BAB 2
LANDASAN TEORI

2.1 Profil Perusahaan

PT Mabra Technology Solution atau biasa disebut Mabratech didirikan pada tahun 2015. Mabratech memulai bisnisnya sebagai perusahaan penyedia jasa konsultasi dan pembangunan perangkat lunak *on-premise*. Jasa konsultasi perangkat lunak yang disediakan oleh Mabratech mulai dari perencanaan, pembangunan, dan optimalisasi perangkat lunak yang berjalan atau pun pembangunan perangkat lunak dari awal.

Sejak awal berdirinya Mabratech telah membangun dan menyediakan layanan perangkat lunak yang berfokus pada keperluan HRIS (Human Resources Information System). Perangkat lunak ini bertujuan pada pengolahan sumber daya manusia pada perusahaan *client* seperti pengolahan data pegawai, absensi, penggajian, pajak karyawan dan permintaan-penerimaan karyawan baru.

Meskipun masih terbilang baru, Mabratech memiliki pengalaman yang cukup dalam pengembangan perangkat lunak terutama perangkat lunak *on-premise*. Berdasarkan pengalaman tersebut, dan melihat pada tren yang ada di pasar saat ini, Mabratech selama dua tahun terakhir aktif dalam pembangunan dan pengembangan perangkat lunak *multi tenant* dengan tujuan untuk menyediakan layanan *Software as a Service* untuk berbagai macam pengguna.

2.2 Multi Tenant Software As A Service

Software as a Service atau sering disingkat SaaS merupakan perangkat lunak yang dijadikan sebagai layanan yang dapat diakses oleh pengguna, pengguna SaaS tidak perlu melakukan proses *deployment* dan perawatan terhadap sistem yang mana jadi tanggung jawab penyedia SaaS. Pada umumnya SaaS merupakan layanan *multi-tenant* atau layanan yang melayani banyak pengguna [9].

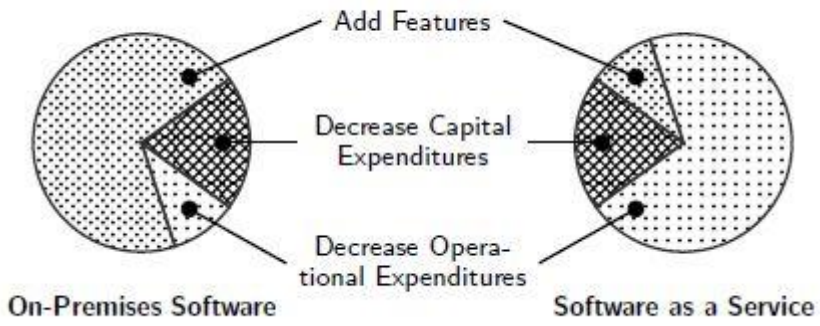
Pada beberapa tahun terakhir, tren "*Everything-as-a-Service*" yang menggunakan model pembayaran *pay-per-use* mendapat banyak perhatian di dunia ICT. Perusahaan-perusahaan mulai meningkatkan adopsi terhadap paradigma ini, dimana perusahaan berusaha untuk mengurangi usaha dalam pengadaan perangkat lunak dan infrastruktur. Sebaliknya, perusahaan yang mengadopsi paradigma ini

hanya akan menggunakan sumber daya tersebut sebagai layanan yang hanya digunakan ketika dibutuhkan tanpa perlu usaha yang besar untuk membuat dan merawatnya. *Cloud computing*, hadir sebagai platform *run-time* yang mewujudkan visi tersebut, menyediakan berbagai layanan mulai dari *Infrastructure-as-a-Service* (IaaS), *Backend-as-a-Service* (BaaS), hingga *Software-as-a-Service* (SaaS) [10].

Dalam sudut pandang penyedia layanan SaaS, keuntungan menyediakan SaaS *multi-tenant* adalah penyedia hanya perlu menyediakan sebuah perangkat lunak yang terpusat dan dapat digunakan oleh banyak penyewa (*tenants*). Hal ini memungkinkan penyedia dapat menekan harga sewa untuk setiap penyewa yang dengan demikian mendorong segmen pasar baru, seperti usaha menengah kebawah yang tidak mampu untuk membeli atau membuat perangkat lunak untuk kebutuhan usahanya [9].

2.2.1 Perbedaan Perangkat Lunak On-Premise dan SaaS

Perangkat lunak *on-premise* merupakan perangkat lunak tradisional yang dirancang dan dibangun untuk memenuhi kebutuhan khusus pada sebuah organisasi atau pelanggan (*single-tenant*) tertentu secara spesifik. Pada perangkat lunak *on-premise* masing-masing pelanggannya akan diberikan sebuah *instance* perangkat lunak yang disesuaikan dengan kebutuhan pelanggannya masing-masing berdasarkan perangkat lunak dasar yang dibuat [9]. Penyesuaian yang dilakukan pada perangkat lunak *on-premise* dapat berupa fitur tambahan, dan penyesuaian bentuk atau skema data data [11].



Gambar 2.1 Prioritas pada perangkat lunak On-Premise dan Software as a Service.

Fokus utama dari perangkat lunak *on-premise* umumnya adalah penyesuaian dalam penambahan fitur. Hal ini tentunya menyebabkan tingginya *cost of ownership* karena tingginya biaya pengembangan. Berbeda dengan perangkat lunak *on-premise*, perangkat lunak SaaS *multi-tenant* dirancang untuk menekan *cost of ownership*. Dalam menekan *cost of ownership* tersebut perangkat lunak SaaS harus dirancang agar dapat menekan kebutuhan akan penambahan fitur [11]. Perbandingan prioritas atau focus diantara perangkat lunak *on-premise* dan perangkat lunak SaaS ini dapat dilihat pada Gambar 2.1 Prioritas pada perangkat lunak

On-Premise dan Software as a Service..

2.2.2 Model SaaS Multi Tenant

Dalam implementasi SaaS *multi tenant*, hal yang sangat perlu untuk diperhatikan adalah bagaimana variabilitas yang dapat muncul karena perbedaan kebutuhan di antara penggunanya. Pada tingkatan abstrak, SaaS merupakan kumpulan dari beberapa layanan, dan masing-masing layanan terdiri dari kumpulan operasi yang dilakukan oleh pengguna. Perbedaan kebutuhan fungsionalitas atas layanan atau pun operasi yang dapat berbeda-beda, menyebabkan perlunya dukungan terhadap variasi dari layanan atau pun operasi yang ada. Mendefinisikan poin-poin yang dapat menghubungkan perbedaan-perbedaan kebutuhan yang ada dapat menjadi acuan untuk implementasi model SaaS. Selain itu, melihat pada serangkaian pola *multi-tenancy* dapat juga membantu membedakan antara

komponen yang akan dibagi di antara semua pelanggan atau khusus untuk beberapa pelanggan. Secara teknis, model tersebut memberikan acuan dasar untuk mendukung variabilitas dalam arsitektur aplikasi SaaS *multi tenant*.

