40+ years (!) of existence, SQL remains one of the most ubiquitous programming languages. StackOverflow's 2017 Developer Survey shows that 51.2% percent of all respondents use SQL, making it the second most-utilized programming language in existence, right behind Java. Looking strictly at data scientist and data engineer respondents, this number increases to 58%.

It cannot be understated how impressive this is for a 40+ year-old programming language. Not only does SQL continue to stand the test of time, but it becomes even more pervasive within the analytics and data science community as it ages. Take a look at any number of data analyst and data scientist job postings, and you'll notice a trend: deep analytical SQL expertise is no longer a luxury, but a necessity.

Two important characteristics account for SQL's staying power. First, its declarative nature (i.e. programming in terms of what you want done, rather than how to do it) makes it easy to learn, write, and comprehend. The logistics of control flow are abstracted away in favor of simplicity and readability. This accessibility has been vital in fostering SQL's ubiquity, particularly within the analytics and data science community.

Second, and perhaps more important, the evolution of data infrastructure (i.e. performant data warehouses such as BigQuery and Redshift) has transfigured SQL into a language suddenly capable of complex data transformations. This evolution has significantly broadened the scope of what is possible using SQL.

Ad-hoc analytical SQL queries that previously may have taken days to complete now take minutes, or even seconds. As a result, analysts, data scientists, and data engineers are able to iterate significantly faster and go significantly further using SQL alone than they ever have before.

1.3 Python for data analysis

In September, StackOverflow revealed evidence to suggest that Python is the fastest growing major programming language. Naturally, this left many of us asking the next logical question.

Why? Python has long attracted a dedicated community that adores it for its expressiveness and versatility. But it began to seem implausible that this alone could be the driving force of Python's accelerating growth.

Shortly after it's publication, StackOverflow put a magnifying glass to this accelerating growth rate as an attempt to reveal the driving factors. Their analysis affirms what many of us surmised. The fastest-growing use case of Python is for data science, machine learning, and academic research. Particularly, the pandas library is the fastest-growing Python-related tag on StackOverflow.

Pandas is a python data analysis library focused on making analysis of structured or tabular data fast, easy, and expressive. Since pandas is designed to work with structured data, it shares many similarities to other structured data manipulation languages, such as SQL. Pandas provides all of the data manipulation capabilities typically provided in relational databases, within the high-performance, array-based computation environment of Python. Its incredible growth in popularity has been accompanied by similar growth in other popular and related Python data science libraries, such as NumPy, matplotlib, Tensorflow, and Keras.

Pandas has been critical in transforming Python into more of an "end-to-end" data science language. When used in tandem with other popular Python data science libraries like those mentioned above, Python becomes a powerful language suitable for the wrangling, preparation, cleaning, exploration, analysis, modeling, and visualization of data. This paradigm shift has changed the way we think about Python as it relates to other data manipulation languages. The future of Python for data science looks brighter still, with a passionate and ever-growing open source community driving innovation at unprecedented speed.

1.4 Python for SQL based warehousing

We still sometimes need hybrid model of SQL and Python for analytics work? When asked about the specifics of how they apply this hybrid approach, the answer typically falls along the same lines; Python starts where SQL ends. Any analytics work that can reasonably be performed using SQL, will be. Python then picks up where SQL leaves off.

There isn't anything inherently wrong with this model. People should feel empowered to use whatever languages they'd like to get the job done. But this model perpetuates the idea that these languages are complementary, yet functionally discrete when it comes to data analysis. It fails to acknowledge a sizable overlap of shared functionality. Exploration of this middle ground can improve our understanding of the capabilities of both languages

Various methods are available now-a-days to use SQL based applications with python. Few of very important ones are using python libraries to fire SQL queries and build ETL process. Another method which is "Python and SQL integration". We will discuss about python libraries in this report.

Example:

