BAB II

LANDASAN TEORI

2.1 Penelitian Terdahulu

Pada penelitian terdahulu, ditemukan beberapa artikel jurnal yang membahas tentang *performance database*, baik untuk *database relational* maupun *database non-relational*.

Tabel 2. 1 Tabel Penelitian Terdahulu

No	Judul Artikel	Jurnal	Pembahasan
1	*Query Optimization in* MySQL *Database Using Index* [10]	**Tahun:** 2022 **Penulis:** Siti Maesaroh, Heru Gunawan, Agung Lestari, Muhammad Sufyan Ats Tsaurie, Mohammad Fauji **Jurnal:** International Journal of Cyber and IT Service Management (IJCITSM), Vol. 2, No. 2.	Penggunaan *index* dalam *database* sangat bermanfaat untuk meningkatkan kecepatan SQL, terutama pada *database* dengan banyak tabel dan baris di setiap tabel. Jika *database* tidak menggunakan *index*, pencarian data akan memakan waktu lebih lama karena database harus memindai semua baris tabel. Namun, akan lebih baik jika berhati-hati saat memutuskan tabel mana yang harus di-*index* atau tidak untuk menghindari kinerja SQL yang buruk.
2	*SQL Inner Join:* MySQL *and* PostgreSQL *Performance* [6]	**Tahun:** 2020 **Penulis:** Ummul Hairah **Jurnal:** International Research Journal of Engineering and Technology (IRJET), Vol. 7, No. 10	*Response time* MySQL dan PostgreSQL pada *inner join* SQL memiliki perbedaan. Hasil penelitian menunjukkan bahwa terdapat pengaruh yang signifikan jumlah data dan relasi terhadap *response time* pada MySQL. *Response time* pada MySQL akan bertambah jika *record* datanya semakin besar. *Response time* pada PostgreSQL juga akan bertambah jika *record* datanya lebih besar, namun nilai peningkatannya tidak sebesar MySQL.
3	*A Performance Benchmark for the* PostgreSQL *and* MySQL *Databases* [7]	**Tahun:** 2024 **Penulis:** Sanket Vilas Salunke, Abdelkader Ouda **Jurnal:** Future Internet, MDPI, Vol. 16, No. 10	Penelitian ini dibagi menjadi dua kategori, yaitu *primary experiment* untuk pengujian *query* sederhana dan *complex experiment* untuk pengujian *query* yang lebih kompleks. Proses pengujian dilakukan dengan melakukan *run*

No	Judul Artikel	Jurnal	Pembahasan
			query sebanyak 100 kali. Hasil dari pengujian adalah baik untuk *query primary experiment* dan *query complex experiment,* PostgreSQL memiliki *execution time* yang lebih singkat dibandingkan MySQL dengan perbedaan waktu yang cukup signifikan. Hasil ini menunjukkan bahwa *database* PostgreSQL adalah *database* yang lebih cocok untuk *automation* sistem secara berkelanjutan karena keunggulannya dalam *handle* SELECT *operations*, baik di *query* sederhana maupun *query* kompleks.
4	Analisis Perbandingan Kecepatan *Privileges Index* pada *Database* Oracle dan *Database* MySQL [11]	**Tahun:** 2023 **Penulis:** Verina Renata Putri, Cintami Prasista Wibowo, Heldha Ayu Setia **Jurnal:** Prosiding Seminar Nasional Teknologi dan Sistem Informasi (SITASI), Vol. 3, No. 1	*Privileges* adalah jenis hak akses yang dapat diberikan ke *user* lain di dalam *database* terkait operasi yang dapat dilakukan pada *index*, sedangkan *index* adalah *object* yang ada di dalam sistem sehingga dapat mempercepat proses pencarian. Hasil akhir pengujian menunjukkan bahwa *database* Oracle meminimalkan waktu eksekusi lebih banyak setelah diberikan *privileges index*, yaitu sekitar 0,32625 detik, dibandingkan dengan *database* MySQL yang hanya meminimalkan waktu sebanyak 0,00835 detik saja.
5	*A Comparative Study of MongoDB and Document-Based MySQL for Big Data Application Data Management* [12]	**Tahun:** 2022 **Penulis:** Cornelia A. Gyorödi, Diana V. Dumse-Burescu, Doina R. Zmaranda, Robert Gyorödi **Jurnal:** Big Data and Cognitive Computing, Vol. 6, No. 2	Performa *database* merupakan salah satu faktor penting yang perlu dipertimbangkan dalam memilih *database* untuk aplikasi *big data*. Pengujian performa *database* MySQL dan *database* MongoDB dilakukan dengan menguji *query* yang berkaitan dengan *create, read, update,* dan *delete* (CRUD). Secara keseluruhan, perbedaan *response time* untuk operasi CRUD dari kedua *database* tidak memiliki perbedaan signifikan, namun untuk *database* MongoDB masih memiliki performa yang lebih baik secara keseluruhan walaupun untuk penulisan *syntax* CRUD juga jauh lebih kompleks dibandingkan dengan MySQL.
6	Perbandingan	**Tahun:** 2023	Ada tiga *query* yang digunakan di

No	Judul Artikel	Jurnal	Pembahasan
	Kinerja *Query* SQL *Join Tables* dengan Menggunakan *Index* [13]	**Penulis:** Kurniawan Eka Permana, Moch Kautsar Sophan, Arif Muntasa, Abdullah Basuki Rahmat **Jurnal:** Jurnal SimanteC, Vol. 11, No. 2	dalam penelitian ini, yaitu *single join*, *triple join*, dan *triple join* dengan pencarian. *Index database* yang dibuat di dalam penelitian ini disesuaikan juga dengan kolom yang dilibatkan dalam SQL *query join.* Hasil penelitian pada *single join* menunjukkan adanya peningkatan *query* yang dijalankan sebesar 39,47%, dari sebelumnya hasil eksekusinya 5,66 detik menjadi 3,43575 detik. Hasil penelitian untuk *triple join* menunjukkan adanya peningkatan sebesar 5,55%, dari sebelum menggunakan *index* ada pada 12,254 detik menjadi 11,574 detik. Hasil penelitian pada *triple join* dengan *condition* juga menunjukkan peningkatan sebesar 43,68%, dari awalnya ada pada 5,382 detik menjadi 3,031 detik. Ketiga hasil ini menunjukkan penggunaan *index* dapat mengurangi jumlah data yang akan diproses dan meningkatkan efisiensi *query* di MySQL.
7	Perbandingan *Response Time* Penggunaan *Index*, *Views*, dan *Materialized Views Database* MySQL [14]	**Tahun:** 2022 **Penulis:** Edi Witono, Parno **Jurnal:** Jurnal Sains Komputer & Informatika (J-SAKTI), Vol. 6, No. 1	*Database* yang digunakan pada penelitian ini adalah *database* MySQL. Ada tiga perbandingan yang dilakukan, yaitu pengujian *response time* dengan *index*, *views*, dan *materialized views.* Hasil *response time* dengan *index* adalah 0,41 detik sampai 0,56 detik, 1,48 detik sampai 2,51 detik dengan *views*, dan 0,01 detik dengan *materialized views*. Dapat disimpulkan bahwa metode *materialized views* adalah metode yang paling cepat untuk meningkatkan performa kecepatan proses *query*, namun metode ini ketika digunakan akan membuat data di dalamnya tidak otomatis ter-*update* apabila ada penambahan atau pengurangan data pada tabel induk.
8	Optimasi *Database* dengan Metode *Index* dan Partisi Tabel *Database*	**Tahun:** 2022 **Penulis:** Samidi, Fadly, Yusuf Virmansyah, Ronal Yulyanto Suladi, Ario Bambang Lesmana	Dalam penelitian ini, *query* yang diujikan dengan *index database* adalah *query single table*, *query single table* dengan WHERE *clause*, *query join table* dengan