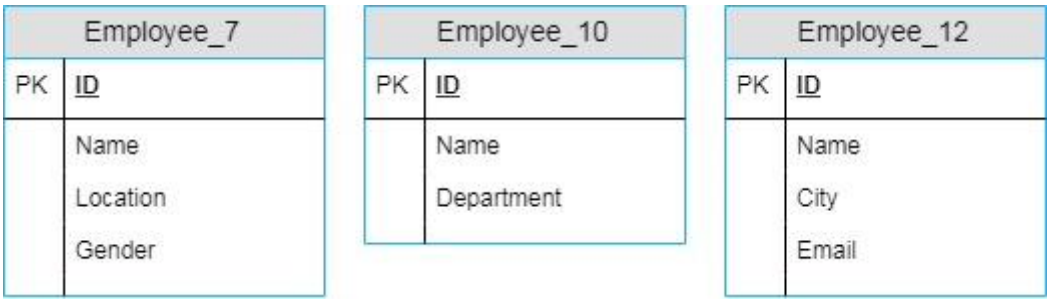
Terdapat beberapa pendekatan untuk menangani variabilitas atas kebutuhan tersebut. Pendekatan pertama, merancang sistem SaaS yang memungkinkan aplikasi-aplikasi di dalamnya dapat disesuaikan untuk memenuhi kebutuhan pengguna baru seiring waktu berjalan. Pendekatan ini dianggap fleksibel dan dapat memastikan kebutuhan seluruh pengguna terpenuhi secara utuh dan lengkap, walaupun dalam implementasinya akan mengalami kesulitan untuk memenuhi prinsip *shareability* pada SaaS. Pendekatan berikutnya adalah merancang sistem SaaS dengan prinsip *fits-for-all* yang mana kustomisasi dijadikan ke dalam bentuk opsi yang dapat dipilih dan disesuaikan langsung oleh pengguna. Pendekatan ini terlihat lebih kaku, membutuhkan perancangan yang lebih rumit dan lebih lama dari pendekatan sebelumnya. Walapun demikian, pendekatan ini dianggap lebih cocok untuk kebanyakan SaaS mengingat pada fokus utama SaaS untuk menekan *cost of ownership* dan kebutuhan akan penambahan fitur.

2.2.3 Skema Data Pada SaaS Multi Tenant

Selain kode sumber, pemodelan skema atau struktur basis data adalah hal yang penting untuk diperhatikan pada perancangan sistem SaaS *multi-tenant*. Untuk dapat memenuhi prinsip *fits-for-all*, skema data harus dibuat sedemikian rupa agar dapat menyesuaikan pada bentuk data dan prosedur yang ada untuk setiap pelanggan.

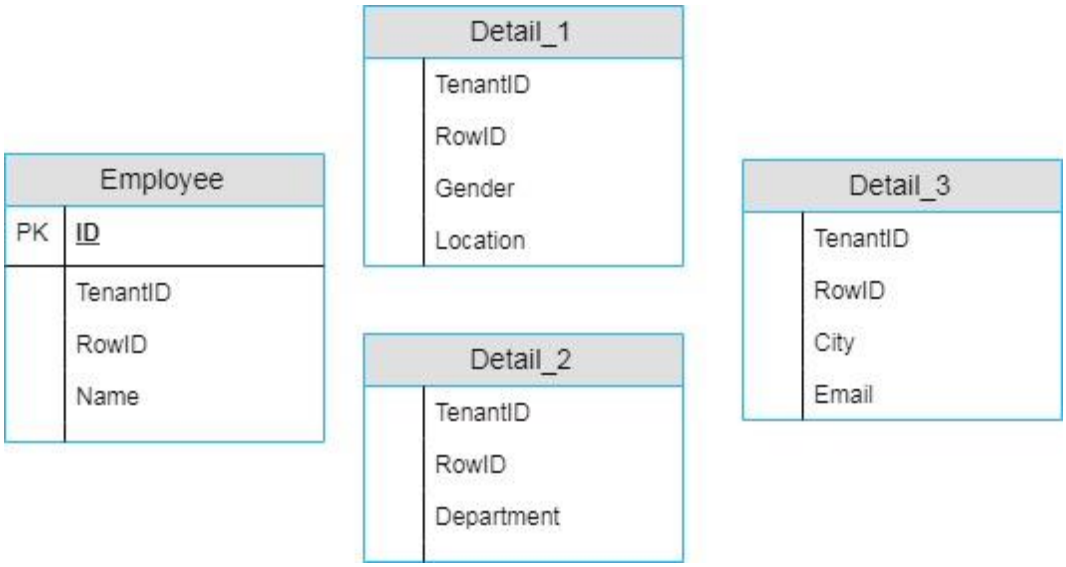
Umumnya, data semua pelanggan tetap disatukan ke dalam basis data yang sama untuk mengurangi total *cost-of-ownership*. Kepemilikan basis data untuk masing-masing pelanggan/*tenant* harus dihindari karena akan menimbulkan masalah dan kerumitan seperti apabila terjadi perubahan skema data yang general atau menyeluruh. Masalah yang sama juga dapat terjadi apabila data pelanggan dibagi ke dalam skema basis datanya masing-masing [11]. Dengan demikian, terdapat beberapa metode yang biasa digunakan agar setiap pelanggan dapat berbagi basis data dengan skema atau struktur yang sama tetapi tetap terpisah dan dapat menangani kebutuhan skema yang berbeda-beda pada masing-masing pelanggan.

a. *Private tables*, memberikan setiap pelanggan tabel-tabel tersendiri dengan tambahan identitas pelanggan ke tabel-tabel tertentu. Pada metode, setiap tabel dapat memiliki atribut yang berbeda-beda disesuaikan dengan masing-masing kebutuhan pelanggan. Kekurangannya adalah jumlah tabel pada basis data akan meningkat secara signifikan semakin bertambahnya jumlah pelanggan dan jumlah tabel yang mungkin akan melewati batas kemampuan basis datanya. Selain itu, menambah kerumitan pada query, karena query yang dibuat perlu disesuaikan dengan identitas pelanggan.



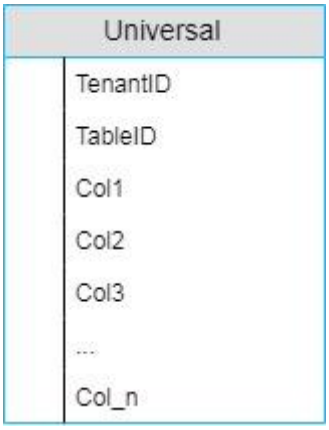
Gambar 2.2 Private Table.

b. *Extension tables*, setiap tabel pada sistem dibuat dengan dalam struktur yang baku dan bersifat general, setiap tabel tersebut harus memiliki kolom id pelanggan(*tenant_id*) sebagai kepemilikan data dan kolom "*row"* yang menyebutkan posisi baris bertujuan untuk proses join antara tabel. Sementara untuk atribut tambahan disimpan ke dalam tabel ekstensi yang disesuaikan.



Gambar 2.3 Extension Table.

c. *Universal tables*, struktur tabel dibuat sangat umum, memiliki jumlah kolom dan bentuk yang bebas. Pada universal tables wajib memiliki setidaknya tiga kolom, kolom "*tenant_id"*, kolom "*table_id*", dan dapat lebih dari satu kolom berisi data atribut tabel. Untuk kolom-kolom dari atribut tabel, kolom tersebut harus memiliki tipe data yang fleksibel seperti VARCHAR dengan kapasitas yang besar dan juga harus dapat memiliki nilai kosong(null). Karena memiliki struktur yang bebas, sistem harus dapat melakukan rekonstruksi kolom-kolom pada tabel dengan atribut yang sebenarnya. Kekurangan dari metode ini dengan jelas terlihat kolom atribut yang diperlukan akan sangat banyak karena akan mengikuti dari tabel yang memiliki atribut terbanyak, juga tipe data yang terlalu fleksibel, pengembang aplikasi harus dengan sangat hati-hati melakukan validasi terhadap data sebelum diinput ke dalam tabel karena tidak ada batasan dari tabel.