Using mysql drivers for python to connect mysql warehouse and load

```
import mysql.connector
import random
from datetime import datetime, timedelta
```

```

mydb = mysql.connector.connect(
    host="localhost",
    user="root",
    password="datamining",
    database="ise_sales"
)

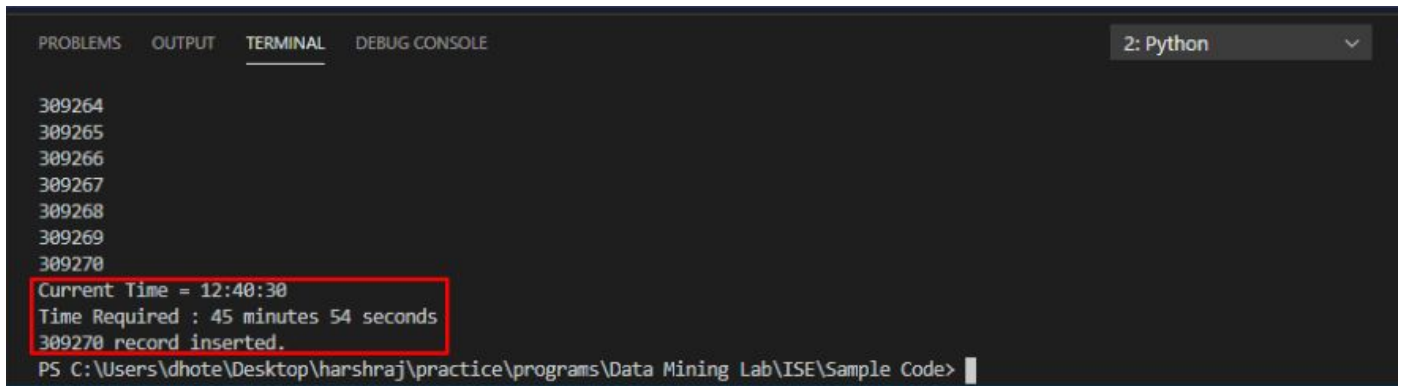
mycursor = mydb.cursor()
now = datetime.now()
start_time = now.strftime("%H:%M:%S")
print("Current Time =", start_time)
# 100000000
for i in range(0,309270):
    sql = "INSERT INTO sales (order_id,product_id,emp_id,customer_id) VALUES (%s, %s,%s,%s) "
    val = (i+1,random.randint(1,9),random.randint(1,6),random.randint(1,6))
    print(i)
    mycursor.execute(sql, val)
    mydb.commit()

now = datetime.now()
current_time = now.strftime("%H:%M:%S")
print("Current Time =", current_time)
#
print(timedelta(hours=start_time.hour,minutes=start_time.minute,seconds=start_time
.second) -
timedelta(hours=current_time.hour,minutes=current_time.minute,seconds=current_time
.second))
print("309270 record inserted.")
# print(mydb)

```

OUTPUT:

