*Notes :
Dalam melakukan praktek pada modul ini, kalian bisa menggunakan sample data public dari snowflake bernama covid19_epidemiological_data. Semua yang dicontohkan pada modul ini yang berbasis pada snowflake menggunakan data tersebut.

Best Practice for Query

Mengquery data secara efisien sangat penting untuk mendapatkan hasil yang akurat dan tepat waktu. Berikut adalah beberapa best practice saat kita mengquery data:

1. **Dengan mengikuti kaidah format penulisan yang baik**, akan memudahkan query di
   baca dan dimengerti. Contoh :
   - Konsistensi dalam penulisan Query
   - SQL Syntax gunakan UPPER CASE
   - Untuk major Syntax (SELECT, WHERE, FROM, GROUP BY, ORDER BY, LIMIT), buat menjadi baris baru
   - Tambah indentasi untuk Syntax seperti berikut (Sub-query, ON, AND/OR, Beberapa kolom yang mau di extraksi)
   - Gunakan alias untuk query Panjang / sub query
   - Selalu tutup query dengan titik-koma(;) sebagai penutup
   - Tambahkan komentar dalam query untuk memberikan informasi mengenai sebagian atau keseluruhan dari query yang ditulis
   - Komentar ditulis dengan menggunakan tanda berikut
     -- untuk satu baris
     /* untuk komentar banyak baris */

```
SELECT
    county, total_population, total_male_population,
    total_female_population
FROM
    covid19_epidemiological_data.public.demographics
WHERE
    state='TX'
ORDER BY
    total_male_population DESC;
```

2. **Optimalkan Query:**
   a. Tulis query secara efisien dengan mengoptimalkan kode SQL kalian.
   b. Gunakan indeks dengan bijak untuk mempercepat pengambilan data.
   c. Pahami distribusi data dan desain query yang memanfaatkannya.
3. **Batasilah Data yang di return:**
   a. Ambil hanya kolom yang dibutuhkan daripada memilih semua (*).
   b. Gunakan klausa LIMIT untuk membatasi jumlah baris yang dikembalikan selama pengembangan dan pengujian.
4. **Filter dan Order Data:**
   a. Terapkan filter untuk membatasi jumlah data yang diproses.

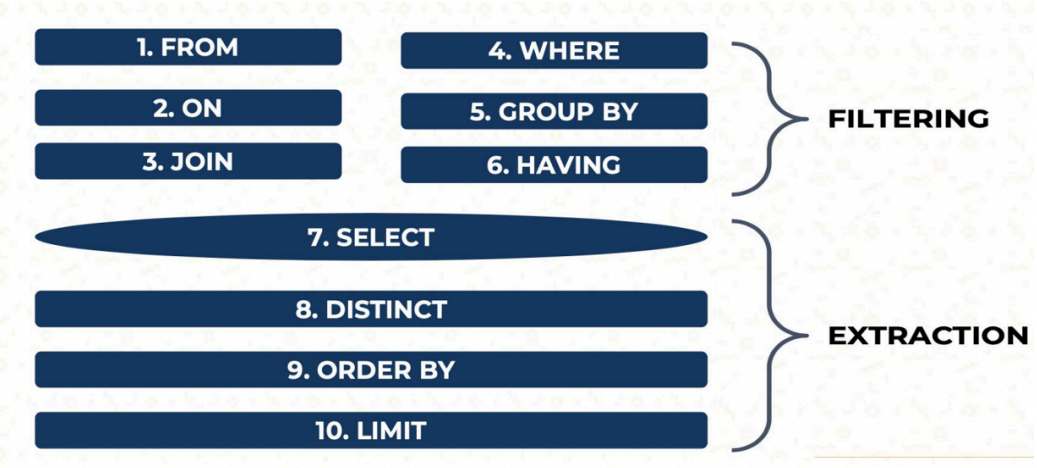
    b. Gunakan klausa WHERE untuk menyaring data sedini mungkin dalam query.
    c. Manfaatkan indeks untuk kolom yang digunakan dalam klausa WHERE.
    d. Pertimbangkan penggunaan ORDER BY dengan bijak untuk menyortir set hasil besar.
5. **Gunakan Join dengan Efisien:**
    a. Pahami jenis join (INNER, LEFT, RIGHT, dll.) dan pilih yang sesuai untuk kasus kalian.
    b. Pastikan kondisi join efisien dan gunakan indeks jika diperlukan.
6. **Pertimbangkan Tipe Data:**
    a. Gunakan tipe data yang sesuai untuk kolom untuk meminimalkan ruang penyimpanan dan meningkatkan kinerja query.
    b. Pahami konversi tipe data implisit yang dapat mempengaruhi kinerja.
7. **Gunakan Caching Bila Sesuai:**
    a. Manfaatkan mekanisme caching, jika tersedia, untuk mengurangi beban pada database.
8. **Hindari SELECT * di Production:**
    a. Hindari menggunakan SELECT * pada query produksi untuk mencegah transfer dan pemrosesan data yang tidak perlu.
9. **Pahami Rencana Eksekusi Query:**
    a. Analisis rencana eksekusi query untuk memahami bagaimana database memproses query Anda.
    b. Gunakan EXPLAIN (atau setara) untuk mendapatkan wawasan tentang rencana eksekusi query.
10. **Monitor dan Optimalkan Secara Berkala:**
    a. Monitor secara berkala kinerja query dan optimalkan query sesuai kebutuhan.
    b. Identifikasi dan atasi query yang berjalan lambat untuk menjaga efisiensi sistem.