Gambar 2.4 Universal Table.

d. *Pivot tables*, merupakan struktur alternatif dari universal tables di mana setiap kolom dari setiap baris terbagi ke dalam baris sendiri. Selain kolom "*tenant_id"*, "*table_id"*, dan "*row"* sebagaimana dijelaskan sebelumnya, struktur ini memiliki kolom "*col"* yang menentukan kolom atau atribut mana yang diwakili oleh baris dan kolom tersebut. Kolom data dapat dibuat spesifik ataupun fleksibel di mana jenis lain dikonversi. Pada dasarnya pivot tables merupakan *universal tables* yang didekomposisi. Untuk mendukung pengindeksan agar lebih efisien, pivot tables dapat dibuat untuk setiap jenis tipe data: satu dengan indeks dan satu tanpa indeks. Setiap nilai ditempatkan tepat di salah satu tabel ini tergantung pada apakah perlu diindeks atau tidak.

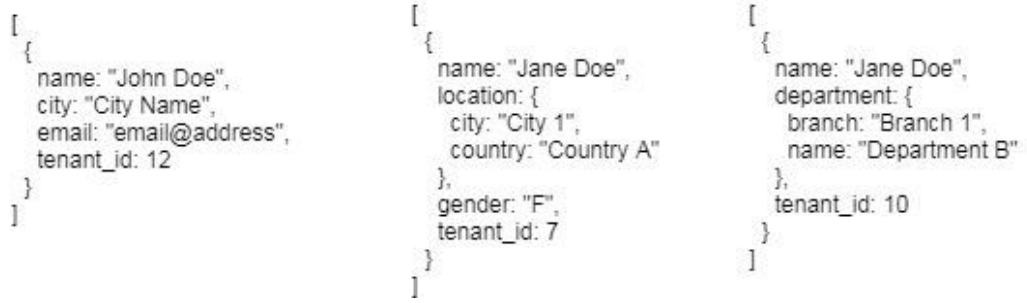
Pendekatan ini menghilangkan kebutuhan untuk menangani banyak nilai kosong (*null*). Namun, pivot tables memiliki lebih banyak kolom meta-data daripada data sebenarnya, merekonstruksi tabel yang memiliki n-kolom akan memerlukan (n - 1) join untuk setiap kolom dan baris. Hal ini dapat menimbulkan masalah overhead runtime yang jauh lebih tinggi hanya untuk menafsirkan meta-data dari jumlah join, ketimbang jumlah yang relatif kecil dalam pada extension table.

Gambar 2.5 Pivot Table.

e. *Schemaless*, seperti namanya pendekatan ini pada dasarnya tidak mengharuskan data untuk memiliki struktur yang baku. Pendekatan ini biasanya hanya dapat dilakukan pada basis data non relasional, hal ini dikarenakan basis data relasional dirancang untuk data yang sudah memiliki struktur dan relasi [12]. Tidak seperti pada basis data relasional, dimana skema data memiliki tabel-tabel yang dapat saling berelasi, bentuk data pada basis data non relasional biasanya berbentuk dokumen yang terkumpul di dalam *collection*. Sebuah *collection* hanya berfungsi seperti layaknya *namespace*, tidak memiliki struktur baku atau mengikat seperti tabel. Dokumen di dalam *collection* biasanya dapat direpresentasikan sebagai JSON(Javascript Object Notation).

   Schemaless dapat memiliki bentuk yang dinamis sangat cocok untuk mendukung kebutuhan data *multi-tenancy*. Walaupun demikian, data *schemaless* memiliki kekurangan, salah satunya adalah proses denormalisasi, denormalisasi diperlukan walaupun memungkinkan duplikasi data. Denormalisasi data dilakukan dengan memasukan data relasi ke dalam dokumen sebagai subdokumen dalam bentuk sub-JSON. Bagaimanapun basis data non-relasional tidak dapat sepenuhnya menggantikan basis data relasional,

pada kebanyakan perangkat lunak, banyak fitur pada basis data relasional tetap diperlukan, seperti relasi, struktur tabel baku, dan juga *transaction*.



Gambar 2.6 Employee Collection.

f. JSON *column*, pendekatan yang terakhir menyimpan struktur data yang berbeda-beda pada kolom tabel yang memiliki tipe data sebagai JSON. Berbeda dengan JSON dalam bentuk *string* biasa, DBMS yang mendukung tipe data JSON bukan hanya dapat menyimpan data dalam bentuk JSON, tetapi juga mendukung *query* terhadap struktur JSON yang tersimpan. Pada pendekatan ini, setiap tabel memiliki struktur yang general untuk menyimpan data general yang ada. Sementara untuk atribut dan struktur tambahan pada setiap kebutuhan khusus *tenant* disimpan dalam bentuk JSON. Hal ini memungkinkan struktur tabel dapat mendukung kebutuhan *multi-tenancy* tanpa memerlukan tabel tambahan di luar tabel general.

Tabel 2.1 Data pegawai dengan kolom data JSON.

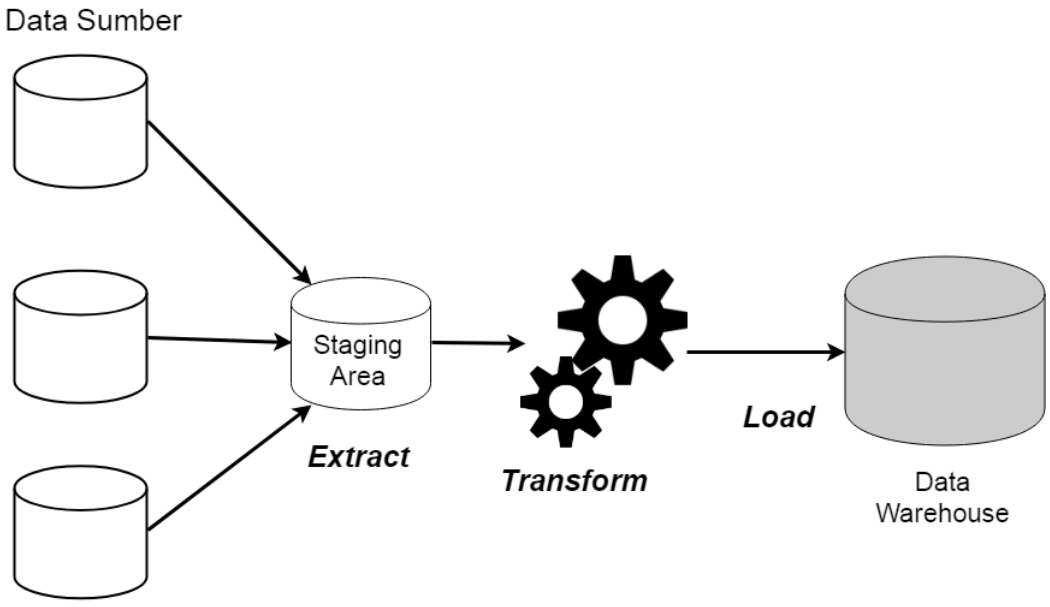
ID	TenantID	Name	Data
100	7	Budi	`{"location": "Bandung", "gender": "L"}`
101	10	Andi	`{"department": "IT"}`
102	12	Ina	`{"city": "Cimahi", "email": "ina@email.com"}`
103	7	Ani	`{"location": "Jakarta", "gender": "P"}`

2.3 Data Warehouse

Data warehouse atau gudang data adalah repositori data gabungan semua data yang dikumpulkan dari berbagai sistem operasional perusahaan, baik fisik maupun logis. *Data warehouse* menekankan pada pengambilan data dari berbagai sumber

data operasional untuk kemudian diakses dan dianalisis menjadi informasi [1]. *Data warehouse* telah digunakan selama bertahun-tahun pada infrastruktur data di berbagai organisasi. Seiring berjalannya waktu, *data warehouse* mengalami banyak perubahan. Seperti *data warehouse* berbasis *cloud*, *scalable*, yang menawarkan kinerja yang baik, dan kemampuan pemrosesan data yang inovatif, semua dengan biaya yang rendah [2].

