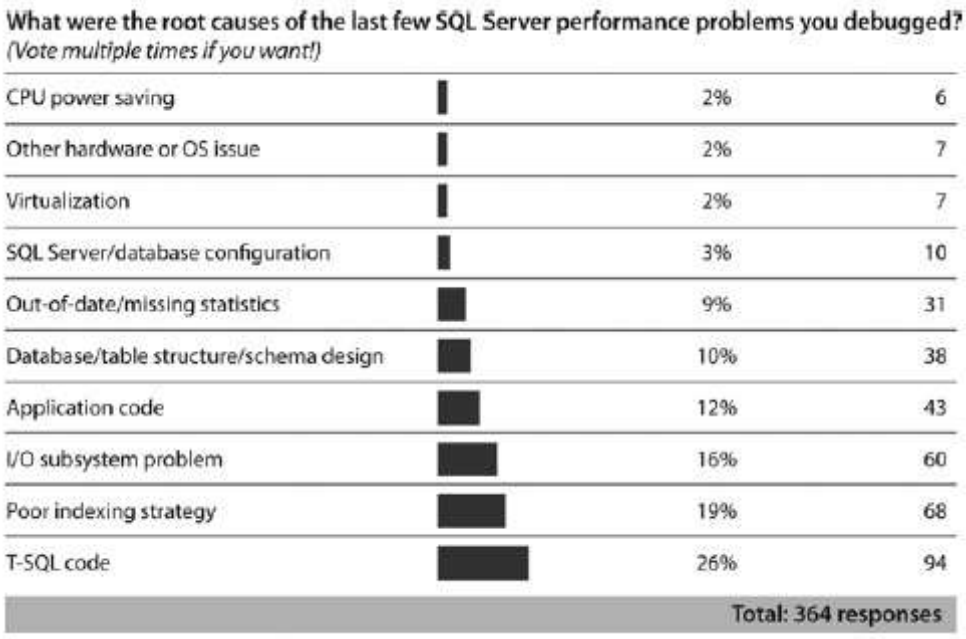
BAB II. LANDASAN TEORI

2.1. SQL Server 2012 *Query Performance Tuning*

Menurut hasil survei yang dilakukan oleh Paul S. Randal telah menyatakan bahwa tingkat akar permasalahan dari performa SQL Server terlihat dari gambar 1.1, terdiri dari beberapa tingkat yang dimulai dari tingkat rendah yaitu dari sisi hardwarenya hingga ke tingkat tinggi yaitu struktur kode T-SQL yang diambil dari 364 responden (Fritchey, 2012).



Gambar 2.1. – Akar permasalahan dari performa SQL (Grant Fritchey, 2012, p. 10)

Dari gambar 2.1. diatas dapat dilihat bahwa akar permasalahan dari performa SQL Server dari tingkat terkecil hingga tertinggi yaitu:

1. Masalah CPU *power saving* merupakan penghematan penggunaan energi yang dapat memicu berkurangnya performa CPU.

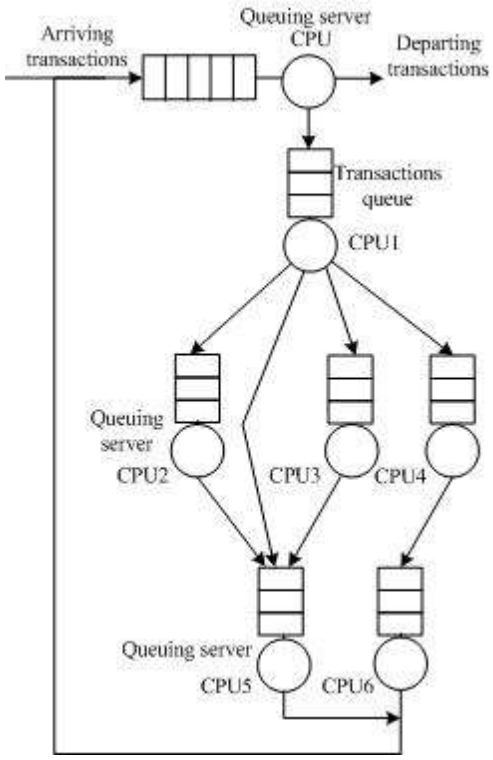
2. *Other hardware or OS Issue* merupakan masalah dari perangkat keras atau isu dari Sistem Operasi.
3. *Virtualization* merupakan sistem komputer fisik utama yang terbagi menjadi beberapa bagian sistem komputer secara virtual.
4. SQL Server / *Database Configuration* merupakan konfigurasi pada *database* atau pada DBMS SQL Server.
5. *Out of date / Missing Statistic* adalah kurangnya nilai statistik indeks dalam pencarian data pada tabel
6. *Database/Table Structure/Schema Design* adalah *database* atau struktur tabel atau desain skema yang bermasalah sehingga dibutuhkan perbaikan dalam perancangannya.
7. *Application Code* adalah kode atau program yang dibuat di dalam aplikasi.
8. *I/O Subsystem Problem* adalah masalah dalam sub- sistem input maupun output dari suatu perangkat (seperti *hardisk*, LAN atau jaringan internal maupun eksternal).
9. *Poor Indexing Strategy* adalah buruknya strategi perancanga *indexing* pada suatu tabel.
10. T-SQL *Code* adalah ada masalah di dalam kode *Transact* SQL yang dibuat.

Jika aplikasi yang dirancang memiliki kelemahan dalam aplikasi desainnya maka banyak yang beranggapan bahwa yang harus ditingkatkan ialah sumber daya (*resources*) dari sisi perangkat kerasnya sehingga merupakan suatu usaha yang mengurangi nilai investasi yang ada dan ada baiknya jika kita memperbaiki

kelemahan yang ada pada aplikasi tersebut. Dengan demikian optimasi query memungkinkan untuk meningkatkan performa pada kinerja aplikasi. Begitu juga dengan hasil survey diatas nilai presentasi akar permasalahan dari performa SQL begitu kecil dari segi sumber daya (*resources*) akan tetapi lebih ke bagian masalah di *database*, *application code* dan lainnya berhubungan dengan *query*.

2.2. Hirarkis Model Jaringan Antrian

Hierarchical Queueing Network (HQN) dipublikasikan oleh (Shao et al., 2015). HQN adalah sistem hirarkis yang mencakup server dan layanan. HQN merupakan sistem yang telah menempati sejumlah sumber yang memprovokasi satu dengan yang lainnya. Dan hal ini digunakan untuk penilaian kinerja (*performance assesment*) untuk sistem yang mengandung sumber daya perangkat lunak dan sumber daya perangkat keras. Kedalaman pemanggilan (*query*) tergantung pada jumlah lapisan panggilan yang berlangsung. Dengan menggunakan jaringan antrian yang berfungsi untuk mengevaluasi kinerja *database*, maka seorang yang menyusun desain *database* harus menggunakan teknik optimasi *query* untuk memperkirakan biaya dari transaksi yang dijalankan pada *database* yang digunakan. *Database* SQL Server dapat digambarkan sebagai sistem jaringan antrian. Hubungan antara server dan transaksi dapat dipetakan ke model hirarki jaringan antrian (HQN).



Gambar 2.2. – Ilustrasi HQN Model (Shao et al., 2015, p. 189)

Dilihat dari gambar 2.2 diatas Server dinyatakan sebagai sumber daya CPU yang terbagi (*shared*). Antrian transaksi akan menggunakan sumber daya CPU tersebut dan CPU melayani transaksi langsung saat tidak ada transaksi yang dijalankan, jika CPU sedang sibuk menjalankan transaksi maka transaksi perlu menunggu di dalam antrian sampai transaksi sebelumnya telah dijalankan dan meninggalkan sistem CPU. Sehingga mekanisme *First In First Out* (FIFO) dijalankan oleh masing-masing CPU yang berada pada server.

Pendistribusian sistem *database* SQL Server memiliki server bersifat *multi-layered*. Proses perangkat lunak yang berjalan pada sumber daya perangkat keras yang dipakai secara bersamaan dapat menyebabkan penundaan karena banyaknya transaksi yang masuk. Sehingga berkaitan dengan nilai dari *thread process*-nya, maka jumlah proses *instance* dan proses alokasi untuk prosesor. Algoritma HQN

berkaitan dengan pendistribusian antrian transaksi di dalam sistem *database* SQL Server yang dapat memprediksikan kinerjanya, Berdasarkan hasil yang diprediksi, tuning kinerja dan optimasi dapat diimplementasikan.

Algoritma untuk kinerja database SQL Server optimasi (HQN-SPO) adalah sebagai berikut:

Input: jumlah server *Np*, waktu berjalan *Tp* untuk server *P*

Output: waktu respon sebelumnya *rp* untuk server *P*, waktu respon saat ini *rc*, tugas dari *throughput Tp* dari Server *P*, waktu respon toleransi yang memungkinkan *ε*, tingkat utilitas *Up* server *P*.

Berikut algoritma yang dibuat:

```
while （|rc-rp|> ε ）
do {
        do while （|rc-rp|> ε ）
        {
                for (L=M-1,L<=1,L--)
                {
                Response times are updated from top to bottom level;
                }
                New performance candidate NC←generated improved performance candidates;
                // candidate C, performance requirements R, Property\initial Property//
                }
                New iterated candidate C is selected;
Return (the optimal performance setting);
}
```

Gambar 2.3. – Algoritma penentuan kandidat performa yang telah ditingkatkan

Desain *database* yang dibuat berisi tabel dan transaksi dapat mengakses isi dari tabel. Indikator penilaian kinerja *database* yang digunakan ialah durasi

waktunya. Total durasi waktu tunggu yang diambil dari penggunaan akses tabel antrian untuk transaksi lainnya yang mengakses tabel waktu penggunaan (*consuming*) dapat dilihat sebagai *query*.

Taccess → Tgetdata → Tsavedata

Taccess = Tgetdata

Tgetdata = Tsavedata

Gambar 2.4. – Skema waktu akses, pengambilan dan penyimpanan data

Total waktu yang dihabiskan untuk mengakses data dari total waktu yang mendapatkan data dan total waktu untuk menyelesaikan operasi penyimpanan data. Mengembalikan data ke pengguna tergantung pada waktu yang dibutuhkan antara klien dan server dan dari kecepatan *response time*-nya. Laporan proses waktu penggunaan ditentukan dari tingkat pengolahan dari klien.

	Processor	MM	OS
Server	Intel Core MP(2.7G/2M)	4GB	MS Windows Server 2005 SP3
Client	Pentium ®4 (2.26G HZ)	1GB	MS Windows 2005 SP3

Gambar 2.5. – Konfigurasi Sistem Server & Client

Situation	Database scale	SQL server version	Database size	MM of server
C1	30	7.0	1.87GB	1GB
C2	30	7.0	3.3GB	4GB
C3	60	7.0	1.87GB	1GB
C4	60	7.0	3.3GB	4GB

Gambar 2.6. – Konfigurasi database SQL Server

Transaction	Least percentage	Average judging time (s)	Least crucial time (s)
New transaction	N/A	15	22
Transaction ending	61	17	5
Transaction status	6	16	7
Transaction delivery	7	9	6

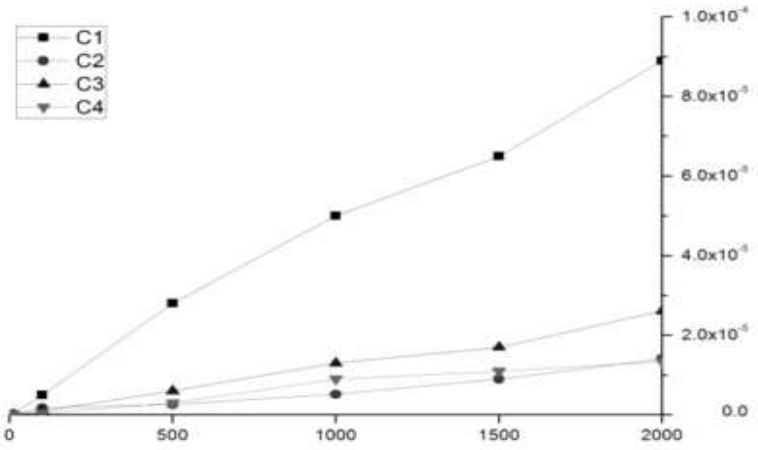
Gambar 2.7. – Transaksi TPC-C Benchmark

Gambar diatas menerangkan nilai persentase, rata-rata waktu yang penting (*crucial time*) dan rata-rata waktu penilaian (*judging time*) dari transaksi yang berbeda.

Hal ini diketahui bahwa gambar diatas transaksi baru tidak dihitung untuk persentasenya karna adanya antrian (queue). Waktu yang terpenting adalah 5 detik, dan 22 detik untuk transaksi baru.

