BAB III

PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Data Engineer Associate Intern merupakan bagian dari divisi *Data Engineer* yang berperan dalam mengelola dan menjaga kelancaran alur data dari berbagai sumber hingga data yang telah diolah, agar dapat digunakan oleh seluruh *Business Unit* (BU) di Kawan Lama Group, yang saat ini berjumlah sekitar 30 BU. Divisi *Data Engineer* sendiri merupakan salah satu dari tiga divisi utama dalam *IT Data Team* yang berada di bawah naungan *Information Technology* (IT). *IT Data Team* terdiri dari tiga divisi utama, yaitu *Data Platform*, *Data Engineer*, dan *Data Analyst*, yang bekerja secara terintegrasi untuk mengelola siklus hidup data dalam perusahaan. Divisi *Data Platform* bertanggung jawab dalam validasi dan pengelolaan kualitas data, baik sebelum maupun setelah data diolah, guna memastikan keakuratan dan konsistensinya sebelum digunakan lebih lanjut. Divisi *Data Engineer* berfokus pada pembersihan, transformasi, pengelolaan, serta penyimpanan data ke dalam infrastruktur yang telah ditentukan, seperti *data lake*, *data warehouse*, dan *data mart*. Ketiga penyimpanan tersebut bertujuan agar data dapat diakses dan digunakan secara optimal oleh berbagai pihak yang membutuhkan. Setelah data tersedia dalam sistem penyimpanan yang telah disiapkan, divisi *Data Analyst* mengambil peran dalam menganalisis data tersebut dan mengolahnya menjadi informasi yang dapat memberikan wawasan strategis bagi perusahaan. Hasil analisis ini umumnya disajikan dalam bentuk *dashboard* interaktif yang membantu pengambilan keputusan berbasis data di seluruh lini bisnis Kawan Lama Group.

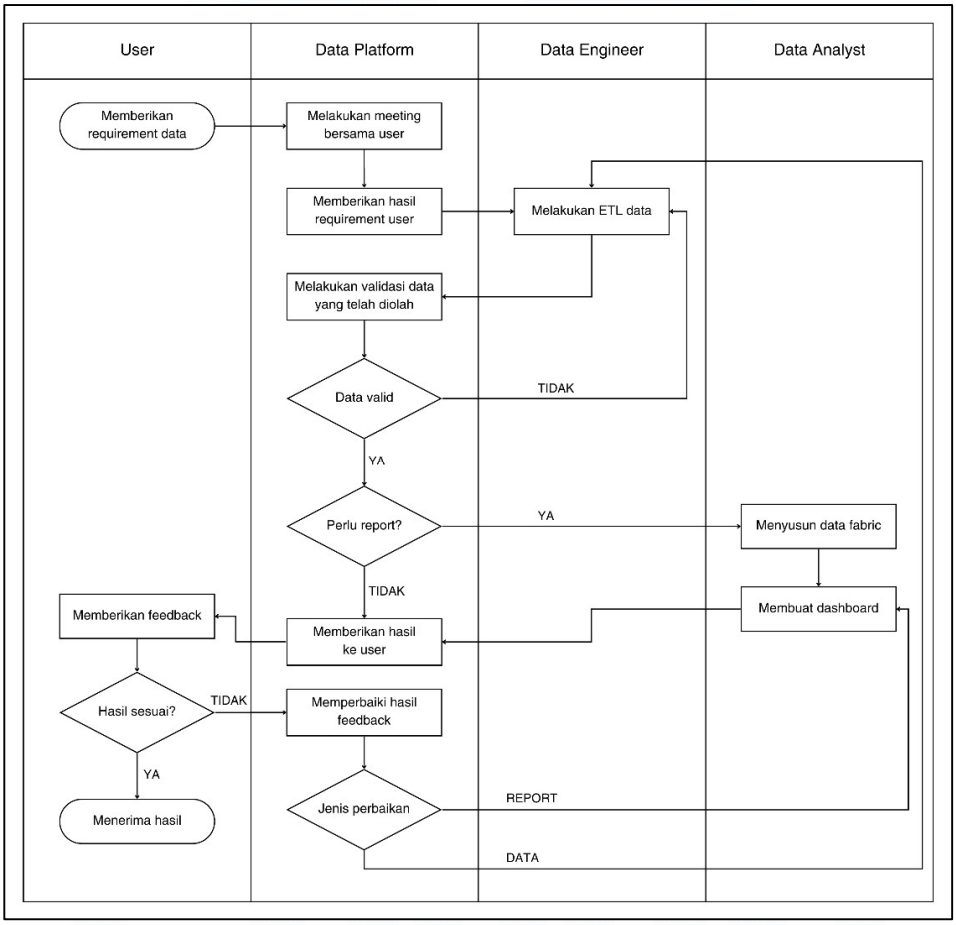
Dalam menjalankan pekerjaannya, *IT Data Team* menerapkan metode kerja berbasis *sprint* dengan durasi satu minggu, dimulai setiap hari Selasa dan berakhir pada hari Senin berikutnya. Metode ini digunakan agar dapat setiap anggota tim dapat bekerja secara terstruktur, dengan tugas yang terencana dan target yang jelas dalam setiap siklus *sprint*. Selama periode *sprint* berlangsung, seluruh anggota tim

berpartisipasi dalam *Daily Stand Up* (DSU) yang diadakan setiap harinya pada pukul 09.00 WIB. Dalam pertemuan ini, masing-masing anggota tim memberikan *update* terkait progres pekerjaan yang sedang mereka tangani, kendala yang dihadapi, serta langkah selanjutnya yang akan dilakukan dalam proyek yang sedang berjalan.

Selain pertemuan harian, di setiap hari Senin juga diadakan sesi *backlog grooming* yang melibatkan seluruh anggota *IT Data Team*. Dalam sesi ini, tim bersama-sama mengevaluasi tugas-tugas yang telah diselesaikan pada *sprint* sebelumnya serta membahas prioritas dan urgensi pekerjaan yang ada di *backlog*. Setelah sesi *backlog grooming* selesai, masing-masing divisi kemudian melanjutkan dengan *sprint planning* internal pada hari yang sama. Untuk divisi *Data Engineer*, sesi ini dilakukan secara lebih spesifik oleh tim di dalamnya. Dalam *sprint planning*, setiap divisi akan memetakan tugas-tugas yang akan dikerjakan selama satu minggu ke depan dan menetapkan *assignee* atau penanggung jawab untuk masing-masing tugas. Proses ini dilakukan guna memastikan seluruh pekerjaan berjalan terstruktur, selaras dengan target bisnis, dan efisien dalam pelaksanaannya.

IT Data Team dalam menjalankan tugasnya berkoordinasi secara sistematis antardivisi untuk memastikan bahwa setiap permintaan data dapat diproses dengan efisien dan menghasilkan output yang sesuai dengan kebutuhan bisnis. Pada Gambar 3.1, alur koordinasi ditunjukkan dengan *swimlane diagram*, yang melibatkan beberapa tahapan yang melibatkan peran dari *User*, *Data Platform*, *Data Engineer*, dan *Data Analyst*. *User* yang dimaksud merupakan anggota tim lain di departemen yang berbeda di Kawan Lama Group, yang memiliki kebutuhan untuk menggunakan data. Proses dimulai ketika *user* memberikan *requirement* data yang diperlukan untuk kebutuhan bisnis tertentu. Kemudian, divisi *Data Platform* mengadakan pertemuan dengan *user* guna memahami kebutuhan tersebut secara lebih mendalam sebelum menyusun hasil *requirement* yang akan diteruskan kepada divisi *Data Engineer* untuk diproses lebih lanjut. Divisi *Data Engineer* bertanggung jawab dalam melakukan proses ETL *(Extract, Transform, Load)* terhadap data yang

telah diterima sebelum dikirimkan kembali ke divisi *Data Platform* untuk dilakukan validasi. Jika data yang telah diolah tidak valid, maka divisi *Data Engineer* perlu melakukan perbaikan sebelum data dapat digunakan lebih lanjut.



Gambar 3.1 Alur Koordinasi *IT Data Team* Kawan Lama Group

Setelah data dinyatakan valid oleh divisi *Data Platform*, langkah berikutnya adalah menentukan apakah hasil akhir harus disajikan dalam bentuk laporan *(report)* atau tidak. Jika diperlukan dalam bentuk *report*, maka divisi *Data Analyst* akan menyusun *data fabric* sebagai tahap awal sebelum pembuatan *dashboard*. Sebaliknya, jika *report* tidak diperlukan, hasil yang telah divalidasi langsung diberikan kepada *user*. Dalam kedua skenario, *user* memiliki kesempatan untuk memberikan *feedback* atas hasil yang diterima. Jika hasilnya belum sesuai dengan ekspektasi, maka dilakukan proses revisi dengan menentukan jenis perbaikan yang

dibutuhkan, baik dalam bentuk data maupun *report*. Proses ini berulang hingga *user* menyetujui hasil yang diberikan.

Untuk mendukung kelancaran proses koordinasi dan pemrosesan data tersebut, divisi *Data Engineer* terbagi menjadi tiga tim dengan fokus tanggung jawab yang berbeda-beda, yaitu tim AI/ML, tim *Data Engineer*, dan tim *Analytics Engineer*. Tim AI/ML berfokus pada pengembangan model kecerdasan buatan, salah satunya adalah *chatbot* berbasis AI yang dirancang untuk membantu para eksekutif dalam mengakses wawasan bisnis perusahaan, baik untuk analisis historis maupun prediktif. *Chatbot* ini berguna untuk memperoleh informasi terkait performa bisnis, seperti prediksi penjualan di masa depan atau rekapitulasi produk dengan angka penjualan tertinggi. Selain itu, tim AI/ML juga mengembangkan *chatbot* lain yang diperuntukkan bagi karyawan di toko *offline*, seperti AZKO dan Informa. *Chatbot* ini mempermudah pencarian produk sesuai dengan kebutuhan pelanggan, misalnya dengan memasukkan kata kunci seperti "lemari anak", *chatbot* akan menampilkan daftar produk furnitur yang tersedia di Informa. Di samping itu, tim ini juga bertanggung jawab dalam pengelolaan dan pemeliharaan infrastruktur *pipeline* data, termasuk menjaga alur otomatisasi data agar berjalan dengan stabil serta menangani *troubleshooting* teknis jika terjadi gangguan pada sistem orkestrasi data yang digunakan. Meskipun tugas utama tim AI/ML berfokus pada pengembangan *chatbot* dan pemeliharaan infrastruktur *pipeline*, mereka juga tetap mendukung tim *Data Engineer* dalam beberapa tugas teknis yang memiliki keterkaitan erat dengan pengelolaan data.

Tim *Data Engineer* memiliki tanggung jawab utama dalam proses pengolahan data melalui *pipeline* ETL ke dalam berbagai lapisan penyimpanan data, seperti *data lake*, *data warehouse*, dan *data mart*. Mereka memastikan bahwa aliran data yang berasal dari berbagai sumber dapat dikelola dengan efisien dan tersimpan dalam sistem yang dapat diakses untuk kebutuhan analisis. Dalam melaksanakan tugas ini, tim *Data Engineer* menggunakan berbagai teknologi dan tools untuk penyimpanan dan pemrosesan data dalam skala besar. Sementara itu, tim *Analytics Engineer* memiliki peran yang sedikit berbeda dibandingkan dua tim lainnya.

Berbeda dengan *Data Engineer* yang berkoordinasi erat dengan *Data Platform* dan *Data Analyst*, tim *Analytics Engineer* berinteraksi langsung dengan pengguna bisnis *(end user)* tanpa melalui perantara dari divisi lain. Fokus utama mereka adalah mengelola platform IBM Planning Analytics (TM1), di mana mereka menyajikan data dalam bentuk laporan interaktif menggunakan *cube* yang dapat diakses oleh pengguna melalui tampilan tabel maupun *dashboard*. Dalam proses ini, tim *Data Engineer* bertanggung jawab untuk memasukkan data ke dalam TM1, yang dapat berasal dari *data lake* maupun *data warehouse*. Sebelum dimasukkan, data tersebut terlebih dahulu dimodifikasi melalui query yang disesuaikan dengan struktur *cube* yang ada di dalam TM1, sehingga dapat digunakan secara optimal oleh pengguna sebagai dasar pengambilan keputusan bisnis. Meskipun memiliki tanggung jawab masing-masing, ketiga tim ini tetap saling berkoordinasi dalam memastikan pengelolaan data berjalan optimal sesuai dengan kebutuhan bisnis Kawan Lama Group.

3.2 Tugas dan Uraian Kerja Magang

Kerja magang di Kawan Lama Group, khususnya dalam lingkup *IT Data Team*, dilaksanakan secara dinamis yang bergantung pada kebutuhan bisnis maupun permintaan khusus dari *user*. Selain menangani permintaan *user* secara langsung *(ad-hoc request)*, kegiatan tim juga mencakup pekerjaan bersifat *enhancement*, yaitu perbaikan atau pengembangan lebih lanjut terhadap *tools* dan *script* yang telah ada untuk mendukung efisiensi dan skalabilitas proses *data engineering*. Lingkungan tim menjalankan berbagai proyek dengan skala yang bervariasi, mulai dari yang kecil hingga berskala besar dan lintas divisi. Setiap proyek memiliki seorang *lead* yang bertanggung jawab atas jalannya koordinasi dan eksekusi tugas.

Untuk proyek berskala kecil, biasanya hanya melibatkan beberapa anggota dari tiap divisi terkait, dengan lingkup kerja yang lebih terbatas dan fokus pada satu kebutuhan spesifik. Sementara itu pada proyek besar dan berdampak luas, permintaan kerja *(task)* biasanya berasal dari *requestor* yang kemudian akan dikoordinasikan dan di-*assign* oleh anggota tim yang memiliki tanggung jawab mengelola distribusi tugas. Seluruh proyek dan *task* yang dikerjakan dipantau

secara sistematis menggunakan platform Jira, yang mendukung metode kerja *sprint-based development*. Dalam setiap siklus *sprint*, masing-masing tugas akan diberi estimasi *story point yang* merepresentasikan tingkat kesulitan dan beban kerja dari tugas tersebut. Jumlah maksimal *story point* yang bisa dimasukkan dalam satu sprint pun telah ditentukan, agar seluruh pekerjaan dapat diselesaikan dalam kurun waktu yang ditetapkan. Proyek yang terlibat selama menjalani program magang sebagai *Data Engineer Associate Intern* meliputi proyek-proyek berskala menengah hingga besar, terutama yang berada dalam tanggung jawab langsung Tim *Data Engineer*. Tabel 3.1 menunjukkan pekerjaan yang telah dilakukan selama masa magang dan durasi pengerjaannya.

Tabel 3.1 Uraian dan Durasi Kerja Magang

No.	Pekerjaan yang Dilakukan	Tanggal Mulai	Tanggal Selesai
1	Pengenalan *tools* dan alur kerja divisi	3 Februari 2025	7 Februari 2025
2	*Backfill* data tabel sales CDHDR SAP ke *data lake*	11 Februari 2025	14 Februari 2025
3	*Ingest* tabel target SQM dan target Traffic TM1 ke *data lake*	17 Februari 2025	18 Februari 2025
4	*Backfill* data *cube* SDSR TM1	19 Februari 2025	25 Februari 2025
5	Pembuatan *daily* DAG SFTP SAP large file	26 Februari 2025	28 Februari 2025
6	*Backfill* data *cube* IM-Aging TM1	3 Maret 2025	7 Maret 2025
7	*Ingest* tabel shipment mapping ke TM1	10 Maret 2025	12 Maret 2025
8	Backfill DWH Inventory	13 Maret 2025	18 Maret 2025
9	Pembuatan *script data mart* SR analysis	19 Maret 2025	24 Maret 2025
10	*Ingest* data AR 100 ke *cube* TM1	25 Maret 2025	8 April 2025
11	Ingest data memberstatuslog (S3) ke *data lake*	9 April 2025	11 April 2025
12	Ingest data ADF report dari *database* Diversion ke *data lake*	14 April 2025	17 April 2025
13	Penambahan kolom pada tabel DWH dim_receive dan fact_transaction_pos	21 April 2025	25 April 2025
14	Pembuatan tabel fact_open_price pada DWH	28 April 2025	30 April 2025
15	Penambahan kolom pada tabel DWH fact_sales_order	1 Mei 2025	2 Mei 2025
16	Ingest data S3 ke *data lake*	5 Mei 2025	9 Mei 2025

No.	Pekerjaan yang Dilakukan	Tanggal Mulai	Tanggal Selesai
17	*Ingest* data dari *database* Prime ke *data lake*	12 Mei 2025	16 Mei 2025
18	Pembuatan *script* otomatisasi *trigger* fabric *pipeline* Power BI pada Airflow	19 Mei 2025	23 Mei 2025

(Sumber olahan peneliti, 2025)

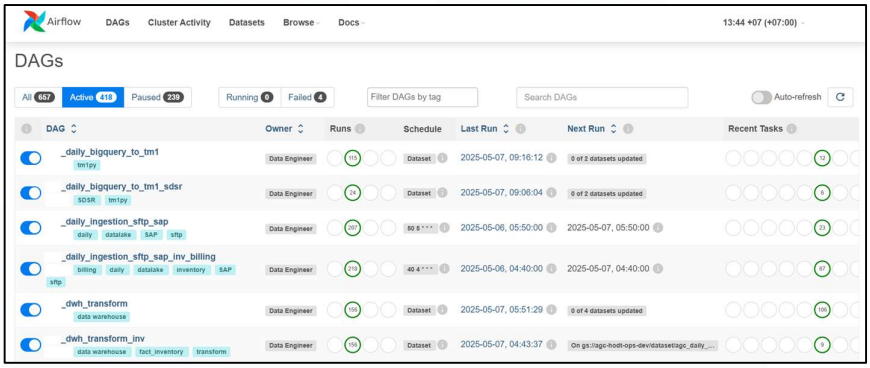
3.2.1 Pengenalan *Tools* dan Alur Kerja Divisi

3.2.1.1 *Tools*

Penguasaan terhadap *tools* dan platform yang digunakan selama kerja magang sangat penting dalam memahami keseluruhan proses *data engineering* yang berjalan dalam perusahaan. Oleh karena itu, perlu adanya pengenalan terhadap *tools* yang akan digunakan. Sejumlah *tools* dan platform utama yang diperkenalkan antara lain adalah Apache Airflow, Visual Studio Code, GitLab, Google BigQuery, serta TM1 (IBM Planning Analytics). Masing-masing *tools* ini memiliki fungsi spesifik dan saling berkaitan dalam mendukung alur kerja tim *Data Engineer*.

Apache Airflow merupakan *tools* utama dalam orkestrasi *pipeline* data di divisi *Data Engineer* Kawan Lama Group. Airflow berperan sebagai *workflow orchestrator*, yaitu sistem yang bertugas menjalankan rangkaian proses otomatis dalam pemrosesan data, mulai dari ekstraksi data hingga penyimpanan ke dalam *data lake* atau *data warehouse*. Selain itu, Airflow juga berfungsi sebagai *scheduler*, di mana setiap *pipeline* dapat dijadwalkan untuk berjalan secara otomatis sesuai dengan kebutuhan, baik harian, mingguan, bulanan, bahkan hingga dalam hitungan jam tertentu. Penjadwalan ini sangat penting untuk memastikan pembaruan data berlangsung secara konsisten tanpa perlu intervensi manual. Dalam Airflow, *pipeline* data didefinisikan dalam bentuk *Directed Acyclic Graph* (DAG), yaitu struktur yang menggambarkan urutan serta ketergantungan antar *task* dalam sebuah proses otomasi data. DAG menjamin bahwa alur eksekusi tidak membentuk siklus *(acyclic)* sehingga setiap task dijalankan berdasarkan urutan dan ketentuan dependensinya. Penggunaan DAG membuat *pipeline* data dapat bekerja secara sistematis dan efisien baik

dijalankan secara paralel maupun berurutan, sehingga proses data yang kompleks menjadi lebih mudah dipantau dan dikelola terutama dalam produksi yang berskala besar.