Data warehouse dapat memberi manfaat bagi penggunanya dari sudut pandang bisnis atau pun teknis, dengan memisahkan data atau pun proses operasional dengan proses analisis. Hal ini dapat meningkatkan kinerja sistem operasional dan memungkinkan pengguna bisnis untuk dapat mengakses data analisis dengan lebih relevan, konsisten, dan lebih cepat sebagai alat untuk pengambilan keputusan. *Data warehouse* pada umumnya terdiri dari tiga lapisan, yaitu data sumber, DSA (*Data Staging Area*), dan bagian utama sebagai gudang data. Tiga lapisan tersebut seperti digambarkan pada Gambar 2.2 masing-masing digunakan berhubungan dengan proses ETL(*Extract, Transform, Load*) [13].



Gambar 2.7 Arsitektur umum *data warehouse*.

2.3.1 ETL

Gambaran umum proses ETL seperti digambarkan pada Gambar 2.7 Arsitektur umum *data warehouse*. Data diekstraksi dari dalam sumber data, lalu diteruskan ke dalam tempat penyimpanan sementara, DSA. Selanjutnya data ditransformasikan dan dibersihkan sebelum akhirnya dimuat ke dalam *data warehouse*. Sumber data dapat memiliki bermacam-macam bentuk struktur atau format, baik berupa basis data relasional, non-relasional, *flat-file*, *spreadsheet*, data log, dan lain-lain [13].

2.3.1.1 Extraction

Tahap awal dari proses ETL merupakan ekstraksi atau memuat data operasional. Data operasional dimuat dari berbagai sumber data yang ada. Setiap sumber data memiliki karakteristik yang berbeda-beda dan memerlukan cara yang berbeda untuk dapat menanganinya secara efektif.

Proses ini harus dapat melakukan integrasi dengan bermacam-macam sistem dari platform yang berbeda-beda, seperti integrasi dengan beberapa sistem basis data, pada sistem operasi yang berbeda dengan protokol komunikasi yang berbeda.

Dalam proses ekstraksi data dari dalam data sumber, proses ETL harus dapat terhubung dengan sistem basis data yang digunakan pada data sumber, dan mengerti skema data yang ada pada data sumber. Proses ekstraksi terbagi lagi ke dalam dua fase, ekstraksi awal atau *initial extraction*, dan ekstraksi perubahan data. Pada tahap ekstraksi awal, seluruh data yang ada di sumber data akan dimuat dan diproses ke dalam *data warehouse*. Proses ini hanya akan dilakukan sekali saja untuk memuat keseluruhan data sumber ke dalam *data warehouse* [1].

Sementara pada proses ekstraksi perubahan data, ETL akan melakukan pembaruan data di *data warehouse* dengan memodifikasi atau pun menambah data berdasarkan perubahan di sumber data. Proses ini biasa disebut *Change Data Capture* (CDC). Proses ini umumnya dilakukan pada siklus waktu tertentu sesuai kebutuhan bisnis dan kemampuan infrastruktur yang ada. Pengambilan perubahan data ini dapat dilakukan dengan banyak cara, misalnya kolom audit, database log, sistem data, atau *delta method*.

2.3.1.2 Transformation

Proses berikutnya pada ETL adalah proses transformasi data. Transformasi data yang telah diekstraksi sebelumnya ini bertujuan untuk melakukan pembersihan dan penyesuaian data agar data yang akan dimuat ke dalam *data warehouse* tetap terjamin kelengkapan, integritas, dan konsistensinya. Pada proses ini terjadi proses mendefinisikan skema *data warehouse* (*star*, *snowflake*, *galaxy*), rincian tabel fakta, dan tabel dimensional. Seluruh aturan transformasi dan skema yang dihasilkan akan disimpan ke dalam *metadata repository*.

2.3.1.3 Loading

Proses memuat data yang telah ditransformasikan ke dalam *data warehouse*. Tahap ini berisi penulisan atau pengubahan data final yang akan diakses secara langsung oleh pengguna atau sistem untuk melakukan tujuan tertentu. Pada *data warehouse* konvensional tahap ini juga termasuk memuat data ke dalam tabel dimensional dan tabel fakta.

2.4 Change Data Capture

Change Data Capture atau biasa disingkat CDC merupakan teknik untuk memonitor operasi perubahan atau penambahan pada suatu basis data operasional [1]. CDC melakukan integrasi data berdasarkan hasil identifikasi, menangkap, dan mengirim perubahan data pada sebuah sistem operasional [14]. Berdasarkan tujuan dari CDC, dapat dilihat bahwa CDC dapat dan telah sering dimanfaatkan pada proses ekstraksi data dalam proses ETL di *data warehouse*.

Proses CDC akan memonitor dan menangkap perubahan atau penambahan data yang terjadi pada data. Dengan CDC, proses ekstraksi pada ETL dapat berjalan lebih efisien karena proses ETL hanya akan memproses data yang ditangkap oleh CDC secara langsung, berbeda dengan proses *batch extraction* dengan jumlah data yang besar.

Pada umumnya, pengaplikasian CDC dapat dikategorikan ke dalam dua kategori, ekstraksi data langsung (*immediate data extraction*) dan ekstraksi data yang ditangguhkan (*deferred data extraction*). Ekstraksi data langsung

memungkinkan ekstraksi data ketika perubahan terjadi pada sumber data secara langsung. Sementara pada ekstraksi data yang ditangguhkan proses ekstraksi dilakukan secara berkala (dalam interval yang ditentukan). Oleh karena itu, data yang diekstraksi adalah data yang telah diubah sejak ekstraksi terakhir [15].

2.4.1 Immediate Data Extraction

Terdapat tiga metode pada ekstraksi data langsung yaitu, menggunakan *transaction log*, *database trigger*, dan ekstraksi langsung pada sistem operasional. Metode CDC yang menggunakan *transaction log*, akan membaca log perubahan yang dihasilkan oleh DBMS. Log transaksi akan mencatat semua kejadian *Data Manipulation Language* seperti *Insert, Update,* dan *Delete*. Proses CDC akan membaca log lalu mencari dan mengolah data yang baru ditambahkan atau diubah berdasarkan dari log transaksi.