Name	Base (byte)	Length of row(byte)	Rows /page
Transaction	5000000	701	5
Old transaction	5000000	48	46
New transaction	1250000	6	271

Gambar 2.8. – Inisiasi skala muatan dari HQN model



Gambar 2.9. – Nilai waktu rata-rata dari transaksi yang berbeda

Hasil eksperimen pada TPC-C *benchmark* menggunakan HQN menunjukkan bahwa optimasi kinerja SQL server dapat mencapai peningkatan kinerja 40% lebih daripada rata-rata dari algoritma klasik.

2.3. Teknik Query Optimization di Relational Database

Dalam jurnal yang berjudul Menjelajahi Teknik Query Optimization di Relational Database oleh (Khan & Khan, 2013). Berisi beberapa literatur yang didapat dan menghasilkan suatu gambaran yang jelas untuk mendapatkan solusi dari permasalahan Optimasi *Query.*

Tabel 2.1. – Ringkasan Teknik Opimasi Query

Research Topic	Author(s)	Problem Discussed and Proposed Solution	Strengths	Limitation/Scope
Query Optimization	Ioannidis [1]	Described the structure of the optimizer and explained the main issues handle by each optimizer module.	Understanding of optimization concepts and main modules of query optimizer.	Dynamic, parallel and distributed optimizations are not discussed.
Query Optimization	Li *et al*. [2]	Discussed intra-query redundancy in sub-queries. Suggest un-nesting and some other heuristics strategies like selection, projection and joining.	Optimizing query by removing intra-query redundancy.	The experiments performed are not related to the techniques mentioned in the paper.
Genetic optimization for join ordering problem	Sun *et al*.[3]	Proposed genetic algorithm to face the problem of efficient selection of join ordering for making an optimal plan by an optimizer.	Better execution time for the query by proper join ordering.	In terms of cost, the proposed algorithm is little expensive for more than 20 joins.
Sub-query optimization in Oracle	Bellamokanda et al. [4]	Avoiding self joins, multi-joins, multi table access and handling NULL values. Proposed sub-query coalescing, sub-query removal using	Improved query execution time.	Simulation results are achieved using parallel CPUs and other high level hardware which may be affected if degree of parallelism becomes low.
Paging Query Optimization of Massive Data	Sun et al.[5]	Paging query in large scale databases. Proposed a sharable stored procedure at server side and applying Oracle techniques like memory	The proposed solution has huge effect on performance by utilizing the server and client resources efficiently.	The paper mainly focused on Oracle user guides.

Autonomic Computing in SQL Server	Mateen et al.[6]	Development of an Automatic Database Management Systems (ADBMS) Introduced self- optimization, self-healing, self-protection, self-	Main advantages of the proposed solution are: less DBA interaction with the system, efficient use of system resources, recovery and protection.	No simulation is performed to show the autonomic nature of the system.
An overview of query optimization	Chaudhuri [7]	Selecting the best execution plan for a query by an optimizer. Discussed the fundamental requirements for search spaces, accurate cost	Best execution plan out of many candidates plans.	No optimization technique is discussed pertaining to memory issues linked with optimization.
Query optimization in centralized, distributed systems	Hameurlain [8]	The optimization in uni-processor, distributed and parallel processing environments. Discussed enumerative and random	Enhance optimization of query in all the environments	The paper lacks algorithmic or prototype support.
Query optimization strategies for distributed databases	Lin [9]	Query optimization in distributed databases. The local data dictionary is added with sentence table to	These strategies enhance the query efficiency and reduce data transmission costs.	As size of the corresponding table becomes large, it takes more CPU time and occupies more
		store mostly-used results to avoid the transmission of large data.		memory.
Optimizing Join queries	Zafarani et al.[10]	Join ordering in heterogeneous distributed databases.	Reduced calculation of join orders result in better response.	The simulation results shown in the paper are complex.
Query optimizers	Chaudhuri [11]	Cardinality estimation and effective cost estimation.	Best response time for a query.	No interface for the optimizer is discussed.
Query optimization strategy of the VDSI system database	Sun et al. [12]	Query optimization of large scale VDSI systems. Proposed indexes algorithm.	Reduces I/O operations and improves the response time of the query.	No cache technique is discussed.
Empirical evaluation of LIKE operator in Oracle	Gupta et al. [13]	Performance issues with LIKE operator, DB design and indexes	High throughput and best response time is achieved.	The paper only discusses LIKE operator, but the experimental diagrams and results are about indexes.

Query Optimization Techniques for Partitioned Tables	Herodotou et al.[14]	Optimization of SQL queries which are running over partitioned tables. Proposed partitioned aware technique for the optimizer.	Fast query processing, access of data in parallel fashion, efficient mechanism to load data and to maintain statistics.	Ambiguity about the type of partition for which the proposed technique can perform better.
Query processing and optimization in Oracle RDB	Antoshenkov et al.[15]	Query processing and its optimization in Oracle relational databases by reworking the query optimizer.	Improved compilation time to reduce complexity and to handle large amount of data.	Partitioning can be introduced while dealing with large amount of data.
Multi-Table Joins Through Bitmapped Join Indices	O'Neil et al. [16]	Query performance in decision support systems and OLAP environments by introducing a method to execute the common multi-table joins (Star	Better evaluation plans.	Indexes seriously affect performance when DML operations are performed.
Power Hints for Query Optimization	Bruno et al. [17]	To improve the poor plan selected by optimizer by query hints.	Better query plan.	Query hints are a non-trivial complex method to achieve better plan.

Mayoritas penelitian yang dilakukan di paper tersebut menyoroti teknik dan model yang digunakan untuk meningkatkan kinerja *query*. Penulis jurnal (Khan & Khan, 2013) berniat untuk mengusulkan teknik yang akan meningkatkan masalah kinerja yang menunjuk pada penelitian yang ada mengenai Optimasi *Query*. Penulis jurnal (Khan & Khan, 2013) berfokus untuk meningkatkan kinerja query melalui teknik optimasi di lingkungan terdistribusi yang memiliki jumlah data besar yang terletak di lokasi yang berbeda.

2.4. Optimasi *Query* dalam Relational Database untuk Mendukung Teknik *Top-k Query Processing*

Dalam jurnal publikasi karya (Kashem, Chowdhury, Deb, & Jahan, 2010), menjelaskan tentang optimasi *query* dengan pendekatan teknik *Top-K Query Proccess*.

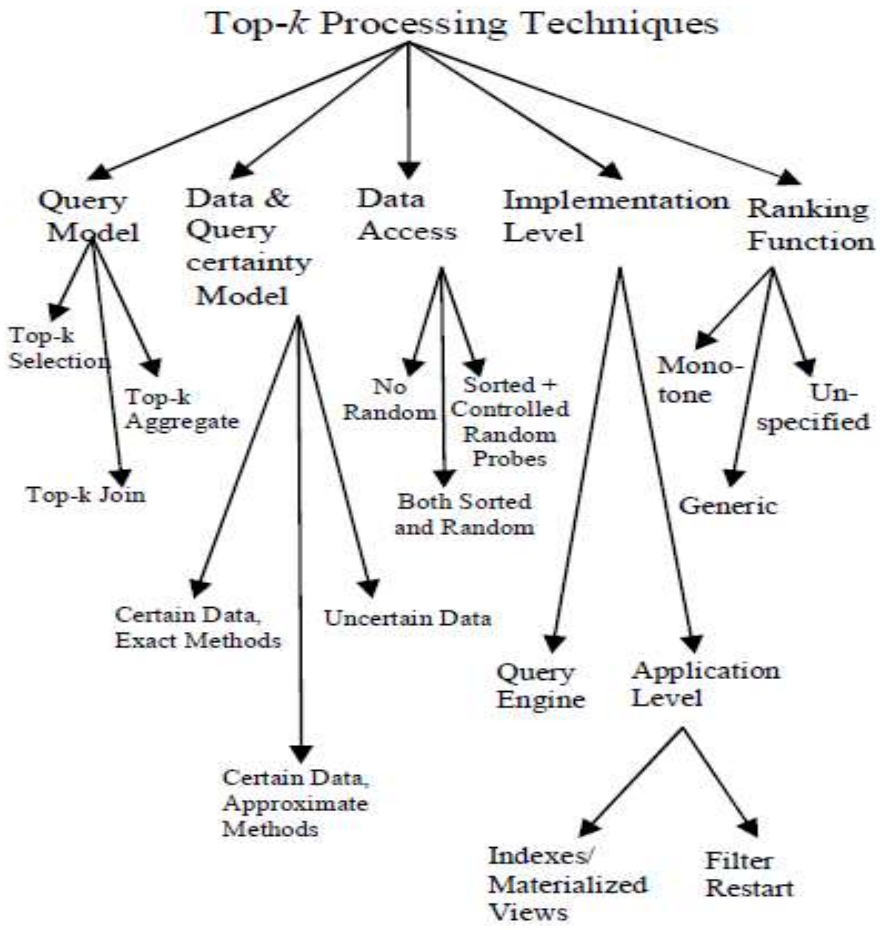
Relational Database adalah kumpulan item data yang terorganisir sebagai satu set tabel secara formal dijelaskan dimana data dapat diakses atau dipasang dengan berbagai cara tanpa harus reorganisasi tabel *database*.

Pengolahan *top-k* yang menghubungkan banyaknya area penelitian *database* termasuk optimasi query, metode *indexing* dan bahasa *query*. Sebagai konsekuensi, dampak dari efisiensi *top-k* menjadikan proses pengolahan lebih jelas dalam peningkatan jumlah aplikasi.

Optimasi *query* adalah untuk memilih strategi eksekusi yang efisien. Proses optimasi *query* ditampilkan di paper tersebut terdiri dari langkah-langkah untuk mendapatkan efisiensi *query* yang behubungan dengan faktor *N* dan menghasilkan rencana yang terbaik untuk *Quer*y *Execution Plan* (QEP). Untuk *query* yang diberikan, ruang pencarian dapat didefinisikan sebagai himpunan *operator tree* yang dapat diproduksi menggunakan aturan transformasi.

Perangkingan *query* berdasarkan *top-k* dapat memperjelas suatu atribut yang dimiliki oleh tabel dan menghubungkan antara relasi yang berkaitan antar tabel.

.



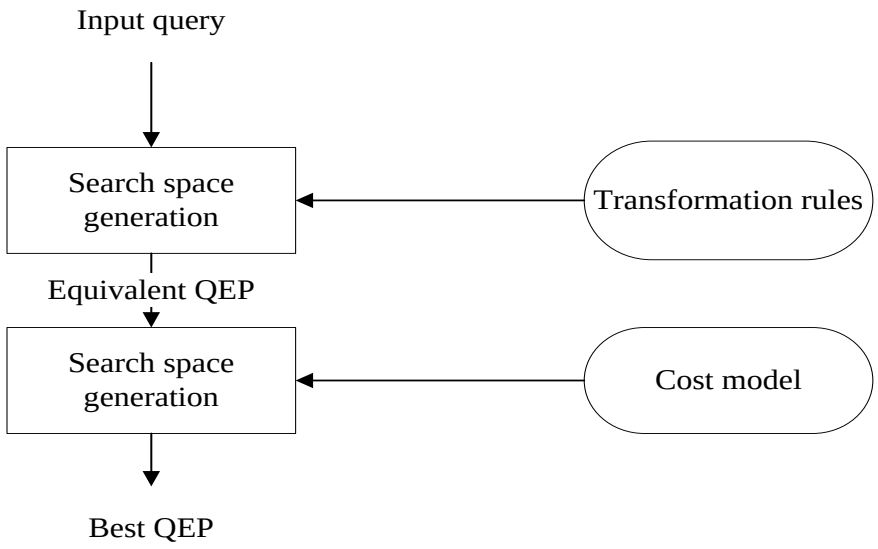
Gambar 2.10. – Top-*k* Processing Techniques (Kashem, Chowdhury, Deb, & Jahan, 2010,p. 55)

Teknik *Top-K Query Processing* adalah Teknik pengolahan *top-k* diklasifikasikan berdasarkan pembatasan mereka yang memaksakan pada fungsi peringkat (*score*). Kebanyakan teknik yang diusulkan dinilai monoton atau kurang efektif terhadap fungsi peringkat (*score*).

2.4.1. Optimasi Query

Tujuan dari optimasi *query* adalah untuk memilih strategi *execution* yang efisien. Proses optimasi query ditampilkan di Gambar 2.2 terdiri dari mendapatkan pertanyaan tentang hubungan *N* dan menghasilkan yang terbaik *Query Execution*

Plan (QEP). Untuk query yang diberikan, ruang pencarian dapat didefinisikan sebagai kumpulan dari himpunan pohon Operator setara yang dapat diproduksi menggunakan aturan transformasi.



Gambar 2.11. – Langkah Optimasi *Query* (Kashem, Chowdhury, Deb, & Jahan, 2010, p. 54)

Optimasi Query dilaksanakan dengan menggunakan pemangkasan arsitektur mengurangi waktu proses *query* drastis dan sehingga mengurangi biaya. Waktu eksekusi yang berbeda untuk nilai *K* yang berbeda perbandingan biaya waktu untuk tradisional dan sistem yang disarankan. Penulis juga menganalisis keakuratan informasi yang didapat dengan cara pendekatan, semua *query* dianalisis dan untuk *Ks* yang berbeda, jawaban dari pendekatan ini ialah 100% sama dengan jawaban yang didapatkan dengan cara pendekatan tradisional.