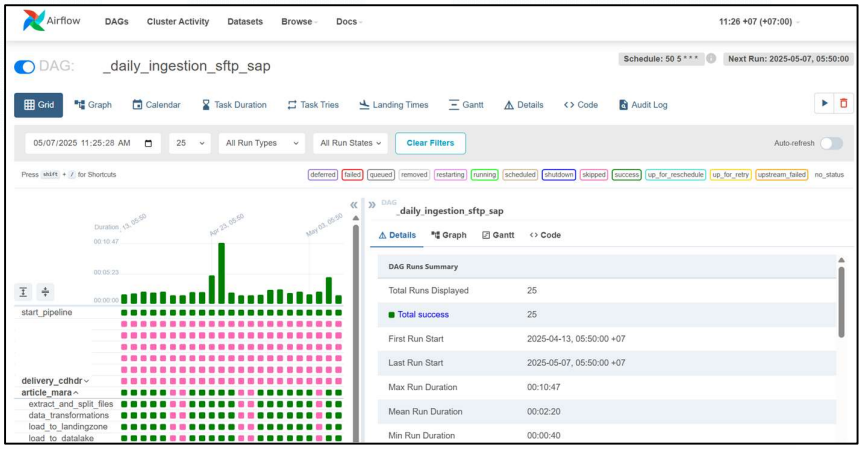


Gambar 3.2 Tampilan Menu Utama Airflow Tim *Data Engineer*

Pada Gambar 3.2, terdapat menu utama Airflow yang menampilkan daftar DAG yang telah dibuat oleh tim Data Engineer. Setiap DAG merepresentasikan satu alur kerja *(workflow)* otomatis yang dirancang untuk menjalankan proses tertentu, mulai dari pengambilan data dari sumber spesifik hingga penyimpanan atau transformasi data untuk tujuan tertentu. Melalui tampilan ini, pengguna dapat melihat sekilas status dari masing-masing DAG, apakah sedang aktif, nonaktif, berjalan, atau mengalami kegagalan. Informasi ini sangat membantu dalam pemantauan dan pengelolaan *pipeline* data yang kompleks agar dapat berjalan dengan lancar dan sesuai jadwal yang telah ditentukan.

Tampilan salah satu DAG dapat dilihat pada Gambar 3.3, di mana DAG tersebut merupakan sebuah *pipeline* harian yang menjalankan proses *ingestion* data. Dalam tampilan ini, kotak-kotak kecil berwarna menandakan status dari setiap task dalam DAG, dengan keterangan warna yang tertera yang juga ada pada tampilan tersebut. Di sisi kanan, terdapat ringkasan eksekusi DAG dan beragam fitur tambahan seperti tab Graph, Gantt, Task Duration, hingga Code yang berguna untuk menganalisis alur, durasi, dan isi kode DAG secara mendalam. Fungsi utama dari tampilan ini adalah sebagai alat pemantauan *real-time* terhadap *pipeline* untuk menjaga

keandalan sistem data perusahaan, sehingga tim dapat dengan cepat mengidentifikasi kendala, seperti task yang gagal atau berjalan terlalu lama, dan segera mengambil tindakan perbaikan. Selain itu, karena *pipeline* dijalankan berdasarkan jadwal, Airflow juga mendukung keberlangsungan data yang up to date tanpa perlu pengerjaan manual secara berkala. Tampilan DAG seperti ini menjadi inti dari operasional data di tim, karena seluruh proses dari awal hingga akhir dapat dikontrol dan dikaji dalam satu tempat terpadu.



Gambar 3.3 Tampilan DAG Airflow Tim *Data Engineer*

Untuk membuat dan mengembangkan *pipeline* tersebut, Visual Studio Code (VS Code) digunakan sebagai code editor utama. VS Code digunakan dalam pembuatan *script* Python maupun file konfigurasi yang diperlukan dalam Airflow, termasuk dalam membangun file DAG yang menjadi jantung dari alur kerja *pipeline*. Setelah *script* selesai dikembangkan di VS Code, proses penyimpanan dan pengelolaan file dilakukan menggunakan GitLab. GitLab berfungsi sebagai repositori untuk menyimpan seluruh *source code* yang berkaitan dengan *pipeline*, serta menyediakan fasilitas *version control* melalui sistem Git. Gambar 3.4 menunjukkan diagram alur kerja Git, di mana Git memiliki area utama serta konsep-konsep penting yang digunakan. Berikut merupakan empat area utama dalam struktur kerja Git yang berperan dalam pengelolaan dan pelacakan perubahan kode:

1. *Working Directory*

   Working Repository adalah direktori kerja di komputer lokal tempat pengembangan dan modifikasi kode dilakukan secara langsung. Di area ini, semua file proyek termasuk skrip Python, file konfigurasi Airflow, dan dokumentasi tersedia dalam bentuk yang dapat dibaca dan diedit. Karena *Working Repository* bersifat dinamis, setiap perubahan atau penambahan berkas, baik itu mulai dari penambahan kode baru, perbaikan bug, hingga eksperimen konfigurasi akan terlihat langsung. *Working Repository* mencerminkan kondisi file terkini sebelum langkah *version control* diterapkan yang bekerja secara terpisah tanpa mengganggu versi stabil proyek di tempat lain, sehingga percobaan dan penyesuaian dapat dilakukan dengan secara fleksibel sebelum perubahan dipublikasikan lebih lanjut.

2. *Staging Area*

   Staging Area, atau yang sering disebut *index*, berfungsi sebagai zona penampungan sementara bagi perubahan yang telah dibuat di *Working Repository* tetapi belum secara permanen diarsipkan. Ketika terdapat modifikasi pada suatu file, modifikasi tersebut dikelompokkan menjadi satu kesatuan logis, sehingga setiap perubahan yang akan dicatat dapat dipilah berdasarkan konteks atau fitur tertentu. Pada area ini, perubahan-perubahan tersebut dapat ditinjau kembali sebelum menjadikannya bagian tetap dari riwayat proyek. *Staging Area* menjembatani antara modifikasi lokal dan penyimpanan permanen di repositori.

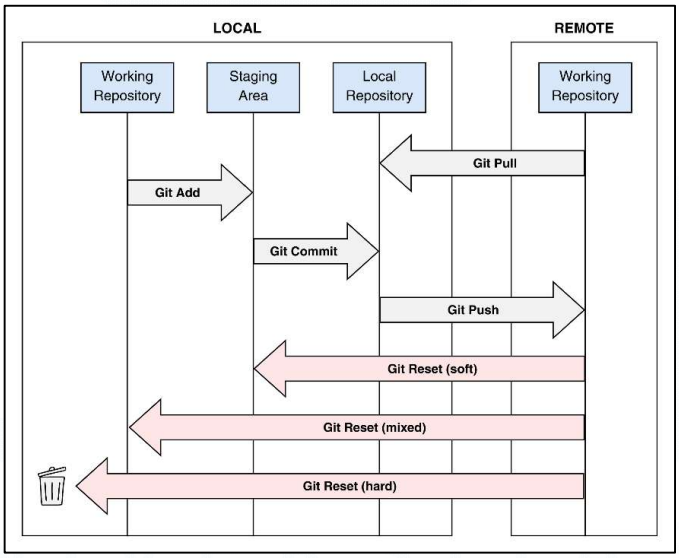
3. *Local Repository*

   Local Repository adalah *database* Git yang terletak di dalam folder tersembunyi di direktori lokal yang berisi seluruh catatan perubahan disimpan secara permanen. Repositori ini dapat digunakan untuk menelusuri kembali titik-titik riwayat proyek, melakukan *rollback* ke kondisi sebelumnya, atau bahkan membuat cabang pengembangan baru berdasarkan versi-versi terdahulu. Karena

Local Repository tersimpan secara lokal, pengerjaan dapat dilakukan secara offline dengan memanfaatkan riwayat lengkap proyek tanpa bergantung pada koneksi internet.

4. *Remote Repository*

   Remote Repository merupakan salinan repositori proyek yang di*hosting* di server pusat (GitLab), untuk mendukung kolaborasi tim. Area ini digunakan untuk seluruh anggota tim agar dapat mengunggah dan men-*download* versi terbaru proyek. Selain sebagai cadangan, *Remote Repository* diintegrasikan dengan sistem *Continuous Integration/Continuous Deployment* (CI/CD), sehingga setiap pembaruan yang masuk dapat secara otomatis diuji dan di-*deploy* ke lingkungan *staging* atau produksi. *Remote Repository* membuat sinkronisasi antar anggota tim menjadi lebih mudah, kolaborasi lintas unit dapat berjalan lancar, dan keamanan versi kode terjaga melalui kontrol akses terpusat.



Gambar 3.4 *Workflow Diagram* Git

Keempat area tersebut membentuk dasar dari mekanisme kerja Git dalam mengelola perubahan kode secara sistematis. Area utama tersebut tidak berdiri sendiri, melainkan menjadi landasan bagi penerapan berbagai

konsep penting dalam pengelolaan kode menggunakan Git. Berikut merupakan beberapa konsep penting yang sering digunakan:

1. *Branching*

   Branching adalah proses membuat cabang baru dari repositori utama (biasa disebut *main* atau *production*) untuk keperluan pengembangan secara terpisah. Cabang ini digunakan untuk mengerjakan fitur, perbaikan *(bug fixing)*, atau pembaruan tanpa mengganggu stabilitas kode yang sudah ada di cabang utama.

2. Git *Add*

   Git add digunakan untuk menandai file yang akan disertakan dalam commit. Setelah perubahan dibuat di VS Code, file yang telah dimodifikasi perlu ditambahkan ke *staging area* menggunakan *add*. File yang sudah di-*add* akan masuk ke daftar siap *commit*.

3. Git *Commit*

   Setelah file ditandai dengan *add*, langkah selanjutnya adalah *commit*, yakni proses menyimpan *snapshot* dari perubahan kode disertai pesan deskriptif *(commit message)* yang menjelaskan perubahan tersebut. *Commit* akan tercatat dalam riwayat perubahan Git dan bisa digunakan untuk menelusuri perkembangan kode dari waktu ke waktu.

4. Git *Push*

   Setelah perubahan file di lingkungan kerja lokal *(local directory)* disimpan dengan melakukan commit, perlu dilakukan git *push* agar perubahan tersebut tersimpan di server GitLab dan bisa dilihat oleh pengguna lain. *Push* akan mengirimkan *commit* yang sudah dibuat dari lokal ke *branch* yang sedang aktif di GitLab. Git *push* bertujuan untuk menyinkronkan pekerjaan lokal dengan repositori utama.

5. Git *Pull*

   Sebaliknya dari *push*, git *pull* digunakan untuk mengambil *update* dari *branch* Production yang ada di GitLab ke dalam repositori lokal. Konsep ini digunakan ketika ingin melakukan perubahan terhadap

script utama, agar *script* yang digunakan merupakan versi terbaru. *Pull* juga penting untuk menghindari konflik apabila ada orang lain yang telah melakukan perubahan di file yang sama.

6. Git *Reset*

   Git *reset* digunakan ketika ingin membatalkan perubahan yang sudah masuk ke *staging area* atau bahkan *commit* yang sudah dilakukan, baik karena kesalahan penulisan kode, file yang salah di-add, atau ingin kembali ke kondisi sebelumnya. *Reset* dapat dilakukan dengan tingkat berbeda, mulai dari menghapus perubahan dari *staging* saja *(soft reset)*, perubahan dari repositori lokal *(mixed reset)*, hingga menghapus seluruh *commit* terakhir *(hard reset)*. Penggunaan *reset* harus dilakukan dengan hati-hati agar tidak kehilangan pekerjaan yang belum dibackup.

7. *Merge*

   Setelah sebuah *branch* selesai dikembangkan dan sudah melalui proses pengecekan atau testing, perubahan dari branch tersebut perlu digabungkan ke branch utama dengan melakukan *merge*. Proses ini biasa dilakukan dengan membuat *merge request* di GitLab. Merge yang disetujui akan menambahkan semua perubahan dari *branch* pengembangan ke *branch* tujuan.

Konsep-konsep di atas penting dalam menunjang keseluruhan alur kerja *pipeline* data, mulai dari tahap pengembangan *script* hingga tahap eksekusi dan penyimpanan akhir data. Data hasil *pipeline* yang dijalankan melalui Airflow disimpan dalam Google BigQuery, platform penyimpanan dan analisis data berbasis *cloud* milik Google. BigQuery menjadi tempat utama untuk menyimpan berbagai jenis data yang dikelola, baik dalam bentuk *data lake*, *data warehouse*, maupun *data mart*. Di sinilah data disimpan dalam skema yang telah ditentukan, dioptimalkan untuk kecepatan query dan efisiensi pemrosesan skala besar. BigQuery memiliki kemampuan yang unggul dalam memproses data dalam jumlah besar secara cepat menggunakan teknologi *distributed computing* tanpa perlu mengelola

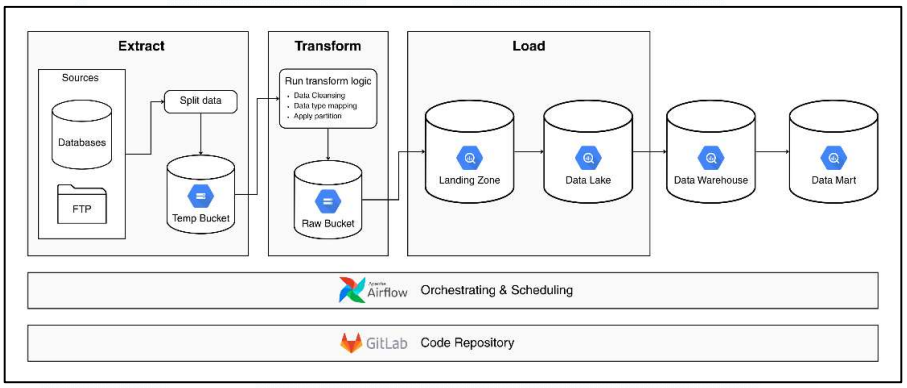
infrastruktur secara manual, sehingga sangat efisien untuk kebutuhan analisis skala besar di perusahaan. Platform ini dirancang dengan arsitektur *serverless*, yang memudahkan dalam menjalankan query tanpa harus melakukan konfigurasi server terlebih dahulu. Hal ini mempercepat proses analisis data sekaligus mengurangi beban operasional tim teknis.

Salah satu penyimpanan selain dalam Bigquery adalah *Table Manager* 1 (TM1) atau IBM Planning Analytics, yang digunakan oleh tim *Analytics Engineer* dan *end-user* sebagai salah satu platform utama untuk mengakses data. TM1 merupakan sistem *Online Analytical Processing* (OLAP) yang dibangun atas struktur *cube*, yang mendukung pemodelan data multidimensi. Salah satu keunggulan TM1 adalah fleksibilitasnya dalam menyajikan data, baik dalam bentuk tabel maupun dashboard yang interaktif. *User* dapat melakukan kustomisasi terhadap data yang ditampilkan, seperti memilih kolom yang ingin ditampilkan atau mengatur filter sesuai kebutuhan mereka. TM1 menjadi pelengkap dari Power BI yang digunakan oleh tim *Data Analyst*, khususnya dalam penyediaan data yang kompleks dan mendetail bagi user bisnis di lingkungan Kawan Lama Group.

3.2.1.2 Alur Divisi

Tim *Data Engineer* mengelola alur proses data secara menyeluruh mulai dari pengambilan data mentah hingga penyediaan data siap pakai untuk keperluan analitik bisnis. Proses ini dijalankan melalui tahapan ETL yang terstruktur dan melibatkan sejumlah komponen serta tools teknis yang telah distandarisasi oleh perusahaan. Alur kerja proses ETL dalam pengelolaan data dapat dilihat pada Gambar 3.5, mulai dari sumber data hingga ke tahap akhir dalam bentuk *data mart*. Proses diawali dengan pengambilan data dari berbagai sumber, seperti *database* maupun *File Transfer Protocol* (FTP). Jika ukuran data terlalu besar, maka data akan di-split menjadi beberapa file CSV. Seluruh file hasil ekstraksi kemudian disimpan sementara dalam bentuk format parquet di *bucket* sementara pada Google Cloud Storage (GCS). Selanjutnya, proses transformasi data dilakukan berdasarkan

karakteristik masing-masing sumber, karena setiap jenis sumber data membutuhkan perlakuan yang berbeda. Proses ini mencakup *data cleansing*, *data type mapping*, serta penerapan partisi data. Hasil dari proses transformasi ini kemudian disimpan di "Raw" *bucket* pada GCS.



Gambar 3.5 Alur Proses Pengolahan Data Divisi *Data Engineer*

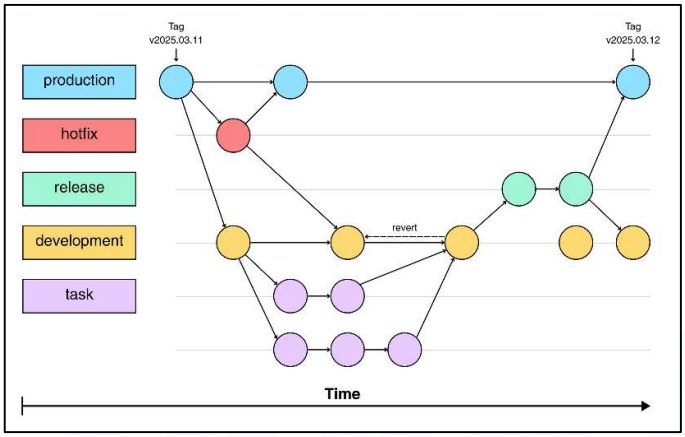
Setelah disimpan di *bucket* GCS, data dilanjutkan dengan proses *load* ke *landing zone* di BigQuery dalam bentuk *temporary table* atau tabel sementara. Data yang dimasukkan ke *landing zone* merupakan gabungan antara data lama yang mengalami perubahan *(changes)* dan data baru. Setiap unit bisnis (*Business Unit*/BU) memiliki *landing zone* masing-masing, sehingga data akan disimpan pada tabel sesuai dengan unit bisnisnya. Data dari *landing zone* kemudian akan digabungkan *(merge)* ke dalam *data lake* berdasarkan BU yang bersangkutan. Tabel-tabel dalam *data lake* ini menjadi dasar pembentuk *data warehouse*, di mana struktur data terbagi menjadi dua jenis, yakni tabel *dimension* dan *fact*. *Data warehouse* berperan sebagai konsolidasi dari keseluruhan data perusahaan, yang memberikan pandangan atas seluruh informasi bisnis yang tersedia. Setelah *data warehouse* terbentuk, dibuatlah penyimpanan data dalam bentuk yang lebih ringkas dan spesifik, yaitu *data mart*. *Data mart* berisi data yang telah disaring dan disesuaikan, serta diambil dari *data lake* maupun *data warehouse*, yang kemudian dimanfaatkan oleh tim *Data Analyst* untuk kebutuhan analitik bisnis dan penyusunan *business intelligence* (BI).

Seluruh proses, mulai dari tahapan *extract* hingga penyimpanan akhir dalam bentuk *data mart*, dijalankan secara otomatis dengan memanfaatkan Apache Airflow sebagai alat orkestrasi data. Airflow berperan sebagai *"orchestrator"* utama yang mengatur dan menjadwalkan *pipeline* data agar dapat berjalan secara terstruktur, efisien, dan sesuai dengan waktu yang telah ditentukan. Setiap proses dalam alur ETL dapat dikelola melalui rangkaian task yang saling terhubung dan dapat dipantau melalui *user interface* secara langsung, sehingga memudahkan tim dalam melakukan *troubleshooting* apabila terjadi kegagalan proses.

Untuk mendukung pengembangan dan pemeliharaan *pipeline* tersebut, seluruh *script* dan konfigurasi yang dijalankan di Airflow disimpan dalam repositori GitLab. Repositori ini terbagi ke dalam dua versi utama, yaitu Development (Dev) dan Production (Prod). Versi Dev digunakan sebagai lingkungan pengembangan, di mana para *engineer* dapat menulis, menguji, dan melakukan eksperimen terhadap kode-kode *pipeline* tanpa memengaruhi jalannya proses produksi. Setelah kode pada versi Dev dinyatakan stabil dan berhasil melalui proses *testing*, maka kode tersebut akan di-*merge* ke dalam *branch* Production, sehingga dapat dijalankan secara resmi di lingkungan operasional perusahaan. Pemisahan versi tersebut bertujuan untuk menjaga kualitas dan stabilitas sistem serta mendukung pengembangan fitur baru yang lebih fleksibel dan terkontrol.

Gambar 3.6 menunjukkan alur kerja Git yang diterapkan oleh divisi *Data Engineer* Kawan Lama Group dalam pengelolaan *source code* melalui GitLab. Alur dimulai dari *branch* utama (Production) yang mencerminkan kondisi sistem yang stabil dan telah siap digunakan. Untuk setiap pengembangan fitur baru, akan dibuat *branch* turunan dari Production maupun Development. Setiap alur task mewakili pekerjaan spesifik yang dilakukan oleh individu, seperti penambahan logika, integrasi sumber data baru, atau perbaikan kecil. Setelah pengembangan di *branch* Task selesai, perubahan akan digabungkan ke *branch* Development menggunakan untuk

pengujian integrasi secara menyeluruh. Jika perubahan dari *branch* Task yang telah digabungkan tidak sesuai atau mengalami error, maka branch dapa di-*revert* ke *commit* sebelum *branch* Task tersebut digabungkan. Apabila diperlukan, misalnya terdapat *bug* kritis yang terjadi di Production, maka *branch* Hotfix akan dibuat dari Production. Perubahan *hotfix* bersifat mendesak dan setelah diuji, akan digabungkan langsung ke Production serta Development agar seluruh jalur pengembangan tetap sinkron. Sementara itu, *branch* Release digunakan sebagai tahap finalisasi sebelum masuk ke Production. Branch ini berfungsi sebagai tempat validasi akhir untuk memastikan semua fitur dalam satu siklus pengembangan sudah layak rilis. Proses *release* ini dilakukan setiap hari kerja menggunakan Tag dengan format tanggal sebagai version release.