Metode CDC yang selanjutnya adalah *database trigger*, proses CDC dilakukan dengan memanfaatkan fitur *trigger* yang disediakan DBMS. *Trigger* perlu dibuat pada entitas-entitas yang terkait. Selanjutnya fungsi *trigger* akan melakukan pengolahan data yang diteruskan dari *trigger*. Metode yang terakhir merupakan proses CDC yang dilakukan pada sistem operasional. Metode ini dilakukan di luar lingkungan DBMS, dimana pengembang sistem perlu melakukan proses CDC pada setiap proses transaksi yang terkait.

Dari ketiga metode ekstraksi data langsung, metode CDC dengan membaca log transaksi akan membutuhkan usaha dan waktu yang lebih sedikit ketimbang dua metode lainnya. Proses CDC dengan log transaksi, hanya perlu membaca log lalu dapat secara dinamis menangkap dan meneruskan data berdasarkan log transaksi yang dibaca, sehingga tidak akan terlalu terpengaruh oleh perubahan pada skema basis data [13].

Beda halnya dengan metode CDC dengan *database trigger* dan CDC pada sistem operasional dimana pengembang sistem atau database administrator harus melakukan *data capture* untuk setiap entitas atau transaksi. Hal ini memerlukan usaha dan waktu yang cukup besar, juga lebih rentan terhadap kesalahan dibandingkan dengan metode dengan log transaksi [3]. Walaupun dalam implementasi CDC dengan log transaksi pun memerlukan usaha yang cukup besar, sistem CDC seperti Debezium dapat menjadi solusi. Debezium merupakan platform

untuk melakukan CDC yang membaca log transaksi basis data dan mendukung beberapa DBMS populer [16].

2.4.2 Deferred Data Extraction

Deferred Data Extraction atau ekstraksi data yang ditangguhkan merupakan proses ekstraksi dilakukan secara berkala pada interval atau periode tertentu dimana data yang diekstraksi merupakan data yang baru atau telah diubah sejak ekstraksi terakhir. Terdapat dua metode pada ekstraksi data yang ditangguhkan, pertama dengan melihat pada *timestamp* data, dan mencari perbedaan antara set data terbaru dan set data saat proses ekstraksi terakhir dilakukan.

Metode ekstraksi dengan melihat pada waktu (*timestamp*) dapat dilakukan apabila entitas data memiliki atribut waktu yang mencatat waktu data tersebut disimpan, diubah atau dihapus (*soft delete*). Atribut tersebut diisi sesuai dengan waktu ketika suatu aktivitas penyimpanan, perubahan, atau penghapusan terjadi. Selanjutnya proses CDC akan melihat data yang atribut waktunya sama atau lebih besar dari waktu ketika proses CDC dilakukan.

Metode yang kedua dengan membandingkan setiap data pada ke dua set data. Perbedaan yang ditemukan pada data diantara kedua set akan dimasukan ke dalam set data yang telah diubah, dan akan diteruskan ke proses selanjutnya. Metode dianggap lebih fleksibel dan lebih akurat dibandingkan dengan metode dengan melihat *timestamp* data, karena akan membandingkan setiap atribut pada setiap *record* di dalam set data. Metode dengan membandingkan dua set data ini memiliki kelemahan yaitu memakan waktu yang lama dan membutuhkan memori yang cukup besar [13].

2.5 Debezium

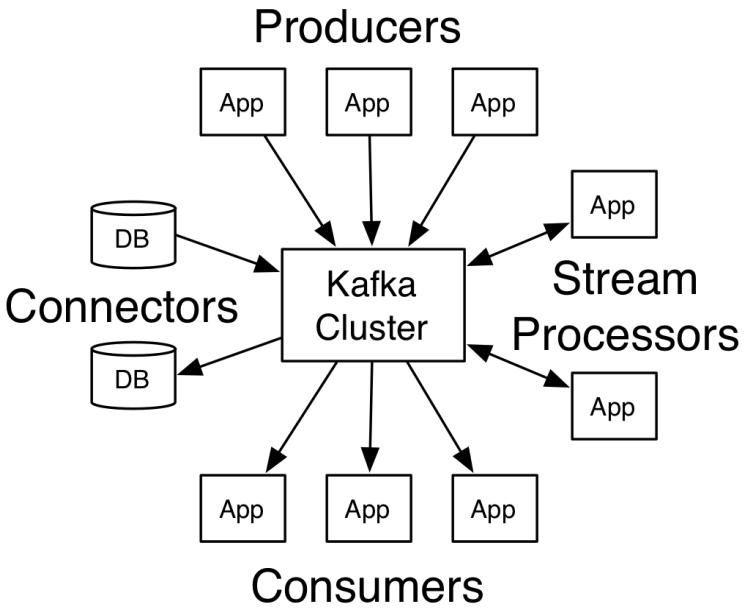
Debezium merupakan platform terdistribusi yang memonitor perubahan data pada basis data. Debezium mengubah perubahan data tersebut ke dalam bentuk *event stream*, sehingga aplikasi penerima dapat secara langsung melakukan menangani perubahan pada setiap *record*. Debezium dibangun diatas Apache Kafka dan menyediakan konektor yang *compatible* dengan Kafka Connect. Konektor yang disediakan oleh Debezium akan memonitor perubahan pada DBMS spesifik.

Debezium akan menyimpan riwayat perubahan pada basis data dalam bentuk *Kafka log*. Log perubahan akan tersimpan dalam bentuk JSON atau Avro. Pada masing-masing log terdapat rincian perubahan, seperti data baru dan data lama, tipe perubahan, dan skema entitas tempat data berada. Selanjutnya aplikasi dapat dengan mudah mengkonsumsi perubahan tersebut pada Apache Kafka [16]. Berikut daftar DBMS yang didukung oleh Debezium:

1. MySQL
2. PostgreSQL
3. SQLServer
4. Oracle
5. MongoDB

2.6 Apache Kafka

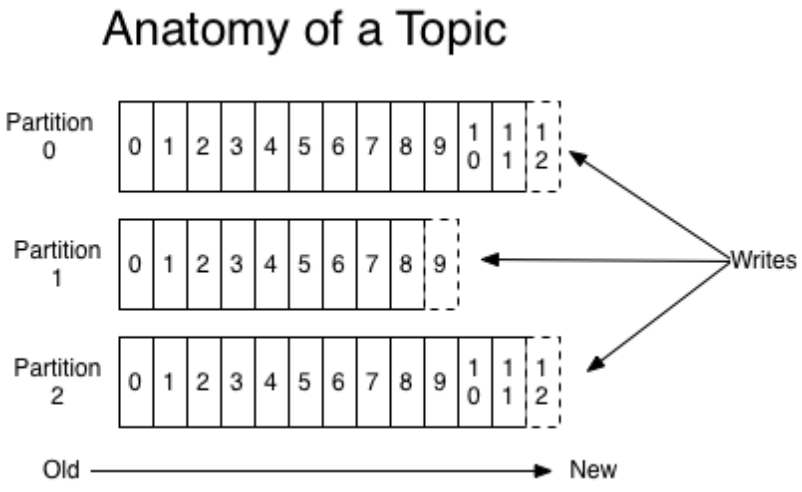
Apache Kafka adalah platform perangkat lunak *stream-processing open source* yang dikembangkan oleh LinkedIn dan kemudian disumbangkan ke Yayasan Perangkat Lunak Apache, ditulis dalam Scala dan Java. Proyek ini bertujuan untuk menyediakan platform yang tergabung, *throughput* tinggi, dengan latensi rendah untuk menangani data secara *real time* [7]. Pada dasarnya, lapisan penyimpanan pada Kafka adalah antrian pesan *publish/subribe* yang *scalable* dan terdistribusikan, membuat Kafka menjadi sangat bernilai bagi infrastruktur perusahaan untuk memproses *streaming* data. Selain itu, Kafka dapat terhubung ke sistem eksternal (untuk impor atau ekspor data) melalui Kafka Connect dan menyediakan Kafka Streams [17].