Ingatlah bahwa best practice dapat bervariasi tergantung pada sistem database yang digunakan (misalnya MySQL, PostgreSQL, Snowflake) dan sifat data dan querynya. Selalu merujuk pada dokumentasi database tertentu Anda untuk tips optimisasi yang spesifik untuk platform.

Urutan Pemrosesan Query

Urutan operasi dalam eksekusi query SQL dapat bervariasi tergantung pada jenis query, struktur tabel, dan indeks yang digunakan. Namun, berikut adalah urutan umum operasi yang terjadi dalam eksekusi query:

1. **Parsing dan Analisis:**
    - Pertama, database membaca dan menganalisis query yang diberikan. Ini melibatkan memeriksa sintaks dan memeriksa apakah query dapat dijalankan.
2. **Optimasi Query:**
    - Database mengevaluasi query dan menciptakan rencana eksekusi query yang optimal. Hal ini melibatkan pemilihan indeks, perhitungan statistik, dan pengaturan optimal lainnya.
3. **Seleksi Tabel (Table Selection):**
    - Jika query melibatkan beberapa tabel, database memilih urutan untuk memilih tabel yang akan digunakan dalam query. Hal ini melibatkan keputusan tentang urutan join atau cara gabung tabel.
4. **Filtering (WHERE Clause):**
    - Jika query memiliki klausa WHERE, database menerapkan filter untuk membatasi jumlah baris yang diambil dari tabel.
5. **Joining Tabel (Join Operations):**
    - Jika query melibatkan operasi join, database menyatukan data dari tabel yang berbeda berdasarkan kriteria yang ditentukan.
6. **Grouping (GROUP BY):**
    - Jika ada klausa GROUP BY, database mengelompokkan data berdasarkan kriteria yang ditentukan.
7. **Pengurutan (ORDER BY):**
    - Jika query memiliki klausa ORDER BY, database mengurutkan hasil query berdasarkan kolom yang ditentukan.
8. **Proyeksi Kolom (SELECT Clause):**
    - Database mengambil kolom yang diperlukan dari hasil query sesuai dengan klausa SELECT.
9. **Paging (LIMIT/OFFSET atau FETCH):**
    - Jika query melibatkan paging, database membatasi jumlah baris yang dikembalikan ke hasil query.
10. **Eksekusi Query:**
    - Setelah rencana eksekusi query diterapkan, database secara fisik mengeksekusi query dan mengambil data yang diperlukan dari tabel.



11. **Penyusunan Hasil:**
    - Database menyusun hasil query dalam format yang sesuai untuk dikirim ke aplikasi atau pengguna akhir.
12. **Pengiriman Hasil:**
    - Hasil query dikirim kembali ke aplikasi atau antarmuka pengguna.