Gambar 3.6 Alur GitLab Divisi *Data Engineer*

3.2.2 Backfill Data

Backfill data adalah proses pengisian ulang data ke dalam suatu sistem penyimpanan data, seperti *database* atau *data lake*, untuk memastikan kelengkapan dan konsistensi data historis yang sebelumnya belum tersedia atau tidak terekam dengan benar. Dalam divisi *Data Engineer*, *backfill* merupakan tugas yang sangat penting untuk menjamin bahwa seluruh data yang dibutuhkan oleh sistem *downstream*, seperti *dashboard* analitik atau sistem pelaporan, dapat berjalan dengan data yang lengkap dan akurat. Proses ini dilakukan secara *batch* dan memerlukan pemahaman yang baik terhadap struktur tabel, logika

bisnis, serta dependensi antar data agar tidak terjadi duplikasi atau ketidaksesuaian.

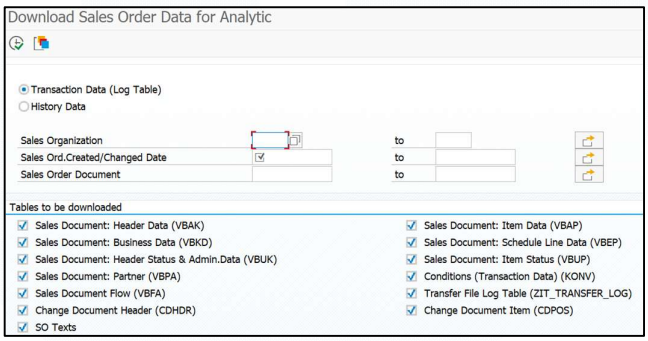
Kebutuhan akan *backfill* data muncul karena adanya data yang tidak terekam dengan baik akibat keterlambatan *ingestion*, kegagalan job harian, atau perubahan pada skema tabel dan sumber data, contohnya seperti ketika terdapat perubahan kolom dalam suatu tabel, atau ketika data historis sebelumnya tidak tersimpan dengan baik karena keterbatasan sistem lama. Dalam beberapa kasus, data yang masuk ke sistem hanya dimulai dari tanggal implementasi *pipeline* baru, sehingga untuk keperluan analisis tren jangka panjang, data dari periode sebelumnya harus diisikan ulang secara manual atau melalui *script* otomatis. Oleh karena itu, *backfill* bukan hanya proses teknis biasa, melainkan juga melibatkan proses verifikasi dan rekonsiliasi terhadap sumber data asli.

Salah satu alasan dilakukannya *backfill* adalah karena adanya proyek besar migrasi data dari platform sebelumnya menuju Google BigQuery sebagai platform utama untuk penyimpanan dan analisis data. Migrasi ini membutuhkan data yang identik antara sistem lama dan sistem baru, baik dalam hal jumlah record, nilai isi kolom, maupun format waktu dan tipe data. Maka dari itu, tim *Data Engineer* harus melakukan proses *backfill* untuk mengisi data lama yang sebelumnya disimpan di platform lain agar data tersebut tersedia di BigQuery. Hal ini dilakukan sering kali dalam beberapa iterasi demi konsistensi total dan keberhasilan proses migrasi. Kebutuhan *backfill* juga muncul dari sisi pengguna data, dalam hal ini tim *Analytics Engineer*. Mereka membutuhkan data dari berbagai sumber untuk dimasukkan ke dalam *cube* TM1, yang digunakan untuk menyusun laporan keuangan, proyeksi, atau analisis performa bisnis. Karena TM1 membutuhkan data yang akurat dan lengkap hingga level granular tertentu, maka ketika terdapat kekurangan data historis, tim *Data Engineer* akan melakukan *backfill* untuk mengisi kekosongan tersebut agar TM1 dapat menghasilkan output yang valid.

Terdapat beberapa jenis *backfill* yang telah dilakukan, di antaranya:

A. SFTP SAP

Backfill jenis ini bertujuan untuk mengisi data tabel-tabel yang bersumber dari sistem SAP. Proses ini dilakukan guna mengisi ulang data historis dari sistem SAP ke dalam *data lake* BigQuery untuk beberapa BU tertentu. Kegiatan ini biasanya dilakukan apabila terdapat kekosongan data, kebutuhan penyesuaian terhadap tabel, atau dalam rangka penyamaan data antar platform, khususnya saat dilakukan integrasi atau konsolidasi dari sistem sumber ke dalam arsitektur data modern berbasis *cloud*.



Gambar 3.7 Tampilan Penurunan Tabel SAP

Proses *backfill* dimulai dengan melakukan penurunan data dari SAP menuju server SFTP. Pengambilan data ini dilakukan berdasarkan konfigurasi tiap BU seperti pada Gambar 3.7, yaitu mencakup jenis tabel yang akan diambil, kode BU yang dituju, serta rentang tanggal historis dari data yang ingin diisi ulang. Setelah data berhasil diturunkan ke server SFTP, proses selanjutnya adalah menjalankan rangkaian task dalam sebuah DAG yang telah tersedia pada platform orkestrasi data, yaitu Apache Airflow. DAG ini dijalankan secara manual di lingkungan Airflow Production karena bersifat *ad-hoc* dan memiliki skenario khusus tergantung jenis data yang diolah.

Proses ETL yang dilakukan untuk *backfill* SAP ini terdiri dari beberapa tahapan utama yang dapat dilihat pada Gambar 3.3, di antaranya:

1. *Extract and split files*

   Pada tahap ini, sistem akan mengekstrak semua file yang telah diturunkan ke server SFTP sesuai dengan konfigurasi yang telah

ditetapkan sebelumnya, seperti jenis tabel dan kode BU. Proses ini juga mencakup mekanisme pembagian file *(split)* menjadi beberapa bagian yang lebih kecil apabila ukuran file melebihi ambang batas tertentu, yakni 200 MB. Tujuan dari mekanisme *split* ini adalah untuk menghindari beban pemrosesan yang terlalu besar pada tahap berikutnya dan mencegah kegagalan saat pemuatan file dalam satu waktu. File yang telah diekstrak kemudian disimpan sementara di GCS dalam format CSV. *Bucket* ini berfungsi sebagai tempat penampungan awal (buffer) sebelum data dimasukkan ke tahap transformasi.

2. *Data transformations*

   Setelah file berhasil diekstrak, tahapan berikutnya adalah melakukan transformasi data. Proses ini mencakup transformasi data untuk memastikan bahwa data yang masuk ke dalam data lake memiliki format, kualitas, dan struktur yang sesuai standar. Transformasi pertama yang dilakukan adalah *data cleansing*, yaitu pembersihan data dari nilai-nilai yang kosong atau tidak konsisten, serta penanganan *regex*. Selanjutnya, dilakukan pemetaan tipe data dengan mengacu pada konfigurasi header yang telah disiapkan sebelumnya dalam bentuk file Excel. File header ini berisi metadata yang menunjukkan tipe data sebenarnya dari masing-masing kolom, serta bagaimana tipe tersebut seharusnya direpresentasikan di lingkungan BigQuery. Misalnya, kolom bertipe *string* yang menyimpan angka perlu diubah ke format numerik atau *object* sesuai konteksnya. Hasil dari transformasi ini kemudian disimpan kembali ke GCS, namun kali ini dalam format parquet yang lebih efisien dan sesuai dengan kebutuhan analitik skala besar.

3. *Load to landing zone*

   Tahap ketiga dalam alur ETL yaitu proses pemuatan hasil transformasi ke dalam *landing zone* di BigQuery. *Landing zone* ini berfungsi sebagai area *staging* awal, di mana data dapat ditinjau

kembali sebelum diproses lebih lanjut. Pada tahap ini, file berformat parquet yang sebelumnya disimpan di GCS akan dimuat ke dalam tabel sementara di BigQuery dengan tetap mempertahankan struktur dan tipe data hasil transformasi.

4. *Load to data lake*

```
CREATE TABLE IF NOT EXISTS `project`.`dataset`.`bu_table`
AS
SELECT columns
FROM `project`.`landingzone`.`table_changes` LIMIT 1;
ALTER TABLE `project`.`dataset`.`bu_table`
ADD COLUMN IF NOT EXISTS created_at DATETIME;
ALTER TABLE `project`.`dataset`.`bu_table`
ADD COLUMN IF NOT EXISTS updated_at DATETIME;
ALTER TABLE `project`.`dataset`.`bu_table`
ADD COLUMN IF NOT EXISTS is_deleted BOOL;
ALTER TABLE `project`.`dataset`.`bu_table`
ADD COLUMN IF NOT EXISTS deleted_at DATETIME;
MERGE INTO `project`.`dataset`.`bu_table` AS `tgt`
USING (
SELECT * EXCEPT (`rn`)
FROM (
SELECT columns , ROW_NUMBER() OVER ( PARTITION BY `column` ORDER BY `upload_at` DESC ) AS `rn`
FROM `project`.`landingzone`.`table_changes`
WHERE TRUE
) WHERE `rn` = 1
) AS `src` ON ( `tgt`.`column` = `src`.`column` )
WHEN MATCHED THEN UPDATE SET `tgt`.`columns` = `src`.`columns`, `tgt`.`updated_at` = DATETIME('2025-05-19 05:35:00')
WHEN NOT MATCHED THEN INSERT ( columns, `created_at`, `updated_at`, `is_deleted`, `deleted_at` ) VALUES ( columns, DATETIME('2025-05-19 05:35:00'),
DATETIME('2025-05-19 05:35:00'), FALSE, NULL )
```

Gambar 3.8 *Query* SQL untuk Pemuatan Data ke *Data Lake*

Setelah melewati tahap verifikasi dan validasi di *landing zone*, data akan dipindahkan ke dalam area *data lake* yang menjadi repositori utama di BigQuery. Proses pemuatan dilakukan dengan mengeksekusi *query* SQL yang telah dikonfigurasi seperti pada Gambar 3.8 untuk mengatur struktur akhir tabel, yaitu dengan melakukan *partitioning* dan penambahan kolom *default* bila diperlukan, melakukan *merge* data dari tabel *landing zone* ke tabel *data lake* untuk menghindari data duplikat, serta memperbarui data yang sudah ada apabila terjadi perubahan pada data tersebut. Tujuan utama dari pemindahan ke data lake adalah untuk memastikan bahwa data yang telah dibersihkan, ditransformasikan, dan diverifikasi dapat tersedia dalam bentuk yang konsisten dan siap digunakan oleh berbagai tim.

B. DWH *Fact Inventory*

Proses *backfill* terhadap data fact_inventory pada *Data Warehouse* (DWH) merupakan komponen penting dalam sistem pelaporan dan analisis persediaan *(inventory)* karena merepresentasikan perubahan kuantitas barang *(movement)*, usia persediaan *(aging)*, serta saldo akhir *(ending*

balance) dari seluruh gudang dan lokasi penyimpanan barang perusahaan. Dalam implementasinya, tabel fact_inventory berkaitan dengan beberapa turunan, seperti *inventory aging* nasional, *inventory aging site*, serta *inventory movement*. Tabel *aging* nasional berisi data usia persediaan dari seluruh gudang atau lokasi di seluruh cabang perusahaan, sementara *aging site* hanya merepresentasikan data *aging* pada satu gudang atau lokasi tertentu. Untuk setiap BU, terdapat set tabel masing-masing yang harus dikelola secara terpisah. Oleh karena itu, proses *backfill* pada level ini memiliki kompleksitas yang cukup tinggi dan membutuhkan ketelitian ekstra.

```
WITH bu_inventory_balance_temp as(
        SELECT
        cast (posting_date as date) as posting_date
        ,store_code
        ,article
        ,storage_location
        ,stock_category
        --,stock_type as stock_type_id
        ,sum (end_qty) as end_qty
        ,sum (end_value_cogs) as end_value_cogs
        FROM gcp-kl-dev-dwh.klg_dl.bu_inventory_balance
        group by 1,2,3,4,5
        --,6
    )
```

Gambar 3.9 *Query* CTE Pemuatan Data *Beginning Balance* ke fact_inventory

```
def gen_fact_inventory(_BU:str,
                       start_date='{start_date}',
                       end_date='{end_date}',
                       _project: str = "                ",
                       _multi_bu_project: str = "                  ",
                       _dl: str = "          ",
                       _dwh: str = "          ") -> str:

    # change
    _bu =
    _BU =
    _CompCode =

    _vQuery_insert = f"""
                Insert into `                          .{_bu}_fact_inventory`
                    (
            period
            ,site_id
            ,product_id
            ,storage_location
            ,flag_stock_category_id
            --,stock_type_id
            ,ending_qty
            ,ending_value_cogs_bw
            ,ending_value_cogs
            ,flag_frequency_id
            ,created_at
            ,updated_at
            ,company_code
```

Gambar 3.10 *Script* Definisi Fungsi untuk *Backfill* fact_inventory

Salah satu hal paling penting dalam *backfill* fact_inventory adalah memastikan bahwa data awal atau *beginning balance* benar dan valid. Hal

ini dikarenakan tabel *inventory* bersifat kumulatif, yakni jika data awal mengandung nilai yang salah, maka seluruh perhitungan selanjutnya, termasuk *movement* harian dan saldo akhir, akan ikut menjadi tidak akurat. Oleh karena itu, tahapan awal sebelum melakukan proses *backfill* adalah memastikan bahwa semua tabel pembentuk fact_inventory telah lengkap dan bebas dari anomali. Tabel-tabel pembentuk ini antara lain adalah data PO *(Purchase Order)* yang terdapat dalam tabel EKPO, data dokumen artikel (MSEG), serta dokumen keuangan seperti *financial document* (BSEG) dan header (BKPF). Ketersediaan dan integritas data pada keempat tabel ini menjadi syarat utama sebelum proses *backfill* terhadap fact_inventory dapat dilakukan.

```
min_date_temp = '2023-12-01'
max_date_temp = '2025-01-31'
min_date = datetime.strptime(min_date_temp, '%Y-%m-%d').date()
max_date = datetime.strptime(max_date_temp, '%Y-%m-%d').date()

while(min_date <= max_date):
    start_date = min_date
    next_month = (start_date + timedelta(days=31)).replace(day=1)
    end_date = (start_date + timedelta(days=31)).replace(day=1) - timedelta(days=1)

    print(f"start_date : {start_date}")
    print(f"end_date : {end_date}")

    query = gen_fact_inventory(_BU='BU',start_date = start_date, end_date=end_date) # change BU
    # print(query)

    try:
        client.query(query).result()
        print(f'executing : {end_date}')
    except Exception as e:
        print(e)
    min_date = next_month
```

Gambar 3.11 *Script Looping* Fungsi gen_fact_inventory

Setelah semua tabel sumber dipastikan siap, proses *backfill* dimulai dengan memuat *beginning balance* dari *data lake* menggunakan *query* khusus untuk memastikan bahwa saldo awal dapat terbentuk secara akurat, berdasarkan kondisi terakhir yang tercatat dalam inventory_balance. Proses ini dilakukan melalui pemanggilan *query* utama untuk memuat data ke dalam tabel fact_inventory yang kemudian dikombinasikan dengan tambahan *Common Table Expression* (CTE) sebagaimana ditunjukkan pada Gambar 3.9. Selanjutnya, proses *backfill* dilakukan secara bertahap berdasarkan periode waktu, yaitu secara bulanan dan harian. Untuk rentang waktu dari awal saldo hingga tiga bulan sebelum waktu terkini, proses

backfill dilakukan dalam periode bulanan. Sementara untuk tiga bulan terakhir hingga tanggal terkini, *backfill* dilakukan harian untuk menjaga tingkat ketelitian dan resolusi data. Proses ini dilakukan dengan bantuan *script* Python yang telah disiapkan pada Google Colab. *Script* dapat dilihat pada Gambar 3.10, di mana *script* ini berisi fungsi untuk mendapatkan *query backfill* fact_inventory yang akan dijalankan, serta kode untuk mengeksekusi *query* secara berulang *(looping)* berdasarkan parameter tanggal yang dikonfigurasi sesuai kebutuhan seperti pada Gambar 3.11. Kode *looping* ini secara dinamis akan memanggil *query* fact_inventory yang sudah dimodifikasi, dan akan terus berjalan hingga tanggal minimum setara atau melebihi tanggal maksimum yang ditentukan.

```
def prereq_aging(_BU:str, _period, _project: str = "              ", _multi_bu_project: str = "              ",_dl='',
    query = f"""select distinct iv.period
                from `{_project}.{_dwh}.{_BU.lower()}_fact_inventory` iv
                    join {_project}.{_dwh}.dim_period dp on iv.period=dp.period
                where cast(iv.updated_at as date) >='{_period}'
                    and (day_of_week=1 or lower(iv.flag_frequency_id)='m')
                order by period asc;
                    """
    return query
```

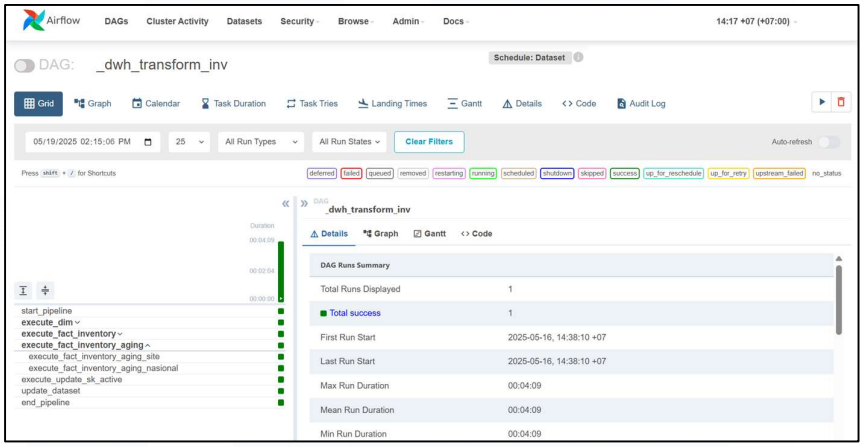
Gambar 3.12 Konfigurasi *Query Prerequisite Inventory Aging*

```
        WHERE
          --  posting_date >= (SELECT min(posting_date) from                                mseg WHERE DATE(updated_at) =
'2025-03-17')
          ##backfill##
          posting_date >= '2025-03-17'
```

Gambar 3.13 Konfigurasi *Query Backfill Inventory Movement*

Adapun untuk proses *backfill* pada tabel turunan seperti inventory aging dan inventory movement. Metode yang digunakan pada dasarnya serupa, namun dengan penyesuaian logika periode di dalam query SQL pada file Python pada repositori lokal. Untuk tabel *inventory aging*, modifikasi *query* dilakukan pada file *prerequisite aging* pada Gambar 3.12, yang mengambil period tabel fact_inventory, di mana period yang diambil dimulai dari sejak *beginning balance* hingga ke waktu terkini. Untuk tabel inventory movement, *query* yang perlu dimodifikasi didapat dari DAG harian untuk BU tersebut, seperti pada Gambar 3.13. *Query* yang telah disesuaikan tersebut kemudian dijalankan melalui DAG yang tersedia untuk DWH pada sistem orkestrasi Airflow lokal yang ditunjukkan pada Gambar 3.14. Karena *aging* dan *movement* hanya bergantung pada perubahan saldo dan waktu

penyimpanan barang, logika perhitungan sudah tertanam di dalam *query*, sehingga eksekusinya bisa dilakukan sekali jalan untuk seluruh periode yang dibutuhkan.



Gambar 3.14 DAG untuk Eksekusi *Script Inventory Aging*

C. TM1

Proses *backfill* dari Data Lake ke sistem TM1 yang dilakukan ditujukan untuk mengisi dua *cube*, yaitu SDSR dan IM-AGING. SDSR merupakan *cube* yang digunakan untuk menarik data transaksi penjualan harian yang terjadi di berbagai cabang. Laporan ini mencakup beberapa nilai numerik seperti kuantitas barang yang terjual *(quantity)*, nilai penjualan *(sales amount)*, dan juga biaya pengiriman *(cost)*. Sementara itu, *cube* IM-AGING merupakan lanjutan dari proses *backfill* pada DWH fact_inventory. Fungsinya adalah untuk menyajikan data usia stok barang berdasarkan kategori umur yang telah ditentukan, seperti 0-30 hari, 31-60 hari, dan seterusnya. Hal ini sangat penting untuk memantau pergerakan persediaan *(inventory movement)* sekaligus menilai risiko penumpukan stok yang terlalu lama tersimpan di gudang.