45 minutes for inserting 309270 rows



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: Python
309264
309265
309266
309267
309268
309269
309270
Current Time = 12:40:30
Time Required : 45 minutes 54 seconds
309270 record inserted.
PS C:\Users\dhote\Desktop\harshraj\practice\programs\Data Mining Lab\ISE\Sample Code>
```

Look at your data; compute raw rows per second (or hour or day or year). There are about 30M seconds in a year; 86,400 seconds per day. Inserting 30 rows per second becomes a billion rows per year. 10 rows per second is about all you can expect from an ordinary machine (after allowing for various overheads). If you have less than that, you don't have many worries, but still you should probably create Summary tables. If more than 10/sec, then batching, etc, becomes vital. Even on spiffy hardware, 100/sec is about all you can expect without utilizing the techniques here.

2.1 Python ETL Tools

Python is a programming language that is relatively easy to learn and use. Python has an impressively active open-source community on GitHub that is churning out new Python libraries and enhancement regularly. Because of this active community and Python's low difficulty/functionality ratio, Python now sports an impressive presence in many diverse fields like game development, web development, application developments, NLP, and computer vision, just to name the few.

In recent years, Python has become a popular programming language choice for data processing, data analytics, and data science (especially with the powerful Pandas data science library). So it should not come as a surprise that there are plenty of Python ETL tools out there to choose from. Let's take a look at the most common ones.

2.1.1 Petl

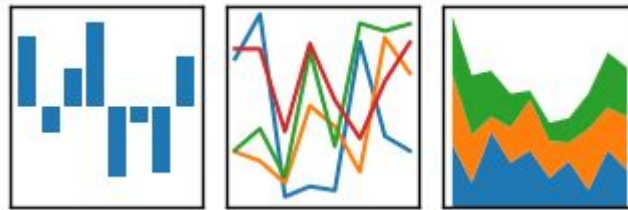
Petl (stands for Python ETL) is a basic tool that offers the standard ETL functionality of importing data from different sources (like csv, XML, json, text, xls) into your database. It is trivial in terms of features and does not offer data analytics capabilities like some other tools in the list. However, it does support all the standard transformations like row operation, sorting, joining, and aggregation.

Petl isn't bad for a simple tool, but it can suffer from performance issues; especially compared to some of the other options out there. So if you just need to build a simple ETL pipeline and performance is not a big factor, then this lightweight tool should do the job. But for anything more complex or if you expect the project to grow in scope, you may want to keep looking.

2.1.2 Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Logo for Pandas, a Python library useful for ETL

Pandas is one of the most popular Python libraries nowadays. Its rise in popularity is largely due to its use in data science, which is a fast-growing field in itself, and is how I first encountered it.

Pandas use *dataframes* as the data structure to hold the data in memory (similar to how data is handled in the R programming language) Apart from regular ETL functionalities, Pandas supports loads of data analytics and visualization features.

Pandas is relatively easy to use and has many rich features, which is why it is a commonly used tool for simple ETL and exploratory data analysis by data scientists. If you are already using Pandas it may be a good solution for deploying a proof-of-concept ETL pipeline.

2.1.3 Mara

Mara is a Python ETL tool that is lightweight but still offers the standard features for creating an ETL pipeline. It also offers other built-in features like web-based UI and command line integration. Web UI helps to visualize the ETL pipeline execution, which can also be integrated into a Flask based app. It uses PostgreSQL as the data processing engine.

If you are looking for an ETL tool that is simple but still has a touch of sophisticated features then Mara can be a good choice.

2.1.4 Airflow



Logo for Apache Airflow

Apache Airflow was created by Airbnb and is an open source workflow management tool. It can be used to create data ETL pipelines. Strictly speaking, it is not an ETL tool itself, instead, it is more of an orchestration tool that can be used to create, schedule, and monitor workflows. This means you can use Airflow to create a pipeline by consolidating various independently written modules of your ETL process.