Gambar 2.2.8 API inti pada Apache Kafka.

2.6.1 Topic dan Logs

Topic merupakan kategori atau nama jenis dari data yang ada Kafka. *Topic* pada Kafka tidak terikat pada satu *subscriber* tertentu, melainkan dapat memiliki banyak *subscriber* atau pun tidak sama sekali. Kafka melakukan partisi pada setiap topik yang ada, seperti digambarkan pada gambar dibawah ini.



Gambar 2.2.9 Log partisi pada topik.

Setiap partisi merupakan antrian sekuensial yang berurutan, dimana setiap data atau *record* baru akan ditambahkan ke dalam urutan tersebut secara terus

menerus. Setiap data di dalam sebuah partisi diberikan nomor identitas atau disebut *offset* yang tujuannya untuk memberikan identitas unik terhadap *record* di dalam sebuah partisi.

Kafka akan menyimpan semua data yang dimuat, selama masa penyimpanan masih berlaku berdasarkan konfigurasi masa penyimpanan, walaupun data telah dikonsumsi. Sebagai contoh, apabila masa penyimpanan dikonfigurasi sebagai dua hari, maka data akan tersimpan dan tersedia untuk dikonsumsi selama dua hari sejak saat data tersebut dimuat. Melewati masa penyimpanan *record* akan dihapus untuk mengosongkan.

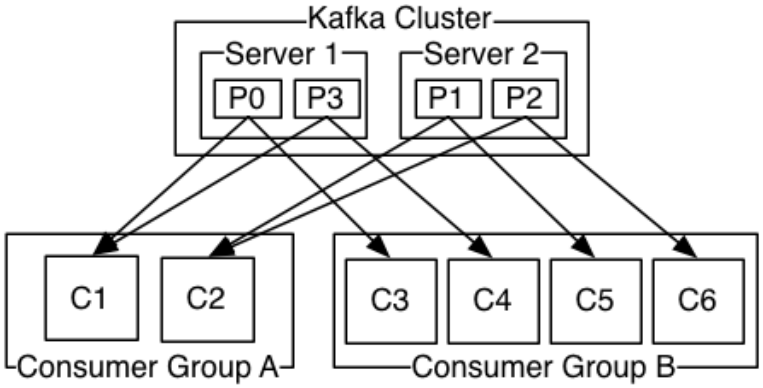
2.6.2 Produser

Produser merupakan sistem eksternal yang akan memuat data ke dalam topik. Produser bertanggung jawab untuk menentukan pastisi pada topik dimana data akan disimpan. Hal ini dapat dilakukan dengan menggunakan teknik *round-robin* untuk menyeimbangkan penyimpanan atau dapat disesuaikan dengan fungsi tertentu.

2.6.3 Konsumen

Konsumen merupakan sistem eksternal yang akan mengkonsumsi data pada topik yang ada. Konsumen perlu melabeli dirinya sendiri dengan nama grup konsumen. Setiap data yang dimuat ke dalam sebuah topik hanya akan dikirim ke satu konsumen pada setiap grup konsumen yang melakukan *subscribe* pada topik tersebut. Konsumen pada grup konsumen yang sama dapat berjalan pada proses yang berbeda atau bahkan pada mesin yang berbeda.

Jika semua konsumen memiliki grup yang sama, maka data akan dimuat secara merata untuk masing-masing konsumen. Sementara apabila konsumen memiliki grup yang berbeda, maka data akan dimuat ke semua proses konsumen.



Gambar 2.10 Pembagian data antara grup konsumen.

2.6.4 Distribusi

Partisi log pada Kafka dapat didistribusikan ke dalam banyak server dimana setiap server akan menangani data dan permintaan untuk sebagian dari partisi. Setiap partisi akan direplikasi ke dalam sejumlah server untuk menangani apabila terjadi kesalahan atau kegagalan sistem.

Setiap partisi memiliki sebuah server yang berperan sebagai “pemimpin” dan dapat memiliki server lain sebagai ”pengikut”. Server yang berperan sebagai pemimpin akan menangani seluruh pembacaan dan penulisan permintaan pada partisi sementara server pengikut akan mereplikasi pemimpinnya. Jika terjadi kesalahan pada server “pemimpin”, salah satu server “pengikut” akan menggantikannya sebagai pemimpin.

2.7 JSON

JavaScript Object Notation (JSON) adalah sebuah format teks yang ringan dan tidak bergantung pada bahasa pemrograman tertentu, bertujuan untuk melakukan serialisasi terhadap struktur data. JSON merupakan turunan dari literal object di bahasa JavaScript, seperti didefinisikan di ECMAScript Programming Language Standard, edisi ke tiga (ECMA-262). JSON dapat merepresentasikan empat jenis data tipe data primitif (string, angka, boolean, dan null), dan dua tipe data struktural (objek dan array) [18]. Berikut rincian dari tipe data yang didukung didalam JSON:

1. String di dalam JSON merupakan kumpulan atau urutan dari nol atau lebih karakter Unicode.

2. Tipe data angka pada JSON dapat berupa angka desimal, negatif atau positif dan dapat menggunakan notasi eksponensial "*E*", tetapi tidak dapat menggunakan karakter lain seperti "*NaN*" seperti pada JavaScript. Untuk ukuran maupun presisi dari angka yang ada pada JSON, hal ini bergantung pada implementasi yang diterapkan oleh encoder JSON pada masing-masing bahasa pemrograman.
3. Boolean, JSON mendukung sintaks boolean yang ada pada JavaScript yaitu "*true*" dan "*false*".
4. Null ditulis sebagai "*null*" pada JSON, melambangkan nilai kosong.
5. Sebuah objek pada JSON merupakan koleksi dari nol atau lebih pasangan kunci dan nilai (key-value pair) yang tidak berurutan, dimana kunci yang ada pada objek harus memiliki tipe data *string* sementara nilainya dapat memiliki tipe data apapun yang diperbolehkan di dalam JSON (string, angka, boolean, null, objek, *array*).
6. *Array* di dalam JSON merupakan sekumpulan dari nol atau lebih nilai dengan tipe data bebas yang kuncinya berupa angka yang berurutan dan berfungsi sebagai indeks. Istilah objek dan *array* merupakan konvensi yang digunakan pada JavaScript.

JSON merupakan format yang sangat popular di kalangan pengembang perangkat lunak. Tidak sedikit teknologi yang memanfaatkan JSON sebagai format data dengan berbagai tujuan seperti pada API dan RPC, file konfigurasi, hingga format penyimpanan pada sistem basis data relasional atau pun non-relasional. Hal ini dikarenakan JSON memiliki banyak kelebihan, diantaranya:

1. JSON merupakan notasi yang diambil dari JavaScript. Notasi JSON memiliki aturan yang lebih ketat ketimbang notasi objek pada JavaScript memastikan kesesuaian penulisan.
2. JSON merupakan format data *self-describing*, dimana JSON tidak memerlukan meta data untuk menjelaskan struktur dari data yang ada pada JSON. Dengan demikian data yang direpresentasikan dengan JSON akan mudah diolah baik dengan struktur data yang sederhana atau pun rumit seperti data hirarki atau bersarang.