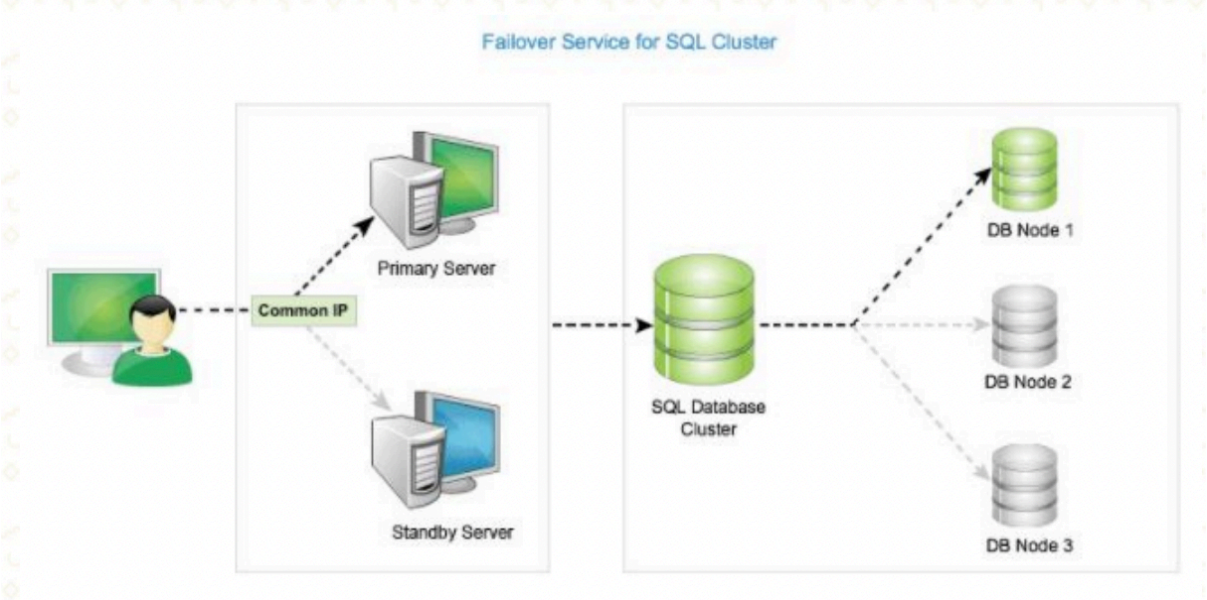
Penting untuk dicatat bahwa database biasanya dirancang untuk menjalankan operasi-operasi ini seefisien mungkin, dan strategi eksekusi dapat bervariasi tergantung pada optimasi yang diterapkan oleh database tersebut.

DB Clustering

Database Clustering adalah suatu pendekatan di mana beberapa server database bekerja bersama sebagai satu kesatuan terkoordinasi. Tujuan dari clustering adalah untuk meningkatkan ketersediaan sistem, ketahanan terhadap kegagalan, dan kinerja. Dalam konteks database clustering, beberapa server atau node bekerja bersama untuk menyediakan layanan database yang handal dan tahan terhadap kegagalan.

Ada beberapa konsep / tipe dari DB Clustering, beberapa yang umum dari konsep itu akan dijabarkan dalam sub kategori dibawah ini

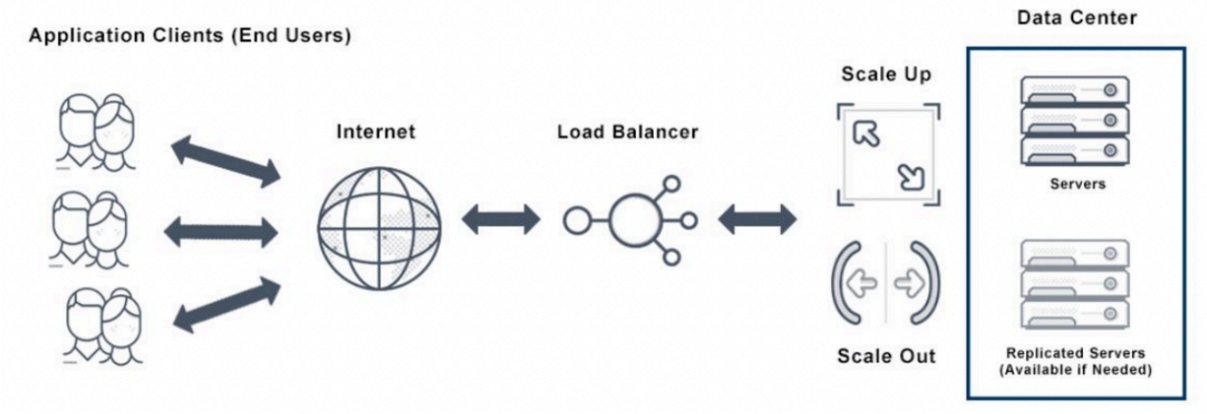
Failover / High Availability Clusters



High Availability (HA) Database Clustering adalah pendekatan untuk merancang dan mengelola sistem database agar tetap aktif dan responsif sepanjang waktu, dengan tujuan untuk meminimalkan waktu downtime atau ketidaktersediaan layanan. Dalam konteks clustering, HA diimplementasikan untuk memastikan bahwa jika satu server atau komponen mengalami kegagalan, sistem tetap dapat beroperasi melalui sumber daya yang masih aktif.