No	Judul Artikel	Jurnal	Pembahasan
	PostgreSQL pada Aplikasi *E-Commerce* (Studi pada Aplikasi Tokopintar) [15]	**Jurnal:** Jurnal Pendidikan Tambusai, Vol. 6, No. 1	WHERE *clause*, dan *query* UPDATE *table.* Ada empat *schema* yang digunakan dalam penelitian, "test0" untuk tabel yang tidak menggunakan *attribute key*, "test1" untuk tabel yang ada *primary key* dan *foreign key*, tetapi tidak ada *index*, "test2" untuk tabel yang ada *primary key*, *foreign key*, dan *index*, dan "test3" yang ada *primary key*, *foreign key*, *index*, dan *partition table.* Berdasarkan hasil penelitian, untuk *query single table*, hasil "test0" dan "test1" memiliki *response time* yang lebih cepat dibandingkan "test2" dan "test3", namun untuk *query single table* dengan WHERE *clause*, *query join table*, dan *query* UPDATE *table*, penggunaan *index* seperti pada "test2" dan "test3" memiliki *response time* yang lebih cepat.
9	*Application of Index for Optimization Query Data in Graduate Information Systems* [16]	**Tahun:** 2022 **Penulis:** Agus Sifaunajah, Rosa Tarbiyatul Khusna **Jurnal:** International Journal of Applied Information Technology (IJAIT), Vol. 71, No. 7	Optimasi *database* perlu dilakukan untuk memastikan *query* berjalan dengan efektif dan efisien agar total waktu pemrosesan *query* bisa lebih cepat. Salah satu metode optimasi yang dapat dilakukan adalah menggunakan *index* di *database*. *Query* yang diujikan dalam penelitian ini difokuskan pada *join tables*, baik *single join table* maupun *multiple join table*. Selain itu, ada juga pengujian *query* SQL dengan *subquery*. Secara garis besar, *query* dengan penggunaan *index* dapat memproses *query* dengan lebih cepat, baik untuk *single join table* maupun pada kondisi *multiple join tables*.
10	*Comparing Oracle and PostgreSQL, Performance and Optimization* [17]	**Tahun:** 2021 **Penulis:** Pedro Martins, Paulo Tomé, Cristina Wanzeller, Filipe S, Maryam Abbasi **Jurnal:** Springer Nature Link	*Relational database* menjadi lebih besar dan lebih kompleks, sehingga DBMS perlu untuk memiliki respons yang bagus. Pada pengujian ini, ada sepuluh *query* diujikan dalam tabel dengan adanya *primary key*, *foreign keys*, dan *index constraints.* Hasil penelitian menunjukkan bahwa Oracle *database* memiliki performa yang lebih stabil pada keseluruhan *query*, dengan hasil yang lebih baik pada

No	Judul Artikel	Jurnal	Pembahasan
			skenario kondisi optimasi yang tidak optimal. Namun, PostgreSQL sendiri menunjukkan hasil *execution times* yang lebih baik setelah dioptimalkan *query*-nya. Ketika dibandingkan antara Oracle dan PostgreSQL, tanpa *index*, Oracle lebih cepat sekitar 64%, namun, dengan *index*, PostgreSQL lebih cepat sekitar 75%.

Berdasarkan tabel penelitian terdahulu yang sudah ditampilkan pada Tabel 2.1 di atas, terdapat beberapa penelitian sebelumnya yang sudah dilakukan dalam rentang waktu di tahun 2020-2024. Penelitian terdahulu menunjukkan hasil performa DBMS, baik *relational database* sepert MySQL dan PostgreSQL maupun *non-relational database* seperti MongoDB. Secara keseluruhan, jika membandingkan *relational database* antara MySQL dan PostgreSQL, penelitian terdahulu menampilkan bahwa *database* PostgreSQL sendiri jauh lebih unggul dibandingkan dengan *database* MySQL untuk menjalankan beberapa operasi *query* sederhana, yaitu operasi *create*, *read*, *update*, dan *delete* (CRUD) [6], [7]. Walaupun begitu, pada penelitian lain, ketika PostgreSQL dibandingkan dengan Oracle, hasil penelitian menunjukkan bahwa Oracle memiliki performa yang lebih stabil tanpa optimasi yang optimal, sedangkan untuk PostgreSQL, *database* baru dapat bekerja dengan lebih optimal setelah *query*-nya dioptimalkan [17].

Selain membandingkan antara dua *relational database*, ada juga penelitian yang membandingkan antara *relational database* dengan *non-relational database*, seperti pada penelitian perbandingan antara MongoDB dan MySQL [12]. Hasil penelitian menunjukkan untuk performa secara keseluruhan, perbedaan *response time* untuk operasi CRUD dari kedua *database* tidak memiliki perbedaan signifikan. Namun, untuk *database* MongoDB masih memiliki performa yang lebih baik walaupun untuk penulisan *syntax* CRUD juga jauh lebih kompleks dibandingkan dengan MySQL.

Salah satu cara untuk meningkatkan performa pencarian data di *database* adalah menggunakan *index database.* Berdasarkan beberapa penelitian terdahulu,

penggunaan *index* pada *database* terbukti dapat membantu meningkatkan *response time* pada *database*. Baik untuk *query* yang sederhana ataupun *query* yang sudah lebih kompleks, seperti *join tables*, *index* pada *database* terbukti dapat mempercepat waktu pencarian data di *database*. Walaupun begitu, penggunaan *index* juga perlu diperhatikan untuk menghindari kinerja SQL yang buruk [13], [16], [18], [11], [15].

Walaupun penggunaan *index* merupakan salah satu metode optimasi *database* yang umum digunakan, namun ada juga penelitian yang membandingkan metode *index, views*, dan *materialized views* untuk membandingkan hasil akhir *response time* dari ketiga metode ini. Hasil akhirnya menunjukkan bahwa metode *materialized views* memiliki *response time* yang paling cepat, lalu diikuti dengan *index*, dan terakhir adalah *views* saja. Namun ada pertimbangan yang perlu diperhatikan dalam menggunakan *materialized views*, yaitu metode *materialized views* akan membuat data di dalamnya tidak otomatis ter-*update* apabila ada penambahan atau pengurangan data pada tabel induk [14].

Pada penelitian yang dilakukan kali ini, proses optimasi yang dilakukan akan membandingkan performa *database* dari tiga *database*, yaitu MySQL, PostgreSQL, dan Microsoft SQL Server. Microsoft SQL Server dipilih sebagai *database* yang berbeda dari beberapa penelitian terdahulu yang sudah dicantumkan pada Tabel 2.1 di atas. Selain itu, proses optimasi juga akan mengkombinasikan beberapa struktur penulisan *query*, seperti penggunaan *join tables*, penggunaan *join reorder* untuk *join tables*, penggunaan *index* pada kolom *database*, dan lain-lain. Hasil optimasi *query* tersebut akan dibandingkan dengan ketiga *database* yang digunakan untuk pengujian untuk mengetahui performa *database* yang paling stabil dengan jumlah data yang banyak.