3. Sederhana dan kompak, karena tidak memerlukan data tambahan seperti halnya meta data, data JSON akan mudah dan ringan untuk dikirimkan antara sistem di berbagai platform.
4. Tipe data pada JSON merupakan tipe data dinamis (*dynamic type*), sehingga bentuk data JSON dapat sangat bervariasi dan berbeda-beda.

Berdasarkan kelebihan-kelebihan diatas, JSON merupakan format data yang dianggap cocok untuk penyimpanan data atau pun untuk transfer data. Seperti pada basis data non-relasional atau NoSQL yang mendukung JSON secara *native*, dimana data pada basis data NoSQL atau sering disebut *document* direpresentasikan dalam bentuk JSON. Bentuk data JSON yang *schema-less* atau tanpa skema yang baku menyebabkan sistem dapat dengan mudah mengolah data di dalam basis data tanpa perlu mendefinisikan atau mengubah skema datanya terlebih dulu. Hal ini sangat penting terutama pada sistem yang mengolah data yang bervariasi dan dinamis, dimana pengembang perangkat lunak tidak perlu secara manual memperbarui dan melakukan normalisasi ulang terhadap skema seperti pada basis data relasional.

2.8 PostgreSQL

PostgreSQL adalah sebuah sistem basis data relasional *open* source dengan lisensi BSD. Perangkat lunak ini merupakan salah satu basis data yang paling banyak digunakan saat ini. PostgreSQL pertama kali dikembangkan pada tahun 1986 sebagai bagian dari proyek POSTGRES di University of California di Berkeley. Semenjak awal pengembangannya, PostgreSQL terus aktif dikembangkan selama lebih dari 30 tahun.

PostgreSQL memiliki reputasi yang baik dan terbukti dapat diandalkan, menjaga integritas, kokoh, dan *extensible*. PostgreSQL didukung oleh komunitas *open source* yang secara konsisten mengembangkan PostgreSQL. PostgreSQL dapat berjalan di banyak sistem operasi, dan memenuhi prinsip ACID sejak 2001.

PostgreSQL memiliki banyak fitur untuk membantu pengembang perangkat lunak atau administrator untuk menjaga integritas data dan membangun lingkungan *fault-tolerant*, dengan jumlah data yang sangat besar sekalipun. PostgreSQL juga

sangat *extensible*, dapat disesuaikan sesuai kebutuhan tanpat perlu melakukan kompilasi ulang terhadap sistemnya [19].

2.9 Go

Go merupakan bahasa pemrograman yang ekspresif, ringkas, rapih, dan efisien. Go memiliki mekanisme *concurrency* yang mudah dan efisien, sehingga memudahkan *programmer* untuk mengembangkan perangkat lunak *multicore* yang berjalan pada suatu jaringan. Go juga memiliki fitur yang memungkinkan pengembangan perangkat lunak yang fleksibel dan modular [20].

Go dirancang pada tahun 2007 oleh Robert Griesemer, Rob Pike, dan Ken Thompson di Google. Go bertujuan untuk meningkatkan kinerja *programmer* di era perangkat lunak *multicore*, jaringan, dan basis kode sumber yang besar [21]. Pada proses perancangannya, para perancang Go mengkritik bahasa-bahasa lain yang digunakan oleh Google, tetapi tetap mempertahankan karakteristik yang dianggap baik, seperti:

1. *Static typing* dan efisien (C++ atau Java).
2. Keterbacaan dan kemudahan (Python atau JavaScript).
3. *High-performance networking* dan *multiprocessing*.

2.10 UML (Unified Modelling Language)

UML adalah keluarga notasi grafis yang didukung oleh meta-model tunggal, yang membantu pendeskripsian dan desain sistem perangkat lunak, khususnya sistem yang dibangun menggunakan pemrograman berorientasi objek [22].

Unified Modeling Language (UML) adalah suatu alat untuk memvisualisasikan dan mendokumentasikan hasil analisa dan desain yang berisi sintak dalam memodelkan sistem secara visual. Juga merupakan satu kumpulan konvensi pemodelan yang digunakan untuk menentukan atau menggambarkan sebuah sistem software yang terkait dengan objek.

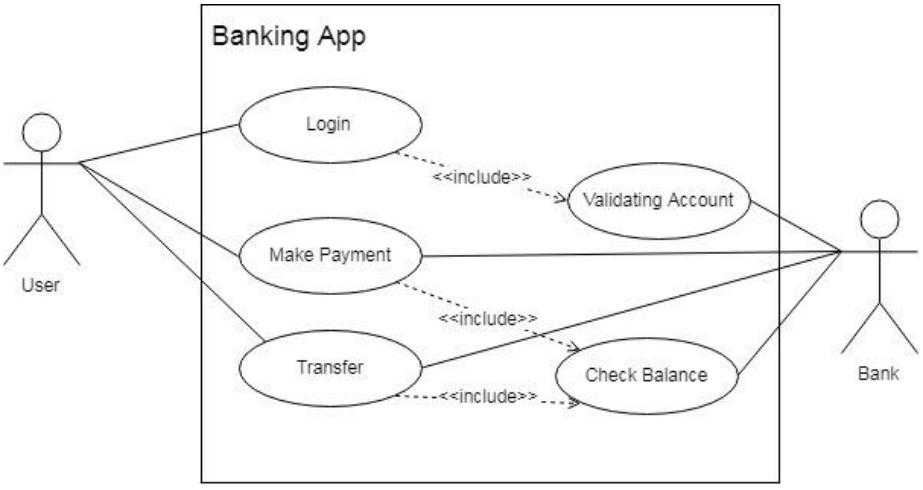
Sejarah UML sendiri terbagi dalam dua fase sebelum dan sesudah munculnya UML. Dalam fase sebelum, UML sebenarnya sudah mulai diperkenalkan sejak tahun 1990, namun notasi yang dikembangkan oleh para ahli analisis dan desain berbeda-beda, sehingga dapat dikatakan belum memiliki standarisasi.

Saat ini sebagian besar para perancang sistem informasi dalam menggambarkan informasi dengan memanfaatkan UML diagram dengan tujuan utama untuk membantu tim proyek berkomunikasi, mengeksplorasi potensi desain, dan memvalidasi rancangan arsitektur perangkat lunak atau pembuat program. Secara filosofi UML, diilhami oleh konsep yang telah ada yaitu konsep pemodelan Object Oriented karena konsep ini menganalogikan sistem seperti kehidupan nyata yang didominasi oleh objek dan digambarkan atau dinotasikan dalam simbol-simbol yang cukup spesifik.

UML memiliki beberapa diagram antara lain: use case diagram, class diagram, activity diagram, sequence diagram. Berikut ini penjelasan untuk beberapa diagram yang akan digunakan dalam penelitian ini [22].

2.10.1 Use Case Diagram

Use Case Diagram adalah pola urutan langkah-langkah yang menggambarkan interaksi antara sistem dan aktor yang berhubungan dalam domain sistem [18]. Aktor merupakan sebuah entitas, baik manusia atau pun sistem eksternal yang berinteraksi dengan sistem, pada *use case diagram* aktor digambarkan dengan simbol aktor. Sementara *use case* merupakan menjelaskan aktifitas atau interaksi yang terjadi, digambarkan dengan symbol oval.