Cluster menyiapkan ketersediaan layanan dengan mereplikasi server dan dengan konfigurasi ulang perangkat lunak dan perangkat keras yang berlebihan. Jadi, setiap sistem mengendalikan yang lain dan bekerja berdasarkan permintaan jika salah satu node gagal. Jenis cluster ini menguntungkan bagi para pengguna yang bergantung pada sistem komputer mereka sepenuhnya. Misalnya e-commerce, website, dll.

High Performance Clusters

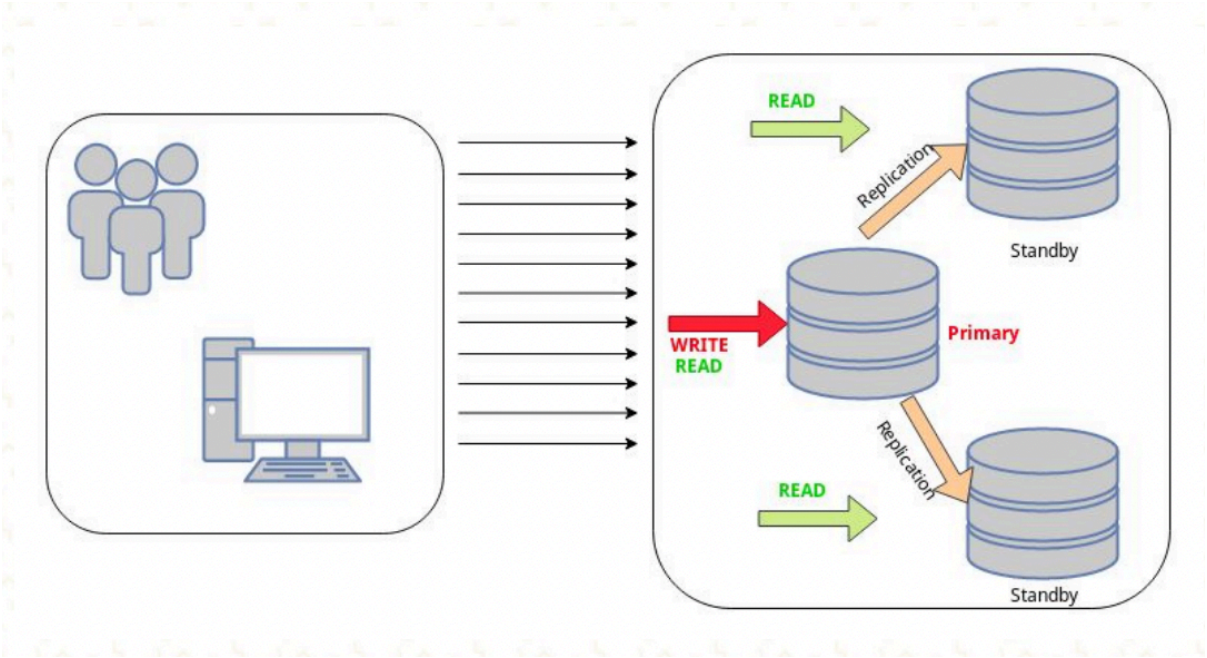


High-Performance Database Clustering adalah konsep di mana beberapa server database bekerja bersama untuk memberikan tingkat kinerja yang tinggi dalam pemrosesan data. Tujuannya adalah untuk mengoptimalkan eksekusi query, responsivitas, dan throughput

secara keseluruhan. Mereka mengoperasikan co-extending program yang diperlukan untuk perhitungan yang menghabiskan waktu. Biasanya digunakan oleh perusahaan yang memerlukan perhitungan / workload tinggi seperti AI Industri dll.

High-performance database clustering adalah bagian integral dari sistem yang membutuhkan kinerja tinggi, ketersediaan yang tinggi, dan penanganan volume data yang besar, seperti sistem yang mendukung aplikasi bisnis kritis atau analisis data skala besar.

Load Balancing Clusters



Load balancing dalam database clustering adalah strategi untuk mendistribusikan beban kerja atau permintaan query secara merata di antara beberapa node atau server dalam sebuah cluster. Tujuan utama dari load balancing adalah mencegah satu atau beberapa node menjadi bottleneck sehingga kinerja keseluruhan sistem dapat dioptimalkan.

Cluster database ini berfungsi untuk mendistribusikan beban antara server yang berbeda. Sistem dalam jaringan ini mengintegrasikan node mereka, dengan bantuan permintaan pengguna yang dibagi rata di seluruh node yang berpartisipasi.