Airflow workflow follows the concept of DAG (Directed Acyclic Graph). Airflow, like other tools in the list, also has a browser-based dashboard to visualize workflow and track execution of multiple workflows. Airflow is a good choice if you want to create a complex ETL workflow by chaining independent and existing modules together

2.1.5 Pyspark



PySpark Logo

Pyspark is the version of Spark which runs on Python and hence the name. As per their website, *"Spark is a unified analytics engine for large-scale data processing."*

The Spark core not only provides robust features for creating ETL pipelines but also has support for data streaming (Spark Streaming), SQL (Spark SQL), machine learning (MLib) and graph processing (Graph X).

The main advantage of using Pyspark is the fast processing of huge amounts data. So if you are looking to create an ETL pipeline to process big data very fast or process streams of data, then you should definitely consider Pyspark. That said, it's not an ETL solution out-of-the-box, but rather would be one part of your ETL pipeline deployment.

2.2 Data Warehouse Conception

As we wanted to act fast, we spent a lot of time designing the best systems to suit our needs. Here is where we ended up:

1. First, we should use a BRIDGE level.
BRIDGES are tables which are the result of the data collection (E from ETL) part. The data in these tables is messy, not that well organised and results from a unique data source.
2. Second, we find a CLOUD level.

CLOUDS are tables which are cleaning the data and where the complex data aggregations and processing are being performed. For instance we remove useless information, we aggregate the “event” data to another level of aggregation (if you have event level information, we transform this to offer level information or company level information etc.). CLOUDS are just a cleaner level of data. These tables are open to our business analysts. Yet, a lot of data levels are still needed for the analysis.

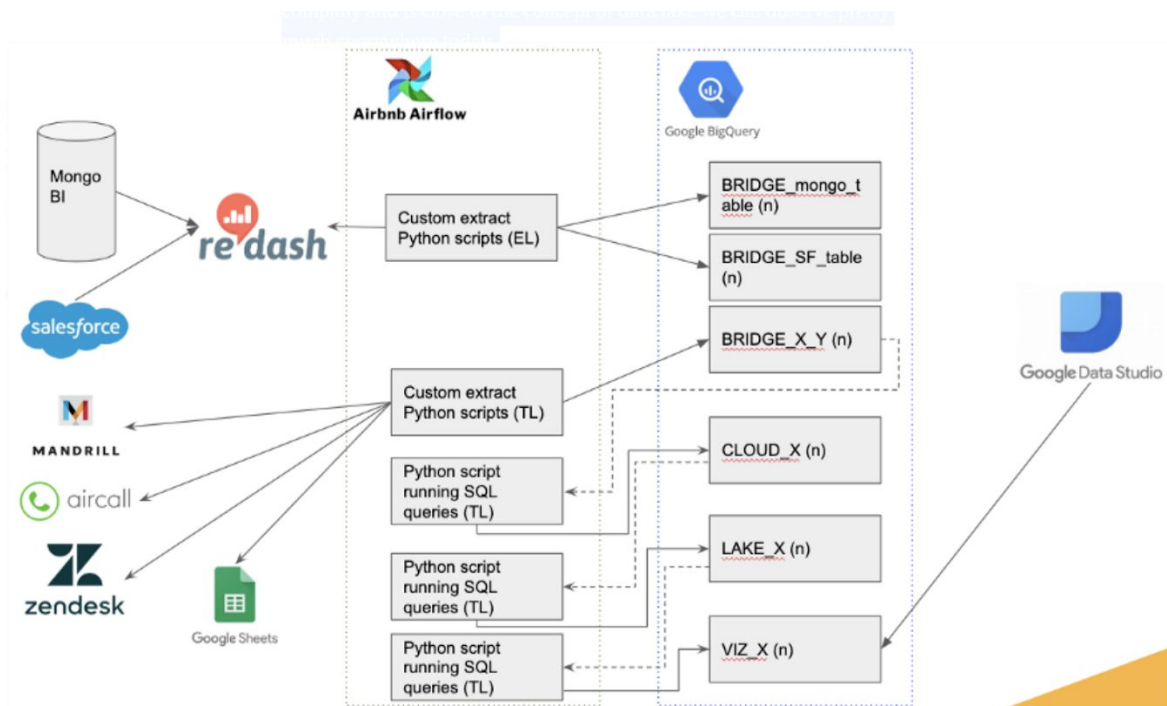
3. Third, we have LAKES.

LAKES are tables aggregating the data from different CLOUDS to provide all the information from a specific angle. For instance, we can find a lake dedicated to the operation team, a lake for the finance team, a lake for the incident part, etc. This data is open to every user of the company and is close to the concept of a data lake we can observe pretty much everywhere today.

4. Then, we use VIZ objects.

VIZ are filtered data from one or several LAKES and are dedicated to plug the data visualisation part. To make it simple, each dashboard inside Google Data Studio is plugged to a VIZ table.

In the end, we ended up with the following design :



3. Aspects to be considered while building warehouse

3.1 Storage

This part of the structure is the main foundation — it’s where your warehouse will live. There are two main options when it comes to storage, an in-house server (Oracle, Microsoft SQL Server) or on the cloud (Amazon S3, Microsoft Azure). An in-house server is internal hardware that’s set up

within your office, and the cloud is a digital storage solution based on external servers. Either is a feasible option when it comes to storage and all depends on your needs.

In House Server:

One big performance killer is UUID/GUID keys. Since they are very 'random', updates of them (at scale) are limited to 1 row = 1 disk hit. Plain disks can handle only 100 hits/second. RAID and/or SSD can increase that to something like 1000 hits/sec. Huge amounts of RAM (for caching the random index) are a costly solution. It is possible to turn type-1 UUIDs into roughly-chronological keys, thereby mitigating the performance problems if the UUIDs are written/read with some chronological clustering. UUID discussion, Hardware, etc:

- Single SATA drive: 100 IOPs (Input/Output operations per second)
- RAID with N physical drives -- 100*N IOPs (roughly)
- SSD -- 5 times as fast as rotating media (in this context)
- Batch INSERT -- 100-1000 rows is 10 times as fast as INSERTing 1 row at a time (see above)
- Purge "old" data -- Do not use DELETE or TRUNCATE, design so you can use DROP PARTITION (see above)
- Think of each INDEX (except the PRIMARY KEY on InnoDB) as a separate table

3. 2 Query Optimization