Proses *backfill* terhadap *cube* SDSR dan IM-AGING dilakukan dengan relatif sederhana, namun tetap memerlukan pemahaman struktur data dan konfigurasi sistem. Proses ini dimulai dengan menjalankan DAG tm1_backfill yang telah tersedia di Airflow Production. Dalam eksekusinya,

pengguna perlu mengisi konfigurasi sesuai kebutuhan, seperti tahun dan company code yang akan dibackfill. Backfill dilakukan per tahun, namun dalam pelaksanaannya, sistem akan membagi proses menjadi per bulan di dalam setiap task DAG. Hal ini bertujuan untuk menjaga efisiensi proses dan menghindari beban data yang terlalu besar dalam satu kali eksekusi. Proses ETL yang digunakan dalam backfill ini terdiri dari beberapa tahapan utama, yaitu:

1. *Extract data*

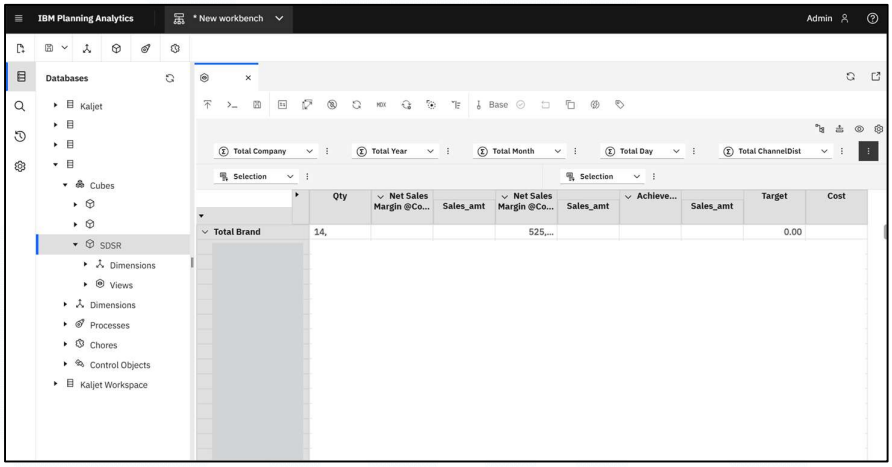
   Proses ETL dimulai dengan tahap extract data, yaitu pengambilan data dari BigQuery dengan menjalankan query yang telah disiapkan sebelumnya di dalam file generate flat. Query ini digunakan untuk mengekstrak data sesuai dengan kebutuhan backfill, baik untuk cube SDSR maupun IM-AGING Inventory. Setelah query dijalankan, hasil ekstraksi data tidak langsung dimasukkan ke sistem downstream, melainkan disimpan terlebih dahulu ke dalam sebuah tabel sementara (temporary table) di BigQuery. Setelah itu, data dari tabel sementara diekspor ke dalam format CSV dan disimpan di *Google Cloud Storage* (GCS) agar dapat dengan mudah diakses dan dibaca oleh sistem TM1 untuk proses input selanjutnya.
2. *Clear cube*

   Tahap berikutnya adalah clear cube, yaitu proses pembersihan cube di TM1 sebelum data baru dimasukkan. Langkah ini bersifat preventif sekaligus kritikal karena fungsinya untuk mencegah duplikasi atau penumpukan data di dalam cube. Tanpa dilakukan proses clear cube, data hasil backfill yang dimasukkan ke TM1 bisa menimpa atau bercampur dengan data lama, yang akan menyebabkan ketidaksesuaian dalam laporan. Pembersihan dilakukan secara selektif, yaitu hanya pada kolom-kolom yang relevan dengan periode dan kode BU yang akan dibackfill, sehingga data lain yang tidak terkait tetap aman dan tidak terhapus. Setelah

proses ini selesai, struktur *cube* akan dalam kondisi bersih dan siap menerima data baru dari tahapan berikutnya.

3. *Insert element*

   Tahap terakhir adalah memasukkan data yang telah dipersiapkan ke dalam cube TM1. Pada tahap ini, file CSV yang tersimpan di GCS dibaca dan dipetakan ke dalam dimensi dan measure sesuai struktur cube. Setiap baris data akan diklasifikasikan berdasarkan beberapa parameter penting seperti waktu (bulan dan tahun), lokasi (BU atau site), serta metrik yang relevan seperti jumlah penjualan, nilai penjualan, atau kategori aging. Proses ini dijalankan oleh *task* khusus dalam DAG Airflow, yang akan memproses data per bulan sesuai dengan konfigurasi yang telah ditentukan sebelumnya. Setelah proses ini selesai, data akan muncul di TM1 dan siap digunakan untuk analisis atau penyusunan laporan bisnis.



Gambar 3.15 Tampilan Tabel *Cube* SDSR pada Platform TM1

Hasil akhir dari proses *backfill* data dapat diakses oleh pengguna melalui platform IBM Planning Analytics, yang berbasis sistem TM1. Platform ini menyediakan tampilan dalam bentuk tabel pivot yang fleksibel dan interaktif seperti pada Gambar 3.15, sehingga *user* dari berbagai unit bisnis untuk melakukan eksplorasi data secara mandiri. Setiap tampilan atau *layout* tabel dalam platform tersebut terdiri dari kolom-kolom dinamis yang

mewakili dimensi dan *measure* yang telah didefinisikan sebelumnya oleh tim *Analytics Engineering*, seperti dimensi waktu (bulan, tahun), *company code*, produk, serta metrik seperti kuantitas penjualan, nilai aging stok, atau nilai biaya distribusi. Data tersebut juga dapat dilihat dalam bentuk grafik batang.

D. Cohesive Loyalty

Proyek Cohesive Loyalty merupakan bagian dari inisiatif strategis yang dijalankan oleh divisi *Customer Relationship Management* (CRM), yang memiliki peran penting dalam pengelolaan data pelanggan untuk kebutuhan kampanye loyalitas, personalisasi layanan, serta pengambilan keputusan berbasis data. Data dari proyek ini perlu dilakukan *backfill* agar seluruh data historis dan terkini yang berkaitan dengan aktivitas pelanggan dapat ditransformasikan dan dimigrasikan secara utuh dan akurat dari sumber penyimpanan awal hingga ke sistem target yang digunakan untuk analitik dan pelaporan. Data yang menjadi objek *backfill* berbentuk file parquet yang telah disimpan di Amazon S3 Browser. Proses ini dilakukan secara bertahap dalam dua fase utama. Fase pertama adalah *backfill* dari S3 Browser ke *data lake*, dan fase kedua adalah pemindahan dari *data lake* ke sistem basis data PostgreSQL. Pada fase pertama, pemilihan data dilakukan secara selektif dengan menyaring file berdasarkan kode BU yang dibutuhkan. Setiap file parquet diperiksa untuk memastikan bahwa tidak mengandung baris kosong, karena hanya file yang memiliki data sah yang akan diproses lebih lanjut. File yang memenuhi kriteria tersebut kemudian disalin ke folder baru di S3 Browser, sebagai bentuk isolasi terhadap data yang valid.

Langkah selanjutnya adalah mengubah prefix atau jalur file pada konfigurasi script ETL, yang terdapat dalam repositori internal, agar sistem ETL dapat membaca file dari folder baru tersebut. Setelah konfigurasi selesai, file dari S3 Browser dipindahkan ke GCS, khususnya ke direktori sementara melalui perintah yang dieksekusi pada Google CloudShell. Folder ini bersifat sementara karena berfungsi sebagai *staging area* sebelum

file diproses oleh *pipeline* Airflow. Seluruh proses ini memastikan bahwa hanya data yang telah lolos validasi dan memiliki struktur yang sesuai yang akan dimasukkan ke dalam sistem data lake perusahaan. Setelah file berada di GCS, task Airflow pada lingkungan Production dijalankan untuk mengeksekusi *pipeline* ETL dan memuat data ke *data lake*.

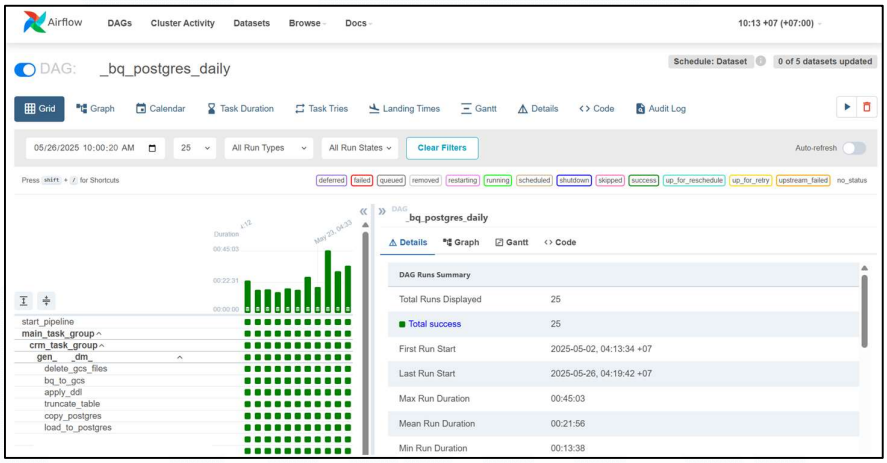
Gambar 3.16 *View* Tabel pada BigQuery

```
def gen_   _dm_transactionattribute(
    _businessUnit,
    _period,
    _multi_bu_project,
    _runType="daily",
    _project="               ",
    _dl="      ",
    _dwh="       ",
    bu_list=["   "],
):
    table_name = "transactionattribute"
    pg_target_table = f"temp_{_businessUnit.lower()}_{table_name}"
    main_table = f"{_businessUnit.lower()}_{table_name}"
    bq_query_extract = f"""
                        SELECT

                        FROM `{_project}.{_dl}.{_businessUnit.lower()}_{table_name}`
                        WHERE date(updated_at) = '{_period}'
                        """
    pg_query_load = f"""
                    DELETE FROM {main_table} main
                    USING {pg_target_table} tmp
                    WHERE main.id = tmp.id;

                    INSERT INTO {main_table}
                    (SELECT * FROM {pg_target_table});

                    TRUNCATE TABLE {pg_target_table};
                    """

    return (
        main_table,
        table_name,
        pg_target_table,
        bq_query_extract,
        pg_query_load,
        bu_list,
    )
```

Gambar 3.17 *Script Data Mart* untuk DAG BigQuery ke PostgreSQL

Setelah data berhasil masuk ke *data lake*, lanjut ke fase kedua, yaitu *backfill* dari *data lake* ke PostgreSQL, yang merupakan basis data yang digunakan untuk proyek Cohesive. Langkah pertama dalam proses ini

adalah memastikan bahwa *view* atau tampilan tabel yang akan digunakan untuk *backfill* telah tersedia. Jika belum, maka perlu dilakukan pembuatan view baru berdasarkan data di *data lake*. Pada Gambar 3.16, *view* untuk tabel berisikan *query* yang berfungsi untuk menggabungkan jenis tabel yang sama dari beberapa BU. *View* ini dirancang untuk memudahkan proses transformasi dan pemuatan data ke PostgreSQL. Kemudian, dibuat *script data mart* untuk mengambil data dari *view* tersebut dan memuat ke dalam basis data, yang ditampilkan pada Gambar 3.17. Selanjutnya, task DAG untuk *backfill* dari BigQuery ke PostgreSQL dijalankan melalui Airflow, dengan konfigurasi harian *(daily)*. Adapun langkah lanjutan, yakni melakukan pengecekan jumlah baris pada akhir proses untuk memastikan bahwa jumlah data di PostgreSQL sesuai dengan yang ada di *data lake*, guna menjamin konsistensi dan integritas data.



Gambar 3.18 DAG *Daily Data Lake* ke PostgreSQL

Gambar 3.18 menunjukkan DAG untuk melakukan *backfill* data dari *data lake* ke PostgreSQL, di mana task-task yang ada pada DAG tersebut merepresentasikan proses ETL yang dilakukan, yakni:

1. *Delete* GCS *files*

   Proses pertama yang dilakukan dalam ETL adalah memastikan tidak ada file duplikat yang tersimpan di direktori GCS, karena file yang dihasilkan dari proses sebelumnya bisa saja memiliki isi data yang

sama, namun dengan nama file yang berbeda. Jika tidak dihapus, file-file tersebut bisa terbaca sebagai data baru dan berpotensi menyebabkan duplikasi saat dimuat ke dalam PostgreSQL. Langkah ini berfungsi sebagai pencegahan terhadap ketidakkonsistenan data.

2. *BigQuery to* GCS

   Setelah direktori bersih dari file duplikat, langkah selanjutnya adalah melakukan ekstraksi data dari BigQuery. Proses ini dilakukan dengan menjalankan *query* SQL pada variabel bq_query_extract yang sudah disiapkan dalam *script data mart* pada Gambar 3.17, di mana *query* tersebut digunakan untuk mengambil data dari skema *data lake* yang sebelumnya telah terstruktur rapi. Hasil dari *query* ekstrak tersebut kemudian diekspor ke dalam format CSV dan disimpan ke dalam folder GCS yang telah ditentukan. Format CSV dipilih karena ringan, mudah diproses, dan kompatibel dengan proses *load* ke PostgreSQL. Proses ini merupakan tahapan transisi dari sistem *cloud-based* (BigQuery) ke *file-based* (GCS), yang kemudian menjadi sumber input bagi proses *loading* berikutnya.

3. *Apply* DDL

   Setelah data berada di GCS dalam bentuk CSV, langkah berikutnya adalah membuat *temporary table* di PostgreSQL dengan menggunakan perintah DDL *(Data Definition Language)*. Tabel sementara ini berfungsi sebagai tempat *staging* untuk menampung data hasil ekstraksi sebelum dipindahkan ke tabel utama. Dalam proses ini, struktur *temporary table* harus disesuaikan dengan struktur file CSV yang akan dimuat, termasuk mendefinisikan tipe data yang sesuai untuk setiap kolom, seperti tipe teks, angka, tanggal, dan sebagainya.

4. *Truncate table*

   Sebelum data baru dimuat ke dalam *temporary table*, langkah *truncate* atau pengosongan isi tabel dilakukan terlebih dahulu. Hal

ini bertujuan untuk memastikan bahwa tidak ada sisa data dari proses sebelumnya yang dapat menyebabkan konflik atau duplikasi. *Temporary table* dikondisikan agar benar-benar bersih dan siap menampung data baru dari file CSV.

5. *Copy to postgres*

   Setelah *temporary table* siap, proses berikutnya adalah memuat file CSV dari GCS ke dalam temporary table PostgreSQL. Data dari CSV akan dipetakan satu per satu ke kolom yang ada di temporary table, dan setiap baris data akan diolah sesuai format yang telah didefinisikan. Langkah ini memastikan bahwa data sudah tersimpan di dalam PostgreSQL dan siap untuk diproses lebih lanjut.

6. *Load to postgres*

   Langkah terakhir adalah memindahkan data dari temporary table ke main table di PostgreSQL menggunakan query pada variabel pg_query_load dalam script data mart pada Gambar 3.17, yang menggunakan DELETE berdasarkan *Primary Key* (PK) untuk menghindari duplikat sehingga tidak mengganggu struktur atau integritas tabel utama. Setelah langkah ini selesai, data pelanggan dari Cohesive Loyalty milik divisi CRM resmi tersedia di sistem operasional dan dapat digunakan oleh berbagai tim untuk keperluan analitik, operasional, maupun bisnis.

Proses *backfill* data dari S3 Browser hingga ke PostgreSQL dalam proyek Cohesive Loyalty tidak hanya berfungsi untuk memindahkan data, tetapi juga sebagai bagian dari proses data *ingestion*. Hal ini karena sebagian besar tabel yang dimuat sebelumnya belum pernah tersedia di dalam sistem penyimpanan perusahaan. Artinya proses ini menjadi langkah awal integrasi data dari sumber eksternal ke dalam ekosistem data perusahaan.

3.2.3 *Ingestion* Tabel

Ingestion tabel merupakan proses sistematis untuk memindahkan data dari berbagai sumber ke dalam *data lake*, maupun ke sistem penyimpanan lainnya

yang digunakan oleh divisi-divisi dalam Kawan Lama Group, seperti *database* SQL atau TM1. Selain memindahkan data, tujuan proses *ingestion* menciptakan integrasi antarplatform, dimana data akan digunakan secara seragam dan konsisten dalam berbagai kebutuhan analisis, pelaporan, serta pengambilan keputusan berbasis data. Dengan adanya *ingestion*, data yang semula tersebar dan tidak terstruktur dapat disatukan dalam satu ekosistem penyimpanan yang mendukung efisiensi dan efektivitas proses analitik. *Ingestion* data menjadi komponen fundamental dalam sistem *pipeline* data modern karena berfungsi sebagai titik masuk utama bagi data mentah ke dalam ekosistem pemrosesan data perusahaan. Proses ini memastikan bahwa setiap sumber data, baik dari sistem ERP seperti SAP, penyimpanan FTP, maupun basis data internal lainnya, dapat dikonversi dan disesuaikan dengan standar penyimpanan dan transformasi data yang berlaku. Salah satu tantangan utama dalam proses ini adalah heterogenitas struktur dan format data dari berbagai sumber, sehingga, *ingestion* tidak hanya berperan sebagai proses transfer data, tetapi juga sebagai tahap awal dalam memastikan kesesuaian skema, konsistensi tipe data, serta kualitas data secara keseluruhan. Dalam pelaksanaan teknis di lingkungan perusahaan, *ingestion* dilakukan secara periodik, baik dalam frekuensi harian, mingguan, maupun sesuai dengan kebutuhan spesifik dari *user*.

3.2.3.1 *Ingestion* Sumber Data ke *Data Lake*

Ingestion dari sumber data ke *data lake* merupakan langkah awal dalam alur pengolahan data perusahaan. Sumber data yang digunakan dalam proses ini meliputi sistem SAP, FTP Server, serta database lain yang digunakan oleh berbagai divisi di dalam perusahaan. Proses ini bertujuan untuk memusatkan seluruh data dari sistem operasional ke dalam satu lingkungan penyimpanan besar berbasis *cloud*, yaitu *data lake*, yang kemudian akan digunakan untuk analisis lebih lanjut. Jenis sumber *ingestion* yang dilakukan selama masa kerja magang, antara lain:

A. SAP dan FTP

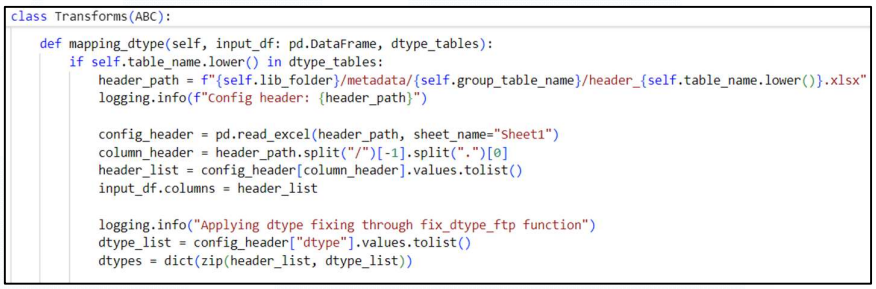
Ingestion data dari SAP dan FTP memiliki proses yang serupa karena keduanya menghasilkan file dalam format CSV tanpa skema atau metadata yang melekat secara eksplisit. Oleh karena itu, langkah pertama yang harus dilakukan adalah menyiapkan file Excel yang berisi nama kolom serta tipe data untuk setiap atribut dalam tabel. File Excel ini berfungsi sebagai panduan header dan pemetaan tipe data yang akan digunakan dalam proses transformasi dan pemuatan data. File tersebut disimpan di folder metadata, agar *pipeline* ETL dapat membaca dan menggunakan informasi tersebut saat memetakan struktur data ke dalam format yang sesuai di BigQuery. Hal ini penting untuk memastikan bahwa setiap kolom ditangani dengan tipe data yang benar, dan tidak terjadi kesalahan parsing atau ketidaksesuaian skema saat data dimuat ke *landing zone* dan *data lake*.

	A	B	C	D
1	header_adhoc_target_sqm	dtype		
2	period	str		
3	store_id	str		
4	deptgrp_name	str		
5	net_sqm	float64		
6	nsa	float64		
7	gross_sqm	float64		
8	created_at_tm1	str		
9	year_month	str		
10				

Gambar 3.19 File Excel untuk Header Tabel

Langkah pertama adalah membuat sebuah file Excel yang berisi daftar nama kolom serta tipe data yang sesuai, seperti pada Gambar 3.19. File Excel ini berfungsi sebagai definisi *schema* eksternal dan akan digunakan untuk proses pemetaan tipe data saat *ingestion* berlangsung. File Excel tersebut harus disimpan pada folder metadata, agar *pipeline* ETL dapat mengenali dan membaca informasi schema secara otomatis saat memetakan struktur data ke dalam format yang sesuai di BigQuery. Hal ini penting untuk memastikan bahwa setiap kolom ditangani dengan tipe data yang benar, dan tidak terjadi kesalahan parsing atau ketidaksesuaian skema saat data dimuat ke landing zone dan data lake.