Berikut adalah beberapa konsep dan strategi yang terkait dengan load balancing dalam konteks database clustering:

1. **Distribusi Permintaan Query:**
    - Load balancing memastikan bahwa setiap permintaan query atau transaksi didistribusikan secara merata di antara node-node dalam cluster. Ini membantu menghindari situasi di mana satu node menerima beban kerja yang berlebihan sementara yang lain kurang dimanfaatkan.
2. **Round Robin Load Balancing:**

    - Metode round-robin adalah pendekatan sederhana di mana setiap permintaan diberikan ke node berikutnya dalam urutan. Ini memastikan bahwa setiap node menerima jumlah permintaan yang seimbang.
3. **Least Connections Load Balancing:**
    - Dalam metode ini, permintaan diberikan ke node dengan jumlah koneksi terendah saat itu. Hal ini membantu menghindari situasi di mana satu node mungkin mengalami beban kerja yang tinggi karena banyak koneksi aktif.
4. **Weighted Load Balancing:**
    - Bobot (weight) dapat ditetapkan untuk setiap node berdasarkan kapasitas atau daya pemrosesan masing-masing. Load balancer akan mendistribusikan beban secara proporsional berdasarkan bobot ini.
5. **DNS Load Balancing:**
    - Menggunakan DNS untuk mendistribusikan permintaan ke beberapa alamat IP yang terkait dengan node dalam cluster. Ini dapat dilakukan dengan cara mengembalikan alamat IP yang berbeda setiap kali permintaan diteruskan ke DNS.
6. **Session Affinity (Sticky Sessions):**
    - Untuk menjaga konsistensi sesi pengguna, load balancer dapat dirancang untuk mengarahkan semua permintaan dari satu pengguna ke node yang sama selama sesi tersebut berlangsung. Ini dikenal sebagai session affinity atau sticky sessions.
7. **Health Checking:**
    - Load balancer dapat secara teratur memeriksa kesehatan node-node dalam cluster untuk memastikan bahwa setiap node dapat menangani beban kerja yang diberikan. Jika sebuah node mengalami kegagalan, load balancer dapat mengarahkan permintaan ke node yang sehat.
8. **Global Server Load Balancing (GSLB):**
    - Untuk distribusi beban di antara pusat data atau lokasi geografis yang berbeda, GSLB dapat digunakan untuk memastikan permintaan diarahkan ke server terdekat atau yang memiliki ketersediaan tinggi.
    - Biasanya digunakan didalam perusahaan besar yang memiliki bisnis yang sangat diharuskan tidak / hampir memiliki zero downtime (seperti perbankan) dan menempatkan secondary cluster nya ditempat yang berbeda (untuk menanggulangi apabila terjadi bencana alam / buatan di daerah tersebut)
9. **Adaptive Load Balancing:**
    - Load balancer dapat diatur untuk merespons secara dinamis terhadap perubahan beban kerja atau kondisi jaringan. Ini bisa melibatkan penyesuaian bobot atau strategi distribusi secara otomatis.
    - Dapat diatur dengan mudah apabila menggunakan Cloud (Biasanya tersedia di masing-masing provider cloud)

Load balancing dalam database clustering adalah aspek penting untuk mencapai kinerja yang optimal, meningkatkan ketersediaan, dan mengelola beban kerja dengan efisien. Memilih metode load balancing yang sesuai dengan kebutuhan dan karakteristik sistem adalah kunci untuk mencapai hasil yang baik.