2.2 Teori Penelitian

2.2.1 *Relational Database Management System* (DBMS)

Relational database management system (RDBMS) adalah salah satu jenis DBMS yang memungkinkan perusahaan mengakses data lebih efisien dibandingkan DBMS. Singkatnya, RDBMS adalah sistem perangkat lunak yang biasa digunakan untuk menyimpan data dalam bentuk tabel. Setiap baris pada

tabel menggunakan ID yang unik dengan tujuan untuk mengidentifikasi dan menghubungkan data antar tabel [3].

Salah satu perbedaan RDBMS adalah DBMS menyimpan data dalam bentuk *file*, sedangkan RDBMS menyimpan data dalam bentuk tabel. Oleh karena itu, RDBMS menggunakan struktur tabel dengan *headers* sebagai nama kolom dan tiap barisnya terdapat *values* yang berkaitan dengan *headers*-nya. Selain dari perbedaan tipe data yang disimpan, RDBMS juga memiliki beberapa keunggulan yang membuat tipe *database* ini banyak digunakan [4]. Pertama, penyimpanan informasi di tabel berbeda dapat menghindari duplikasi data. Selain itu, data yang disimpan di RDBMS dapat dengan mudah diakses dengan *graphical user interface* (GUI). Terakhir, RDBMS dapat diakses dengan penggunaan SQL, sehingga memudahkan untuk mengganti data di *database* yang berbeda.

2.2.1.1 *Structured Query Language* (SQL)

Bahasa yang digunakan untuk berinteraksi dengan *relational database* adalah *structured query language* (SQL) [4]. SQL sendiri adalah bahasa standar terstruktur yang digunakan antara pengguna dan program aplikasi di RDBMS. SQL adalah salah satu bahasa *query* yang paling banyak digunakan untuk *relational database* karena mudah untuk dipahami dan dapat digunakan untuk berbagai tipe *database*, sehingga memudahkan untuk melakukan transfer data antar *relational database* [19]. SQL sendiri dapat digunakan untuk operasi *create, read, update,* dan *delete* di dalam *relational database.* Ada beberapa *command* SQL yang dapat diklasifikasikan berdasarkan kegunaannya di dalam *database* [20]. Berikut adalah penjelasan terkait masing-masing *command* SQL yang umum digunakan.

1) ***Data Definition Language* (DDL)**

   DDL adalah bagian dari *database query* untuk menentukan struktur dan skema *database*. Sama seperti arti dari istilahnya, perintah DDL memungkinkan pengguna untuk mendefinisikan entitas dan hubungan entitas di *database*. Beberapa perintah DDL yang umum digunakan adalah CREATE, ALTER, dan DROP.

2) ***Data Manipulation Language* (DML)**

Berbeda dengan DDL, DML merupakan bagian dari *query database* untuk memanipulasi data dalam tabel. DML memungkinkan pengguna untuk menambah, menghapus, mengubah, atau mengambil data dalam tabel database. Beberapa perintah dari DML adalah INSERT, UPDATE, DELETE, dan SELECT.

3) *Data Control Language* (DCL)

Perintah yang masuk dalam kategori DCL merupakan perintah yang bertujuan untuk mengontrol hak akses *database* (*privileges*) dan memanipulasi pengguna *database*. Perintah yang termasuk dalam kategori DCL adalah GRANT dan REVOKE.

2.2.1.2 *Entity Relationship Diagram* (ERD)

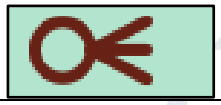
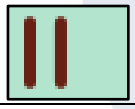
Entity relationship diagram atau dapat disingkat sebagai ERD adalah *diagram* yang biasanya dibuat di dalam tahap perancangan sistem untuk mendesain skema *database* [21]. ERD secara visual ditampilkan di dalam bentuk notasi grafik dan memiliki satu atau lebih relasi antar tabel. ERD sudah banyak digunakan di dalam analisis struktur *database* dan pembuatan konsep model karena dapat berguna untuk menunjukkan relasi antar data di dalam *database*.

Secara garis besar, ERD memiliki tiga elemen dasar, yaitu entitas, atribut, dan relasi [22]. Entitas adalah objek yang akan menjadi perhatian. Beberapa contoh dari entitas adalah produk, *suppliers*, *customers*, dan lain-lain. Atribut merupakan informasi yang ada di dalam entitas. Jadi, di dalam entitas harus memiliki *primary key* sebagai ciri khas setiap entitas dan ada atribut deskriptif. Relasi adalah hubungan antara dua atau lebih entitas. Relasi sendiri digambarkan dengan garis atau *line* untuk menghubungkan antar komponen dalam ERD.

Dalam pembuatan ERD, terdapat beberapa tipe relasi yang berbeda juga antar tabel, sehingga perlu untuk mengetahui jenis relasi yang umum dipakai dalam pembuatan ERD agar relasi antar entitas dapat tergambar dengan jelas. Tabel 2.2 di bawah ini merupakan penjelasan terkait jenis-jenis relasi yang dapat digunakan untuk ERD dengan menggunakan *Crow's Foot*.

Kombinasi ini sering disebut sebagai *cardinality* dalam notasi *Crow's Foot notation*.

Tabel 2. 2 Tipe Relasi pada ERD Menggunakan *Crow's Foot Notation* [23]

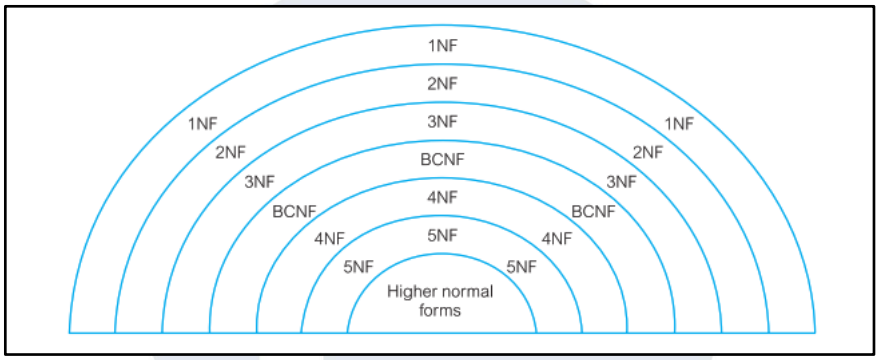
Simbol Crow Foot	Kardinalitas	Keterangan
	(0, N)	*Zero or many*, bagian "*many*" termasuk opsional.
	(1, N)	*One or many*, bagian "*many*" termasuk *mandatory* atau wajib.
	(1, 1)	*One and only one*, bagian "1" termasuk *mandatory* atau wajib.
	(0, 1)	*Zero or one*, bagian "1" termasuk opsional.

2.2.1.3 Normalisasi *Database*

Normalisasi dalam *database* adalah teknik desain *database* yang dimulai dari memeriksa hubungan antara atribut pada data. Normalisasi sendiri memiliki tujuan utama untuk mengidentifikasi jumlah atribut yang sesuai untuk *requirements* data perusahaan. Selain itu, normalisasi juga dapat mengurangi duplikasi data dengan memastikan bahwa setiap atribut hanya muncul satu kali saja, dengan pengecualian untuk atribut yang berperan juga sebagai *foreign keys*, karena *key* tersebut penting untuk kebutuhan *join tables* antar tabel [24]. Dalam proses pengembangan *database*, pendekatan *top-down* ataupun *bottom-up* tetap dapat menggunakan normalisasi untuk memastikan skema *database* tidak memiliki duplikat data [19].