2.5. Meningkatkan Optimasi *Query* SQL Server dengan Teknik Penulisan Query yang Efisien dan Merancang *Query* yang lebih Cepat

Menurut jurnal yang dibuat oleh (Kumari, 2012) didapatkan informasi untuk meningkatkan performa kinerja *query* SQL Server. Berikut tips dalam langkah optimasi query yang diberikan didalam jurnal tersebut ialah sebagai berikut:

1. Sebuah tabel harus memiliki minimal satu *clustered index* dan jumlah yang tepat dari non *clustered index* yang dibuat pada kolom tabel berdasarkan *query* yang berjalan mengikuti prioritas order sebagai klausa WHERE, maka klausa JOIN, maka ORDER BY klausa dan akhirnya klausa SELECT.
2. Hindari penggunaan Trigger yang merupakan perintah sintaks yang otomatis dijalankan, gunakanlah jika dibutuhkan. Gunakan gabungan Trigers di dalam *stored prosedure*.
3. Setiap tabel harus memiliki *primary key*.
4. Cobalah untuk menggunakan seleksi *constraints* daripada menggunakan *triggers*, bila memungkinkan. *Constraints* lebih efisien daripada *triggers* dalam meningkatkan performa. Gunakanlah *constraints* daripada *triggers*.
5. Cobalah untuk menggunakan variabel tabel bukan tabel sementara. Variabel tabel diperlukan sumber daya *less locking* serta sumber daya *less logging* dibanding tabel sementara, sehingga variabel tabel sebagai pilihan jika memungkinkan untuk digunakan.
6. Hindari penggunaan *views* atau mengganti *views* sama dengan tabel asli.

7. Cobalah menghindari penggunaan klausa DISTINCT untuk menampilkan data tanpa adanya redundan/duplikasi data, gunakanlah seperlunya. Sebagai klausa DISTINCT akan menghasilkan penurunan kinerja *query*, kita harus menggunakan klausa ini hanya bila diperlukan atau tidak dapat dihindari sama sekali.
8. Cobalah untuk menambahkan pernyataan (*statement*) SET NOCOUNT ON ke dalam *stored procedure* karena dapat memberhentikan pesan yang menunjukkan jumlah baris yang mempengaruhi SQL. Hal ini juga mengurangi lalu lintas jaringan, karena klien kami tidak akan menerima pesan apapun yang menunjukkan jumlah baris yang dipengaruhi oleh pernyataan (*statement*) SQL.
9. Cobalah untuk menggunakan kata kunci TOP atau pernyataan (*statement*) SET ROWCOUNT yang berfungsi sebagai jumlah baris *query* yang akan dieksekusi, jika kita perlu kembali hanya baris *n* pertama. Hal ini dapat meningkatkan kinerja dari pertanyaan (*statement*) kita, karna lebih kecil *result set*-nya *maka* akan dikembalikan. Hal ini juga dapat mengurangi lalu lintas antara server dan klien.
10. Cobalah untuk menggunakan fungsi *user-defined* untuk menjaga kode tetap terenkapsulasi untuk digunakan kembali di masa depan.
11. *User-defined Functions* (UDFs) mengandung satu atau lebih pernyataan SQL yang dapat digunakan untuk merangkum kode yang dapat digunakan kembali. Menggunakan UDFS juga dapat mengurangi lalu lintas jaringan.
12. Jika memungkinkan dapat mengganti logika UDFs ke SP (*Stored*

Procedure) juga.

13. Jika dperlukan untuk menghapus semua baris dari tabel, cobalah untuk menggunakan perintah TRUNCATE TABLE bukan perintah DELETE. Menggunakan TRUNCATE TABLE adalah cara yang lebih cepat untuk menghapus semua baris dari tabel, karena menghapus semua baris dari tabel tanpa login setiap baris hapus.
14. Menghapus yang fungsi JOIN tabel jika tidak diperlukan.
15. Jika ada *cursor* digunakan dalam *query*, maka perlu diperhatikan apakah ada cara lain untuk menghindari penggunaan ini (baik dengan cara SELECT ... INTO atau INSERT ... INTO, dll). Cobalah untuk menghindari menggunakan kursor bila memungkinkan. SQL Server yang menggunakan *cursor* dapat mengakibatkan beberapa penurunan kinerja dibandingkan dengan memilih pernyataan(*statement*). Cobalah untuk menggunakan korelasi *sub-query* atau tabel yang diturunkan untuk operasi baris-demi-baris pada tabel.
16. Saat menulis sub-query (SELECT dalam WHERE atau HAVING klausa SQL *Statement* lainnya):
    a.) Cobalah untuk menggunakan *correlated* (mengacu pada sedikitnya satu nilai dari permintaan dari luar) *sub-query* ketika pengembalian relatif kecil dan / atau kriteria lain yang efisien yaitu jika tabel dalam *sub-query* memiliki efisiensi *index*.
    b.) Cobalah untuk menggunakan *non-correlated* (tidak mengacu pada permintaan luar) *sub-query* ketika berhadapan dengan tabel besar yang diharapkan yaitu pengembalian besar (banyak baris) dan / atau

jika tabel dalam *sub-query* tidak memiliki efisiensi *index*.

c.) Pastikan bahwa beberapa *sub-query* berada di urutan yang paling efisien.

d.) Ingat bahwa menulis ulang *sub-query* sebagai *join* terkadang dapat meningkatkan efisiensi.

17. Gunakan untuk tipe data *char* atau *varchar*, bukan tipe data *nchar* / *nvarchar* jika kita tidak perlu menyimpan data *unicode*. Data nilai *char* atau *varchar* hanya menggunakan satu *byte* untuk menyimpan satu karakter; sedangkan nilai *nchar* atau *nvarchar* menggunakan dua *bytes* untuk menyimpan satu karakter, sehingga kolom *char* atau *varchar* menggunakan dua kali lebih sedikit ruang untuk menyimpan data dibandingkan dengan kolom data *nchar* atau *nvarchar*.
18. Cobalah untuk menggunakan *stored procedure* bukan *query* yang berbobot berat karena dapat mengurangi lalu lintas jaringan..
19. Cobalah untuk mengembalikan nilai *integer* dari RETURN *statement* bukannya mengembali nilai *integer* sebagai bagian dari satu set *record*. RETURN *statement* umumnya digunakan untuk pengecekan error, tetapi kita juga bisa menggunakan pernyataan (*statement*) ini untuk mengembalikan nilai *integer* untuk alasan lain. Menggunakan pernyataan RETURN dapat meningkatkan kinerja SQL Server dan tidak akan membuat set *record*.
20. Cobalah untuk mengapus *index* yang tidak terpakai. Karena setiap *index* memakan ruang *disk* dan memperlambat operasi DML (*Data Manipulation Language*), Hapus *index* yang tidak digunakan. Kita dapat

menggunakan *Index Wizard* untuk mengidentifikasi index yang tidak digunakan dalam *query* SQL.

21. Cobalah untuk membuat index pada kolom yang memiliki nilai *integer* daripada nilai-nilai bersifat karakter. Karena nilai *integer* memiliki ukuran kurang dari ukuran nilai karakter, sehingga kita dapat mengurangi jumlah halaman *index* yang digunakan untuk menyimpan kunci *index*. Hal ini akhirnya mengurangi jumlah *read* yang diperlukan untuk membaca *index* dan meningkatkan kinerja keseluruhan *index*.
22. Jika *join* diperlukan di beberapa *table* sangat sering digunakan, maka kita harus mempertimbangkan untuk membuat *index* pada kolom *join* yang secara signifikan dapat meningkatkan kinerja *query* terhadap *join* tabel.
23. Cobalah untuk menghindari operasi apapun pada *fields*. Beberapa operasi akan mencegah penggunaan index pada *field*.

2.6. Teknik Optimasi Query di Microsoft SQL Server

Di dalam jurnal yang dipublikasikan karya (Corlăţan et al., 2014) bahwa banyak problem dalam rendahnya kinerja sistem SQL Server. Dan setelah melakukan peningkatan sistem *hardware*, sistem operasi dan pengaturan SQL Server telah ditemukan faktor utama yang mempengaruhi proses eksekusi *query*, yaitu:

1. Tidak digunakannya *Indexing*.

   Faktor *indexing* dapat mempengaruhi kinerja server. Ketika hilangnya *indexing* pada tabel, sistem SQL Server harus melakukan pencarian

(*scanning*) satu persatu pada seluruh tabel untuk mendapatkan data yang diinginkan. Hal ini dapat menyebabkan *overloading* memori RAM dan CPU, sehingga dapat meningkatkan waktu lamanya proses eksekusi *query*.

2. Statistik yang kurang tepat.

   Sistem manajemen *database* SQL Server sebagian besar bergantung pada berdasarkan *cost based optimization*, sehingga statistik yang tepat sangat penting untuk efisiensi penggunaan indeks. Tanpa ini, sistem tidak dapat memperkirakan persis jumlah baris yang dipengaruhi oleh *query*. Kuantitas data yang akan diambil dari satu atau lebih tabel (dalam kasus *join*) cukup penting ketika memutuskan metode optimasi eksekusi *query*. *SQL Server query optimizer* biasanya berdasarkan pada biaya (*cost*), yang berarti bahwa ia memutuskan mekanisme yang terbaik terhadap pengaksesan data. Dengan menerapkan strategi identifikasi secara selektidf. Setiap statistik memiliki indeks yang terpasang (*attached*). Terkadang ada statistik dibuat secara manual, pada kolom yang bukan tidak memiliki indeks. Dengan menggunakan statistik, *optimizer* dapat membuat perkiraan yang cukup masuk akal mengenai waktu yang dibutuhkan untuk sistem untuk mendapatkan satu set hasil dari *query*.

3. *Query* yang ditulis sangat buruk.

   Efisiensi indeks bergantung kepada cara *query* yang ditulis. Mengambil jumlah baris yang sangat besar dari tabel dapat menyebabkan tidak efisiensinya indeks. Untuk meningkatkan kinerja, *query* SQL harus

ditulis dengan menggunakan indeks yang ada.

Disamping cara penulisan *query* yang benar. Terdapat beberapa penyebab lambatnya optimasi query, seperti metode pemblokiran untuk tabel, metode penggunaan indeks tertentu, garis pemblokiran tabel dan lain-lainnya. Berikut ada beberapa jenis yang dimaksud:

- NOLOCK – READUNCOMMITED : NOLOCK hanya dapat diterapkan bila data diblokir. Ketika Skema Stabilitas pada tabel diblokir maka dapat langsung menggunakan proses eksekusi *query* secara langsung dengan metode NOLOCK. Hal ini mengindikasikan tidak ada pemblokiran dan tidak menghentikan dalam mengakses data yang akan diambil. Termasuk ketika sedang berjalannya proses modifikasi data tersebut. Pada proses ini kemungkinan besar data yang diambil adalah data sebelum proses modifikasi sedang berlangsung.
- INDEX : Mendorong penggunaan indeks pada tabel.
- READPAST : Menunjukan kepada sistem untuk tidak membaca baris data yang diblokir oleh transaksi lainnya.
- ROWLOCK : Sebuah baris data yang diblokir.
- TABLOCK : Menunjukan bahwa pemblokiran berada pada tingkat tabel.
- HOLDLOCK : Metode ini setara dengan tingkat SERIALIZABLE
- TABLOCKX : Menunjukan pemblokiran tabel secara ekslusif.

4. *Blocking* yang berlebihan (*Deadlocks*)

SQL Server adalah sebuah perangkat lunak yang bersifat "**ACID**" yang

merupakan singkatan untuk *Atomicity, Consistensy, Isolation* dan *Durability*. Hal ini diyakini bahwa variasi yang dibuat oleh transaksi yang berkelanjutan perlu terisolasi dari transaksi lainnya. Sesuai dengan aturan dari transaksi adalah wajib untuk keamanan data.

- *Atomicity*: satu transaksi adalah suatu atom, jika sesuai dengan prinsip "semua atau tidak". Ketika transaksi berhasil semua maka variasinya menjadi permanen, ketika transaksi gagal, maka semua variasi dibatalkan.
- *Consistency*: satu transaksi menciptakan *state* baru dan validasi data. Jika kegagalan terjadi maka sistem mengembalikan semua data ke keadaan sebelum transaksi dimulai.
- *Isolation*: suatu transaksi akan terisolasi jika tidak menyangkut transaksi lain atau tidak memperhatikan transaksi lainnya pada data yang sama (level dari isolasi).
- *Durabillity*: suatu transaksi akan bertahan lama jika hal itu bisa dilakukan atau dapat terbalik meskipun sistem sedang *breakdown*. Operasi T-SQL tidak bergantung pada satu set hasil (*cursors*).