Table Partitioning

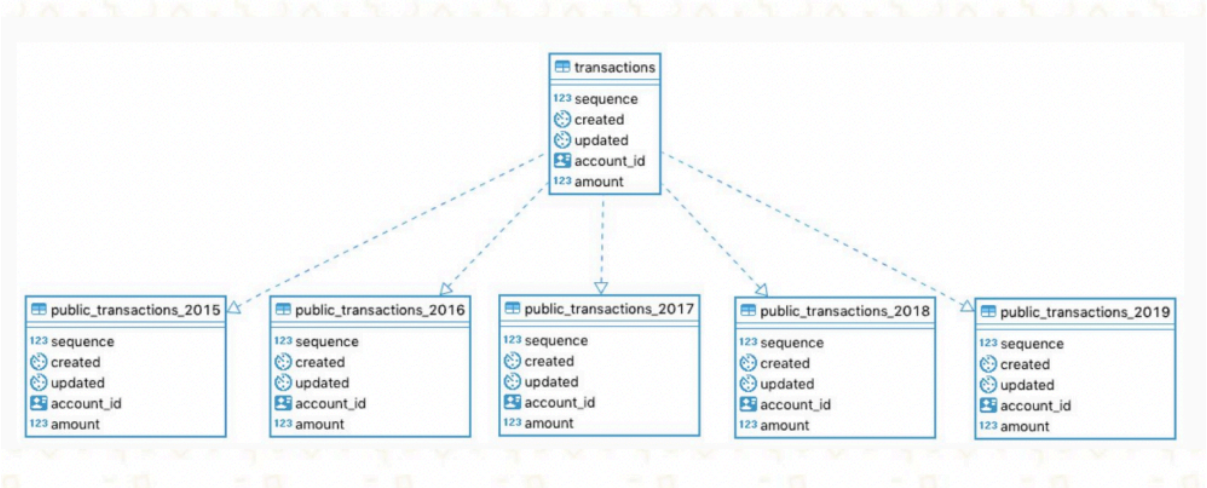


Table partitioning adalah teknik pengorganisasian data dalam tabel menjadi beberapa bagian yang disebut partisi, berdasarkan kriteria tertentu. Partisi dapat membantu meningkatkan kinerja, mengelola ukuran tabel, dan mempermudah operasi pemeliharaan pada basis data.

Partisi dapat memberikan beberapa manfaat, termasuk:

- **Peningkatan Kinerja:** Query yang hanya melibatkan satu partisi dapat dieksekusi lebih cepat karena database hanya perlu memindai bagian tertentu dari data.
- **Pemeliharaan Lebih Mudah:** Pemeliharaan seperti backup, restore, dan pengindeksan dapat dilakukan pada partisi tertentu tanpa memengaruhi seluruh tabel.
- **Manajemen Data yang Lebih Efisien:** Mengelola data per partisi dapat mempermudah operasi seperti pemindahan dan penghapusan data.

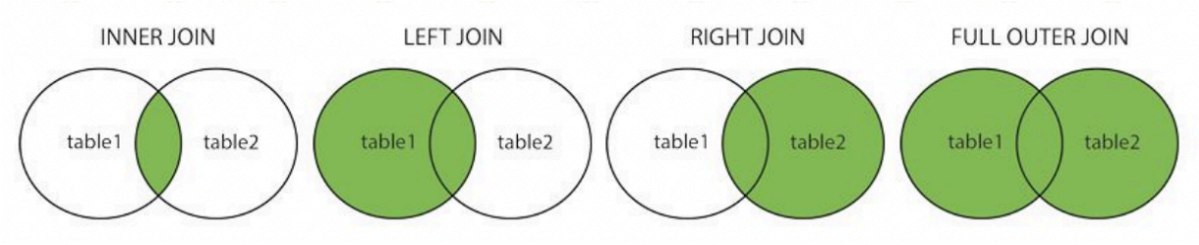
Berikut contoh query table partitioning dalam snowflake :

```
CREATE TABLE TESTING.PUBLIC.DEMOGRAPHICS PARTITION BY (state) AS
SELECT
    county, total_population, total_male_population,
    total_female_population
FROM
    covid19_epidemiological_data.public.demographics
WHERE
    state='TX'
ORDER BY
    total_male_population DESC;
```

Join Operation

Join operation adalah operasi dalam bahasa SQL yang digunakan untuk menggabungkan baris dari dua atau lebih tabel berdasarkan kolom atau kondisi yang sesuai. Join

memungkinkan pengguna untuk mengambil data yang terkait dari tabel-tabel yang berbeda. Ada beberapa jenis operasi join yang umum digunakan, termasuk:



1. **Inner Join:**
    - Inner join menghasilkan hasil gabungan yang hanya mencakup baris yang memiliki nilai yang sesuai di kedua tabel yang dijoin. Ini adalah jenis join paling umum.

```
SELECT *
FROM table1
INNER JOIN table2 ON table1.column = table2.column;
```

2. **Left (Outer) Join:**
    - Left join menghasilkan semua baris dari tabel kiri (tabel pertama dalam pernyataan join) dan baris yang sesuai dari tabel kanan. Jika tidak ada kesesuaian, nilai-nilai untuk kolom-kolom tabel kanan akan menjadi NULL.

```
SELECT *
FROM table1
LEFT JOIN table2 ON table1.column = table2.column;
```

3. **Right (Outer) Join:**
    - Right join mirip dengan left join, tetapi menghasilkan semua baris dari tabel kanan dan baris yang sesuai dari tabel kiri.

```
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.column = table2.column;
```

4. **Full (Outer) Join:**
    - Full join menghasilkan gabungan dari semua baris dari kedua tabel, dengan nilai NULL pada kolom-kolom yang tidak sesuai.

```
SELECT *
FROM table1
FULL JOIN table2 ON table1.column = table2.column;
```