File Excel yang berisi header tabel akan dibaca dan digunakan tahap transformasi, di mana potongan *script* pada Gambar 3.20 membaca header dari *path* yang disimpan pada repositori dalam variabel header_path.



```
class Transforms(ABC):

    def mapping_dtype(self, input_df: pd.DataFrame, dtype_tables):
        if self.table_name.lower() in dtype_tables:
            header_path = f"{self.lib_folder}/metadata/{self.group_table_name}/header_{self.table_name.lower()}.xlsx"
            logging.info(f"Config header: {header_path}")

            config_header = pd.read_excel(header_path, sheet_name="Sheet1")
            column_header = header_path.split("/")[-1].split(".")[0]
            header_list = config_header[column_header].values.tolist()
            input_df.columns = header_list

            logging.info("Applying dtype fixing through fix_dtype_ftp function")
            dtype_list = config_header["dtype"].values.tolist()
            dtypes = dict(zip(header_list, dtype_list))
```

Gambar 3.20 *Script* Pemetaan Tipe Data pada Tahap Transformasi ETL

Setelah file metadata selesai disiapkan, langkah berikutnya adalah menambahkan konfigurasi *ingestion* untuk tabel yang akan dimuat. Konfigurasi ini dituliskan dalam format YAML yang dikelompokkan berdasarkan BU dan disimpan dalam folder khusus konfigurasi dalam repositori *pipeline*. Dalam konfigurasi tersebut, perlu adanya penentuan untuk beberapa parameter penting seperti apakah tabel tersebut menggunakan partisi, serta jenis *insert* yang akan digunakan. Terdapat tiga jenis *insert* yang biasa digunakan, di antaranya:

1. *Merge*

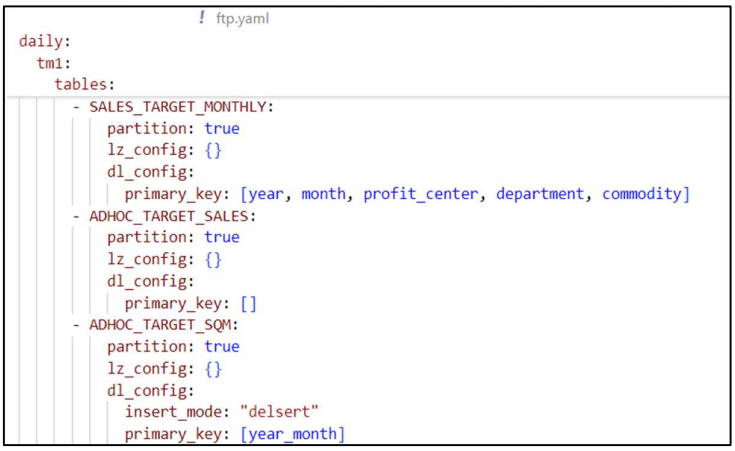
   Jenis ini digunakan jika ingin menyimpan data baru hanya jika belum ada data dengan *Primary Key* (PK) yang sama di dalam tabel. Jika PK pada data baru sudah ada di tabel, maka data tidak akan ditambahkan (hanya *merge* jika berbeda). Untuk menggunakan merge, konfigurasi harus menetapkan kolom PK dengan jelas.
2. *Truncate-Insert (Trunsert)*

   Dalam jenis ini, setiap kali data baru masuk, seluruh data lama dalam tabel akan dihapus *(truncate)*, lalu data baru dimuat sepenuhnya. Ini berguna jika ingin selalu menyimpan snapshot

terbaru dari data. Untuk mengaktifkan *trunsert*, bagian PK pada konfigurasi YAML harus dikosongkan.

3. *Delete-Insert (Delsert)*

   Jenis ini serupa dengan *merge*, namun lebih agresif. Jika ditemukan data baru dengan PK yang sama, data lama akan dihapus terlebih dahulu lalu data baru dimasukkan. Jika tidak ditemukan PK yang sama, data langsung dimuat. Untuk mengaktifkan *delsert*, perlu menetapkan PK dan menambahkan insert_mode: delsert dalam konfigurasi.



Gambar 3.21 Konfigurasi Tabel FTP pada File YAML

Contoh dari pengaplikasian jenis *insert* dapat dilihat pada Gambar 3.21, di mana tabel-tabel tersebut secara berurutan merupakan tabel jenis *merge*, *trunsert*, dan *delsert*. Setelah konfigurasi pada file YAML diperbarui, seluruh perubahan disimpan dan dapat diuji coba terlebih dahulu pada Airflow Local. Dalam Airflow, DAG *ingestion* akan dijalankan sesuai dengan jenis sumber data yang ditentukan. Jika pada saat proses ingestion ternyata tabel belum tersedia baik di *landing zone* maupun *data lake*), maka *pipeline* akan membuat tabel tersebut berdasarkan struktur metadata yang telah ditentukan sebelumnya. Hal ini membuat proses *ingestion* menjadi lebih fleksibel dan mudah untuk diskalakan, terutama dalam proyek-proyek data integrasi yang melibatkan banyak tabel dari berbagai sumber data.

B. *Database* Diversion

Ingestion data dari sumber database Diversion berbasis MySQL dilakukan untuk memindahkan data atau tabel terkait ADF *report*, yaitu administrasi barang *defect* yang berasal dari sistem operasional gudang. Data tersebut memiliki peran penting dalam mendukung evaluasi kualitas logistik serta pemantauan efektivitas penanganan barang yang mengalami kerusakan atau cacat produksi. Akses data dilakukan melalui aplikasi DBeaver dengan menambahkan koneksi ke *database* sumber menggunakan kredensial dan informasi host yang telah disediakan oleh pemilik sistem. Koneksi ini kemudian dilengkapi dengan konfigurasi tambahan agar dapat dikenali oleh *pipeline* Airflow. Pada tahap awal, daftar tabel yang diperlukan harus dituliskan dalam berkas konfigurasi YAML seperti pada Gambar 3.22 untuk mengenali struktur serta metode pemuatan data yang sesuai agar dapat teridentifikasi. Informasi dalam konfigurasi tersebut meliputi nama tabel, status partisi, serta jenis insert yang disesuaikan dengan karakteristik dan kebutuhan masing-masing tabel, di mana jenis insert yang digunakan adalah *merge*.

! sql.yaml

```
2   daily:
3     mssql:
230     server_8:
231       - adf_h:
232           partition: true
233           lz_config: {}
234           dl_config:
235             primary_key: [company_code, no_adf, company]
236       - adf_d:
237           partition: true
238           lz_config: {}
239           dl_config:
240             primary_key: [no_adf, stockcode]
241       - adf_d_history:
242           partition: true
243           lz_config: {}
244           dl_config:
245             primary_key: [no_adf, stockcode, DELETED_DATE]
```

Gambar 3.22 Konfigurasi YAML Tabel Terkait ADF *Report*

Selain konfigurasi, proses ini juga melibatkan penyusunan *query* SQL untuk setiap tabel yang akan di-*ingest*. *Query* ditunjukkan pada Gambar 3.23, yang berisi SELECT *statement* dan kondisi untuk delta tarikan data yang dibutuhkan. Seluruh rangkaian ini kemudian perlu

diuji agar tidak terjadi kesalahan pada *pipeline* menjalankan DAG *ingestion* SQL berdasarkan konfigurasi yang telah ditentukan. Ketika proses berjalan dengan lancar, tabel hasil *ingestion* akan muncul di *data lake* dan siap digunakan.

Write_Off.sql

```
SELECT
    wo.WO_No,
    wo.Doc_No,
    wo.status_wo,
    wo.qty_wo,
    wo.Photo_1,
    wo.Photo_2,
    wo.Photo_3,
    wo.Photo_4,
    wo.createdBy,
    wo.createdDate,
    wo.approvedBy,
    wo.approvedDate
FROM Write_Off wo
WHERE CAST(wo.createdDate AS DATE) >= '{{ data_interval_start.in_timezone("Asia/Jakarta").to_date_string() }}';
```

Gambar 3.23 *Query* Ekstrak Data ADF *Report*

C. *Database* Prime

Ingestion data dari *database* Prime yang berbasis Microsoft SQL Server (MSSQL) bertujuan untuk memproses data laporan piutang pelanggan atau *Account Receivable* (AR) *Aging*. Laporan ini berisi informasi mengenai status tagihan, mulai dari informasi kontrak, nilai tagihan, umur piutang, hingga status tindak lanjut penagihan. Tidak seperti *ingestion* dari *database* SQL yang biasa dilakukan sebelumnya, data pada sistem Prime tidak diperoleh langsung dari tabel, melainkan dari *Stored Procedure* (SP). SP digunakan karena mencakup serangkaian logika bisnis kompleks yang telah dikemas dalam fungsi siap pakai oleh pemilik sistem. Oleh karena itu, untuk *ingest* data dari SP memerlukan penyesuaian tertentu dalam alur ekstraksi data.

report_aging.sql

```
DECLARE @CutOffDate DATE = DATEADD(DAY, -1, CAST(GETDATE() AS DATE));
EXEC [dbo].[sp_rpt_AgingView] @CutOffDate,'BLDG','SUTS';
```

Gambar 3.24 *Query* Eksekusi SP Report Aging

Salah satu tantangan dalam mengambil data dari SP adalah absennya metadata terkait struktur tabel, seperti nama kolom dan tipe datanya. Berbeda dengan *query* SELECT biasa, hasil dari SP tidak memberikan skema bawaan yang dapat langsung digunakan oleh *pipeline* ETL. Data

dalam *database* ini juga tidak dapat dimodifikasi karena tidak memiliki akses untuk DDL, sehingga minim aksi yang dapat dilakukan. Maka, *ingestion* data dari Prime akan menggunakan metode yang digunakan untuk *ingestion* dari sumber FTP, yaitu dengan mendefinisikan struktur melalui file Excel yang berisi nama kolom dan tipe data sesuai dengan hasil keluaran dari SP. File ini kemudian diletakkan dalam folder metadata agar dapat dikenali oleh *pipeline* transformasi saat proses *ingestion* berlangsung. Setelah itu, *query* eksekusi SP disimpan dalam file SQL, yang digunakan untuk mengekstrak data dari SP tersebut. Pada Gambar 3.24, *query* eksekusi mencakup tiga parameter, yakni @CutOffDate, @SiteFrom, dan @SiteTo. Parameter @CutOffDate digunakan untuk menetapkan tanggal *cut-off*, yaitu tanggal hingga data akan diambil, yang diset ke H-1 dari tanggal saat ini.. Sedangkan parameter @SiteFrom dan @SiteTo merupakan parameter untuk menentukan rentang *site*, yaitu dari *site* mana hingga *site* mana data akan diambil.

```
class SQL(Transforms):
    def __init__(self, bu, raw_bucket, table_name, group_table_name):
        super().__init__(bu, raw_bucket, table_name, group_table_name)

    def cleansing_dataframe():
        pass

    def mapping_dtype(self, sql_conn, sql_hook, df):
        if sql_conn == "mssql_   _db_10":
            logging.info("Using FTP mapping_dtype...")
            return super().mapping_dtype(
                df, dtype_tables=ConfigHelper().dtype_tables("ftp")["list_table"]
            )

        dtype_mapping = ConfigHelper().sql_dtype_mapping()

        column_name = "column_name"
        data_type = "data_type"

        sql_tables = ConfigHelper().ingestion_configs(business_unit=self.bu.lower())
        join_table = sql_tables["join_table"] if "join_table" in sql_tables else []
```

Gambar 3.25 *Script* Kondisi untuk Data dari *Database* Prime

Proses *ingestion* ini juga memerlukan modifikasi pada bagian transformasi SQL dalam *script* Python. Karena *pipeline ingestion* awalnya tidak dirancang untuk menangani hasil dari SP, maka perlu penambahan logika khusus dalam *script* untuk menangani kondisi ketika sumber data berasal dari Prime. Logika *script* pada Gambar 3.25

digunakan untuk memetakan *output* SP ke dalam format yang sesuai dengan skema *data lake*, serta menyesuaikan jalur pengolahan data agar mengikuti alur *ingestion* dari FTP. Penambahan logika ini dituliskan dalam bentuk kondisi pemisah pada kode Python, yang menunjukkan bagaimana *pipeline* menangani jenis sumber yang berbeda. Penyesuaian ini bersifat modular, sehingga tidak mengganggu jalannya ingestion untuk sumber-sumber lain. Setelah penyesuaian tersebut diterapkan, *pipeline* Airflow dapat menjalankan *ingestion* data Prime seperti pada sumber data lainnya. Data dari SP dieksekusi melalui koneksi *database* MSSQL, kemudian hasilnya dikonversi menjadi file berformat parquet dalam GCS, dan selanjutnya diproses melalui *pipeline* yang telah ditentukan. *Output* akhir dari proses ini adalah pemuatan data laporan AR *Aging* ke dalam *data lake*.

3.2.3.2 *Ingestion Data Lake* ke Penyimpanan Data

Ingestion data dari *data lake* ke sistem penyimpanan lainnya merupakan tahapan lanjutan dalam arsitektur *pipeline* data yang menghubungkan antara penyimpanan terpusat dan sistem-sistem *downstream*, seperti *database*. Setelah data berhasil dikumpulkan dan diproses dalam *data lake*, data tersebut dipindahkan ke sistem lain yang lebih spesifik dalam penggunaannya. Proses tersebut melibatkan proses penyesuaian struktur serta pemetaan format agar data yang dikirim bisa digunakan secara langsung oleh sistem tujuan. Setiap sistem penyimpanan memiliki standar dan ekspektasi format yang berbeda, sehingga *ingestion* dari data lake ke penyimpanan lain memerlukan perhatian terhadap struktur, kualitas, serta konteks data yang akan digunakan.

Proses *ingestion* dari data lake ke sistem penyimpanan lanjutan yang dilakukan adalah menuju ke platform TM1 untuk *ingestion* data *Account Receivable* (AR) *Credit*, atau yang biasa disebut dengan AR 100. Meskipun *cube* AR 100 sudah tersedia sebelumnya di TM1, *cube* tersebut belum pernah terisi oleh data apa pun. Pengolahan data dari *data lake* agar bisa

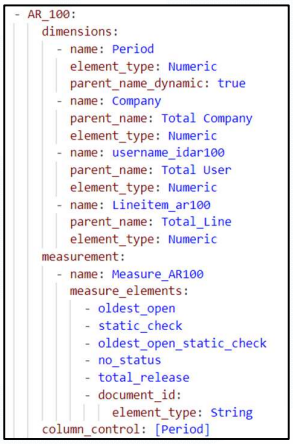
dikonsumsi oleh *cube* TM1 menggunakan *script* Python seperti *ingestion* pada umumnya. Walaupun *script* Python untuk AR 100 sudah tersedia, tetap perlu dilakukan sejumlah penyesuaian pada bagian query SELECT. Penyesuaian ini berkaitan dengan nama kolom yang digunakan, yang harus diselaraskan dengan nomenklatur yang berlaku di cube TM1. Hal ini penting untuk menghindari ketidaksesuaian antara struktur data sumber dengan struktur yang diharapkan oleh cube. Gambar 3.26 menunjukkan penyesuaian nama kolom pada bagian SELECT, yang mencerminkan adaptasi agar nama-nama atribut di *source* data identik dengan struktur cube. Penyesuaian ini juga dilakukan untuk mendukung proses pemetaan dimensi dan *measure* yang akan digunakan dalam pelaporan.

```
SELECT
Period,
Company,
username_idar100,
ROW_NUMBER() OVER(PARTITION BY Period, Company, username_idar100) AS Lineitem_ar100,
oldest_open,
static_check,
oldest_open_static_check,
no_status,
total_release,
document_id,
parent_Period
FROM (
    SELECT
        ff.period as Period,
        COALESCE(NULLIF(TRIM(ff.company_id), ''),'#') as Company,
        COALESCE(NULLIF(TRIM(ff.create_by_user), ''),'#') as username_idar100,
        ifnull(ff.oldest_open,0) as oldest_open,
        ifnull(ff.static_check,0) as static_check,
        ifnull(ff.oldest_open_static_check,0) as oldest_open_static_check,
        ifnull(ff.no_status,0) as no_status,
        ifnull(ff.total_release,0) as total_release,
        COALESCE(NULLIF(TRIM(ff.document_id), ''),'#') as document_id,
        LEFT(CAST(Period AS STRING), 4) AS parent_Period
    FROM
        delivery ff
    UNION ALL
    SELECT
        ff.period as Period,
        COALESCE(NULLIF(TRIM(ff.company_id), ''),'#') as Company,
        COALESCE(NULLIF(TRIM(ff.create_by_user), ''),'#') as username_idar100,
        ifnull(ff.oldest_open,0) as oldest_open,
        ifnull(ff.static_check,0) as static_check,
        ifnull(ff.oldest_open_static_check,0) as oldest_open_static_check,
        ifnull(ff.no_status,0) as no_status,
        ifnull(ff.total_release,0) as total_release,
        COALESCE(NULLIF(TRIM(ff.document_id), ''),'#') as document_id,
        LEFT(CAST(Period AS STRING), 4) AS parent_Period
    FROM
        so ff
```

Gambar 3.26 *Query* Penyesuaian Kolom pada *Script* AR 100

Kolom-kolom yang akan dimuat ke dalam TM1 tidak seluruhnya berasal dari *query* awal, melainkan difilter berdasarkan file konfigurasi mapping *cube* yang telah ditetapkan. File konfigurasi tersebut berisi daftar dimensi dan measure yang dibutuhkan oleh *cube* AR 100, dan hanya kolom-kolom

yang terdapat di file itulah yang akan diteruskan ke proses pemuatan. Penyesuaian ini penting untuk menjaga konsistensi dengan definisi struktur *cube* yang telah dibangun sebelumnya oleh tim *Analytics Engineer*, yang bertanggung jawab atas TM1. Gambar 3.27 menunjukkan tampilan file konfigurasi pemetaan *cube* yang digunakan dalam proses ini. Dengan adanya konfigurasi ini, proses *ingestion* menjadi lebih terarah dan sesuai dengan standar pelaporan yang telah ditentukan.

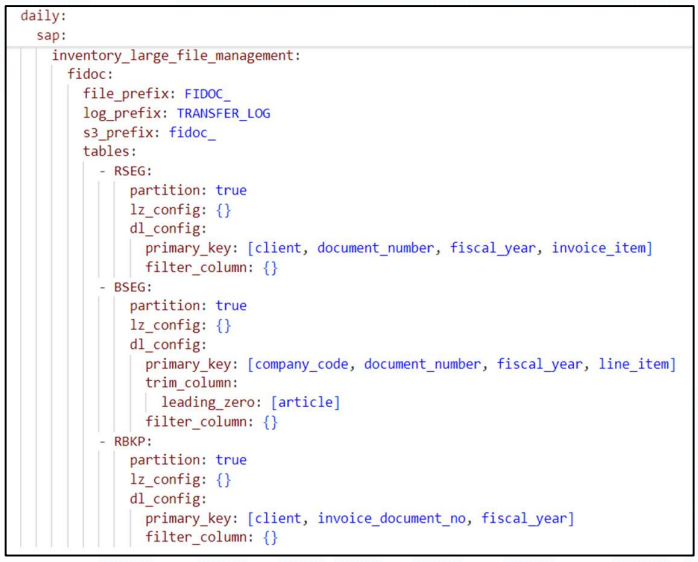


Gambar 3.27 Konfigurasi Pemetaan *Cube* AR 100 TM1

3.2.4 Pembuatan Daily DAG Baru pada Airflow

Proses pengelolaan data harian melalui Airflow saat ini membutuhkan efisiensi waktu eksekusi yang optimal untuk menangani tabel-tabel berukuran besar dari sumber SAP. Untuk itu, dibuatlah DAG baru yang secara khusus ditujukan untuk menangani *ingestion* tabel-tabel SAP dengan ukuran file yang sangat besar *(large file)*, seperti RSEG, RBKP, dan BKPF. Ketiga tabel ini merupakan bagian dari kelompok data SAP FIDOC, yang sebelumnya berada dalam satu DAG harian bersama tabel-tabel SAP lainnya. Namun karena ukurannya yang signifikan besar dibanding tabel lainnya, keberadaan ketiga tabel tersebut dalam satu DAG menyebabkan waktu eksekusi menjadi terlalu lama dan membebani sistem. Oleh karena itu, ketiganya dipindahkan ke dalam satu DAG baru yang terpisah agar dapat diproses lebih optimal. Pembuatan DAG ini bertujuan untuk meringankan beban DAG utama dan mendistribusikan beban kerja dengan lebih seimbang.