Karna sifat ACID tersebut jika banyaknya transaksi yang dilakukan pada sistem SQL Server maka akan terjadi blocking berlebihan atau disebut dengan *deadlocks*.

5. Operasi T-SQL yang tidak didasarkan pada satu set hasil (kursor)

   Mengingat bahwa Transact -SQL adalah bahasa berbasis set (*set based*), kita tahu bahwa itu pengoperasian terjadi pada set. Menulis kode

berbasis non-set dapat menyebabkan penggunaan kursor yang berlebih dan *loop* berkepanjangan. Untuk meningkatkan kinerja, dianjurkan untuk menggunakan *query* berdasarkan set, tidak menggunakan pendekatan baris per baris, sehingga dalam waktu akses data yang jauh lebih tinggi. Penggunaan yang berlebihan dari kursor menaikan tingkat stres pada SQL server dan mengakibatkan kinerja sistem berkurang.

6. Fragmentasi yang berlebihan terhadap *Index*.

   Biasanya, data diorganisir dengan cara yang teratur. Namun, dalam kasus di mana halaman berisi data yang terfragmentasi atau mengandung sejumlah kecil data karena selalu melakukan pembagian halaman (*page division*), jumlah operasi *read* yang diperlukan untuk mengembalikan data akan lebih tinggi dari biasanya. Meningkatnya jumlah *read* disebabkan karena adanya fragmentasi indeks.

7. Seringnya penggunaan kompilasi ulang *query*.

   Cara yang paling umum untuk memberikan rencana eksekusi yang dapat digunakan kembali (*reusable execution plan*). Tidak bergantung dari variabel yang digunakan dalam *query*, adalah dengan menggunakan *stored procedure* atau *parameterized query*. Dengan menciptakan *stored procedure* untuk mengeksekusi satu set *query* SQL, sistem *database* menciptakan rencana eksekusi parameter (*parameterized execution plan*) yang independen dari parameter selama eksekusi. *Execution plan* yang dihasilkan akan digunakan kembali hanya jika SQL Server tidak perlu mengkompilasi ulang pernyataan individu dari *stored procedure* yang setiap kali dijalankan (misalnya urutan dari SQL

yang dinamis). Seharusnya dengan membangun kembali eksekusi *query* menyebabkan meningkatnya nilai waktu pengeksekusian *query*.

2.7. Mengatur Performa Pada DBMS SQL Server

Dalam jurnal yang dipublikasikan di *International Journal of Computer Science and Mobile Computing Vol*. 4 *Issue*. 6, karya (Dahiya & Ahlawat, 2015) menjelaskan tentang variasi *bottleneck* yang berdampak negatif terhadap performa sistem.

Bottleneck terjadi ketika sumber daya mencapai kapasitasnya, menyebabkan performa seluruh sistem melambat. Sumber daya yang tidak mencukupi atau salah pada bagian konfigurasi sistem, terdapat beberapa komponen yang tidak berfungsi, dan permintaan sumber daya yang salah biasanya menyebabkan terjadinya *bottleneck* oleh suatu program. Ada lima area sumber utama yang dapat menyebabkan *bottleneck* dan mempengaruhi performa server, seperti: *Physical Disk*, *Memory*, *Process*, CPU, dan *Network*.

1. *Hard Disk Bottleneck*

   Logical Disk \% Free Space : Untuk mengukur persentase ruang kosong pada *logical disk drive* yang dipilih.

   Physical Disk \% Idle Time: Untuk mengukur persentase waktu *disk* saat *idle* selama interval sampel akan dijalankan.

   Physical Disk \ Avg. Disk Sec / Read : Untuk mengukur waktu rata-rata, dalam hitungan detik, untuk membaca data dari disk

   Memory \ Cache Bytes : Untuk menunjukkan jumlah memori yang digunakan untuk cache sistem file. Kemungkinan terjadinya *bottleneck*

pada disk jika nilai ini lebih besar dari 300MB.

2. *Memory Bottleneck*

   Memory\Available Mbytes : Untuk mengukur jumlah memori fisik, dalam ukuran *megabyte*, jumlah *memory* yang tersedia untuk menjalankan proses. Jika nilai ini kurang dari 5 persen dari total RAM fisik, maka memori tidak mencukupi dalam menjalankan proses, dan meningkatkan aktivitas *paging*. Untuk mengatasi masalah ini, Anda cukup menambahkan lebih banyak memori.

   Checkpoint Pages/sec : Digunakan secara otomatis oleh *database engine* sql server untuk *flush dirty pages* ke disk.

   High number of Lazy writes/sec : *Lazy Writer* digunakan ketika ada tekanan memori pada sql server.

   High number of Page reads/sec : Berfungsi untuk mengingatkan bila jumlah *disk* yang dibaca lebih banyak dari jumlah *cache* yang dibaca.

   Low Buffer cache hit ratio : Nilai yang konsisten di bawah 90% menunjukkan bahwa dibutuhkan lebih banyak memori fisik di server.

   Low Page Life Expectancy : Idealnya nilai ini tidak boleh kurang dari 300 detik.

3. CPU *Bottleneck*

   Penggunan CPU yang tinggi merupakan *bottleneck* yang cukup penting, yang berdampak buruk pada performa sistem. Jika SQL Server menggunakan CPU tinggi kita perlu menghentikan beberapa *batched* (Program yang tertumpuk) yang bertanggung jawab atas lambatnya performa CPU.

% *Processor Time* : *Processor time* memberikan nilai waktu penggunaan CPU dalam persentase

% *Privilege Time* : menunjukan waktu yang dibutuhkan prosesor untuk mengeksekusi panggilan dari sistem.

4. Deadlocks and Blocking

   Pada SQL Server, pemblokiran terjadi ketika satu SPID (*Server Process* ID) memegang kunci pada sumber daya tertentu dan upaya SPID kedua untuk memperoleh tipe kunci yang saling bertentangan pada sumber yang sama.

   Blocking (Pemblokiran) sangat menurunkan kinerja sistem. *Deadlocks* terjadi ketika dua atau lebih tugas saling memblok satu sama lain oleh setiap tugas yang memiliki kunci pada sumber daya, sementara tugas lainnya mencoba untuk mengunci.

Cara untuk mengatasi *bottlenecks* ialah sebagai berikut:

1. Menerapkan Indexing yang tepat pada kolom tabel yang ada di dalam *database*.

   Dengan adanya *Indexing* dapat memberikan pencapaian performa terbaik dengan nilai waktu yang tercepat dalam *production system*.

   Menggunakan *Indexes* di dalam *database* tidak membutuhkan modifikasi pada aplikasi juga tak perlu membangun (build) dan menyebar (deployment) ulang.

2. Optimasi Query

   SQL Server *optimizer* menggunakan statistik terkini untuk mengoptimalkan *query* dan memilih rencana eksekusi terbaik yang ada.

3. Menggunakan SQL Server *Performance Tools*

   SQL *Performance Tools* terdapat 3 bagian yaitu:

   a. SQL Server *Profiler* dan *Tuning Advisor*

      Microsoft SQL Server Profiler merupakan *Graphical User Interface* (GUI) untuk SQL *Trace* untuk memonitori *instance* dari *Database Engine* atau *Analysis Services*. Dapat menyimpan data setiap transaksi ke *file* atau tabel untuk dianalisis.

      Tuning Advisor membantu mendapatkan laporan performa yang dihasilkan oleh SQL *Profiler* dan memberikan *indexing* yang sesuai. Dibutuhkan satu atau lebih *statement* SQL sebagai masukan dan menggunakan *Automatic Tuning Optimizer* untuk melakukan SQL *Tuning* pada *statements*.

   b. SQL Query Analyzer

      Di dalam aplikasi SQL Server 2000 terdapat SQL *Query Analyzer* yang dapat membantu dalam menganalisa *query* dan *script* lainnya seperti *Stored Procedured*. Juga dapat men-*debug Stored Procedured*, *Query Performance Problems* dan *tools* lainya yang membantu dalam proses perbaikan *query*.

   c. SQL DMV

      SQL DMV singkatan dari SQL *Dynamic Management Views*. Dengan melakukan query satu DMV, sys.dm_os_performance_counters yang tepat. Dapat mengumpulkan informasi *counter*s yang akan diterima dari

PerfMon (*Performance Monitor*) dari variasi SQL Server *counters*. SQL DMV di dalamnya terdapat subjek yang luas yang ditutupi. Hal ini dapat digunakan untuk menangkap setiap penghitung (*counters*) performa apakah terkait dengan kinerja OS, fragmentasi indeks, penguncian, deadlock, tipe tunggu, sistem infi dan lain lainnya.

2.8. Metodologi Investigasi Novel untuk Mengidentifikasi Kinerja Query SQL Server

Berdasarkan jurnal publikasi *Indian Journal of Science and Technology Vol*. 8 *Issue*. 27, dibuat oleh (Murugesan, Karthikeyan, & Sivakumar, 2015) menerangkan bahwa suatu protokol dapat digunakan untuk menemukan dan memecahkan masalah kinerja SQL. Rincian dari *Specific Technique* dapat digunakan untuk menghilangkan penyebab masalah performa SQL dan menyelesaikan permasalahan tersebut.

End Users akan mengalami frustrasi ketika aplikasi memiliki isu performa yang lambat. Selama ini, tidak akan ada ide di mana untuk memulai mengidentifikasi *performance bottleneck* secara proaktif akan meningkatkan kinerja aplikasi dan menghemat banyak waktu dan biaya. Pendekatan baru yang tersedia dalam jurnal tersebut memberikan solusi untuk:

a. Setiap orang yang menghadapi masalah performa pada aplikasi yang ada.
b. Saat SQL *statements* membutuhkan waktu lama untuk dijalankan. Atau lebih buruk lagi, *query* dijalankan begitu lama (terkadang bisa mencapai berjam-jam).

c. Setiap orang yang berencana untuk meningkatkan bisnis mereka.

Untuk memungkinkan performa yang lebih baik, dibutuhkan konfigurasi sistem yang baik dan diperlukan *software database* yang benar. Konfigurasi sistem tergantung pada *throughput* sistem, stabilitas sistem dan kemampuan pengguna memelihara (*maintenance*) aplikasi.

Terdapat metodologi yang berbeda-beda dan pengukuran eksekusi *query* yang diukur dengan istilah diantaranya:

a. *Most Expensive Queries*

   Bagian ini memberikan informasi rinci tentang apa saja untuk semua *query* yang memerlukan lebih banyak waktu untuk mengkonsumsi CPU dan *query* apa saja paling mahal (berat/lebih lama proses eksekusinya) yang berjalan di dalam *database*. Utilisasi CPU telah dihitung berdasarkan penyelesaian *query* dalam hitungan detik. SQL *Statements* menyediakan semua informasi yang dibutuhkan untuk menemukan *query* paling mahal yang saat ini sedang dijalankan, atau baru-baru ini dieksekusi di mana rencana eksekusi *query* masih dalam tersimpan di dalam *cache*.

b. *Most Frequently Executed Queries*

   Bagian ini memberikan informasi rincian tentang *query* yang paling sering dieksekusi. Juga tidak menunjukkan *query* yang lambat dan akan memberikan informasi dari *statement* yang dieksekusi berkali-kali. Rencana eksekusi *query* sangat berguna untuk memahami karakteristik performa suatu *query*. Menghitung dengan cara yang paling efisien untuk menerapkan *request* yang ditunjukkan oleh T-SQL *Query*.

Rencana eksekusi menunjukan sebuah *query* yang akan dijalankan dan mengidentifikasi masalah yang ada di dalam kode SQL. Bila SQL *statement* membutuhkan waktu lama untuk menyelesaikannya, rencana eksekusi akan membantu dalam menentukan *tuning* yang tepat.

c. *Most I/Os per Executions*

Mengidentifikasi dari sebagian besar I / O *queries* juga merupakan salah satu faktor terpenting untuk mengetahui *performance bottleneck*. Performa aplikasi pada dasarnya dibatasi oleh *disk input* / *output* (I / O). Selain itu, aktivitas CPU harus ditangguhkan (*suspended*), sementara aktivitas I / O selesai. SQL *query* harus dirancang agar performanya tidak dibatasi oleh I / O. Dengan mencapai *database* I / O yang tinggi dan menemukan T-SQL *statements* secara spesifik dengan mengkonsumsi sejumlah besar I / O, masalah *query* tersebut dapat diidentifikasi dengan cepat. Dengan mengurangi I / O pada *query* dan *database* I / O yang besar, dengan ini dapat membantu untuk menyempurnakan server dari perspektif I / O.

d. *CPU and Memory Utilization and Availability*

Penggunaan CPU dan *memory* merupakan faktor terpenting dalam mengidentifkasi *performance bottleneck*. Setiap sistem memiliki kapasitas maksimum untuk memproses permintaan SQL *server* secara bersamaan. Ketika SQL *server* mengirimkan permintaan terus-menerus melebihi kapasitas maksimum server, server dapat menggunakan CPU 100% atau *memory* dan *virtual server memory* secara otomatis akan terisi. Kemudian, *server* harus me-restart SQL server-nya secara

berkala. Selama situasi kritis ini, penggunaan CPU dan *memory* harus dipantau secara berkala untuk mengidentifikasi *query* atau *store procedured* mana yang menyebabkan masalah ini.