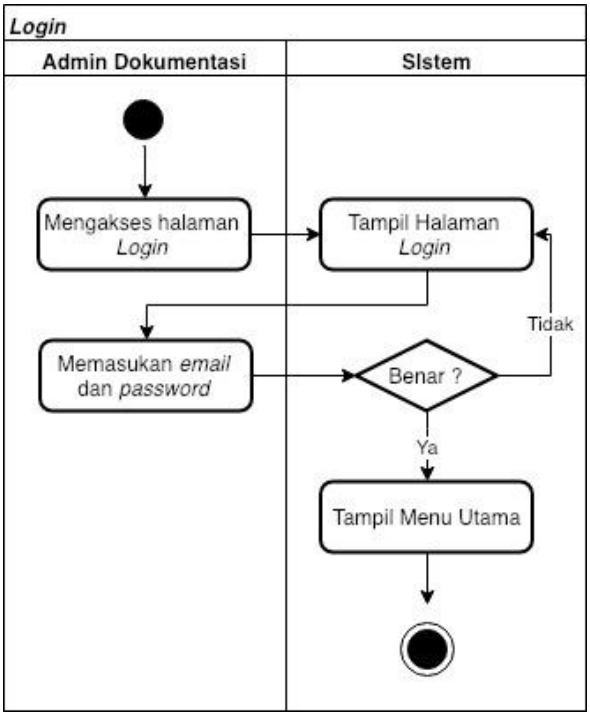


Gambar 2.11 Contoh Use Case Diagram.

2.10.2 Activity Diagram

Activity diagram adalah teknik untuk menggambarkan logika prosedural, proses bisnis, dan jalur kerja. Dalam beberapa hal, *activity diagram* memainkan

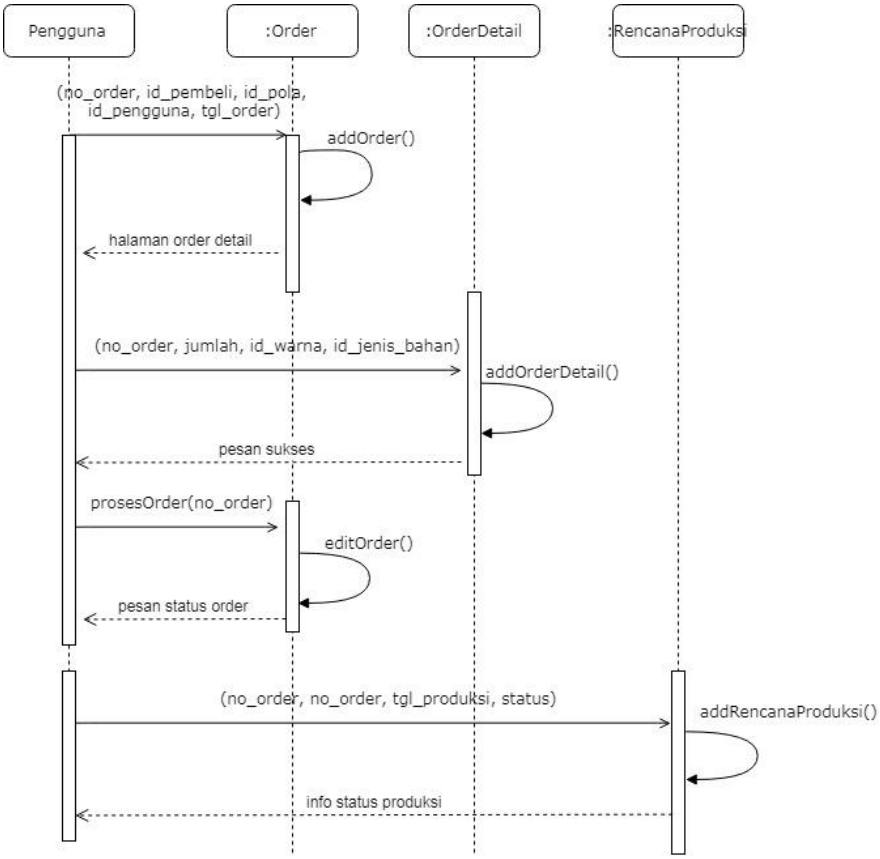
peran mirip diagram alir, tetapi perbedaan prinsip antara notasi diagram alir adalah *activity diagram* mendukung *behavior* paralel. *Node* pada sebuah *activity diagram* disebut sebagai *action*, sehingga diagram tersebut menampilkan sebuah *activity* yang tersusun dari *action* [22].



Gambar 2.12 Contoh Activity Diagram.

2.10.3 Sequence Diagram

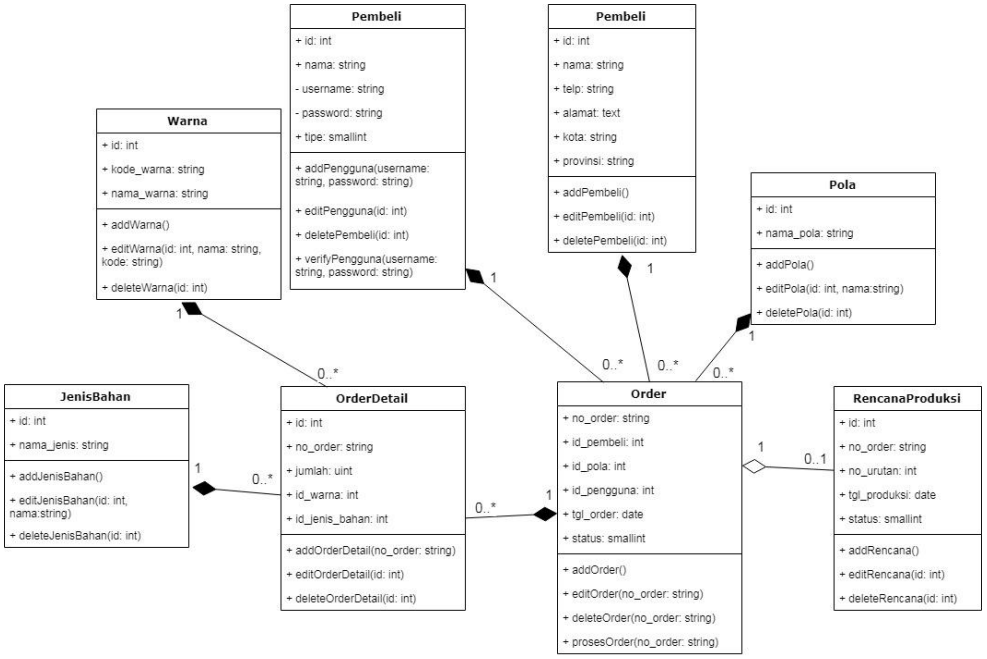
Sequence diagram adalah grafik dua dimensi dimana objek ditunjukkan dalam dimensi horizontal, sedangkan *lifeline* ditunjukkan dalam dimensi vertikal [23].



Gambar 2.13 Contoh Sequence Diagram.

2.10.4 Class Diagram

Class diagram merupakan himpunan dari objek-objek yang sejenis. Sebuah objek memiliki keadaan sesaat (*state*) dan perilaku (*behavior*). *State* sebuah objek adalah kondisi objek tersebut yang dinyatakan dalam *attribute/properties*. Sedangkan perilaku suatu objek mendefinisikan bagaimana sebuah objek bertindak/beraksi dan memberikan reaksi [23].



Gambar 2.14 Contoh Class Diagram.