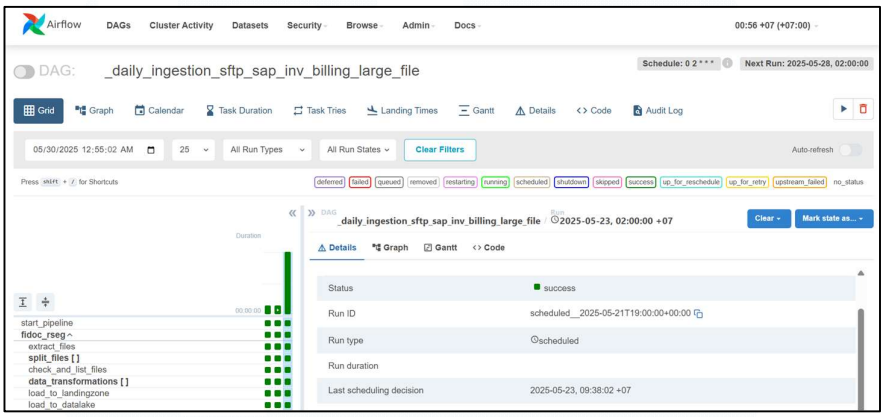
Struktur DAG *large file* dibuat dengan mengacu pada *template* DAG yang telah tersedia sebelumnya, sehingga tidak dilakukan penulisan ulang dari awal, melainkan hanya perlu melakukan penyesuaian pada konfigurasi YAML dan *script* Python DAG yang terkait dengan tabel-tabel tersebut. Pada awalnya, konfigurasi tabel RSEG, RBKP, dan BKPF berada dalam satu bagian umum bersama konfigurasi tabel SAP lainnya. Namun, agar lebih terorganisir dan dapat dibaca oleh *script* Python untuk DAG tersebut, konfigurasi tersebut dipisahkan dan dipindahkan ke bagian baru dengan nama inventory_large_file_management. Gambar 3.28 menunjukkan ilustrasi pemisahan struktur konfigurasi YAML yang dilakukan sebagai bagian dari implementasi DAG baru ini.



Gambar 3.28 Konfigurasi Tabel pada DAG *Large File* SAP

Perbedaan DAG *large file* dengan DAG ingestion SAP terletak pada saat transformasi data. Jika sebelumnya task transformasi berjalan secara berurutan, maka pada DAG *large file*, tahapan transformasi tersebut dibagi ke dalam beberapa mapped tasks yang dapat dieksekusi secara paralel. Sehingga, proses transformasi tidak lagi harus menunggu task sebelumnya selesai, melainkan dapat langsung dijalankan bersamaan sesuai dengan pemetaan yang telah ditentukan. Langkah ini mempercepat waktu pemrosesan dan mengurangi

bottleneck dalam *pipeline* data harian. Implementasi mapped tasks tersebut dituliskan secara eksplisit dalam *script* Python DAG, dan dikaitkan dengan pemanggilan konfigurasi YAML yang telah diperbarui. Hasil DAG ditunjukkan pada Gambar 3.29.



Gambar 3.29 DAG Large File SAP

3.2.5 Penyusunan Script Data Mart

Penyusunan *script data mart* dilakukan sebagai bagian dari upaya untuk memenuhi kebutuhan penyediaan data yang dibutuhkan oleh tim audit internal di Kawan Lama Group. *Script* ini berfungsi untuk mengambil data yang sudah diproses di *data warehouse*, atau dalam beberapa kasus juga dari *data lake*, dan kemudian memuatnya ke dalam database operasional seperti Microsoft SQL Server (MSSQL). Dalam kasus ini, *database* tujuan merupakan milik departemen Audit, yang membutuhkan data dengan struktur khusus untuk melakukan analisis lebih lanjut, terutama terkait *Sales Return* (SR) dari empat BU utama yang berada dalam cakupan tanggung jawab mereka. *Script* ini bersifat modular dan dijalankan secara rutin, karena akan diintegrasikan ke dalam scheduler Airflow untuk otomatisasi proses setiap bulannya.

Script disusun dengan mempertimbangkan kebutuhan spesifik dari masing-masing BU, karena struktur tabel di sumber data tidak sepenuhnya seragam. Beberapa BU memiliki nama tabel yang berbeda meskipun isi atau struktur datanya sebenarnya sama, sehingga pada *script* bagian awal, ditambahkan kondisi konfigurasi menggunakan *if-else* yang akan membedakan nama tabel,

sebagaimana dapat dilihat pada Gambar 3.30. Variabel-variabel tersebut ialah receive_table_name, master_company_table_name, dan trans_table_name yang digunakan sebagai *default*, dan nilainya bisa disesuaikan tergantung BU yang sedang diproses. Setelah konfigurasi disesuaikan, bagian utama dari script berisi *query* pada Gambar 3.31 digunakan untuk mengekstrak data SR. Data ini bersumber dari kombinasi *data lake* dan *data warehouse*, tergantung kebutuhan dan ketersediaan tabel. *Query* ini mencakup penggabungan beberapa tabel seperti trans, receive, dan juga master_company, dengan kondisi yang difilter berdasarkan bulan sebelumnya agar data yang di-*load* tetap relevan.

```
def gen_audit_dm_sr_analysis(
    _businessUnit,
    _project="              ",
    _dl="      ",
    _dwh="      ",
    _bu_list=["BU1", "BU2", "BU3"],
):
    table_name = "sr_analysis"
    pg_target_table = f"temp_{_businessUnit.lower()}_{table_name}"
    main_table = f"{_businessUnit.lower()}_{table_name}"

    receive_table_name = f"`{_project}.{_dl}.{_businessUnit.lower()}_receive`"
    master_company_table_name = (
        f"`{_project}.{_dl}.{_businessUnit.lower()}_master_company`"
    )
    trans_table_name = f"`{_project}.{_dl}.{_businessUnit.lower()}_trans`"
    filter_dim_dp = f"AND dp.business_unit = '{_businessUnit.upper()}'"
    dim_product_table = f"`{_project}.{_dwh}.dim_product`"
    select_rank = ""

    if _businessUnit in ("BU2"):
        filter_dim_dp = "AND dp.business_unit IN ('BU2','BU2A','BU2B')"
        select_rank = f"with product as ( SELECT * FROM (SELECT *, row_number() over(partition by product_id) as rn FROM `{_project}.
        dim_product_table = "product"
    elif _businessUnit in ("BU3"):
        receive_table_name = f"`{_project}.{_dl}.{_businessUnit.lower()}_receive_sap`"
        master_company_table_name = (
            f"`{_project}.{_dl}.{_businessUnit.lower()}_master_company_sap`"
        )
        trans_table_name = f"`{_project}.{_dl}.{_businessUnit.lower()}_trans_sap`"
```

Gambar 3.30 *Script Data Mart* SR Analysis

```
bq_query_extract = f"""
    {select_rank}
    SELECT site_sap.site_code as site_id, ds.site_name as site_name, sr.return_no as sr_no,
    sr.receive_no as transaction_number, trim(sr.description) as sr_transaction_number,
        r3.receive_no as transaction_number_after_sr,
        CASE WHEN sr.trans_type = 1 THEN 'REFUND MONEY'
            WHEN sr.trans_type = 2 THEN 'GOODS EXCHANGE'
            WHEN sr.trans_type = 3 THEN 'CREDIT NOTE'
            WHEN sr.trans_type = 4 THEN 'CREDIT NOTE PARTIAL DELIVERY'
        END as sr_type,
        r2.transdate as calendar_day,
        COALESCE(r.transdate, CAST('1970-01-05 00:00:01' AS TIMESTAMP)) as transaction_date,
        r.cust_id as member_id, r.card_id as member_card_id, dp.product_id_sap as article_id,
        dp.product_name as article_name, hrc.comm_name as ah_commodity, rd.notes as note,
        (rd.qty_ret * rd.stock_price) as return_value, rd.qty_ret as return_qty
    FROM `{_project}.{_dl}.{_businessUnit.lower()}_sreturn` sr
    LEFT JOIN {receive_table_name} r ON r.receive_no = sr.receive_no
    LEFT JOIN {master_company_table_name} site_sap ON upper(site_sap.company_code) = upper(r.company_code)
    LEFT JOIN `{_project}.{_dwh}.dim_site` ds ON ds.site_id = site_sap.site_code
    LEFT JOIN {receive_table_name} r2 ON trim(r2.receive_no) = trim(sr.description)
    LEFT JOIN `{_project}.{_dl}.{_businessUnit.lower()}_sreturnd` rd ON sr.return_no = rd.return_no
    LEFT JOIN {dim_product_table} dp ON dp.product_id = ltrim(trim(rd.stock_code),'0')
        {filter_dim_dp}
    LEFT JOIN `{_project}.{_dwh}.dim_product_hrc_scd` hrc ON hrc.product_id_sap = dp.product_id_sap
        {filter_dim_dp} AND hrc.hrc_status = 'active'
    LEFT JOIN {trans_table_name} p ON trim(p.doc_no) = trim(sr.return_no)
    LEFT JOIN {receive_table_name} r3 ON trim(r3.receive_no) = trim(p.receive_no)
    WHERE
        DATE(r2.transdate) >= DATE_SUB(DATE_TRUNC(CURRENT_DATE(),month),INTERVAL 1 month)
        AND DATE(r2.transdate) <= DATE_SUB(DATE_TRUNC(CURRENT_DATE(),month),INTERVAL 1 day)
        AND r3.posid <> 0
```

Gambar 3.31 *Script* Bagian *Query* Ekstrak Data SR

Selain bagian ekstraksi, *script* juga dilengkapi dengan bagian *load* ke dalam database MSSQL milik departemen Audit. Karena tabel tujuan merupakan tabel baru yang sebelumnya belum tersedia, maka pada saat proses pertama kali dijalankan perlu dibuat struktur tabelnya terlebih dahulu. Pembuatan tabel ini juga mencakup penambahan *constraint* dengan *unique key* (PK) untuk menjamin keunikan data agar ketika proses load dijalankan ulang di kemudian hari, data yang sudah ada tidak mengalami duplikasi. Apabila ada baris yang sudah memiliki *key* yang sama, maka baris tersebut tidak akan di-*insert* ulang, tetapi cukup diperbarui pada kolom-kolom lain yang bukan bagian dari key. Hal ini dilakukan dengan menambahkan kondisi ON DUPLICATE KEY UPDATE di *query load* pada Gambar 3.32 dan penambahan *constraint* pada *database* tujuan pada Gambar 3.33.

```
pg_query_load = f"""
    INSERT INTO {main_table}
    (
    site_id,site_name,sr_no,transaction_number,sr_transaction_number,transaction_number_after_sr,sr_type,
    calendar_day,transaction_date,member_id,member_card_id,article_id,article_name,ah_commodity,note,return_value,return_qty
    )
        SELECT
        t.site_id, t.site_name, t.sr_no, t.transaction_number, t.sr_transaction_number, t.transaction_number_after_sr,
        t.sr_type, t.calendar_day, t.transaction_date, t.member_id, t.member_card_id, t.article_id, t.article_name,
        t.ah_commodity, t.note, t.return_value, t.return_qty
        FROM {pg_target_table} t
    ON DUPLICATE KEY UPDATE
    site_id = t.site_id,
    site_name = t.site_name,
    sr_type = t.sr_type,
    calendar_day = t.calendar_day,
    transaction_date = t.transaction_date,
    member_id = t.member_id,
    member_card_id = t.member_card_id,
    article_name = t.article_name,
    ah_commodity = t.ah_commodity,
    note = t.note,
    return_value = t.return_value,
    return_qty = t.return_qty;
"""

return (
    main_table,
    table_name,
    pg_target_table,
    bq_query_extract,
    pg_query_load,
    _bu_list,
)
```

Gambar 3.32 *Script* Bagian *Query Load* ke Tabel sr_analysis

Columns	Name	Column	Owner	Type	Check expression
Constraints	no_duplicate	–	ahi_sr_analysis	UNIQUE KEY	
Foreign Keys	sr_no	sr_no	–	–	–
References	transaction_number	transaction_number	–	–	–
Triggers	sr_transaction_number	sr_transaction_number	–	–	–
Indexes	transaction_number_after_sr	transaction_number_after_sr	–	–	–
	article_id	article_id	–	–	–

Gambar 3.33 *Constraint* Kolom Unique Key pada Tabel sr_analysis

Setelah *Script* dirancang agar dapat dijalankan secara rutin di Airflow, dengan jadwal bulanan. Setiap kali dijalankan, *script* hanya akan mengambil data untuk bulan sebelumnya, sesuai dengan kondisi WHERE yang ditambahkan dalam query ekstrak. Hal ini penting agar volume data yang ditarik tidak terlalu besar, dan juga supaya data yang masuk ke database tetap terorganisasi dengan baik per periode. Proses ini juga mengurangi beban jika sewaktu-waktu perlu dilakukan pengecekan ulang atas data tertentu karena sudah terbagi per bulan. Pola seperti ini sangat umum digunakan dalam pengelolaan data *pipeline* yang bersifat periodik, terutama dalam konteks audit yang memerlukan laporan dan analisis data secara berkala.

3.2.6 Pengembangan dan Penyesuaian Struktur Tabel

Pengembangan dan penyesuaian struktur tabel merupakan langkah penting dalam menjawab kebutuhan data baru yang muncul seiring berjalannya operasional dan permintaan analisis dari berbagai pihak dalam perusahaan. Struktur tabel destinasi yang semula hanya mencakup informasi dasar, terkadang perlu dikembangkan dengan menambahkan atribut-atribut baru agar data yang tersimpan menjadi lebih kaya dan relevan. Salah satu bentuk pengembangan ini terlihat dari penambahan kolom pada tabel-tabel dalam *data warehouse* (DWH) untuk mengakomodasi data yang sebelumnya tidak tersedia, terutama data yang dibutuhkan oleh divisi atau departemen tertentu. Proses ini tidak hanya mencakup penambahan struktur secara fisik pada tabel, tetapi juga penyesuaian pada *pipeline* pemrosesan data jika diperlukan agar kolom-kolom baru tersebut dapat terisi dengan benar. Proses penyesuaian tersebut sering dilakukan dengan menyesuaikan kebutuhan per BU dan tidak serta-merta diterapkan secara general untuk seluruh entitas. Penyesuaian struktur tabel yang telah dilakukan selama masa kerja magang di antaranya:

A. Penambahan kolom pada DWH dim_receive

   Penyesuaian ini dilakukan untuk mendukung kebutuhan analisis dan pelacakan yang lebih rinci oleh departemen Audit, terutama terkait dengan proses *Sales Return* (SR). Departemen Audit memerlukan akses terhadap

informasi pengembalian barang setelah barang diterima, yang sebelumnya tidak tercakup dalam struktur tabel yang ada. Oleh karena itu, dilakukan penambahan tiga kolom baru, yakni receive_after_sr, return_id, dan sr_trans_type, yang secara langsung berkaitan dengan identifikasi transaksi SR. Ketiga kolom ini dirancang untuk menangkap data mengenai apakah suatu penerimaan merupakan hasil SR, nomor referensi dari pengembalian, serta jenis transaksi pengembalian tersebut. Tambahan ini bersifat penting untuk kebutuhan audit internal pada empat BU yang aktif melakukan SR.

```
"join_after_sr": f"""
    LEFT JOIN (
        SELECT
            *
        FROM (
            SELECT
                DISTINCT t.receive_no,
                t.doc_no,
                t.transdate,
                ROW_NUMBER() OVER(PARTITION BY doc_no ORDER BY t.receive_no ASC) rn
            FROM
                `{_project}.{_dl}.{_BU.lower()}_trans` t
            LEFT JOIN
                `{_project}.{_dl}.{_BU.lower()}_receive` r3
            ON TRIM(r3.receive_no) = TRIM(t.receive_no)
            WHERE CAST(r3.posid AS STRING) <> '0'
        )
        WHERE rn = 1
) sreturn_2 ON sreturn_2.doc_no = sreturn.return_id
""",
```

Gambar 3.34 Logika *Join* untuk Kolom Baru SR pada *Script* dim_receive

Untuk mengisi kolom-kolom yang ditambahkan, dilakukan penyesuaian pada *query* yang digunakan dalam *pipeline* pengolahan data, tepatnya pada *script* dim_receive. Proses awal melibatkan pemahaman terhadap logika *query* yang telah ada, guna menentukan titik yang tepat untuk menyisipkan logika tambahan tanpa mengganggu struktur yang sudah berjalan. Cara yang digunakan adalah dengan membuat variabel khusus untuk menampung logika *left join* antara tabel trans dan receive, yang berfungsi menarik data SR. Namun karena tidak semua BU membutuhkan kolom ini, maka logika tersebut tidak diletakkan langsung di bagian query utama. Sebaliknya, logika join dimasukkan dalam bentuk *dictionary* bernama default_dict, seperti yang diperlihatkan pada Gambar 3.34. Di dalam *script* dim_receive, setiap BU memiliki logika tersendiri, dan pada bagian inilah dictionary digunakan untuk menentukan apakah kolom-kolom SR perlu dimasukkan atau tidak. Untuk mengintegrasikan kolom hasil *join* ke dalam SELECT

utama, kolom-kolom yang baru harus ditambahkan pada variabel insert_column dalam *dictionary* ini sesuai BU yang diperlukan. Sebagai contoh, pada Gambar 3.35 ditunjukkan bagaimana salah satu BU mengaktifkan insert_column dan menggunakan *key-key* tambahan untuk menjalankan logika *join* yang sudah dibuat sebelumnya beserta *key-key* lain yang dibutuhkan selain untuk keperluan SR. Sebaliknya, pada Gambar 3.36, BU yang tidak memerlukan kolom-kolom SR akan memiliki value kosong (“”) pada *key dictionary*-nya, sehingga proses join maupun pemanggilan kolom tidak dijalankan.

Gambar 3.35 Contoh Nilai *Key Dictionary* BU yang Membutuhkan Kolom SR

Gambar 3.36 Contoh Nilai *Key Dictionary* BU yang Tidak Membutuhkan Kolom SR

Setelah seluruh penyesuaian pada level logika dan *script* selesai, tahapan berikutnya adalah memperbarui struktur fisik tabel dim_receive pada BigQuery. Hal ini dilakukan menggunakan perintah ALTER TABLE untuk menambahkan ketiga kolom baru secara permanen ke dalam tabel. Namun karena *pipeline ingest* tidak langsung mengisi data historis yang sudah ada, maka perlu dilakukan proses *update* data secara terpisah. Proses

ini dilakukan dengan menyusun *query update* yang dirancang khusus untuk mengisi nilai-nilai ketiga kolom berdasarkan hasil join sebelumnya. Gambar 3.37 menunjukkan contoh dari *query update* yang digunakan untuk mengisi nilai kolom receive_after_sr, return_id, dan sr_trans_type. Dengan langkah ini, kolom-kolom baru tidak hanya tersedia dalam struktur, tetapi juga berisi informasi yang relevan sesuai kebutuhan analisis dari departemen terkait.

```
-- update dim receive pos
UPDATE `project.dwh.dim_receive` dr
    SET receive_after_sr = cte.receive_after_sr, return_id = cte.return_id, sr_trans_type = cte.sr_trans_type
FROM (
    SELECT up.receive_no as receive_id,sreturn.return_id,sreturn.sr_trans_type, sreturn_2.receive_no as receive_after_sr,
    FROM (
        select receive_no,createtime as period_time,posid as pos_id,'xxx' as trx_type, lpad(cast(salesman_id as string),7,'0') as cashier_id, cust_id
        FROM `project.dl.abc_receive` r
    ) up
    LEFT JOIN (
        SELECT
            CASE WHEN TRIM(r.old_um_receive_no) <> '' THEN TRIM(r.old_um_receive_no) ELSE r.receive_no END AS receive_no,s.description,s.return_id,s.sr_trans_type
        FROM (
            SELECT receive_no,old_um_receive_no,transdate FROM `project.dl.abc_receive`
            UNION ALL
            SELECT receive_no,NULL AS old_um_receive_no,transdate FROM `project.dl.abc_umreceive`  ) r
        LEFT JOIN (
            SELECT * FROM (
                SELECT receive_no,TRIM(description) AS description,TRIM(return_no) AS return_id,trans_type AS sr_trans_type,
                    ROW_NUMBER() OVER(PARTITION BY sr.description ORDER BY paid desc, sr.trans_date desc) rn
                    FROM `project.dl.abc_sreturn` sr
                    WHERE TRIM(description) <> '' AND trans_date >= '2021-01-01'
            ) WHERE rn = 1 ) s
        ON r.receive_no = s.receive_no
        WHERE s.description IS NOT NULL
    ) sreturn on sreturn.description = up.receive_no
    LEFT JOIN (
        SELECT * FROM (
          SELECT *,ROW_NUMBER() OVER(PARTITION BY doc_no ORDER BY receive_no ASC) rn
          FROM (
                SELECT DISTINCT t.receive_no,t.doc_no, t.transdate
                FROM `project.dl.abc_trans` t
                LEFT JOIN `project.dl.abc_receive` r3
                ON TRIM(r3.receive_no) = TRIM(t.receive_no)
                WHERE CAST(r3.posid AS STRING) <> '0'
          ))
        WHERE rn = 1 ) sreturn_2
    ON sreturn_2.doc_no = sreturn.return_id
) cte
WHERE dr.receive_id = cte.receive_id AND upper(business_unit) = 'ABC' AND source_data = 'POS';
```

Gambar 3.37 *Query* untuk *Update* Data DWH dim_receive

B. Penambahan kolom pada DWH fact_transaction_pos

Penambahan kolom pada DWH fact_transaction_pos juga merupakan bagian dari proses penyesuaian struktur data untuk mendukung kebutuhan departemen Audit terhadap data transaksi SR. Terdapat dua kolom baru yang ditambahkan, yakni notes dan period_return. Keduanya berfungsi sebagai pelengkap informasi atas transaksi yang berhubungan dengan pengembalian barang. Kolom notes digunakan untuk menampung catatan tambahan terkait transaksi pengembalian, seperti alasan pengembalian, keterangan dari karyawan, atau informasi lain yang relevan secara operasional. Sementara itu, kolom period_return berfungsi untuk menunjukkan tanggal pengembalian dilakukan.