Proses normalisasi *database* memiliki beberapa tahapan, mulai dari *unnormalized form*, *first normal form* (1NF), *second normal form* (2NF), *third normal form* (3NF), *Boyce-Codd normal form* (BCNF), dan *fourth normal form* (4NF). Namun, tujuan yang paling penting adalah memastikan bahwa setiap tabel sudah berhasil dinormalisasi setidaknya sudah sampai pada tahap ketiga, pada *third normal form* (3NF). Normalisasi yang lebih

tinggi juga dapat digunakan, namun tahapan normalisasi seperti *fifth normal form* (5NF) ataupun *domain-key normal form* (DKNF) umumnya tidak dapat ditemukan dalam lingkup bisnis, namun lebih pada lingkup teoretis [23]. Gambar 2.1 merupakan gambar terkait proses normalisasi *database*.



Gambar 2. 1 Proses Normalisasi *Database* [19]

Unnormalized form adalah tahapan pertama untuk mengumpulkan atribut dan juga entitas yang memiliki satu atau lebih *repeating groups*. Selanjutnya, pada *first normal form* (1NF), tahapan yang dilakukan adalah mengidentifikasi dan menghapus *repeating groups* dari masing-masing tabel. Ada dua pendekatan yang biasanya dilakukan pada tahapan normalisasi pertama, yaitu mengisi bagian yang kosong dengan cara menduplikasi data yang tidak berulang, atau mengisi data yang berulang dengan *copy* dari atribut *original key*.

Setelah melakukan normalisasi pertama, tahapan selanjutnya adalah *second normal form* (2NF) yang bertujuan untuk memastikan bahwa setiap atribut yang bukan *primary key* semuanya dapat bergantung pada *primary key*. Terakhir, untuk *third normal form* (3NF), normalisasi yang dilakukan adalah memastikan tidak adanya atribut non-*primary key* yang bergantung secara transitif ke *primary key*. Apabila ada ketergantungan transitif, atribut yang memiliki dependensi transitif dihapus dan ditempatkan dalam relasi baru bersamaan dengan salinan determinan [19].

2.2.2 Optimisasi Performa *Database*

Database merupakan salah satu peran yang krusial dalam suatu sistem ataupun aplikasi, sehingga perlu diperhatikan performa dari *database*. Apabila

performa *database* menjadi lebih pelan, tentu akan berdampak pada aplikasi, produk, *user*, dan pertumbuhan bisnis di suatu perusahaan. Oleh karena itu, optimisasi performa *database* adalah salah satu hal yang penting dalam melakukan manajemen *database* dengan tujuan untuk memastikan *query* berjalan dengan efektif dan efisien agar total waktu pemrosesan *query* bisa lebih cepat.

Pengoptimalan kinerja *database* memiliki beberapa manfaat, yaitu meningkatkan *user experience* yang nyaman untuk pengguna, efisiensi operasional, dan skalabilitas. Dari sisi *user experience*, data yang lebih cepat dikembalikan akan meningkatkan kepuasan dari sisi *user*. Optimalisasi *database* juga dapat mengurangi *load* pada *server* dan *operational costs*. Dari sisi *scalability*, *database* yang sudah dioptimalkan dapat menerima *volume* data yang terus bertambah tanpa mengalami penurunan performa [25]. Secara umum, ada dua teknik untuk melakukan optimasi *query* SQL, yaitu *heuristic optimizations* dan *cost-based optimizations*. Kedua teknik optimasi *query* ini seringkali digabungkan untuk mendapatkan hasil *query* yang lebih optimal dan maksimal.

Heuristic optimizations adalah teknik pengoptimalan *query* yang digunakan untuk mengurangi kompleksitas dari *query* dengan mengurangi sebanyak mungkin baris data yang tidak diperlukan dalam prosesnya [26]. Jadi, tabel yang ada di dalam *query* akan di-*filter* terlebih dahulu sesuai dengan kebutuhan analisis, selanjutnya baru digunakan agar *database* tidak perlu melakukan *scan* seluruh baris data di tabel tersebut. Ada sejumlah aturan dalam penggunaan *heuristic optimizations*, yaitu melakukan operasi *selections* dan *projections* untuk mengurangi jumlah baris dan atribut pada tabel dan melakukan proses *selections* dan *join table* yang paling kecil keluarannya sebelum melakukan operasi lainnya [27]. Salah satu keunggulan dari pendekatan *heuristic* adalah proses optimasi yang sederhana dan mudah untuk diimplementasikan. Dengan mengikuti serangkaian aturan yang sudah ditentukan, *optimizer* dapat dengan cepat membuat perencanaan eksekusi untuk *query* tertentu [28].

Cost-based optimizations adalah proses mengoptimasikan *cost* yang digunakan dari beberapa alternatif yang tersedia, lalu dipilih salah satu yang menjadi *cost* terendah [27]. Faktor terpenting dalam menentukan perencanaan *query* adalah *cardinality*, jadi semakin sedikit jumlah baris data yang perlu diproses oleh SQL *operator*, *query* akan berjalan semakin cepat [29]. Berikut ini adalah beberapa cara yang dapat dilakukan untuk mengoptimalkan *query* SQL dari sisi struktur penulisan.

2.2.2.1 *Join Tables*

Join table adalah penggabungan dua tabel atau lebih dengan menggunakan *query* yang dapat dilakukan dengan menggunakan *key* tertentu yang mempunyai nilai terkait untuk memperoleh data yang lengkap dari tabel yang lain. *Join table* diperlukan karena dapat mengurangi duplikasi dan pemborosan. Selain itu, penggunaan *join table* memungkinkan untuk pengambilan data secara efisien, sehingga dapat menghasilkan observasi dan analisis yang bermanfaat di dalam bisnis.

Penggunaan *query join table* umumnya dilakukan dengan menggabungkan kolom *primary key* di tabel satu dan kolom *foreign key* di tabel lainnya. *Join table* sendiri bisa ditambahkan dengan kondisi tertentu dengan menggunakan klausa WHERE untuk melakukan *filter* data. Ada beberapa jenis *join table* yang dapat digunakan sesuai kebutuhan, yaitu *inner join*, *full outer join*, *union join*, *left join*, dan *right join*. Untuk kasus yang membutuhkan *join table* lebih dari satu tabel, dapat menggunakan *multi-join tables* dengan struktur penulisan yang sama, hanya berbeda di jumlah tabel yang digabungkan [30].

2.2.2.2 *Index Database*

Index database adalah objek-objek dalam sistem *database* yang dapat mempercepat proses pencarian *query* data. Penggunaan *index database* bisa digambarkan mirip dengan daftar isi buku yang dapat mempermudah pencarian data, sehingga tidak perlu mencari setiap *record* data pada tabel yang diperlukan [15]. Ada beberapa tipe *index database* yang umum digunakan, yaitu *hash index*, B-*tree index*, dan *bitmap index* [31].

Hash index adalah adalah *index* berdasarkan *ordered list* pada *hash values*, sehingga cocok untuk tipe *query* yang memiliki operasi simpel. Algoritma *hash* dibuat untuk membuat *hash value* dari kolom *key*. Umumnya, *index* ini dapat digunakan untuk pencarian yang cepat, seperti penggunaan LNAME=”Scott” dan FNAME=”Shannon” [31].