2.9. Desain & Pengembangan *Advanced Database Management System* Menggunakan *Multiversion Concurrency Control Model* untuk Lingkungan *Multiprogramming*

Pada jurnal yang dipublikasikan oleh IOSR *Journal of Computer Engineering* Vol. 17 Issue. 2, karya (Hussain, Premchand, & Someswar, 2015). Menerangkan bahwa di dalam lingkungan *multiprogramming* dimana lebih dari satu transaksi terdapat proses eksekusi secara bersamaan (*concurrency*). Ada kebutuhan protokol untuk mengendalikan transaksi yang bersifat *concurrency* untuk memastikan *atomicity* dan terisolasi dari transaksi. *Concurrency control protocols* bertugas untuk memastikan transaksi *serializability* dianggap paling banyak digunakan. *Concurrency control protocols* dapat dibagi secara luas menjadi dua kategori, yaitu:

a. *Lock based protocols*

Sistem basis data telah dilengkapi dengan protokol *lock-based*, menggunakan mekanisme dimana transaksi tidak dapat membaca atau menulis data sampai memperoleh kunci yang tepat di dalamnya terlebih dahulu. Kunci terdiri dari dua jenis, yaitu:

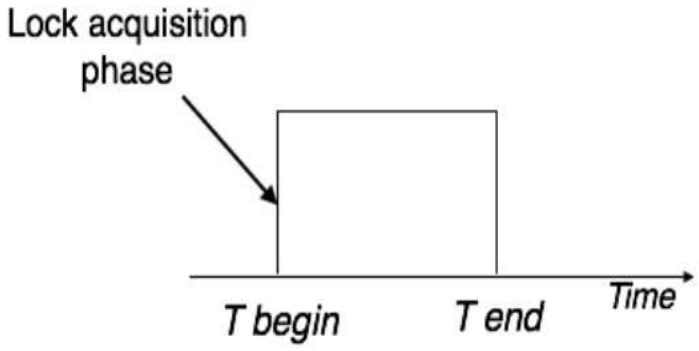
Binary Locks : Kunci pada item data bisa berada di antara dua *states*. Yaitu terkunci atau tidak terkunci.

Shared / exclusive : Jenis mekanisme penguncian ini membedakan kunci berdasarkan penggunaannya. Jika kunci diperoleh pada item data untuk melakukan operasi tulis, itu merupakam kunci eksklusif (*exclusive lock*). Karena memungkinkan lebih dari satu transaksi untuk menulis pada item data yang sama akan menyebabkan *database* menjadi tidak konsisten. *Reads Locks* akan terbagi (*shared*) karena tidak ada nilai data yang diubah.

Ada empat tipe *lock protocols* yang tersedia, yaitu:

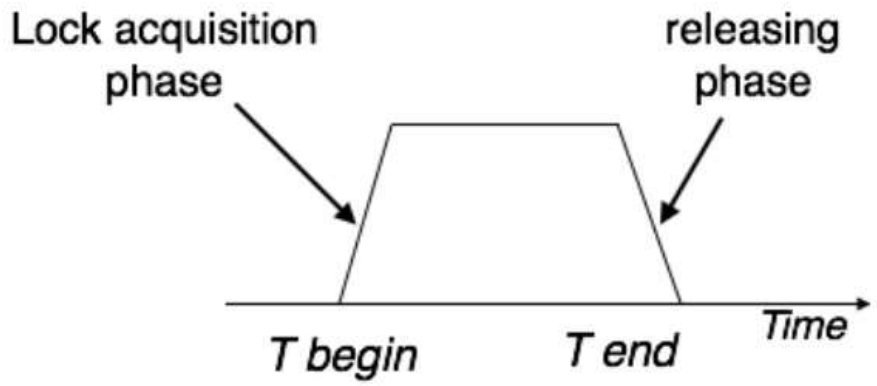
Simplistic : *Simplistic lock based protocols* memungkinkan transaksi untuk mendapatkan kunci pada setiap objek sebelum operasi "tulis" dilakukan. Begitu operasi "tulis" telah selesai, transaksi dapat membuka item data.

Pre-claiming : Pada protokol ini, transaksi mengevaluasi operasinya dan membuat daftar item data yang menjadi suatu objek yang membutuhkan *locks*. Sebelum memulai eksekusi, permintaan transaksi sistem untuk semua *locks* yang dibutuhkan sebelumnya. Jika semua *locks* diberikan, setiap transaksi akan mengeksekusi dan melepaskan semua *locks* saat semua operasinya telah selesai. Jika semua *locks* tidak diizinkan, transaksi akan kembali dan menunggu sampai semua *locks* dizinkan.



Gambar 2.12. – *Pre-claiming* (Hussain, Premchand, & Someswar, 2015, p.38)

***Two Phase Locking* - 2PL** : Protokol *locks* ini membagi fase atas eksekusi transaksi menjadi tiga bagian. Pada bagian pertama, kapan transaksi mulai dijalankan, transaksi mencari perizinan untuk *locks* yang dibutuhkannya saat akan dijalankan. Bagian kedua adalah tempat transaksi memperoleh semua *locks* dan tidak ada *locks* lain yang diperlukan. Transaksi akan terus menjalankan operasinya. Begitu transaksi melepaskan *locks* pertamanya, fase ketiga dimulai. Pada tahap ini transaksi tidak dapat menuntut *locks* apapun tetapi hanya melepaskan *locks* yang diperoleh.

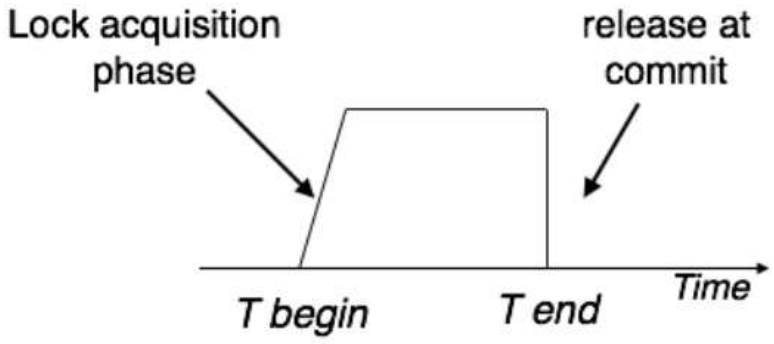


Gambar 2.13. – *Two Phase Locking* - 2TP (Hussain, Premchand, & Someswar, 2015, p. 38)

2TP memiliki dua fase. Fase pertama ialah tahap berkembang, dimana semua *locks* diperoleh dari transaksi. Fase yang kedua tahap penyusutan, di mana *locks* yang dipegang oleh transaksi akan dilepaskan. Untuk mendapatkan *exclusive locks* (*write*), transaksi harus terlebih dahulu memperoleh *shared locks* (*read*) dan kemudian dilanjutkan ke tingkat *exclusive locks*.

Strict Two Phase Locking : Tahap pertama dari *Strict*-2PL sama dengan 2PL sebelumnya. Setelah mendapatkan semua *locks* pada tahap pertama, transaksi terus berlanjut secara normal. Tapi berbeda dengan 2PL, *Strict*-2PL tidak melepaskan

locks secara langsung setelah tidak diperlukan lagi, namun dengan memegang semua *locks* hingga keadaan telah selesai. *Strict*-2PL melepaskan semua *locks* sekaligus pada titik *commit* pada transaksi.



Gambar 2.14. – *Strict Two Phase Locking – Strict* 2TP (Hussain, Premchand, & Someswar, 2015, p. 38)

b. *Time stamp based protocols*

Protokol *concurrency* yang paling sering digunakan adalah *time-stamp based protocol*. Protokol ini menggunakan waktu sistem atau *logical counter* untuk digunakan sebagai *time-stamp*. Protokol *lock-based* mengelola urutan antara pasangan yang bentrok di antara transaksi pada saat eksekusi sedangkan protokol berbasis *time-stamp* mulai bekerja segera setelah transaksi telah dibuat.

2.10. Database Optimizing Services

Pada jurnal yang dipublikasikan oleh *Database Systems Journal* Vol. 1 No. 2, karya (Ghencea & Gieger, 2010). Tujuan dari jurnal ini bermaksud untuk menyajikan gagasan representatif tentang proses dasar optimasi pada *database*, dengan menggunakan estimasi matematis terhadap biaya dalam berbagai jenis

query, tinjauan atas tingkat performa yang telah dicapai, dan pengaruh struktur akses fisik yang berbeda dalam sampel *query* tertentu. *Target group* harus terbiasa dengan SQL dan konsep dasar dalam *database* relasional. Dengan cara ini, strategi eksekusi untuk *query* yang kompleks dapat dilakukan, memungkinkan penggunaan pengetahuan dalam memperoleh informasi dengan biaya lebih rendah. *Database* dijalankan melalui serangkaian transformasi sampai kepada penggunaan akhir, dimulai dengan pemodelan data (*data modelling*), perancangan basis data (*database designing*) dan pengembangan (*development*), dan diakhiri dengan pemeliharaan (*maintenance*) dan optimasi (*optimization*).

1. ***Data Modelling***

   Data Model lebih difokuskan pada data yang dibutuhkan secara terorganisir dan kurangnya operasi yang akan dibuat pada data. Tahap pemodelan data melibatkan struktur, integritas, manipulasi dan *query*. Ada beberapa aset yang berhubungan dengan hal ini, seperti:

   1. Mendefinisikan cara data harus teratur atau terorganisir (*hierarchical network, relational* dan *object-focused*). Menyediakan definisi aturan yang membatasi contoh struktur yang diizinkan / diterima.

   2. Menawarkan protokol *update* data.

   3. Menawarkan metode untuk *query* data.

   Terdapat beberapa metode *data modelling* diantaranya:

   a. *Customized databases/ Database development*

      Database yang dibangun dan di kostumisasi (*customized*) untuk menjawab permintaan dari kostumer. Hal penting dari *custom database* adalah bersifat produk komersial dari pelayanan secara langsung hingga

memungkinkan mencapai target yang diinginkan oleh konsumen.

b. *Database Designing*

Struktur komunikasi data sederhana yang mudah dimengerti oleh *final user* adalah hasil pemodelan data yang sebenarnya. Jika *database* memiliki masalah seperti berikut: kerusakan, data tidak aman atau tidak akurat atau *database* telah terdegradasi dan kehilangan fleksibilitasnya, maka itulah saat untuk membuat *database* baru.

c. Data Mining

Data mining adalah ilmu penggalian informasi yang berguna dari setiap dataset dan *database* dengan skala yang lebih besar.

d. Database Migration

Database Migration mewakili transfer (atau migrasi) skema *database* dasar dan data ke dalam pengelolaan basis data, seperti Oracle, IBM DB2, MS-SQL Server, MySQL dll. Sistem migrasi basis data dapat memungkinkan keandalan dan integritas data.

e. Database Maintenance

Database Maintenance merupakan proses yang sangat penting dalam setiap organisasi. Setelah pengembangan *database* yang aman, proses selanjutnya yang sangat penting adalah pemeliharaan *database*, yang menawarkan *update, backup* dan keamanan yang tinggi.

2. *Database Optimization*

Pada setiap akses pada aplikasi telah mengalami jutaan transfer data, *database optimization* menjadi domain riset utama untuk institusi penelitian universitas, dan juga untuk organisasi perusahaan. Dari sudut pandang

pengembangan perangkat lunak perusahaan, *relational database* sering melayani aplikasi perangkat lunak pada domain tersebut, dan kurangnya peningkatan performa sebagai menopang biaya yang signifikan bagi pelanggan dan perusahaan.

Berikut metode yang ada di dalam *database optimization*:

a. *Indexing*

Salah satu cara agar database dapat dioptimalkan adalah dengan melakukan *indexing*. Hal ini dilakukan untuk meningkatkan kinerja *query*, yang terdapat variasi dari *database* ke *database* lainnya, namun secara umum semuanya menguntungkan dari indeks yang efisien. Indeks yang efisien memungkinkan *query* untuk menghindari pemindaian keseluruhan struktur tabel untuk mencari data satu persatu.

b. *Types of indexing*

Index terbagi menjadi dua kategori: *clustered* atau *nonclustered*. Perbedaan utama dari kedua kategori tersebut adalah bahwa *nonclustered index* tidak mempengaruhi pemesanan indeks yang terletak pada posisi yang sulit. Sementara *clustered index* dapat melakukannya. Karena *clustered index* mempengaruhi urutan pesanan secara fisik dari *disk*, dapat menjadi *indexed cluster* untuk setiap tabel. Pembatasan yang sama tidak dapat diterapkan pada *nonclustered index*, sehingga menciptakan ruang pada *disk* dapat diterapkan (meskipun tidak mewakili solusi yang terbaik).

c. *Cost estimating for optimized databases*

Cost estimating adalah proses penerapan ukuran eksekusi yang

konsisten dan signifikan dari biaya untuk *query* tertentu. Metrik yang berbeda dapat digunakan untuk menentukan estimasi biaya, namun metrik yang paling relevan dan paling umum adalah jumlah *block* yang mengakses *query carts*. Karena pada *disk, input* atau *output* mewakili operasi yang cukup memakan waktu. Oleh karena itu tujuannya adalah untuk meminimalkan jumlah akses *block*, tanpa mengorbankan fungsionalitas.