Kolom notes dan period_return ditambahkan pada DWH melalui proses *query* yang melibatkan *join* ke dua tabel sumber, yaitu sreturn dan sreturnd,

di mana kedua tabel ini berisi detail transaksi pengembalian barang. Agar struktur *script* tetap terorganisir dan fleksibel, *join* dan kolom-kolom tambahan tidak langsung ditulis di query utama, namun disusun sedemikian rupa dengan menggunakan *dictionary* pada *script* Python untuk setiap komponen *query* yang bersifat dinamis. Nilai *dictionary* ini dibedakan setiap BU yang membutuhkan data SR, yaitu sebanyak empat BU. Masing-masing BU memiliki *key* tersendiri yang akan dipanggil pada saat eksekusi *script* sebagaimana ditampilkan pada Gambar 3.38, yang menunjukkan bagaimana konfigurasi *dictionary* untuk proses *insert* kolom, pemilihan kolom *(select)*, serta logika *join* pada BU yang bersangkutan, di mana untuk menyertakan nilai dari masing-masing *key* ke dalam query utama, yang dilakukan adalah memanggil *dictionary* dan *key* yang dibutuhkan. Struktur ini bersifat fleksibel karena dapat dikembangkan lebih lanjut jika BU baru memerlukan kolom tambahan atau jika logika *join* mengalami perubahan. Hal tersebut akan membuat *script* menjadi lebih mudah dipelihara dan dapat diatur dengan granularitas yang sesuai kebutuhan masing-masing unit bisnis.

```
sr_notes_period = {
    "insert_col": ",notes, period_return",
    "select_col": ",notes, period_return",
    "join": f"""
            LEFT JOIN(
                SELECT * FROM (
                    SELECT trim(sr.description) AS receive_return, srd.stock_code, srd.notes,
                    date(sr.trans_date) period_return, paid, sr.return_no,
                    ROW_NUMBER() OVER(PARTITION BY sr.description ORDER BY paid desc, sr.trans_date desc) rn
                    FROM `{_project}.{_dl}.{_BU.lower()}_sreturn` sr
                    LEFT JOIN `{_project}.{_dl}.{_BU.lower()}_sreturnd` srd
                    ON sr.return_no = srd.return_no
                ) where rn = 1
            ) sr on sr.receive_return = receive_id and ltrim(sr.stock_code, '0') = product_id
            """,
}
```

Gambar 3.38 Konfigurasi Kolom Baru SR dalam Bentuk *Dictionary*

```
_vCTE = f"""
    SELECT
    ...
    {sr_notes_period['select_col']}
    ...
        ...
    ) )
    WHERE RN = 1
) abc
{sr_notes_period['join']}
```

Gambar 3.39 Pemanggilan Variabel *Dictionary* SR pada *Query* Utama

Setelah konfigurasi selesai, bagian *query* utama dalam script fact_transaction_pos kemudian diperbarui. Pemanggilan *key* dari *dictionary* dimasukkan tepat di bawah SELECT kolom terakhir serta di bagian paling akhir dari *join-query*, sebagaimana ditunjukkan pada Gambar 3.39. Setelah seluruh logika pada *query* selesai disesuaikan di *script* Python, maka langkah berikutnya adalah melakukan penyesuaian struktur tabel di sisi penyimpanan, yaitu BigQuery. Sama seperti pada penyesuaian dim_receive, kolom notes dan period_return ditambahkan ke dalam tabel fact_transaction_pos menggunakan ALTER TABLE agar tidak mengubah data yang sudah ada sebelumnya pada tabel tersebut.

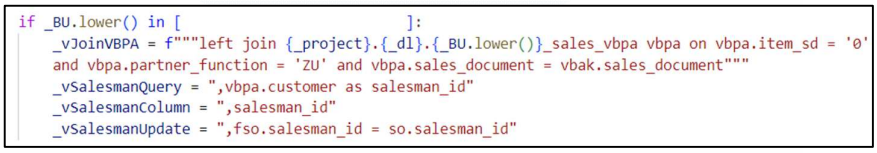
```
-- update fact
UPDATE
`project.dwh.abc_fact_transaction_pos_cc` ft
    SET notes = cte.notes,
        period_return = cte.period_return
    FROM
    (
        SELECT * FROM (
            SELECT trim(sr.description) AS receive_return, srd.stock_code, srd.notes,
            date(sr.trans_date) period_return, paid, sr.return_no,
            ROW_NUMBER() OVER(PARTITION BY sr.description
            ORDER BY paid desc, sr.trans_date desc) rn
            FROM `project.dl.abc_sreturn` sr
            LEFT JOIN `project.dl.abc_sreturnd` srd
            ON sr.return_no = srd.return_no
        ) where rn = 1
    ) cte
      WHERE cte.receive_return = ft.receive_id
      AND ltrim(cte.stock_code, '0') = ft.product_id ;
```

Gambar 3.40 *Query* untuk *Update* Data DWH fact_transaction_pos

Namun, penambahan kolom saja belum cukup karena data pada kolom-kolom baru tersebut masih kosong. Oleh sebab itu, perlu dilakukan proses update data menggunakan query tersendiri. *Update query* ini kembali menggunakan join terhadap tabel sreturn dan sreturnd sebagai referensi, serta menyesuaikan kondisi-kondisi per BU. Proses ini digambarkan dalam Gambar 3.40, di mana diperlihatkan bagaimana pemetaan value dilakukan agar data yang dimasukkan sesuai dengan konteks masing-masing transaksi. Setelah diperbarui, data akan divalidasi oleh PIC dari divisi *Data Platform* yang memiliki tanggung jawab atas kualitas dan akurasi data. Validasi ini mencakup pengecekan format, kesesuaian isi kolom, serta integrasi antar tabel. Jika hasil validasi telah sesuai, maka data tersebut dapat dikatakan siap digunakan oleh departemen yang memerlukannya.

C. Generalisasi dan penambahan kolom pada DWH fact_sales_order

Penyesuaian struktur pada tabel fact_sales_order memiliki dua penyesuaian utama, yaitu penambahan satu kolom baru bernama ship_to_party_id serta generalisasi kolom salesman_id agar dapat digunakan oleh seluruh BU tanpa pengecualian. Kedua kolom ini bersumber dari tabel relasi *partner* pada suatu *sales document* di sistem SAP yang disebut VBPA, namun masing-masing mengambil nilai berdasarkan *partner function* yang berbeda. *Partner function* sendiri merupakan kode yang digunakan untuk mengidentifikasi peran pihak terkait dalam suatu dokumen penjualan. Dalam hal ini, salesman_id berasal dari partner function "ZU", yang merepresentasikan pihak *sales representative* atau tenaga penjual, sedangkan ship_to_party_id berasal dari partner function "WE", yang menandakan pihak penerima barang atau tujuan pengiriman. Kedua nilai merepresentasikan relasi yang berbeda dalam satu proses penjualan yang sama.



Gambar 3.41 Perubahan *Query* Utama untuk Generalisasi dan Penambahan Kolom SR

Sebelum dilakukan generalisasi, kolom salesman_id hanya digunakan pada beberapa BU saja. Hal ini membuat *script* pada fact_sales_order menjadi cukup kompleks karena harus membedakan perlakuan antara BU yang menggunakan dan yang tidak menggunakan kolom tersebut. *Script* disusun dengan banyak kondisi yang hanya aktif untuk BU tertentu, sehingga logika pengolahan data tidak bersifat universal. Sekarang, dilakukan kolom tersebut akan digeneralisasi dengan cara menghapus kondisi-kondisi eksklusif terkait kolom salesman_id dan menggabungkannya ke dalam struktur *query* utama, sehingga seluruh BU akan mendapatkan perlakuan yang sama tanpa perlu membuat pengecualian di dalam *script*. Semua logika yang sebelumnya tersebar di dalam *dictionary*

berdasarkan BU kemudian dipindahkan ke dalam satu blok yang konsisten, sebagaimana ditunjukkan pada Gambar 3.41. Proses pemindahan ini juga mencakup penyusunan ulang bagian *select*, *insert*, hingga *join*, agar semuanya berada di tempat yang sesuai dan tidak saling bertabrakan.

Gambar 3.42 Perubahan *Query* Utama untuk Generalisasi dan Penambahan Kolom SR

Penambahan kolom ship_to_party_id memiliki proses yang secara garis besar cukup mirip dengan proses penambahan kolom salesman_id, karena keduanya sama-sama bersumber dari tabel relasi VBPA yang terdapat dalam sistem SAP dan keduanya dihubungkan dengan menggunakan field yang sama sebagai kunci utama untuk proses *join*. Namun, satu *sales document* dalam SAP bisa memiliki lebih dari satu baris di tabel VBPA, masing-masing dengan *partner function* yang berbeda. Oleh karena itu, proses *join* untuk mengambil nilai ship_to_party_id tidak bisa disatukan dengan *join* untuk salesman_id. Jika kedua kolom ini diambil dengan satu kali *join* dan menggunakan kondisi partner_function IN ('ZU', 'WE'), maka hasilnya akan menyebabkan duplikasi baris. Misalnya, satu *sales document* bisa muncul dua kali jika memiliki *partner function* ZU dan WE sekaligus. Hal ini tentu tidak diinginkan karena akan menyebabkan data menjadi tidak akurat.

Untuk menghindari hal tersebut, yang dilakukan adalah memisahkan proses *join* untuk masing-masing kolom. *Join* untuk kolom salesman_id dilakukan secara terpisah dengan filter khusus untuk partner_function =

'ZU', sedangkan *join* untuk kolom ship_to_party_id dilakukan dengan filter partner_function = 'WE'. Kedua *join* ini memanggil tabel VBPA, namun dengan alias yang berbeda agar pada saat pemilihan kolom, nilai kolom yang berasal dari kolom customer pada tabel VBPA dapat diarahkan ke kolom salesman_id dan ship_to_party_id sesuai pemanggilan alias. Seluruh perubahan dan penyesuaian ini kemudian diintegrasikan ke dalam query utama fact_sales_order, untuk kolom baru yang ditambahkan. Gambar 3.42 menampilkan bagaimana kedua kolom tersebut dipanggil dalam *query* utama pada bagian *select*, *insert*, *join*, dan *update*.

```
# Update
client = ""
_project= ""
_dl= "dl"
_dwh= "dwh"
_bu_list=["BU1", "BU2"]

for bu in _bu_list:
  if bu.lower() == "BU1":
    client = bigquery.Client(project='project1', location='asia-southeast1')
    _project= "project1"
  else:
    client = bigquery.Client(project='project2', location='asia-southeast1')
    _project= "project2"

  vUpdate = f"""
        UPDATE `{_project}.{_dwh}.{bu.lower()}_fact_sales_order` so
              SET salesman_id = cte.salesman_id, ship_to_party_id = cte.ship_to_party_id
              FROM (
                  SELECT
                    vbak.sales_document as sales_order_id, zuvbpa.customer as salesman_id, wevbpa.customer as ship_to_party_id
                  FROM `{_project}.{_dl}.{bu.lower()}_sales_vbak` vbak
                  LEFT JOIN `{_project}.{_dl}.{bu.lower()}_sales_vbpa` zuvbpa on zuvbpa.item_sd = '0' and zuvbpa.partner_function = 'ZU'
                  LEFT JOIN `{_project}.{_dl}.{bu.lower()}_sales_vbpa` wevbpa on wevbpa.item_sd = '0' and wevbpa.partner_function = 'WE'
              ) cte
                WHERE cte.sales_order_id = so.sales_order_id """

  client.query(vUpdate).result()
  print(f"{bu} updated")
```

Gambar 3.43 *Script* Looping untuk *Update* Data DWH fact_sales_order

Setelah *script* Python diperbarui, penyesuaian berlanjut ke tahap eksekusi *query* SQL untuk menambahkan kolom baru pada tabel di BigQuery menggunakan perintah ALTER TABLE. Karena merupakan kolom baru, maka data di dalamnya masih kosong dan perlu diisi ulang berdasarkan data yang tersedia di tabel VBPA. Untuk itu, dilakukan proses *update* data dengan cara menjalankan *update query* secara berulang *(looping)* untuk setiap BU yang ada pada Google Colab. Gambar 3.43 menunjukkan struktur *looping* pada *update query*, di mana setiap BU akan diproses satu per satu dengan menjalankan *update query* yang akan memperbarui nilai kolom salesman_id dan ship_to_party_id apabila memenuhi kondisi.

3.2.7 Pembuatan Tabel Fact pada Data Warehouse

Pembuatan *script* untuk *fact* baru bernama fact_open_price bertujuan dalam untuk mengintegrasikan data dari berbagai tabel sumber yang tersimpan di data lake menjadi satu tabel tabular yang lebih terstruktur dan siap digunakan untuk proses analisis lanjutan, di mana data ini berguna untuk kebutuhan departemen Audit. *Fact* ini dirancang untuk menyimpan data yang berkaitan dengan aktivitas otorisasi perubahan harga *(open price)* seperti harga spesial atau diskon yang terjadi di toko-toko perusahaan. Data tersebut mencakup atribut-atribut penting seperti waktu transaksi, toko, produk, serta rincian harga sebelum dan sesudah perubahan. Gambar 3.44 menunjukkan *script* yang dikembangkan menggunakan fungsi def pada Python yang dirancang dengan sejumlah parameter wajib, seperti kode BU, project, skema *data lake* maupun *data warehouse*, jenis eksekusi (runType) dan periode data yang diambil, di mana parameter-parameter tersebut diperlukan baik untuk eksekusi otomatis dalam ETL *pipeline* maupun untuk eksekusi manual jika diperlukan. Parameter _runType dalam fungsi ini memiliki peran penting untuk membedakan mode eksekusi, apakah *script* akan dijalankan dalam mode *daily* atau mode *backfill*, yang sesekali dibutuhkan untuk mengambil data dari periode tertentu.

```
def gen_fact_open_price(
    _BU,
    _project: str = "project",
    _dl: str = "dl",
    _dwh: str = "dwh",
    _runType="daily",
    _period='{{ data_interval_end.in_timezone("Asia/Jakarta").to_date_string() }}',
    _bu_list=["BU1", "BU2", "BU3", "BU4"],
):
    if _BU.upper() in _bu_list:
        _vPeriodCondition = ""

        if _runType.lower() == "daily":
            _vPeriodCondition = f"""WHERE date(a.updated_at) = '{_period}'"""
        elif _runType.lower() == "backfill":
            _vPeriodCondition = f"""WHERE date(a.tanggal) >= '{_period}'"""

        _vCTE = f"""
            SELECT date(a.tanggal) as period,a.sessionid as session_id,b.site_code AS store_id,a.posid as pos_id,
                a.stockcode as product_id,a.authorized_by as authorized_id,a.cashier_by as cashier_id,a.receive_no as receive_id,
                CASE WHEN a.terpakai = true THEN 1 ELSE 0 end as flag_terpakai_id,
                CASE WHEN a.status = true THEN 1 ELSE 0 end as flag_status_id,
                CASE WHEN a.reject = true THEN 1 ELSE 0 end as flag_reject_id,a.remark as remark_id,a.qty,a.unitprice,
                a.specialprice_old,a.disc_old,a.specialprice_new,a.disc_new,TIMESTAMP(CURRENT_DATETIME('Asia/Jakarta')) AS created_at,TI
            FROM `{_project}.{_dl}.{_BU.lower()}_openprice_authorization` a
            LEFT JOIN `{_project}.{_dl}.{_BU.lower()}_master_company` b
            ON a.company_code = b.company_code
            {_vPeriodCondition} """

        _vValues = """
                period,session_id,store_id,pos_id,product_id,authorized_id,cashier_id,receive_id,flag_terpakai_id,flag_status_id,flag_re
        """
```

Gambar 3.44 *Script* Konfigurasi dan CTE Tabel fact_open_price

Struktur fungsi ini secara umum terdiri dari empat bagian utama yang ditunjukkan pada Gambar 3.45, yaitu *query* untuk proses ekstraksi data, *insert* data, *merge* data, dan penghapusan data. Pada bagian awal dari fungsi, script akan melakukan validasi terhadap parameter BU yang diberikan. Jika BU tersebut tidak termasuk dalam daftar BU yang ditargetkan untuk dimasukkan ke dalam *pipeline*, maka *query* tidak akan dijalankan. Namun jika BU yang dimaksud tersedia, proses akan dilanjutkan dengan membangun *query* yang disesuaikan dengan kondisi dan kebutuhan spesifik dari masing-masing BU. *Query* ekstraksi data dibangun berdasarkan tabel utama, yaitu openprice_authorization, yang tersimpan di *data lake*. Tabel ini berfungsi sebagai log dari seluruh aktivitas pengajuan dan persetujuan perubahan harga di toko. Tabel ini kemudian di-*join* dengan tabel master_company untuk mendapatkan store_id yang berasal dari kolom site_code. Pengambilan ekstraksi data diatur berdasarkan periode tanggal sesuai runType yang digunakan. Untuk runType *daily*, data yang diambil hanya mencakup transaksi pada tanggal saat job dijalankan. Data ini diambil berdasarkan kolom updated_at, yang menunjukkan kapan data terakhir kali dimodifikasi. Sebaliknya, untuk runType *backfill*, kondisi waktu menggunakan kolom tanggal transaksi aktual.

Setelah proses ekstraksi data selesai, langkah berikutnya adalah memasukkan data tersebut ke dalam tabel *fact* di *data warehouse*. Pada runType daily, proses yang digunakan adalah *merge* dan *insert*. Dengan menggunakan merge, sistem dapat membandingkan data hasil *query* dengan data yang sudah ada di tabel *fact*. Proses *merge* ini dilakukan berdasarkan kolom session_id dan store_id, yang digunakan untuk mengidentifikasi setiap transaksi *open price*. Jika nilai dari kedua kolom tersebut belum ada, maka sistem akan menambahkan data baru menggunakan *insert*. Sementara itu pada runType *backfill*, data yang sudah ada untuk periode tertentu akan dihapus terlebih dahulu dari tabel *fact* melalui *query delete* untuk menghindari duplikasi, kemudian data hasil *query* akan langsung di-*insert*.

```
    _vInsert = f"""
        INSERT
        --INTO `{_project}.{_dwh}.{_BU.lower()}_fact_open_price`
        ( {_vValues}) """

    _vDelete = f"""
        DELETE FROM `{_project}.{_dwh}.{_BU.lower()}_fact_open_price`
        WHERE date(created_at) >= '{_period}' """

    _vMerge = f"""
        MERGE INTO `{_project}.{_dwh}.{_BU.lower()}_fact_open_price` op
        USING ( {_vCTE} ) cte
        ON op.session_id = cte.session_id AND op.store_id = cte.store_id
        WHEN NOT MATCHED
        THEN
        {_vInsert}
        VALUES ( {_vValues} ); """

    if _runType.lower() == "daily":
        query = _vMerge

    elif _runType.lower() == "backfill":
        query = f"""
            {_vDelete};
            {_vInsert.replace("--", " ")}
            {_vCTE};
        """

    return query

else:
    return "SELECT 'THIS BUSINESS UNIT IS NOT AVAILABLE';"
```

Gambar 3.45 *Script* Bagian *Query Load* ke Tabel fact_open_price

Seluruh *query* baik untuk ekstraksi, *insert*, *merge*, maupun *delete* disusun dalam bentuk string yang digabungkan secara dinamis sesuai dengan parameter yang diberikan. Komponen *query* dirancang secara modular sehingga jika ada perubahan di masa mendatang, seperti penambahan kolom baru atau perubahan sumber data, hanya bagian tertentu dari *script* yang perlu diubah, tanpa harus menulis ulang keseluruhan *query*. Sebagai contoh, daftar kolom yang digunakan untuk proses *insert* (vValues) ditulis secara terpisah dari *query* utama agar dapat digunakan kembali pada proses *merge* atau *insert* lainnya.