Tipe *index* selanjutnya yang umum digunakan adalah B-*tree index*. B-*tree index* adalah struktur data berurutan yang diorganisasikan sebagai *upside-down tree*. Pohon *index*-nya disimpan terpisah dari data aslinya. Semakin rendah *leaves* pada B-*tree index*, berarti semakin dekat dengan baris data yang aktual. B-*tree index* bersifat “*self-balanced*” yang berarti membutuhkan waktu yang sama untuk mengakses baris tertentu dalam *index*. B-*tree index* merupakan tipe *index* yang paling umum dari berbagai jenis *index* yang digunakan di dalam *database* [31].

Tipe *index* terakhir yang juga banyak digunakan adalah *bitmap index*. *Bitmap index* menggunakan *bit array* (0s dan 1s) untuk merepresentasikan eksistensi dari *value* atau kondisi. *Bitmap index* umumnya paling banyak digunakan di aplikasi *data warehouse* dengan tabel yang memiliki jumlah baris yang banyak dan jumlah kolom yang sedikit. *Bitmap index* cenderung untuk menggunakan ruang penyimpanan yang lebih sedikit dibandingkan B-*tree index* karena *bitmap index* menggunakan *bits*, bukan *bytes* untuk menyimpan baris datanya [31].

Di dalam SQL, *index* dapat dibuat dengan satu kolom atau beberapa kolom apabila *index* tersebut digunakan. Artinya, hasil dari *index* dapat berupa satu kolom (*column index*) dari sebuah tabel, atau bisa juga beberapa kolom (*multiple column index*) dari sebuah tabel . Semua tipe data yang ada di *database* dapat diberikan *column index*, sehingga menggunakan *index* pada kolom yang merupakan cara yang sesuai untuk meningkatkan performa *database* pada operasi *selections*. *Multiple column indexes* dapat dikatakan sebagai *array* yang memiliki nilai-nilai yang digabungkan dengan nilai pada kolom yang dibuat *index*.

2.2.2.3 *Partition Table*

Partition table adalah salah satu teknik yang umum digunakan di dalam penulisan SQL untuk membagi tabel dengan data yang besar menjadi beberapa bagian kecil yang dapat diakses, disimpan, dan dipertahankan secara independen [32]. Dengan memecah tabel yang besar menjadi lebih kecil, *query* hanya akan mengakses sebagian kecil dari data yang dapat membuat kinerja *query* menjadi lebih cepat karena jumlah data yang perlu di-*scan* menjadi lebih sedikit. Umumnya, *database* dapat digunakan untuk jangka waktu yang lama, sehingga jumlah *record* akan sering mengalami peningkatan hingga mencapai angka yang besar. Hal ini secara langsung dapat mempengaruhi waktu kecepatan *query* ketika dijalankan. Oleh karena itu, *partition table* memang bermanfaat untuk mengurangi dampak dari penambahan jumlah data pada *database* terhadap kecepatan *response time* pada *query* yang dijalankan [32].

Contoh tabel yang sering digunakan untuk dipartisi adalah tabel yang berisi *sales records* berdasarkan kolom tanggal transaksi terjadi. Satu tabel partisi bisa memiliki semua *sales records* di bulan Januari, lalu tabel partisi lainnya bisa memiliki data *sales records* untuk bulan Februari, dan seterusnya. Ada tiga tipe partisi yang umum digunakan untuk bagian tanggal secara horizontal, yaitu partisi per-tahun, partisi per-kuarter, dan partisi per-bulan. Walaupun begitu, dalam banyak kasus, 80% yang digunakan adalah partisi per-bulan [33].

Pada contoh tabel yang berisi *sales records*, *partition table* yang dilakukan adalah partisi secara horizontal di bagian tanggal transaksi. Partisi tabel secara horizontal umumnya digunakan untuk *query* yang berkaitan dengan rentang tanggal ataupun untuk kondisi, sehingga dapat membantu mengurangi jumlah data yang perlu di-*scan* untuk *query* yang lebih spesifik dan memudahkan untuk melakukan manajemen data [34]. Selain partisi secara horizontal, tabel juga dapat dilakukan partisi secara vertikal ataupun kombinasi antara horizontal dan vertikal. Partisi vertikal adalah partisi untuk membagi tabel berdasarkan kolom yang berguna ketika memiliki tabel dengan kolom yang banyak dan tidak semua kolom diakses secara bersamaan. Partisi vertikal ini dapat meningkatkan performa *query* dengan

mengurangi I/O dan memudahkan untuk efisiensi *indexing* pada kolom yang relevan [34].

2.2.3 *Database Performance Measurement*

Secara umum, performa adalah pengukuran terkait efisiensi *software* dalam menyelesaikan tugasnya. Efisiensi sendiri dapat diukur dengan berbagai hal, baik diukur dengan *response time*, banyaknya *concurrent user* yang dapat dilayani oleh sistem, atau efisiensi sistem yang dimiliki pada *software*. Semakin banyak sistem meng-*handle* data, semakin besar juga ketergantungan performa pada *software* terhadap *database*. *Database* juga sering menjadi aspek pertama yang diperiksa apabila ada masalah pada performa *software*. Oleh karena itu, sangat penting untuk selalu mengecek dan menguji performa *database* untuk memastikan bahwa performa *database* selalu dalam keadaan yang baik untuk mengurangi adanya permasalahan performa pada *software*. Walaupun DBMS dapat melakukan beragam aktivitas selain pada *querying*, namun umumnya dalam pengukuran DBMS, *querying* adalah aspek yang akan diukur untuk mengecek performa pada *database* [35].

Di dalam *database*, performa umumnya diukur dengan *response time*, *throughput*, dan dalam beberapa kasus, menggunakan *computing resources* [35]. *Response time* adalah waktu yang diperlukan untuk mengembalikan hasil setelah *end-user* mengirimkan *request* ke aplikasi *software*, yang akan dilanjutkan ke *request* ke DBMS. Hasil jawaban pada *request* DBMS baru akan dikembalikan ke aplikasi dan ditampilkan sesuai dengan tampilan yang diatur pada masing-masing *device* [35]. *Response time* biasanya dianggap sebagai waktu yang berlalu sejak *user* memasukkan *command* atau mengaktifkan fungsi sampai dengan waktu aplikasi menunjukkan bahwa *command* atau fungsi telah selesai dijalankan [36].

Setiap *user* tentu menginginkan agar aplikasi ataupun *database* dapat memberikan waktu *response time* yang cepat, agar data yang diminta *user* untuk ditampilkan ke dalam *software* juga dapat semakin cepat ditampilkan. *Response time* dapat mempengaruhi *user experience* juga, seperti toleransi, penerimaan, dan kepuasan dari *user* setelah menggunakan *software*. *Response time* yang

semakin lama akan memberikan efek negatif terbesar dari kepuasan *user*, lalu penerimaan, dan terakhir adalah toleransi [37].

Selain dari *response time*, pengukuran performa DBMS juga bisa dilihat dari *throughput*. *Throughput* adalah jumlah data yang berhasil diproses sistem dalam satuan waktu tertentu, bisa dengan satuan *byte per second* (bps), *megabyte per second* (Mbps), ataupun *gigabyte per second* (Gbps). Satuan yang digunakan ini bisa beragam tergantung pada volume data yang ditransfer. *Throughput* sendiri dapat diukur dengan berbagai cara disesuaikan dengan konteks penggunaannya [35], [38]. Dalam proses sistem transaksi *database*, umumnya *throughput* diukur di dalam *transactions per second* (TPS) atau *transactions per minute* (TPM) [36]. Pengukuran dengan TPS atau TPM ini penting untuk menilai kinerja pada *website* ataupun aplikasi yang berbasis transaksi.