2.11. Prediksi Performa Aplikasi Web Dengan *Performance Modelling* Dengan Menggunakan Jmeter

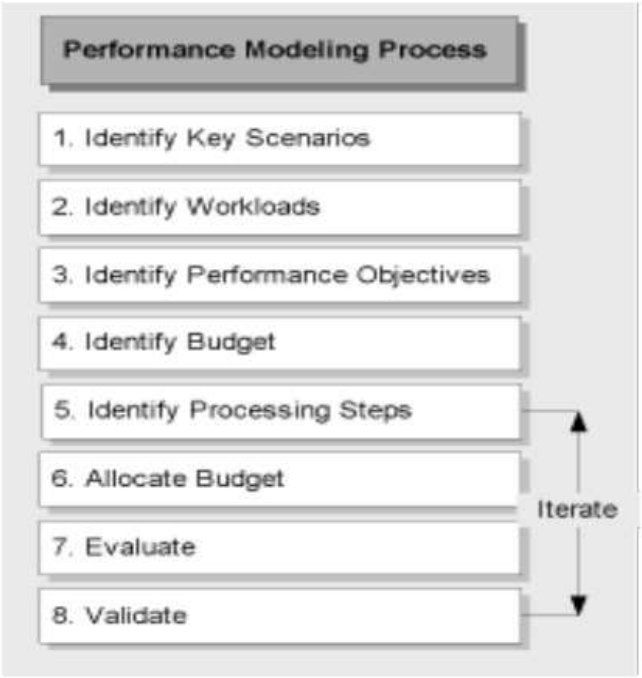
Menurur jurnal ilmiah karya (Patil, Joshi, & Dhotre, 2012) yang dipublikasikan di *International Journal of Scientific & Engineering Research, Vol*.3, *Issue*.12. Menjelaskan bahwa suatu proyek *software* mengikuti *Software Development Life Cycle* (SDLC) dalam proses pengembangannya. Seiring *user* menginginkan aplikasi *web* yang cepat dan responsif. Dalam pengembangan *software* diikuti oleh seluruh proses SDLC yang menjalankan proses secara sistematis yang terarah dengan baik dengan mengggunakan *Software Performace Engineering* (SPE). Bagian proses yang berjalan pada SPE seperti pengukuran (*measurement*) dan pemodelan performa (*performance modelling*) yang digunakan untuk mempelajari suatu sistem. Untuk menilai secara langsung apakah sebuah aplikasi akan memenuhi tujuan performa yang dipersyaratkan berdasarkan sumber daya yang ada, untuk keperluan perencanaan kapasitas (*capacity planning*).

Dengan menggunakan *performance modelling*, penggunaan dalam perencanaan kapasitas dengan cara memprediksi performa sistem dan mengamati

jika ada terjadinya sistem *bottleneck*. Penggunaan pemodelan lainnya meliputi penyediaan kapasitas yaitu mengalokasikan dan menyiapkan sumber daya untuk menangani tuntutan dan menemukan parameter konfigurasi aplikasi yang memenuhi tujuan yang diinginkan.

Performance modelling memberikan jalan untuk menemukan apa yang tidak kita ketahui. Manfaat *Performance modelling* meliputi:

- Dapat membuat titik pencapaian performa di setiap tahap proses *development*.
- Dapat mengevaluasi *tradeoffs* yang dibuat sebelumnya dalam siklus hidup berdasarkan pengukuran.
- Pengujian kasus menunjukkan arah pencapaian, apakah dapat mendekati atau menjauh dari sasaran performa aplikasi selama siklus hidup aplikasi yang berjalan.



Gambar 2.15. – *Performance Modelling Process* (Patil, Joshi, & Dhotre, 2012, p. 3)

Dari gambar 2.15 terdapat langkah prose dalam penggunaan *Performance Modelling*, yaitu sebagai berikut:

1. *Identify Key Scenarios* : Mengidentifikasikan kunci skenario dalam penentuan performa yang diinginkan
2. *Identify Workloads* : Mengidentifikasikan beban kerja yang akan diuji pada aplikasi.
3. *Identify Performance Objectives*: Mengidentifikasikan target performa yang diinginkan sesuai dengan keinginan *user*.
4. *Identify Budget* : Mengidentifikasikan *budget* yang akan digunakan dalam proses peningkatan performa aplikasi.
5. *Identify Processing Steps* : Mengidentifikasikan langkah-langkah proses dalam pencapaian performa pada aplikasi.
6. *Allocate Budget* : Mengalokasikan *budget* yang telah dibuat.
7. *Evaluate* : Mengevaluasi hasil performa aplikasi.
8. *Validate* : Memvalidasi performa ketika sudah digunakan oleh *user*.

Pengujian performa atau kinerja mempunyai data yang berbentuk angka dan metrik. JMeter adalah aplikasi gratis *open source*, alat ukur performa yang ditulis menggunakan Java. *Thread Group* adalah elemen dasar dari rencana pengujian JMeter. Setiap *thread* mewakili beberapa *user* yang dibuat. *Sampler* melakukan pengetesan dan berinteraksi ketika *server loading*. Untuk pengujian web, pengujian menggunakan *Sampler* 'HTTP *Request*' yang ada pada aplikasi Apache JMeter.

Informasi yang didapatkan dari beberapa sampel akan digunakan oleh *listener* dari JMeter. *Listener* yang biasa digunakan adalah: *Graph Results,View*

Results Tree,Simple Data Writer. *Logical Controllers* menentukan urutan Sampel yang diproses dari skrip yang lebih kompleks maka digunakan *Loops, Interleave, Random, If*. *Configuration Elements* menetapkan nilai kesalahan untuk bagian lain dari *Test Plan* dan juga variabel dari konfigurasi:

- Dengan CSV *Data Set Config* dapat melakukan *Data-Driven Testing* (DDT) dalam JMeter.
- HTTP *Cookie Manager* secara otomatis akan mencegat dan mengirim *cookies* sesuai dengan permintaan.
- HTTP *Requests Default* dapat menghemat waktu ketika melakukan banyak HTTP *Sampler*.

Dengan penggunaan Apache JMeter, didapatkan nilai yang berbeda seperti *Throughput* (*responses* / *minute*), *Average* (ms), dan *Median* (ms). Apache JMeter merupakan alat yang cukup fleksibel. Tidak hanya memungkinkan Anda untuk menguji HTTP *server* namun juga untuk menguji beban penggunaan *Web Services*. *Web Services* dan SOAP *Sampler* adalah fitur terbaru dari JMeter. Dengan menggunakan fungsionalitas Apache JMeter yang ada dan antarmuka pengguna yang disediakan, pengguna dapat mensimulasikan sebuah beban dari 5 *threads* secara serentak yang mencapai penggunaan server dengan penundaan 10 dan 5 ms.

2.12. Studi Banding Pengujian Performa dengan JMeter

Berdasarkan jurnal yang dipublikasikan di *International Journal of Advanced Research in Computer and Communication Engineering Vol*. 5 *Issue*. 2. Dari hasil karya yang dibuat oleh (Niranjanamurthy et al., 2016) telah membuat

studi banding atas dua aplikasi untuk pengujian performa untuk mengukur seberapa besar kinerja aplikasi yang dapat dibebani oleh *user* yang mengakses secara simultan dan bersamaan pada aplikasi tersebut.

Pada jurnal tersebut membahas perbandingan aplikasi antara LOAD RUNNER dengan JMETER. Berikut dibahas perbandingan antara keduanya:

Tabel 2.2. – Tabel Perbandingan antara aplikasi JMeter dengan Load Runner

ITEM	**LOAD RUNNER**	**JMETER**
Load Generator tak terbatas	Tidak	Ya
Mudah untuk dipasang (*installation*)	Tidak	Ya
Performa *download* yang tinggi	Ya	Tidak
Memberikan laporan hasil pengujian	Ya	Tidak
Membutuhkan biaya	Tidak	Ya
Tingkat Level Teknis	Ya	
Stabilitas aplikasi	Netral	Tidak

Perbandingan antara JMeter dengan Load Runner dari segi kelebihan dan kekurangannya:

Kelebihan JMeter :

a. *Tools* yang cukup ringan dan mudak diaplikasikan.
b. Sangat sederhana dalam proses pemuatan 50-100 sampel dan bisa dilakukan pemantauan.

c. Tidak ada biaya untuk pembelian lisensi karna bersifat *open source*.
d. Cukup mudah dalam penambahan *plugins* untuk mendapatkan variasi hasil laporan yang diinginkan.
e. Tidak ada infrastruktur untuk menginstall aplikasi ini.
f. Baik jika digunakan selama pengujian unit maka gangguan pada performa aplikasi yang diuji dapat diidentifikasi pada tahap awal.

Kekurangan JMeter:

a. Hanya bisa dilakukan pengetesan untuk aplikasi *web*.
b. Pengujian tingkat stres (*stress test*) terkadang memberikan laporan performa yang ambigu ketika JMeter menggunakan sumber daya yang cukup banyak. *Distributed testing* diutamakan untuk mencegah hal itu terjadi.
c. Di satu sisi harus mengikuti cara praktis untuk mempelajari aplikasi JMeter. Dikarenakan masih digunakannya *command line execution*, dan minimum laporan yang dihasilkan.
d. Jika skenario yang dilakukan cukup rumit. JMeter akan kesulitan dalam proses pengujian aplikasi *web*.
e. Seseorang harus memahami secara teknis aplikasi web sebelum menggunakan Jmeter seperti mekanisme *client-server*.
f. Prosedur *recording* cukup rumit karena melibatkan untuk memulai *server proxy*, mengubah pengaturan *browser*, menambahkan think time secara manual, dan lain-lainnya.

Kelebihan Load Runner:

a. Tidak dibutuhkan proses instalasi pada *server* ketika proses pengujian.

b. Mengunakan ANSI C sebagai bahasa pemrogramannya sedangkan aplikasi lainnya menggunakan aplikasi seperti Java dan Visual Basic.
c. Pemantauan yang cukup sempurna. Antar muka analisinya terdapat laporan yang mudah dimengerti dengan *charts* dan *graphics* yang berwarna.
d. Mendukung banyak protokol yang cukup banyak.
e. Dapat dibuat korelasi yang cukup mudah
f. *Graphical User Interface* (GUI) yang cukup bagus untuk *generated scripts* dengan sekali klik dalam proses *recording*.
g. Tutorial yang diberikan mudah dipahami, dokumentasi yang lengkap dan tersedia *active tool support* yang diberikan HP.

Kekurangan Load Runner:

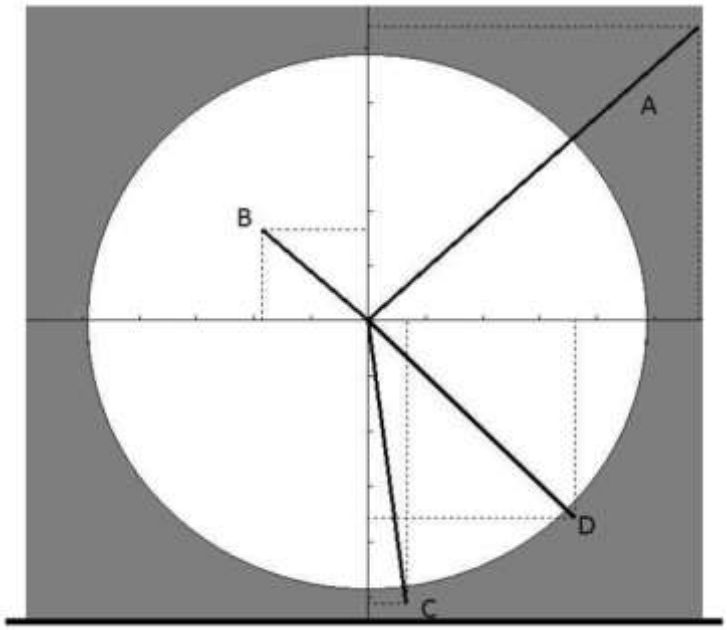
a. Bahasa Pemrograman / *Scripting* digunakan untuk mewakili data protokol dan memanipulasi data untuk diputar ulang.
b. Menggunakan protokol dalam proses komunikasi antara klien dengan sistem.
c. Korelasi adalah cara untuk mengganti nilai dalam data dinamis sehingga memungkinkan proses *playback* berhasil.