Untuk memastikan eksekusi otomatis, fungsi ini nantinya akan dipanggil oleh DAG Airflow yang dijalankan setiap hari atau sesuai kebutuhan audit. Parameter waktu (periode) dalam fungsi ini memanfaatkan *template* Jinja dari Airflow, sehingga tanggal eksekusi dapat ditentukan secara dinamis berdasarkan jadwal *pipeline*, sehingga *pipeline* dapat berjalan secara otomatis tanpa memerlukan pengaturan tanggal secara manual setiap kali dijalankan. Struktur tersebut akan mempermudah dalam memperluas *script* fact_open_price ke BU lain jika diperlukan di masa depan. Selama struktur tabel dan format data antar-BU seragam, penambahan BU baru cukup dilakukan

dengan menambahkan nama BU tersebut ke dalam daftar BU yang diizinkan dalam fungsi. Seluruh *pipeline* akan tetap berjalan dengan baik tanpa memerlukan pengembangan *script* baru dari awal. Pendekatan yang konsisten akan memberikan efisiensi dan kontrol yang lebih baik dalam proses pengembangan, khususnya ketika harus menangani banyak unit bisnis dalam skala besar seperti yang dimiliki oleh Kawan Lama Group.

3.2.8 Otomatisasi Trigger Fabric Pipeline Power BI

Divisi *Data Analyst* telah membuat *pipeline* Fabric untuk menjalankan proses *load* data di Power BI Fabric untuk tiga BU Kawan Lama Group. *Pipeline* Fabric akan di-*trigger* menggunakan Airflow agar lebih terorganisir dan dapat mengikuti jadwal DAG *daily* yang lain, di mana *trigger* Fabric *pipeline* dilakukan bersamaan dengan proses *refresh dashboard* Power BI yang sudah ada. Fungsi ini diintegrasikan ke dalam DAG Airflow sehingga *pipeline* Fabric dapat dijalankan bersamaan dengan task *refresh dashboard*. Tujuan utamanya adalah untuk memanfaatkan *resource* paralel dan efisiensi eksekusi, terutama saat ada kebutuhan pengolahan data berskala besar dari beberapa BU yang berbeda. Untuk menjalankan *pipeline* ini, perlu ditambahkan metode *trigger pipeline* Fabric pada *script* Python. Dengan mengintegrasikan fungsi ini ke dalam DAG di Airflow, *pipeline* Fabric dapat dijalankan bersamaan dengan *task refresh dashboard*, sehingga kedua proses berjalan secara paralel tanpa saling menunggu.

Pipeline Fabric yang dibuat terdiri dari dua jenis metode *load*, yaitu *incremental* dan *full load*. Jenis *incremental load* mengacu pada proses pemuatan data yang hanya mengambil perubahan (delta) sejak terakhir kali *pipeline* dijalankan. Sebaliknya, jenis *full load* mengambil seluruh data dari sumber tanpa memperhatikan perubahan. *Pipeline* jenis *incremental load* ini hanya memproses data yang benar-benar baru atau diubah, sehingga dapat menghemat waktu pemrosesan dan beban sistem, sementara jenis *full load* digunakan untuk memuat ulang seluruh data dari awal, yang sering digunakan untuk *reprocess* skala besar, atau setelah terdapat perubahan struktur pada

dataset. Jenis *full load* lebih berat dibandingkan *incremental load*, tapi berguna dalam situasi tertentu ketika dibutuhkan *data re-alignment* atau untuk validasi skala besar. Kedua jenis tersebut dikonfigurasi dalam Google Sheets, yang akan dipanggil oleh *script* Python melalui API untuk dijalankan di Airflow. Gambar 3.46 menunjukkan *script* Python yang berisi *class-class* yang dibutuhkan untuk *trigger pipeline* Fabric dan juga *refresh dashboard* Power BI.

```python
class call_pbi_api:

    def access_token(self) -> str:
        auth_url = self.auth_url + self.tenant_id
        token = msal.PublicClientApplication(client_id=self.client_id, authority=auth_url)
        token_response = token.acquire_token_by_username_password(username=self.username, password=self.password, scopes=self.scopes)

        if "access_token" not in token_response:
            raise Exception(token_response["error_description"])
        else:
            access_token = token_response.get("access_token")

        return access_token

    def run_pipeline(self, pipelineId: str, workspaceId: str) -> dict:
        access_token = self.access_token()
        headers = {
            "Authorization": f"Bearer {access_token}",
            "Content-Type": "application/json",
        }
        endpoint = f"https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{pipelineId}/jobs/instances?jobType=Pipeline"
        response = requests.post(endpoint, headers=headers, json={})
        if response.status_code in (200, 202):
            message = "No error message found"
            return response.status_code, message
        else:
            message = response.json().get("message", "No error message found")
            return response.status_code, message
```

Gambar 3.46 *Class* untuk Memanggil API Power BI dan Menjalankan *Pipeline* Fabric

Akses dan *trigger* Fabric *pipeline* dilakukan menggunakan *class* bernama call_pbi_api yang berfungsi sebagai *class* utama dalam proses komunikasi dengan API Power BI. Dalam *class* tersebut, terdapat fungsi access_token yang bertugas untuk mengambil token autentikasi menggunakan *Microsoft Authentication Library* (MSAL). Fungsi ini akan memanggil kombinasi tenant ID, client ID, dan kredensial pengguna untuk mendapatkan token yang valid. Token ini kemudian digunakan dalam seluruh proses komunikasi API dengan *endpoint* Fabric. Jika proses pengambilan token gagal, error akan langsung dilemparkan, dan eksekusi *pipeline* akan dihentikan. Pada *class* ini juga memiliki method yang baru ditambahkan bernama run_pipeline, yang bertanggung jawab untuk menjalankan *pipeline* berdasarkan ID *pipeline* dan ID *workspace* yang diberikan. Endpoint REST API yang digunakan dalam proses ini berfungsi untuk membuat *instance* baru dari *pipeline* yang ditentukan. Jika API memberikan respons dengan status 200 atau 202, eksekusi *pipeline*

dianggap berhasil. Jika eksekusi berhasil, maka akan mengembalikan pesan *"No error message found"*. Namun jika terjadi kegagalan, pesan error akan diambil langsung dari JSON *response* API dan digunakan untuk pelaporan.

```python
class call_space_api:
    def __init__(self, webhook_url: str) -> None:
        self.webhook_url = webhook_url

    def send_notification(
        self,status: str,message: str,business_unit: str,pipelineName: str,pipelineId: str,workspaceName: str,
    ) -> int:
        if status == "success":
            message_json = {
                "text": (
                    "*Fabric Pipeline Trigger:* SUCCESS\n\n"
                    f"*Business Unit:* {business_unit}\n"
                    f"*Pipeline Name:* {pipelineName}\n"
                    f"*Pipeline ID:* {pipelineId}\n"
                    f"*Workspace Name:* {workspaceName}\n"
                    f"*Message:* {message}\n"
                    f"*Triggered At:* {str(datetime.now())}\n"
                )
            }
            response = requests.post(
                self.webhook_url,data=json.dumps(message_json),headers={"Content-Type": "application/json"},)
            return response.status_code
        else:
            message_json = {
                "text": (
                    "*Fabric Pipeline Trigger:* FAILED\n\n"
                    f"*Business Unit:* {business_unit}\n"
                    f"*Pipeline Name:* {pipelineName}\n"
                    f"*Pipeline ID:* {pipelineId}\n"
                    f"*Workspace Name:* {workspaceName}\n"
                    f"*Message:* {message}\n"
                    f"*Triggered At:* {str(datetime.now())}\n"
                )
            }
            response = requests.post(
                self.webhook_url,data=json.dumps(message_json),headers={"Content-Type": "application/json"},
            )
            return response.status_code
```

Gambar 3.47 *Class* untuk Mengirimkan Notifikasi Google Space

Setiap eksekusi *pipeline* akan menghasilkan *output* status yang perlu diinformasikan. Oleh karena itu, dibuat juga *class* call_space_api yang berfungsi untuk mengirimkan notifikasi pesan ke Google Space. Pada Gambar 3.47, format notifikasi disusun dalam bentuk JSON dan dikirim ke *webhook* yang telah dikonfigurasi di file pbi_fabric_config.yml akan ditambahkan pada repositori. Pesan tersebut berisi informasi seperti status eksekusi, nama *pipeline*, ID *pipeline*, *workspace*, pesan respons dari API, serta waktu *trigger*. Pesan dibedakan antara yang sukses dan gagal agar divisi *Data Analyst* dapat langsung mengetahui hasilnya dan mengambil aksi jika perlu.

	A	B	C	D	E
1	**Business Unit**	**Pipeline Name**	**Pipeline ID**	**Workspace Name**	**Workspace ID**
2	BU1	BU1 - Main Incremental Load		BU1 - Fabric	
3	BU1	BU1 - Main Full Load		BU1 - Fabric	
4	BU2	BU2 - Main Incremental Load		BU2 - Fabric	
5	BU2	BU2 - Main Full Load		BU2 - Fabric	
6	BU3	BU3 - Main Incremental Load		BU3 - Fabric	
7	BU3	BU3 - Main Full Load		BU3 - Fabric	

Gambar 3.48 Spreadsheet *Mapping Pipeline* Fabric Power BI

```python
def run_fabric_pipeline(username,password,client_id):
    import os
    import yaml
    import gspread
    import logging
    from google.oauth2.service_account import Credentials
    from lib.task.pbi import call_pbi_api, call_space_api

    CONFIG_PATH = f"{os.path.dirname(__file__)}/../lib/config/pbi_fabric_config.yml"
    SERVICE_ACCOUNT = f"{os.path.dirname(__file__)}/../lib/credential/gcp/data_analyst_service_account.json"

    with open(CONFIG_PATH) as f:
        CONFIG = yaml.safe_load(f)

    TENANT_ID = CONFIG["Application Key"]["tenant_id"]
    AUTH_URL = CONFIG["Authorization"]["auth_url"]
    SCOPES_FABRIC = CONFIG["Authorization"]["scopes"]
    SHEET_ID = CONFIG["Spreadsheets"]["sheet_id"]
    SCOPES_SPREADSHEET = CONFIG["Spreadsheets"]["scopes"]
    WEBHOOKS_URL = CONFIG["Google Space"]["webhooks_url"]

    # READ SPREADSHEET
    creds = Credentials.from_service_account_file(
        SERVICE_ACCOUNT, scopes=SCOPES_SPREADSHEET
    )
    client = gspread.authorize(creds)
    data = client.open_by_key(SHEET_ID).sheet1.get_all_values()
```

Gambar 3.49 Penambahan Method Baru pada *Script* untuk *Trigger Pipeline* Fabric

```python
# RUN FABRIC PIPELINE AND SEND NOTIFICATION TO GOOGLE SPACE
call_pbi_api = call_pbi_api(username=username,password=password,client_id=client_id,tenant_id=TENANT_ID,
                            auth_url=AUTH_URL,scopes=SCOPES_FABRIC,)
call_space_api = call_space_api(webhook_url=WEBHOOKS_URL)

for i in data[1:]:
    if i[0] == f"{BUSINESS_UNIT}":
        print("Run Fabric Pipeline...")
        print(f"Pipeline Name: {i[1]} - Workspace Name: {i[3]}")
        fabric_status_code, fabric_message = call_pbi_api.run_pipeline(pipelineId=i[2], workspaceId=i[4])
        if fabric_status_code in (200, 202):
            gspace_response = call_space_api.send_notification(
                status="success",
                message=fabric_message,
                business_unit=i[0],
                pipelineName=i[1],
                pipelineId=i[2],
                workspaceName=i[3],
            )
            print(f"✅ Successfully Triggered Fabric Pipeline\nResponse Code: {fabric_status_code}\nMessage: {fabric_message}")
        else:
            gspace_response = call_space_api.send_notification(
                status="failed",
                message=fabric_message,
                business_unit=i[0],
                pipelineName=i[1],
                pipelineId=i[2],
                workspaceName=i[3],
            )
            print(f"❌ Failed to Trigger Fabric Pipeline\nResponse Code: {fabric_status_code}\nMessage: {fabric_message}")
        logging.info(f"Notification has been sent to Google Space: {gspace_response}")
```

Gambar 3.50 *Script* Menjalankan *Pipeline* Fabric untuk Setiap Row dalam Spreadsheet

Dalam file DAG Airflow, konfigurasi pipeline dibaca dari dua sumber, yaitu file YAML dan spreadsheet. File YAML berisi informasi autentikasi, seperti tenant ID, *scope*, dan URL *authorization* yang digunakan untuk mengakses API Power BI. Sementara itu, spreadsheet digunakan untuk menyimpan pemetaan data antara BU dengan nama *pipeline*, *pipeline* ID, *workspace*, dan *workspace* ID. Spreadsheet dapat dilihat pada Gambar 3.48, di mana spreadsheet penggunaan spreadsheet sebagai sumber data akan mempermudah penambahan

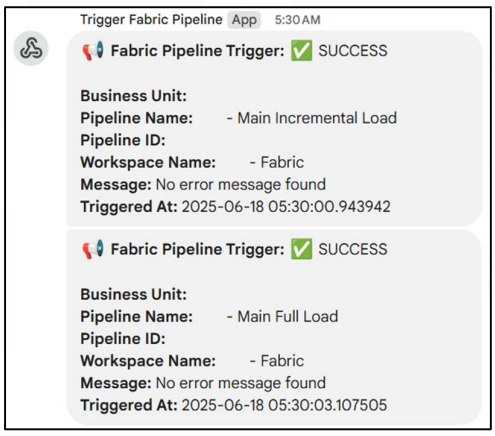
BU baru karena tidak perlu mengubah *script* secara langsung, namun dengan menambahkan baris baru pada spreadsheet. Spreadsheet ini diakses menggunakan *library* gspread dan *service account* dari Google Cloud Platform (GCP) dengan menggunakan *script* pada Gambar 3.49, yang memastikan akses yang aman dan efisien ke data konfigurasi. Setelah semua konfigurasi dan data *pipeline* tersedia, *script* pada Gambar 3.50 akan membaca data dari *spreadsheet* secara berulang untuk tiap baris lalu memfilter berdasarkan BU yang sedang diproses. Jika terdapat baris yang cocok, maka akan dijalankan fungsi run_pipeline dengan ID-ID yang sesuai. Ketika eksekusi *pipeline* selesai, respons dari API akan dikembalikan dan langsung diproses. Jika respons berhasil, maka notifikasi dengan status *success* dikirim ke Google Space. Jika gagal, maka status *failed* dikirim dengan pesan dari API.

```
with TaskGroup(group_id="trigger_fabric_pbi") as trigger_group:
    trigger_fabric = PythonOperator(
        task_id="run_fabric_pbi",
        python_callable=run_fabric_pipeline,
        on_failure_callback=task_fail_alert,
        op_args=[
            "{{ var.json.power_bi_credentials_secret.username }}",
            "{{ var.json.power_bi_credentials_secret.password }}",
            "{{ var.json.power_bi_credentials_secret.client_id }}",
        ],
    )

broadcast = PythonOperator(
    task_id="broadcast_message",
    python_callable=broadcast_message,
    on_failure_callback=task_fail_alert,
)

update_dataset = PythonOperator(
    task_id="update_dataset",
    python_callable=update_log_dataset,
    outlets=[AHI_PBI_DATASET],
    on_failure_callback=task_fail_alert,
)

end_pipeline = EmptyOperator(task_id="end_pipeline")

start_pipeline >> [topic_group, trigger_group] >> broadcast >> update_dataset >> end_pipeline
```

Gambar 3.51 Struktur TaskGroup untuk *Trigger Pipeline* Fabric

Gambar 3.51 menunjukkan *script* struktur *task* DAG yang bertanggung jawab untuk men-*trigger pipeline* Fabric dikelompokkan dalam sebuah TaskGroup, yang berisi task utama untuk menjalankan fungsi run_fabric_pipeline menggunakan PythonOperator. Fungsi ini menerima tiga parameter utama dari Airflow Variable, yaitu *username*, *password*, dan *client* ID, yang digunakan dalam proses autentikasi. TaskGroup ini dihubungkan

langsung setelah task start_pipeline dan dijalankan secara paralel dengan task *refresh dashboard* Power BI sehingga kedua task tersebut dapat berjalan bersamaan tanpa saling menunggu, sehingga jika ada satu BU yang gagal men-*trigger pipeline* Fabric, hal tersebut tidak akan mempengaruhi proses *refresh dashboard* BU lain, dan begitu juga sebaliknya. *Logging* dibuat secara eksplisit agar semua hasil eksekusi baik dari respons API maupun notifikasi yang berhasil atau gagal dikirim ke Google Space tercatat dalam log DAG Airflow. Hal ini memudahkan proses *debugging* jika terdapat ada anomali atau error.



Gambar 3.52 Notifikasi Hasil Eksekusi *Pipeline* Fabric pada Google Space

Otomatisasi *trigger pipeline* Fabric akan membantu sinkronisasi antara proses ETL di Airflow dan visualisasi di Power BI agar dapat berjalan secara efisien. Proses manual untuk men-trigger *pipeline* dari antarmuka Power BI tidak lagi diperlukan, karena semua telah diotomatisasi berdasarkan jadwal DAG. Hasil dari eksekusi dapat dilihat pada Gambar 3.52 dalam bentuk notifikasi Google Space, di mana hasil ini dapat dipantau langsung oleh divisi *Data Analyst* sehingga mereka dapat mengetahui status dari eksekusi *pipeline*.

3.3 Kendala yang Ditemukan

Semasa pelaksanaan kerja magang di Kawan Lama Group, ditemukan beberapa kendala atau kesulitan, di antaranya:

1. Ketidaksesuaian format data antara data yang diminta dengan data yang diterima dari *user*, seperti perbedaan nama kolom, tipe data yang tidak

sesuai, atau penamaan file yang tidak konsisten dengan standar yang telah digunakan dalam sistem.

2. Kurangnya ketelitian baik dari diri sendiri maupun dari anggota tim dalam menangani tugas-tugas tertentu, terutama yang berpengaruh terhadap *pipeline* data harian, sehingga mengharuskan adanya tindakan korektif yang cukup menyita waktu, serta menyebabkan keterlambatan pada eksekusi rutin yang seharusnya berjalan otomatis. Ketika proses ETL terganggu atau mengalami keterlambatan, dampaknya tidak hanya terbatas pada sisi teknis, tetapi juga langsung dirasakan oleh *user*. Banyak *user* dari berbagai divisi, terutama yang memiliki kebutuhan analisis operasional harian, sangat mengandalkan ketersediaan data yang selalu *up to date*, di mana mereka sudah mengharapkan data telah siap untuk diakses. Jika proses ETL gagal atau tertunda, maka laporan atau dashboard yang mereka gunakan tidak akan menampilkan informasi terbaru, yang dapat mengganggu pengambilan keputusan harian maupun proses kerja selanjutnya.
3. Respons yang lambat dari *Person in Charge* (PIC) *task* yang berada di bawah koordinasi supervisor, sehingga menunda penanganan atau penyelesaian *task* yang telah diberikan.

3.4 Solusi atas Kendala yang Ditemukan

Dari kendala-kendala yang telah dihadapi, muncul solusi untuk mengatasi kendala atau kesulitan tersebut, yakni:

1. Berkoordinasi langsung dengan PIC yang bersangkutan agar dapat melakukan penyesuaian data dengan *user* untuk mengeneralisasi format data serta membuat penyesuaian pada *script* agar dapat menangani ketidakkonsistenan data apabila diperlukan, semisal data tidak dapat diubah dari sisi sumber.
2. Menumbuhkan kebiasaan untuk melakukan *cross-check* terhadap setiap pekerjaan baik sebelum maupun sesudah *deployment*, serta membangun budaya kerja yang saling terbuka dan saling mengingatkan antar anggota tim. Sikap yang kritis terhadap pekerjaan sendiri maupun pekerjaan rekan

akan menjadi salah satu bentuk upaya preventif untuk meminimalkan terjadinya kesalahan akibat kelalaian kecil yang bisa berdampak besar pada kelangsungan proses ETL harian.

3. Melakukan *follow-up* dan diskusi internal dalam tim. Apabila PIC yang dituju tidak dapat dihubungi atau membutuhkan waktu lama untuk merespons, maka dapat bertanya atau berkonsultasi kepada Supervisor magang atau rekan tim yang memiliki pengalaman menangani *task* serupa.