Terakhir, performa *database*, dalam beberapa situasi, juga dapat diukur dengan *resource utilizations*. Beberapa contohnya seperti dengan mengukur performa *database* dari CPU *time*, I/O, alokasi memori, atau konsumsi energi di dalam sistem untuk kebutuhan efisiensi energi, bisa karena *battery power* yang terbatas, atau karena permasalahan lingkungan. Pengukuran performa dari sisi *throughput* secara singkat dapat dianggap sebagai proses yang membutuhkan simulasi pada tingkat tertentu, sedangkan pengukuran performa dari sisi *response time* adalah suatu metode dengan angka yang tepat.

2.3 Teori *Framework* yang Digunakan

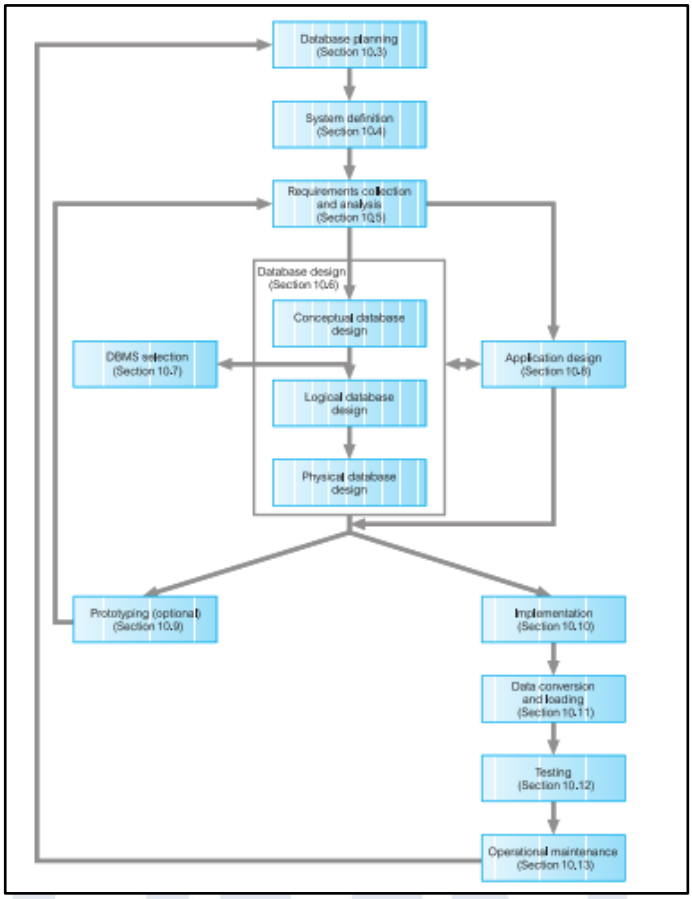
2.3.1 *Database System Development Lifecycle* (DSDLC)

Umumnya, dalam pengembangan *software*, pendekatan yang dilakukan untuk memudahkan manajemen *progress* dari pembuatan *software* adalah dengan *Software Development Lifecycle* (SDLC). Konsep SDLC sendiri menjadi dasar untuk model pengembangan *software*, seperti *Waterfall* dan *Rapid Application Development* (RAD) [39]. *Waterfall* adalah metode yang memiliki kerangka kerja terstruktur dan perlu untuk melewati beberapa tahapan untuk mendapatkan hasil akhir yang diinginkan [40]. Selain *Waterfall*, ada juga RAD yang merupakan salah satu metode yang mudah diimplementasikan

karena pengembangan difokuskan pada setiap tahapan pengembangan, sehingga bisa memakan waktu yang lebih singkat setiap tahapannya. Hasil akhir dari *software* yang dikembangkan juga dapat terlihat dengan jelas dengan durasi waktu pengerjaan yang singkat [41].

Pendekatan SDLC juga bisa disesuaikan dengan produk yang akan dikembangkan. Contohnya adalah jika sedang melakukan pengembangan *website*, nama dari *lifecycle* tersebut bisa disesuaikan menjadi *Website Development Lifecycle* (WDLC). Siklus dari WDLC ini secara garis besar tidak memiliki perbedaan yang signifikan dengan SDLC, namun WDLC dapat digunakan untuk berbagai tipe pengembangan yang berfokus pada produk *website*, seperti dari *waterfall*, *prototyping*, dan *spiral* [42].

Selain WDLC, masih ada turunan *lifecycle* lain yang dapat digunakan. Ketika produk yang dikembangkan lebih difokuskan untuk pengembangan *database*, nama dari *lifecycle* tersebut berubah menjadi *Database System Development Lifecycle* (DSDLC) [43]. Siklus pengembangan *database* ini akan sangat penting terutama untuk pengembangan *database* skala menengah sampai *database* skala besar dengan jumlah pengguna mencapai lebih dari ribuan pengguna.



Gambar 2. 2 *Database System Development Lifecycle* (DSDLC) [19]

Berdasarkan Gambar 2.2 di atas, ada delapan tahapan dalam pengembangan siklus *database.* Di dalam DSDLC terdapat beberapa tahapan yang bersifat repetisi atau mengulang seperti *looping*. Adanya pengulangan ini didapatkan dari *feedback* di dalam pengembangannya, sehingga ada beberapa tahapan yang digambarkan *looping* untuk menunjukkan bahwa tahapan siklus ini bisa menyesuaikan dengan situasi dan kondisi dalam pengembangan. Berikut adalah penjelasan dari delapan tahap siklus pengembangan *database system* atau DSDLC.

1) *Database Planning*

   Tahap ini adalah tahapan untuk merencanakan tujuan dari pengembangan *database* dan alur rencana pembuatan *database* dengan cara yang paling efektif dan efisien.

2) *System Definition*

Tahap kedua ini adalah tahap untuk memberikan deskripsi cakupan dan batasan dari *database* yang akan dikembangkan dan juga target pengguna yang akan menggunakan *database*.

3) *Requirements Analysis*

Tahapan ketiga dari proses DSDLC adalah *requirements analysis*. Tahap ini adalah tahapan untuk mengumpulkan *requirements* yang diperlukan untuk mengembangkan *database*. Informasi yang dikumpulkan di tahap ini umumnya akan terkumpul secara berantakan, sehingga perlu dibuat agar lebih terstruktur dengan *requirement specification*. Beberapa teknik yang masuk ke dalam *requirement specification* adalah Structured Analysis *and* Design (SAD), Data Flow Diagrams (DFD), dan Hierarchical Input Process Output (HIPO) *charts*.

4) *Database Design*

Tahap keempat adalah merancang *database* sesuai dengan informasi yang sudah diidentifikasi di tahap ketiga. Ada dua pendekatan yang sering dipakai dalam melakukan desain *database*, yaitu *bottom-up* dan *top-down*. Kedua pendekatan ini memiliki keunggulan dan kekurangannya tersendiri, sehingga akan lebih baik untuk menggunakan kedua pendekatan ini sesuai dengan kondisi saat akan melakukan pengembangan *database* agar pendekatan yang digunakan bisa lebih efektif.

Pendekatan *bottom-up* dimulai dari tahapan paling dasar, yaitu mengidentifikasi atribut dan relasi antar atribut di *database*, lalu dilanjutkan dengan tahapan pembuatan struktur *database* yang lebih kompleks. Pendekatan ini lebih cocok untuk sistem yang sederhana karena pendekatan ini akan menjadi lebih rumit ketika diterapkan pada sistem *database* yang lebih kompleks. Selain itu, pendekatan *bottom-up* juga lebih cocok untuk *redesign* sistem yang sudah ada sebelumnya.