Techniques that should be applied to the huge Fact table.

- id INT/BIGINT UNSIGNED NOT NULL AUTO_INCREMENT
- PRIMARY KEY (id)
- Probably no other INDEXes
- Accessed only via id
- All VARCHARs are "normalized"; ids are stored instead
- ENGINE = InnoDB
- All "reports" use summary tables, not the Fact table
- Summary tables may be populated from ranges of id (other techniques described below)

There are exceptions where the Fact table must be accessed to retrieve multiple rows. However, you should minimize the number of INDEXes on the table because they are likely to be costly on INSERT.

3.2.1 Batching the load of the Fact table

When talking about billions of rows in the Fact table, it is essentially mandatory that you "batch" the inserts. There are two main ways:

- INSERT INTO Fact (.,.,.) VALUES (.,.,.), (.,.,.), ...; -- "Batch insert"
- LOAD DATA ...;

A third way is to INSERT or LOAD into a Staging table, then

- INSERT INTO Fact SELECT * FROM Staging; This INSERT..SELECT allows you to do other things, such as normalization. More later.

3.2.2 Batched INSERT Statement

Chunk size should usually be 100-1000 rows.

- 100-1000 an insert will run 10 times as fast as single-row inserts.
- Beyond 100, you may be interfering replication and SELECTs.
- Beyond 1000, you are into diminishing returns -- virtually no further performance gains.
- Don't go past, say, 1MB for the constructed INSERT statement. This deals with packet sizes, etc. (1MB is unlikely to be hit for a Fact table.) Decide whether your application should lean toward the 100 or the 1000.

If your data is coming in continually, and you are adding a batching layer, let's do some math. Compute your ingestion rate -- R rows per second.

- If $R < 10$ ($= 1\text{M/day} = 300\text{M/year}$) -- single-row INSERTs would probably work fine (that is, batching is optional)
- If $R < 100$ (3B records per year) -- secondary indexes on Fact table may be ok
- If $R < 1000$ (100M records/day) -- avoid secondary indexes on the Fact table.
- If $R > 1000$ -- Batching may not work. Decide how long (S seconds) you can stall loading the data in order to collect a batch of rows.
- If $S < 0.1\text{s}$ -- May not be able to keep up

If batching seems viable, then design the batching layer to gather for S seconds or 100-1000 rows, whichever comes first.

3.2.3 Normalization (Dimension) table

Normalization is important in Data Warehouse applications because it significantly cuts down on the disk footprint and improves performance. There are other reasons for normalizing, but space is the important one for DW.

3.2.4 Purging old data

Typically the Fact table is PARTITION BY RANGE (10-60 ranges of days/weeks/etc) and needs purging (DROP PARTITION) periodically. This discusses a safe/clean way to design the partitioning and do the DROPS: Purging PARTITIONS.

3.2.5 Master / slave

For "read scaling", backup, and failover, use master-slave replication or something fancier. Do ingestion only on a single active master; it replicates to the slave(s). Generate reports on the slave(s).

3.2.6 Sharding

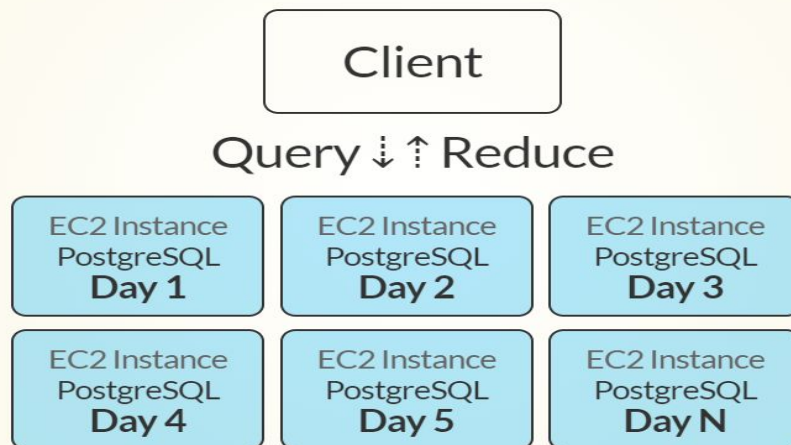
"Sharding" is the splitting of data across multiple servers. (In contrast, replication and Galera have the same data on all servers, requiring all data to be written to all servers.)

With the non-sharding techniques described here, terabyte(s) of data can be handled by a single machine. Tens of terabytes probably requires sharding.

3.3 Retrieval Efficiency

Nature of Data in a data warehouse is mostly static. Once data is inserted into the warehouse it never changes. Hence data is immutable. Every machine has its limitations to process the data, suppose every machine can efficiently work for 1 million rows of data. So we can split the 1 trillion rows of immutable data to multiple machines so they can respond to multiple queries individually and efficiently.

QUERYING TBs OF IMMUTABLE DATA IS EASY!



4. REFERENCES

1. <https://medium.com/everoad/building-a-data-warehouse-in-six-months-what-did-we-learn-e058e42446f1>
2. <https://medium.com/datadriveninvestor/complete-data-analytics-solution-using-etl-pipeline-in-python-edd6580de24b>
3. <https://tuulos.github.io/pydata-2014/#/>
4. https://www.youtube.com/watch?v=xnfnv6WT1Ng&ab_channel=PyData
5. <https://kubernetes.io/docs/home/>