5. **Cross Join:**
    - Cross join menghasilkan hasil gabungan dari semua baris dalam tabel kiri dengan semua baris dalam tabel kanan. Ini menghasilkan jumlah baris yang sama dengan perkalian jumlah baris dalam kedua tabel.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

Apabila ingin mengetahui lebih lanjut mengenai join pada snowflake, dapat dilihat dokumentasi nya pada :

https://docs.snowflake.com/en/sql-reference/constructs/join

Berikut contoh join yang dilakukan didalam snowflake :

```
SELECT
    OV.DATE, OV.ISO3166_1, OV.COUNTRY_REGION ,OV.DAILY_VACCINATIONS, OV.PEOPLE_VACCINATED, IBA.INPATIENT_BEDS_OCCUPIED,
    IBA.TOTAL_INPATIENT_BEDS, IBA.INPATIENT_BEDS_IN_USE_PCT
FROM
    COVID19_EPIDEMIOLOGICAL_DATA.PUBLIC.OWID_VACCINATIONS AS OV
LEFT JOIN
    COVID19_EPIDEMIOLOGICAL_DATA.PUBLIC.CDC_INPATIENT_BEDS_ALL AS IBA
    ON OV.ISO3166_1 = IBA.ISO3166_1 AND OV.DATE = IBA.DATE
```

Join operation sangat penting dalam pemrosesan data di basis data relasional, memungkinkan pengguna untuk menggabungkan dan mengambil informasi dari berbagai tabel yang terkait. Pemilihan jenis join yang tepat tergantung pada kebutuhan spesifik query dan struktur data tabel yang dijoin.

Sorting and Grouping Operation

Operasi Sorting dan Grouping dalam SQL digunakan untuk mengurutkan data dan mengelompokkannya berdasarkan nilai tertentu. Berikut adalah penjelasan singkat untuk kedua operasi tersebut:

Sorting (Pengurutan):

Operasi sorting dilakukan dengan menggunakan klausa `ORDER BY` dalam pernyataan SQL. Tujuannya adalah untuk mengurutkan hasil query berdasarkan satu atau beberapa kolom.

Contoh Pengurutan Ascending:

```
SELECT column1, column2
FROM my_table
ORDER BY column1 ASC;
```

Contoh Pengurutan Descending:

```
SELECT column1, column2
FROM my_table
ORDER BY column1 DESC;
```

Dalam contoh-contoh tersebut, data akan diurutkan berdasarkan nilai di `column1`. Ascending (ASC) mengurutkan dari nilai terkecil ke terbesar, sedangkan Descending (DESC) mengurutkan dari nilai terbesar ke terkecil.

Satu hal yang perlu diperhatikan, Proses ORDER merupakan salah satu Expensive Process pada SQL, hanya pergunakan jika dibutuhkan.

Grouping (Pengelompokan):

Operasi grouping dilakukan dengan menggunakan klausa `GROUP BY` dalam pernyataan SQL. Ini digunakan untuk mengelompokkan baris data yang memiliki nilai yang sama dalam satu atau lebih kolom. Operasi agregasi seperti COUNT, SUM, AVG, MAX, atau MIN sering digunakan bersamaan dengan operasi grouping.

Contoh Pengelompokan dan Penghitungan:

```
SELECT department, COUNT(*) as employee_count
FROM employees
GROUP BY department;
```

Dalam contoh ini, data dikelompokkan berdasarkan departemen, dan untuk setiap kelompok, dihitung jumlah karyawan di setiap departemen.

Penggunaan Bersama:

Penggunaan sorting dan grouping seringkali dikombinasikan untuk menyusun dan menganalisis data dengan lebih baik. Misalnya, kita dapat mengurutkan hasil grouping berdasarkan nilai tertentu untuk melihat data dengan urutan tertentu.

Contoh Sorting dan Grouping Bersama:

```
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department
ORDER BY avg_salary DESC;
```

Dalam contoh ini, data dielompokkan berdasarkan departemen, dihitung rata-rata gaji untuk setiap departemen, dan hasilnya diurutkan dari rata-rata gaji tertinggi ke terendah.

Penting untuk memahami bahwa operasi grouping biasanya melibatkan klausa `GROUP BY`, sementara operasi sorting melibatkan klausa `ORDER BY`. Keduanya bekerja bersama untuk menyusun dan menyajikan data dengan cara yang lebih terstruktur dan bermakna.

Sub Query

Subquery (subquery atau subselect) adalah query yang tertanam di dalam query utama. Subquery digunakan untuk mengambil data yang akan digunakan oleh query utama untuk mencapai hasil yang diinginkan. Subquery dapat muncul dalam klausa SELECT, FROM, WHERE, atau HAVING dari query utama.