2.13. Keutamaan dalam *Multivariate Analysis of Variance* (MANOVA) untuk para peneliti (*Scientists*)

Berdasarkan jurnal publikasi *Practical Assessment, Research & Evaluation Vol.* 19 *Issue*. 17. Dijelaskan oleh (Warne, 2014) tentang keutamaan *Multivariate*

Analysis of Variance (MANOVA) yang digunakan oleh para peneliti dalam membuat keputusan dari hasil statistik yang telah dibuat.

MANOVA merupakan dari ANOVA (*Analisys of Variance*) secara matematis diperluas yang dapat diterapkan ke dalam situasi dimana terdapat dua atau lebih variabel dependen.



Gambar 2.16. – Contoh logika analisis pengujian MANOVA untuk variabel yang berkorelasi sempurna (**Warne, 2014, p. 2)**

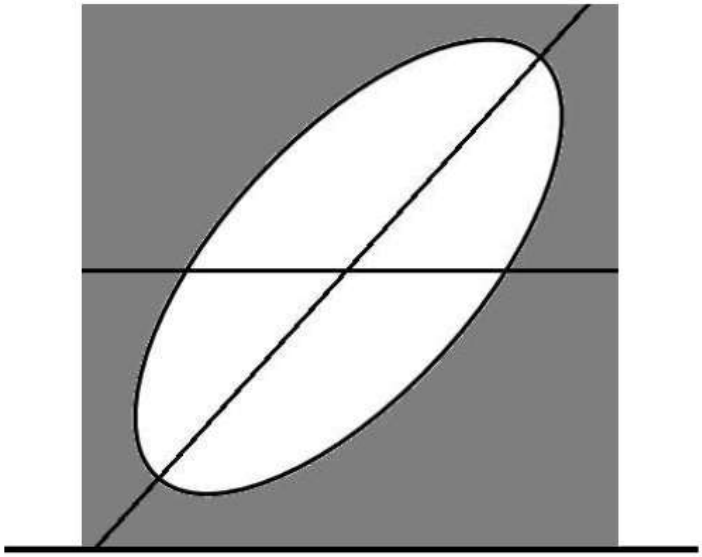
Dari gambar 2.16 diatas daerah yang diarsir adalah daerah penolakan (*rejected*). Jika vektor untuk pengujian berakhir di daerah penolakan, maka hipotesis nol ditolak. Jika garis putus-putus yang berasal dari ujung vektor memenuhi sumbu di daerah yang diarsir, maka ANOVA untuk variabel dependen sumbu tersebut juga akan ditolak.

Mengingat sifat MANOVA yang lebih rumit, beberapa peneliti mungkin mempertanyakan apakah itu layak ditambah kompleksitasnya. Alternatif penggunaan MANOVA adalah melakukan ANOVA untuk setiap variabel

dependen. Namun, pendekatan ini tidak menguntungkan karena (a) melakukan beberapa ANOVA meningkatkan kemungkinan melakukan kesalahan *Type* I, dan (b) beberapa ANOVA tidak dapat menentukan apakah variabel independen terkait dengan kombinasi dari variabel dependen, seringkali lebih banyak informasi yang berguna bagi ilmuwan atau peneliti yang mempelajari variabel dependen yang berkorelasi.

2.13.1. Menghindari inflasi dari *Type* I *error*

Dengan adanya beberapa variabel dependen, penggunaan MANOVA cukup bermanfaat karena mengurangi kemungkinan kesalahan *Type* I. Probabilitas kesalahan *Type* I setidaknya satu kali dalam rangkaian ANOVA (disebut *experiment-wise error*) dengan nilai sekitar $1 - (1 - 0{,}05)^{k}$, di mana k adalah jumlah ANOVA yang dilakukan. Oleh karena itu, jika seorang peneliti memilih nilai α tradisional 0,05 untuk dua ANOVA, maka *experiment-wise error Type* I dapat setinggi .0975 - tidak mencapai .05. Meskipun α untuk masing-masing ANOVA adalah 0,05. Namun, kesalahan *Type* I untuk dua variabel dependen MANOVA yang sama hanya 0,05. Tingkat parahnya inflasi *Type* I pada beberapa ANOVA bergantung pada seberapa korelasi ada variabel dependen satu dengan yang lain, dengan inflasi *Type* I terparah bila variabel dependen tidak berkorelasi. Dengan mengenali inflasi *Type* I yang terjadi dengan beberapa ANOVA karena inflasi jenis eksperimen *Type* I juga terjadi saat melakukan uji *multiple t-test*. Sebenarnya, salah satu alasan utama untuk melakukan ANOVA adalah dengan menghindari melakukan banyak *t-test*, yang menambah kesalahan *Type* I yang terjadi saat peneliti melakukan banyak pengujian *t-test*.



Gambar 2.17. – Contoh logika analisis pengujian MANOVA untuk variabel berkorelasi secara *moderate* (Warne, 2014, p. 2)

Pada gambar 2.16 diatas daerah yang diarsir merupakan are penolakan.

2.13.2. Post Hoc Procedures

Prosedur *post hoc* sering diperlukan setelah hipotesis nol ditolak dalam ANOVA atau MANOVA. Ini karena hipotesis nol untuk prosedur ini seringkali tidak memberi para peneliti semua informasi yang mereka inginkan . Misalnya, jika hipotesis nol ditolak dalam ANOVA dengan tiga atau lebih kelompok, maka peneliti mengetahui bahwa setidaknya satu kelompok rata-rata secara statistik berbeda dari setidaknya satu kelompok lainnya. Namun, kebanyakan peneliti akan tertarik untuk belajar yang mana *mean* yang berbeda dari mean kelompok lainnya. Banyak pengujian ANOVA post hoc telah ditemukan untuk memberikan informasi ini, walaupun *Tukey's Test* sejauh ini adalah tes yang paling umum yang dilaporkan dalam literatur psikologis. Sama seperti ANOVA, MANOVA memiliki prosedur *post hoc* untuk menentukan mengapa hipotesis nol ditolak. Bagi MANOVA kasus

ini biasanya berhubungan dengan DDA (*Descriptive Discriminant Analysis*), merupakan prosedur statistik yang menciptakan satu rangkaian persamaan linier korelasi yang sempurna bersama-sama memodelkan perbedaan antar kelompok di MANOVA. Manfaat dari persamaan yang tidak berkorelasi dengan DDA adalah bahwa setiap fungsi akan memberikan informasi unik tentang perbedaan antar kelompok, dan informasinya dapat dikombinasikan dengan cara aditif untuk interpretasi yang mudah.

2.13.3. Melakukan Penghitungan MANOVA

Contoh sampel data dalam penelitian ini diambil dari *National Longitudinal Study of Adolescent Health* (*Add Health*), sebuah studi longitudinal perkembangan remaja dan dewasa yang dimulai pada tahun ajaran 1994-1995 ketika para pesertanya berada di kelas 7 hingga kelas 12. Data sampel ini diambil dari publik menggunakan *file* dari variabel yang dikumpulkan selama periode pengumpulan data awal dan diunduh dari situs *website Inter-University Consortium for Political and Social Research*. Untuk contoh ini, satu-satunya variabel independen adalah kelas, dan dua variabel dependen tersebut adalah tanggapan peserta terhadap pertanyaan:

a. Pada bulan terakhir, seberapa sering Anda merasa tertekan?
b. Pada bulan lalu, Seberapa sering Anda mengalami kesulitan untuk rileks?

Dari kedua tanggapan diatas kemudian dicatat pada skala *Likert* 5 poin (0 = Tidak pernah, 1 = Jarang, 2 = Kadang-kadang, 3 = Sering, dan 4 = Setiap hari). Untuk kenyamanan, variabel-variabel ini akan disebut "depresi" dan "kecemasan". Dari penelitian sebelumnya telah menemukan bahwa kecemasan dan gangguan

depresi seringkali komorbid. Komorbid adalah berkenaan dengan suatu penyakit atau proses patologi lain yang berlangsung secara bersamaan.

Tabel 2.3. – Statistika deskriptif dari variabel yang digunakan untuk melakukan MANOVA

Grade	n	Depression		Anxiety	
		M	SD	M	SD
7	662	0.88	1.114	0.75	1.048
8	664	1.08	1.188	0.80	1.058
9	778	1.17	1.188	0.93	1.080
10	817	1.27	1.232	0.96	1.109
11	790	1.37	1.204	1.12	1.156
12	673	1.34	1.140	1.10	1.105

Pada tabel 2.3 menunjukkan nilai statistika deskriptif untuk variabel dan sampel data bersumber dari studi *Add Health*. MANOVA dilakukan untuk mengetahui hubungan antara *grade* dan gabungan variabel depresi (*Depression*) dan kecemasan (*Anxiety*). Hasil dari SPSS 19.0 ditunjukkan pada tabel 2.4, yang termodifikasi sedikit agar lebih mudah dibaca.

Tabel 2.4. – Contoh hasil MANOVA dari *Output* SPSS

Effect	Test Statistic	Value	F	df	Sig. (*p*)	Partial η^2	Noncentrality Parameter
Intercept	Pillai's Trace	0.533	2470.323	2, 4337	< .001	.533	4940.647
	Wilks' Lambda	0.467	2470.323	2, 4337	< .001	.533	4940.647
	Hotelling's Trace	1.139	2470.323	2, 4337	< .001	.533	4940.647
	Roy's Largest Root	1.139	2470.323	2, 4337	< .001	.533	4940.647
Grade	Pillai's Trace	0.022	9.834	10, 8676	< .001	.011	98.340
	Wilks' Lambda	0.978	9.873	10, 8674	< .001	.011	98.726
	Hotelling's Trace	0.023	9.911	10, 8672	< .001	.011	99.112
	Roy's Largest Root	0.021	18.388	5, 4338	< .001	.021	91.939

Pada tabel 2.4 berisi informasi catatan tambahan. Pertama, ukuran efek (yaitu, nilai parsial η2) semuanya dari nilai 0,011 dan 0,021 yang menunjukkan

bahwa kelas tidak memiliki hubungan statistik yang sangat kuat dengan variabel dependen. Kedua, parameter *noncentrality* untuk setiap uji statistik ditunjukkan pada tabel 2.4. Parameter *noncentrality* ini adalah parameter statistik yang diperlukan untuk memperkirakan distribusi *noncentral* yang memodelkan distribusi dari uji statistik jika hipotesis nol tidak benar. Bila parameter *noncentrality* sama dengan nol, hipotesis nol sangat sesuai dengan data dan oleh karena itu tidak boleh ditolak. Oleh karena itu, dapat masuk akal bahwa keempat uji statistik memiliki nilai parameter *noncentrality* yang tinggi (antara 91.939 dan 99.112) pada tabel 2.4. dikarenakan oleh semua uji statistik menunjukkan bahwa hipotesis nol harus ditolak, dan masing-masing memiliki *p-values* yang sangat kecil ($p < .001$). Pada akhirnya, kolom yang diberi label "Observed Power" memberi kekuatan statistik posterior dari desain MANOVA untuk setiap statistik uji. Angka di kolom ini akan selalu berbeda berbanding terbalik dengan *p-values* yang sesuai, sehingga tidak memberikan informasi selain yang diberikan oleh *p-values* di tabel tersebut.

2.13.4. *Post Hoc* DDA

Dengan hasil MANOVA pada Tabel 2.4 menunjukkan bahwa hipotesis nol harus ditolak, perlu dilakukan post hoc DDA untuk menentukan fungsi matematis yang membedakan kelompok kelas satu dengan yang lain pada nilai variabel dependen. Jumlah fungsi yang dibuat akan bervariasi, namun nilai minimum akan selalu menjadi jumlah variabel dependen atau jumlah kelompok dikurangi 1 (nilai mana saja yang lebih kecil). Karena itu, dalam contoh kasus *Add Health* akan ada dua fungsi diskriminan karena ada dua variabel dependen.

Dalam SPSS, analisis diskriminan deskriptif dapat dilakukan melalui alat "diskriminan". Program komputer menghasilkan sebuah tabel yang diberi label "Wilks 'Lambda," dan menampilkan hasil uji yang signifikan untuk dua fungsi diskriminan. Dalam contoh ini fungsi diskriminan pertama secara statistik signifikan ($p < .001$), sedangkan yang kedua tidak ($p = .125$), walaupun ukuran sampelnya besar ($n = 4,384$). Oleh karena itu, tidak perlu menafsirkan fungsi diskriminan kedua.