Pendekatan *top-down* merupakan kebalikan dari pendekatan *bottom-up*. Tahapan awal dari pendekatan *top-down* dimulai dengan *level* yang lebih tinggi, baru diikuti dengan *detail* yang lebih spesifik. Berbeda dengan *bottom-up*, tahapan *top-down* lebih cocok untuk sistem yang

lebih kompleks karena memudahkan memberikan pemahaman secara menyeluruh pada sistem. Tahapan *top-down* ini umumnya dapat diilustrasikan dengan konsep *entity relationship* (ER) *model*.

Walaupun ada dua pendekatan yang dapat digunakan dalam desain *database*, namun secara garis besar ada tiga fase yang perlu dilakukan dalam desain *database*, yaitu *conceptual*, *logical*, dan *physical design*. Berikut adalah poin-poin penjelasan dari tiga fase tersebut.

1. *Conceptual database design*

   Pembuatan desain secara konseptual merupakan tahapan pertama dari *database design*. Tahapan *conceptual database design* hanya perlu fokus pada model data yang akan digunakan tanpa harus memikirkan logika penyimpanan *database* dan pertimbangan fisik *database*.

2. *Logical database design*

   Database yang telah dirancang secara abstrak secara konseptual akan diubah menjadi *logical design* yang memiliki relasi antar tabel. Tahapan yang dilakukan di *logical design* umumnya adalah melakukan normalisasi *database* untuk melihat kesesuaian model data.

3. *Physical database design*

   Physical design adalah tahapan yang difokuskan untuk mengimplementasikan *logical database design* yang sudah dibuat. Tahapan ini lebih difokuskan ke struktur *database* dan integritas data yang akan disimpan di *database* tersebut.

5) *Prototyping*

Tahapan *prototyping* ini merupakan tahapan opsional, jadi bisa disesuaikan kebutuhannya dalam pengembangan *database*. *Prototyping* sendiri adalah tahapan untuk membangun model dari *database*, yang memberikan izin kepada *user* untuk mengevaluasi tampilan dan fungsi dari hasil akhir sistem yang akan dibuat.

6) *Implementation*

Apabila sudah membuat desain *database* dan/ atau membuat *prototyping*, tahapan selanjutnya adalah implementasi dan penerapan rancangan desain *database* ke dalam bentuk *database* asli. Tahapan implementasi mencakup pembuatan *database* dan relasi antar tabel di *database* sesuai dengan rancangan desain.

7) *Data Conversion and Loading*

Tahapan selanjutnya adalah tahapan *loading* data untuk memasukkan data ke *database* yang telah dibuat. Jika terdapat data dari sistem yang lama, nantinya akan dikonversi menjadi data dengan *format* sistem yang baru.

8) *Database Testing*

Setelah memasukkan data ke dalam *database*, tahapan selanjutnya adalah melakukan pengujian *database*. *Database* akan diuji dan divalidasi berdasarkan *requirements* yang ditentukan dari *users* untuk memastikan proses kerja *database* sudah berjalan dengan baik.

9) *Operational Maintenance*

Tahapan terakhir dalam pengembangan *database* adalah *operational maintenance*. Tahapan ini dilakukan untuk memelihara dan memastikan *database* tetap berjalan dengan baik.

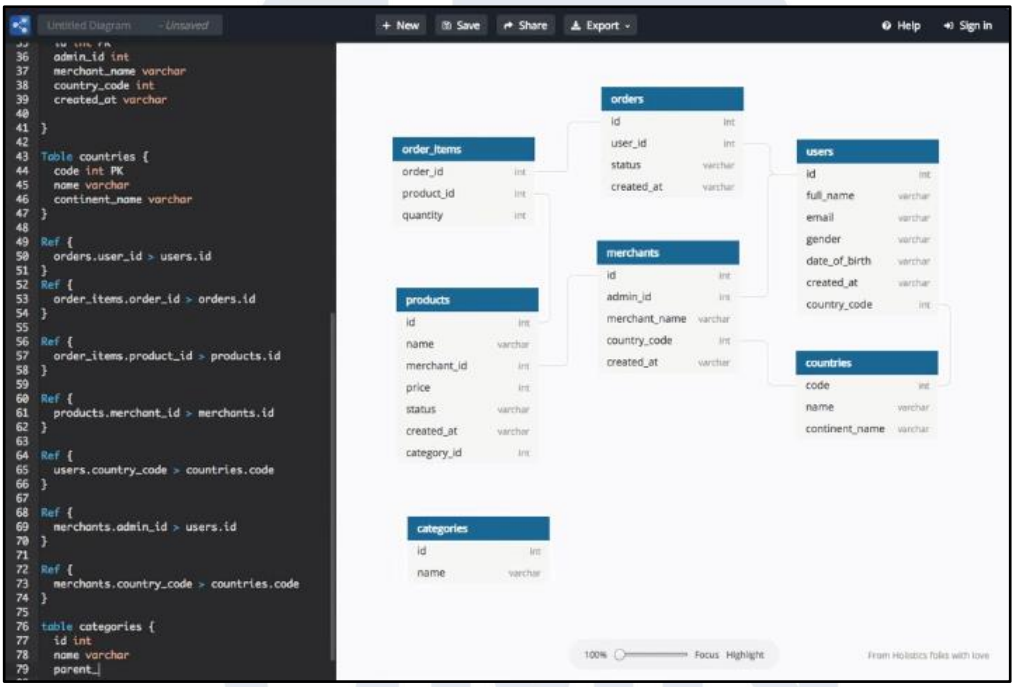
2.4 Teori Tentang *Software* yang Digunakan

2.4.1 dbdiagram.io

Sebelum membuat suatu *database*, akan lebih baik jika membuat rancangan skema *database* terlebih dahulu dengan membuat *Entity Relationship Diagram* (ERD). Secara garis besar, ERD adalah diagram berbentuk notasi grafis yang berada dalam pembuatan *database* yang menghubungkan antara data satu dengan data lainnya. ERD sendiri dapat digunakan sebagai alat bantu dalam pembuatan *database* dan memberikan gambaran terkait pembuatan *database* kedepannya [44].

Salah satu *tools* yang dapat digunakan untuk memudahkan pembuatan ERD adalah dbdiagram.io. *Tools* ini dirancang oleh Holistics.io, salah satu perusahaan untuk produk berbasis *business intelligence* di Singapura.

Penggunaan dbdiagram.io ini memudahkan untuk membuat ERD karena pengguna hanya perlu untuk menuliskan kode sederhana, lalu dbdiagram.io akan secara otomatis membuat visualisasi dari ERD sesuai dengan kode yang diketikkan. Oleh karena itu, *tools* ini umumnya dapat digunakan oleh Developer dan Data Analyst [45]. Gambar 2.3 di bawah ini merupakan tampilan dari penggunaan *tools* dbdiagram.io.



Gambar 2. 3 Tampilan dbdiagram.io untuk pembuatan ERD [45]

Selain kemudahan yang diberikan dalam pembuatan ERD, dbdiagram.io juga memberikan beberapa manfaat lain. Beberapa contoh manfaatnya adalah hasil kode yang sudah dituliskan dapat langsung di-*generate* menjadi *query* SQL sesuai dengan kebutuhan RDBMS. Selain itu, kode ERD juga dapat di-*share* secara *online* kepada *customers* atau anggota tim IT lainnya dengan mudah dan dapat diintegrasikan juga dengan *web frameworks* yang popular, seperti Rails atau Django dengan cara meng-*upload file* schema.rb atau models.py [45].