Ada dua jenis subquery utama: scalar subquery dan table subquery.

Scalar Subquery:

Scalar subquery menghasilkan satu nilai (skalar) yang akan digunakan dalam ekspresi atau kondisi di dalam query utama.

Contoh Scalar Subquery dalam Klausa SELECT:

```
SELECT column1, (SELECT MAX(column2) FROM another_table) AS
max_value
FROM my_table;
```

Dalam contoh ini, subquery `(SELECT MAX(column2) FROM another_table)` menghasilkan nilai maksimum dari `column2` dalam `another_table`, dan nilai ini digunakan dalam hasil query utama.

Table Subquery:

Table subquery menghasilkan satu set hasil (tabel) yang dapat digunakan dalam klausa FROM, JOIN, atau WHERE di dalam query utama.

Contoh Table Subquery dalam Klausa FROM:

```
SELECT column1, column2
FROM (SELECT * FROM another_table WHERE condition) AS
subquery_table;
```

Dalam contoh ini, subquery `(SELECT * FROM another_table WHERE condition)` menghasilkan sebuah tabel yang digunakan dalam klausa FROM query utama.

Penggunaan Subquery dalam Klausa WHERE:

Subquery juga sering digunakan dalam klausa WHERE untuk membandingkan nilai atau hasil dari subquery dengan nilai dalam tabel utama.

Contoh Subquery dalam Klausa WHERE:

```
SELECT column1, column2
FROM my_table
WHERE column3 > (SELECT AVG(column4) FROM another_table);
```

Dalam contoh ini, subquery `(SELECT AVG(column4) FROM another_table)` menghasilkan nilai rata-rata dari `column4` dalam `another_table`, dan hanya baris-baris di `my_table` dengan nilai `column3` yang lebih besar dari rata-rata tersebut yang akan dipilih.

Contoh lainnya, dari table berikut :

StudentID	Name
V001	Abe
V002	Abhay
V003	Acelin
V004	Adelphos

StudentID	Total_marks
V001	95
V002	80
V003	74
V004	81

Kita ingin mencari siswa mana saja yang mendapatkan nilai diatas KKN (> 80) maka kita dapat melakukan query berikut :

```
1  SELECT a.studentid, a.name, b.total_marks
2  FROM student a, marks b
3  WHERE a.studentid = b.studentid AND b.total_marks >
4  (SELECT total_marks
5  FROM marks
6  WHERE studentid =  'V002');
```

Sehingga mendapatkan return :

studentid	name	total_marks
V001	Abe	95
V004	Adelphos	81

Subquery memberikan fleksibilitas dan kekuatan tambahan dalam membuat query yang lebih kompleks dan kontekstual. Penting untuk merancang subquery dengan cermat dan memastikan bahwa subquery menghasilkan output yang diharapkan sebelum digunakan dalam query utama.

Function Operation

User-Defined Function (UDF) dalam SQL adalah fungsi yang dibuat oleh pengguna untuk melaksanakan tugas tertentu. Fungsi ini dapat diintegrasikan ke dalam pernyataan SQL untuk memudahkan dan mengotomatiskan proses-proses tertentu.

Seperti fungsi bawaan yang bisa kita panggil dari SQL, logika UDF biasanya memperluas atau menyempurnakan SQL dengan fungsionalitas yang tidak dimiliki atau tidak dilakukan dengan baik oleh SQL. UDF juga memberi kita cara untuk meng enkapsulasi fungsionalitas sehingga kita dapat memanggilnya berulang kali dari berbagai tempat dalam kode.

UDF Function using SQL

Sebagai contoh, kita ingin membuat fungsi dimana fungsi tersebut dapat mengalikan 2 int / float maka kita dapat membuat fungsinya sebagai berikut :

```
CREATE FUNCTION multiply1 (a number, b number)
  RETURNS number
  COMMENT='multiply two numbers'
  AS 'a * b';
```

Lalu kita dapat mengeksekusi nya dengan menggunakan query :

```
SELECT multiply1(5, 7) AS result;
```

UDF Function using Python

Sebagai contoh, kita ingin membuat fungsi dimana fungsi tersebut dapat mengalikan int / float yang kita masukkan lalu dikalikan dengan 3.14 atau *phi,* tapi kali ini menggunakan python, kita dapat membuat function tersebut dengan cara :

```
create or replace function py_udf(a int)
  returns variant
  language python
  runtime_version = '3.8'
  handler = 'udf'
as $$
def udf(a):
    b = a * 3.14
    return b
$$;
```

Lalu eksekusi seperti eksekusi function sql seperti diatas.