SPSS menampilkan sejumlah tabel lain yang membantu dalam menafsirkan fungsi diskriminan, dua yang paling penting di antaranya menunjukkan *Standardized Discriminant Coefficients* (SDCs) dan struktur untuk fungsi diskriminan. SDC dapat digunakan untuk menghitung skor masing-masing subjek untuk setiap fungsi diskriminan. Nilai fungsi untuk fungsi pertama dapat dihitung sebagai:

DDA Score1 = .700Depression + .431Anxiety

Nilai SDC ini ditafsirkan dengan cara yang sama seperti koefisien regresi standar dalam regresi berganda. Oleh karena itu, nilai .700 dalam persamaan berarti bahwa untuk setiap 1 peningkatan standar deviasi dalam nilai skor depresi peserta ujian skor DDA mereka diprediksi meningkat dengan nilai standar deviasi .700 jika semua variabel lainnya dipertahankan konstan. Secara substantif, karena fungsi diskriminan memaksimalkan perbedaan di antara kelompok kelas, dapat dilihat bahwa walaupun kedua depresi dan kecemasan berkontribusi terhadap perbedaan kelompok, depresi menunjukkan variasi antar kelompok. Hal ini didukung oleh data pada tabel 2.3, yang menunjukkan bahwa dari kelas 7 sampai 11 depresi dan skor kecemasan meningkat dengan mantap, walaupun perubahan skor depresi lebih

dramatis. Hasil ini juga mendukung penelitian yang menunjukkan bahwa baik gangguan kecemasan maupun depresi sering dimulai pada masa remaja.

Tabel 2.5. – Contoh *Standardized Discriminant Coefficients and Structure Coefficients*

Standardized Discriminant Coefficients[a]		
	Function 1	Function 2
Depression	.700	-.958
Anxiety	.431	1.105
Structure Coefficients[b]		
	Function 1	Function 2
Depression	.932	-.364
Anxiety	.808	.590
Parallel Discriminant Ratio Coefficients		
	Function 1	Function 2
Depression	(.700)(.932) = .652	(-.958)(-.364) = .349
Anxiety	(.431)(.808) = .348	(1.105)(.590) = .652

Namun, penting untuk diketahui bahwa SDC tidak dapat menunjukkan variabel mana yang lebih "penting" untuk membedakan kelompok. Cara termudah seseorang dapat menentukan kepentingan variabel adalah dengan menghitung paralel *Discriminant Ratio Coefficient* (DRC).

Untuk menghitung DRC paralel, kita harus mengalikan setiap SDC dengan koefisien struktur yang sesuai, seperti yang ditunjukkan di bagian bawah Tabel 2.4. Untuk fungsi pertama DRC paralel adalah 0,652 untuk depresi dan 0,348 untuk setiap tingkat kecemasan yang mengindikasikan bahwa tingkat depresi merupakan

variabel yang lebih penting dari keduanya dalam membedakan antara kelompok variabel independen dalam contoh ini.

2.14. Literatur Reviewer

Dalam jurnal yang dibuat oleh (Shao et al., 2015). HQN adalah sistem hirarkis yang mencakup server dan layanan. Dapat digunakan untuk penilaian kinerja (*performance assesment*) untuk sistem yang mengandung sumber daya perangkat lunak dan sumber daya perangkat keras. Kedalaman pemanggilan (*query*) tergantung pada jumlah lapisan panggilan yang berlangsung. HQN dapat ditentukan terutama oleh panggilan hubungan antara sumber (data).

Pendistribusian sistem *database* SQL Server memiliki server bersifat *multi-layered*. Proses perangkat lunak yang berjalan pada sumber daya perangkat keras yang dipakai secara bersamaan dapat menyebabkan penundaan karena banyaknya transaksi yang masuk. Sehingga berkaitan dengan nilai dari *thread process*-nya, maka jumlah proses *instance* dan proses alokasi untuk prosesor. Algoritma HQN berkaitan dengan pendistribusian antrian transaksi di dalam sistem *database* SQL Server yang dapat memprediksikan kinerjanya, Berdasarkan hasil yang diprediksi, tuning kinerja dan optimasi dapat diimplementasikan. Hasil eksperimen pada *benchmark* TPC-C menggunakan HQN menunjukkan bahwa optimasi kinerja SQL server dapat mencapai peningkatan kinerja 40% lebih daripada rata-rata dari algoritma klasik.

Berdasarkan jurnal karya (Khan & Khan, 2013). Berisikan beberapa literatur yang didapat dan menghasilkan suatu gambaran yang jelas untuk mendapatkan solusi dari permasalahan Optimasi *Query*.

Dalam jurnal publikasi karya (Kashem et al., 2010), menjelaskan tentang optimasi *query* dengan pendekatan teknik *Top-K Query Proccess*.

Konvensional *Top K Query Processing* adalah Pengolahan key teratas menghubungkan terlalu banyak riset *database* termasuk optimasi query, metode pengindeksan dan permintaan bahasa. Sebagai konsekuensi, dampak yang efisiensi *top-k* yaitu pengolahan menjadi jelas dalam peningkatan jumlah aplikasi.

Optimasi *query* adalah untuk memilih strategi eksekusi yang efisien. Proses optimasi *query* untuk mendapatkan efisiensi *query* yang behubungan dengan faktor *N* sehingga menghasilkan rencana yang terbaik untuk *Quer*y *Execution Plan* (QEP).

Optimasi Query dilakukan dengan menggunakan pemangkasan arsitektur mengurangi waktu proses *query* drastis dan sehingga mengurangi biaya. Waktu eksekusi yang berbeda untuk nilai *K* yang berbeda perbandingan biaya waktu untuk tradisional dan sistem yang disarankan. Sehingga pendekatan ini dapat digunakan untuk memilih strategi eksekusi *query* yang dapat meningkatkan performa *query*.

Menurut jurnal yang dibuat oleh (Kumari, 2012) didapatkan informasi untuk meningkatkan performa kinerja *query* SQL Server. Terdapat tips didalamnya seperti pemilihan *clustered* index atau *non clustered index*, pemilihan penggunaan *triggers*, *constraints*, *DISTINCT*, *stored procedure*, *User-defined Functions* (UDFs), *join, views*.

Pada jurnal yang dipublikasikan karya (Corlăţan et al., 2014) bahwa banyak problem dalam performa *query di* dalam SQL Server. Dan setelah meningkatkan performa *hardware*, sistem operasi dan konfigurasi SQL server ditemukan faktor utama yang mempengaruhi, seperti tidak digunakannya *Indexing*, statistik yang kurang tepat, *query* yang ditulis sangat buruk, operasi T-SQL tidak bergantung pada

satu set hasil (*cursors*), fragmentasi yang berlebihan terhadap *Index*, Seringnya pengunaan kompilasi ulang *query*.

Pada jurnal yang dibuat oleh (Dahiya & Ahlawat, 2015) membahas tentang pengaturan performa pada DBMS SQL Server. Dengan mengidentifikasikan faktor terjadinya *bottelneck* yang disebabkan oleh beberapa faktor yang mempengaruhi performa *server* diantaranya : *Physical Disk*, *Memory*, *Process*, CPU, dan *Network*.

Pada jurnal yang dibuat oleh (Murugesan et al., 2015) menerangkan bahwa terdapat metodologi yang berbeda untuk mengidentifikasikan masalah performa eksekusi *query*, yaitu: *Most Expensive Queries, Most Frequently Executed Queries* , *Most I/Os per Executions* dan *CPU and Memory Utilization and Availability*.

Menurut (Hussain et al., 2015) dalam menangani konkurensi (*concurrency*) dapat dilakukan dengan *Concurrency control protocols* yang terbagi luas, diantaranya : *Locks Based Protocols* dan *Time Stamp Based Protocols*.

Dalam jurnal yang dibuat oleh (Ghencea & Gieger, 2010) telah menyajikan gagasan representatif tentang proses dasar optimasi pada *database*. Dengan dijalankannya serangkaian transformasi awal sampai kepada penggunaan akhir, dimulai dengan pemodelan data (*data modelling*), perancangan basis data (*database designing*) dan pengembangan (*development*), dan diakhiri dengan pemeliharaan (*maintenance*) dan optimasi (*optimization*).

Menurur jurnal ilmiah karya (Patil et al., 2012) menjelaskan bahwa dengan *performance modelling* penggunaan dalam perencanaan kapasitas dengan cara memprediksi performa sistem dan mengamati jika ada terjadinya sistem *bottleneck*. Dan juga didalamnya terdapat langkah-langkah dalam proses *performance modelling* yaitu : *Identify Key Scenarios*, *Identify Workloads*, *Identify Performance*

Objectives, Identify Budget, Identify Processing Steps, Allocate Budget, Evaluate dan *Validate*.

Pengujian performa atau kinerja mempunyai data yang berbentuk angka dan metrik. *Thread Group* adalah elemen dasar dari rencana pengujian JMeter. Setiap *thread* mewakili beberapa *user* yang dibuat. *Sampler* melakukan pengetesan dan berinteraksi ketika *server loading*. Untuk pengujian web, pengujian menggunakan *Sampler* 'HTTP *Request*' yang ada pada aplikasi Apache JMeter.

Dari hasil karya yang dibuat oleh (Niranjanamurthy et al., 2016) telah membuat studi banding atas dua aplikasi untuk pengujian performa untuk mengukur seberapa besar kinerja aplikasi yang dapat dibebani oleh *user* yang mengakses secara simultan dan bersamaan pada aplikasi tersebut. Bahwa aplikasi JMeter lebih baik dibandingkan aplikasi Load Runner dari segi biaya dan segi performa penggunaan aplikasi tersebut.

Menurut (Warne, 2014) telah menjelaaskan tentang keutamaan *Multivariate Analysis of Variance* (MANOVA) yang digunakan oleh para peneliti dalam membuat keputusan dari hasil statistik yang telah dibuat. Bahwa penerapan MANOVA merupakan perluasan dari ANOVA secara matematis diperluas yang dapat diterapkan ke dalam situasi dimana terdapat dua atau lebih variabel dependen (*multivariate*). Langkah-langkah dalam pengujian statistik terdiri dari: Menghindari inflasi dari Type I *error* dan *Post Hoc Procedure*. Dan terdapat contoh proses penghitungan MANOVA di dalamnya.

Tabel 2.6. – Literatur Reviewer

Author	Title	Year	Kontribusi dalam penulisan
Shao, Jingbo Liu, Xiaoxiao Li, Yingmei Liu, Jingyu	Database Performance Optimization for SQL Server Based on Hierarchical Queuing Network Model	2015	Dapat mengetahui konsep penggunaan HQN dalam pendistribusian *database* SQL Server dapat memprediksikan kinerja Query dan dapat dilakukan tuning kinerja dan optimasi Query
Khan, Majid Khan, M N A	Exploring Query Optimization Techniques in Relational Databases	2013	Memberikan informasi detail mengenai teknik tuning dan optimasi dari literatur jurnal yang kemudian dirangkum di dalam jurnal
Kashem, M A Chowdhury, Abu Sayed Deb, Rupam Jahan, Moslema	Query Optimization on Relational Databases for Supporting Top-k Query Processing Techniques	2010	Jurnal yang menjelaskan tentang optimasi query menggunakan Top- K Query yang berguna untuk meningkatkan performa dan menghasilkan QEP terbaik
Kumari, Navita	SQL Server Query Optimization Techniques - Tips for Writing Efficient and Faster Queries	2012	Berisi informasi tips untuk meningkatkan performa kinerja *query* SQL Server
Luca, Valentina Petricică, Octavian Teodor	Query Optimization Techniques in Microsoft SQL Server	2014	Membahas faktor apa saja yang dapat mempengaruhi performa Query pada Database SQL Server

Dahiya, Sapna Ahlawat, Pooja	Performance Tuning In Microsoft SQL Server DBMS	2015	Menjelaskan tentang bagaimana terjadinya *bottleneck* yang disebabkan oleh beberapa faktor
Murugesan, Muthkumar Karthikeyan, K Sivakumar, K	Novel Investigation Methodologies to Identify the SQL Server Query Performance	2015	Menerangkan dengan protokol dapat digunakan untuk menemukan dan memecahkan masalah kinerja SQL
Hussain, Mohammed Waheeduddin Premchand, P Someswar, G Manoj	Design & Development of an Advanced Database Management System Using Multiversion Concurrency Control Model for a Multiprogramming Environment	2015	Menjelaskan bahwa di dalam lingkungan *multiprogramming* akan mempunyai masalah apabila terjadinya proses eksekusi secara bersamaan (*concurrency*)
Patil, Sheetal S Joshi, S D Dhotre, S S	Prediction Of Performance For Web Application By Performance Modeling With JMeter	2012	Menjelaskan tentang prediksi performa aplikasi dengan *Performance Modelling* dapat digunakan untuk mempelajari suatu sistem
Niranjanamurthy, M S, Kiran Kumar Saha, Anupama Chahar, Dharmendra	Comparative Study on Performance Testing with JMeter	2016	Menjelaskan tentang perbandingan aplikasi JMeter dengan Load Runner dalam proses pengujian performa untuk mengukur seberapa besar kinerja aplikasi
Warne, Russell T	A Primer on Multivariate Analysis of Variance (MANOVA) for Behavioral Scientists	2014	Menjelaskan langkah-langkah pengujian statisting menggunakan MANOVA