2.4.2 MySQL

MySQL adalah salah satu RDBMS yang populer digunakan untuk pengembangan sistem. *Database* MySQL memiliki kelebihan utama, yaitu mudah digunakan dan memiliki kinerja yang stabil karena *database* ini mampu

menangani beragam data, mulai dari data dengan ukuran *small to medium size* sampai dengan data dengan ukuran yang besar atau *big data* [46]. Fleksibilitas yang dimiliki MySQL dalam mengolah berbagai ukuran data menjadikan *database* ini menjadi salah satu yang banyak digunakan oleh Developer untuk mengembangkan sistem, seperti contohnya adalah pembuatan *website* dan juga *project-project* lainnya.

MySQL adalah RDBMS yang diciptakan oleh Michael Widenius, David Axmark, dan Allan Larsson di tahun 1995 pada perusahaan Swedia bernama MySQL AB dengan bahasa pemrograman C dan C++[47]. Di tahun 2000, MySQL baru memperoleh popularitasnya karena *database* bersifat *open-source* [48]. Di tahun 2001, MySQL baru diakuisisi oleh Oracle Corporation dan disebarikan tanpa biaya dibawah General Public License (GPL) [49]. MySQL sendiri sudah banyak digunakan di perusahaan, baik perusahaan skala kecil maupun perusahaan skala besar, seperti Google, Facebook, dan Twitter.

2.4.3 PostgreSQL

Database PostgreSQL adalah salah satu *open-source* RDBMS yang juga populer digunakan untuk pengembangan sistem. PostgreSQL sendiri dirilis pada naungan lisensi PostgreSQL, yang memiliki kemiripan dengan lisensi MIT. PostgreSQL juga merupakan kombinasi dari *relational* dan *object oriented database system*, yang memungkinkan adaptasi yang presisi untuk kebutuhan pengembangan sistem . Salah satu keunggulan dari PostgreSQL adalah *database* ini menggunakan MVCC (*Multi Version Concurrency Control*) sebagai metode *concurrency*, sehingga memungkinkan banyak pengguna mengakses *object* dan *resource* yang sama dari *database* pada waktu yang bersamaan.

Database PostgreSQL pertama kali dikembangkan di tahun 1986 sebagai tindak lanjutan dari INGRES. INGRES sendiri adalah *open-source relational database* yang sudah mulai dikembangkan sejak awal tahun 1970an, namun saat ini namanya lebih dikenal dengan PostgreSQL. PostgreSQL sendiri adalah *database* yang dikembangkan oleh Michael Stonebraker, seorang profesor ilmu komputer di Berkeley Computer Science Department. Saat ini, PostgreSQL sudah didukung oleh banyak *platform* dan bebas lisensi [50].

Database PostgreSQL dapat digunakan di banyak situasi dan di banyak industri bisnis. PostgreSQL adalah salah satu *database* yang cocok digunakan untuk manajemen *Online Transaction Processing* (OLTP) [51]. PostgreSQL dapat bekerja dengan baik di berbagai kasus, seperti *e-commerce*, *customer relationship management* (CRM), dan buku besar keuangan.

2.4.4 Microsoft SQL Server

Microsoft SQL Server adalah salah satu RDBMS yang dikembangkan oleh perusahaan Microsoft. *Database* ini merupakan aplikasi *desktop* yang dirancang untuk menyimpan dan mengelola data dengan *Structured Query Language* sebagai bahasa *query* yang utama. Microsoft SQL Server banyak digunakan dalam aplikasi, mulai dari *online transaction processing* (OLTP), *data warehousing*, *web applications*, dan juga *business intelligence*. Selain itu, mengingat Microsoft SQL Server juga merupakan produk dari Microsoft, hal ini juga menjadi salah satu keunggulan karena bisa dengan mudah terintegrasi dengan produk Microsoft lainnya, seperti Microsoft Excel dan Power BI, sehingga bisa meningkatkan efektivitas dalam dunia bisnis [52].

Dalam mengelola Microsoft SQL Server, umumnya menggunakan bantuan aplikasi bernama SQL Server Management Studio (SSMS). SSMS sendiri memiliki fasilitas lengkap yang dapat membantu seorang Database Administrator (DBA) dalam mengelola *database* seperti objek *database*, mulai dari tabel, kolom, *view*, dan objek-objek lainnya, mengelola data dalam *database*, dan melakukan proses manajemen *database*, mulai dari *backup*, *restore*, *import*, dan *export* [53].

2.4.5 Apache JMeter

Apache JMeter adalah aplikasi *open source* menggunakan bahasa pemrograman Java sebagai basis pengembangannya, karena itu Apache JMeter kompatibel untuk hampir semua *operating system* (OS) yang menggunakan Java versi 6 atau versi Java terbaru. Apache JMeter dapat digunakan untuk melakukan pengujian pengukuran performa dan *functional behavior*. Apache JMeter sendiri awalnya lebih diarahkan untuk melakukan pengujian aplikasi *website*, namun saat ini Apache JMeter sudah berkembang untuk pengujian fungsi-fungsi lainnya [54]. Beberapa fungsi layanan lain yang tersedia di

Apache JMeter adalah layanan *database* menggunakan *Java Database Connectivity* (JDBC), *Lightweight Directory Access Protocol* (LDAP), *Message-Oriented Middleware* (MO) melalui *Java Messaging Service* (JMS), dan layanan lainnya yang dapat digunakan sesuai kebutuhan.

Penggunaan Apache JMeter untuk pengujian *database* akan memungkinkan untuk mengobservasi CPU, *memory*, *latency*, *response time*, dan beberapa indikator lain yang ada di *database*. Pengujian *database* sendiri diperlukan untuk menemukan dan mengidentifikasi masalah yang ada di *database*, terutama ketika *database* sedang bekerja dengan jumlah *workloads* yang banyak dengan jumlah pengguna yang banyak.

Jika difokuskan untuk pengujian performa *database*, maka Apache JMeter bisa digunakan untuk menguji beberapa RDBMS yang cukup populer, seperti MySQL, PostgreSQL, Oracle *Database*, Microsoft SQL Server, dan *database* lainnya. Untuk menguji RDBMS di Apache JMeter perlu menggunakan bantuan dari *driver* bernama *Java Database Connectivity* (JDBC). Jadi, nantinya *user* bisa melakukan *download driver* terlebih dahulu sesuai dengan RDBMS yang ingin diujikan performanya.

Salah satu kelebihan dari penggunaan Apache JMeter adalah dapat digunakan dengan dua cara, yaitu dengan *graphical user interface* (GUI) dan non-*graphical user interface* (non-GUI) dengan menggunakan *command line*. GUI dapat digunakan untuk melakukan *testing* yang lebih ramah bagi *user* karena memiliki *interface* untuk memudahkan dalam proses *testing*. Selain adanya dua tipe tampilan, Apache JMeter juga mendukung pengujian *scalability* dan *load testing* dan menyediakan hasil analisis *response time* pada *database* yang akurat. Walaupun begitu, Apache JMeter sendiri juga memiliki kekurangan, salah satunya adalah membutuhkan *memory* dan *processing power* untuk skala *testing* yang besar.