BAB II

LANDASAN TEORI

2.1 Penelitian Terdahulu

Pada bab ini membahas berbagai penelitian terdahulu yang berfokus pada penerapan *partitioning* dan *indexing* dalam pengoptimalan kinerja *database*. Kajian terhadap penelitian-penelitian sebelumnya bertujuan untuk memperkuat landasan teoritis penelitian ini serta memberikan bukti pendukung mengenai efektivitas kedua teknik tersebut dalam meningkatkan performa sistem *database*. Pada tabel 2.1 merupakan tabel perbandingan penelitian terdahulu.

Tabel 2. 1 Perbandingan Penelitian Terdahulu

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
1	*Looking Deeply into the Magic Mirror: An Interactive Analysis of Database Index SELECT ion Approaches* Penulis: Halfpap (2024) [11]	Proceedings of the VLDB Endowment (PVLDB)	Evaluasi delapan algoritme *index SELECT ion* (AutoAdmin, DB2 Advisor, CoPhy, Dexter, dsb.)	Pemilihan *index* sesuai *query workload* menurunkan biaya eksekusi hingga **50%**; kombinasi *index* perlu dioptimasi untuk menghindari *redundant maintenance*.
2	*The Effect of Partitioning and Indexing on Data Access Time* Penulis: Salgova & Matiasko (2022) [8]	29th FRUCT Conference	Menggunakan *Local Partitioned Index* dengan *Range* dan *List Partitioning* di Oracle 18c.	Kombinasi *range partition* dan *index* lokal mempercepat waktu akses hingga **10**×; partisi berlebihan memperlambat *INSERT*.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
3	*Robust Partitioning Scheme for Accelerating SQL Database* Penulis: Khan (2022) [12]	IEEE ICESIT Conference	Metode yang digunakan Neo4j dengan konfigurasi default dan Oracle 11g (EE) dengan *range partitioning*.	*Partitioning* meningkatkan kinerja *query* hingga **50%**, mengurangi *scan cost*, dan mempercepat agregasi historis.
4	*PostgreSQL Table Partitioning Strategies: Handling Billions of Rows Efficiently* Penulis: Avula, S. B. (2024) [13]	IJETCSIT	Implementasi *Range*, *List*, dan Hash *Partitioning*.	*Range partitioning* menurunkan waktu eksekusi 35-60%; *hash partitioning* efektif untuk *load balancing*.
5	*Comparing Oracle and PostgreSQL, Performance and Optimization* Penulis: Martins (2021) [14]	WorldCIST	B-Tree *Indexing* dan *tuning work_mem*, *shared_buffers*.	PostgreSQL meningkat **91%** setelah optimasi *index*; menunjukkan pentingnya *cost-based optimizer*.
6	Tabel Partisi Pada STARS: Konsep Dan Evaluasi (Studi Kasus STARS UKSW)	Jurnal Pengembangan Informatika dan Teknologi (JPIT)	Partisi *vertical* dan *list partition*.	Partisi tabel meningkatkan kinerja *SELECT /UPDATE/DELETE*, sedangkan *INSERT* memiliki kinerja yang buruk untuk tabel partsi.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
	Penulis: Boymau (2023) [15]			
7	*Performance Optimization in SAP HANA: A Comprehensive Guide to Database Tuning* Penulis: Malli & Jayaprakash (2025) [16]	IJSAT	*Range Partitioning*, Hash *Partitioning*, dan Clustered *Index* pada SAP HANA.	Kombinasi *range partition* dan *clustered index* menurunkan latensi hingga **45%**; *round-robin partition* tidak efisien untuk data berurutan.
8	*Optimizing Database Performance: Strategies for Efficient Query Execution and Resource Utilization* Penulis: Basavegowda, V. (2023) [17]	IJCTT	*Index*, *partitioning*, *query rewriting*, dan Data *Partitioning* berbasis waktu.	Kombinasi *index* dan partisi menurunkan *query latency* hingga **40%** dan meningkatkan *throughput* sistem.
9	*Unlocking Peak Performance: Advanced Techniques for Optimizing Database Efficiency* Penulis: Thallapally,	International Journal of Science and Research Archive (IJSRA)	Pembahasan teori *Indexing* (B-Tree, Bitmap) dan *Sharding*/ *Partitioning*.	*Indexing* dan *sharding* mempercepat performa sistem besar; direkomendasikan penggunaan *range partitioning* untuk data analitik.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
	N. (2021) [18]			
10	*Indexing techniques and structured queries for relational databases management systems* Penulis: Saidu, I. C. (2024) [19]	Journal of the Nigerian Society of Physical Sciences	Eksperimen B-Tree vs Hash *Indexing* pada PostgreSQL 15.	*Hash index* unggul untuk *INSERT-heavy workload*, sementara *B-Tree* lebih efisien untuk *query SELECT /UPDATE*.
11	*Comparing database optimisation techniques in PostgreSQL : Indexes, query writing and the query optimiser* Penulis: Inersjö (2021) [20]	Examensarbete Inom Teknik, Grundnivå, 15 Hp	Eksperimen perbandingan teknik optimasi PostgreSQL termasuk *indexing*, *query rewriting*, dan *planner tuning*.	Penggunaan *indexing* dan *query optimization* menurunkan waktu eksekusi *query* kompleks hingga 70%; *tuning* parameter *work_mem* dan *shared_buffers* berpengaruh signifikan terhadap performa sistem.
12	*Elastic Indexes: Dynamic Space vs. Query Efficiency Tuning for In-Memory Database Indexing* Penulis: Hershcovitch (2022) [21]	Advances in *Database* Technology - EDBT	Implementasi *Elastic B+-Tree* yang dapat menyesuaikan ukuran node secara dinamis berdasarkan tekanan memori.	*Elastic indexing* memungkinkan efisiensi ruang penyimpanan meningkat hingga 60% dengan penurunan performa *query* kurang dari 25%, menjaga keseimbangan antara kecepatan dan efisiensi memori.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
13	*Performance Tuning* Oracle 11g *Database* Melalui Inisial Paramater, Structure *Database* dan SQL *Tuning*. Studi Pada ERP SISFORBUN Dana Pensiun Perkebunan (DAPENBUN) Penulis: Samidi & Hariyanto (2023)	Techno.com, 2023, Vol 22, Issue 2, p400	Studi eksperimental pada sistem Oracle 11g menggunakan *SQL Tuning Advisor* serta pengaturan parameter inisialisasi (*optimizer_mode*, *db_cache_size*, *shared_pool_size*).	Penelitian menunjukkan peningkatan performa sistem sebesar 55% setelah *tuning* parameter dilakukan. Kombinasi antara *index optimization* dan *cost-based optimizer* terbukti mempercepat proses *query* dan efisiensi penggunaan sumber daya.
14	*Tailored Partitioning for Healthcare Big Data: A Novel Technique for Efficient Data Management and Hash Retrieval in RDBMS Relational Architectures* Penulis: Soltanmohammadi (2025) [22]	Journal of Data Analysis and Information Processing	Pendekatan *two-layer partitioning* dengan algoritma *hash-based anonymization* dan *segmented range partitioning* pada PostgreSQL.	Model *tailored partitioning* meningkatkan efisiensi *query* 2,5× dan mendukung *fault tolerance* lebih baik melalui distribusi data yang seimbang antar partisi.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
15	*To Partition, or Not to Partition, That is the Join Question in a Real System* Penulis: Bandle M, Giceva J, Neumann T (2021) [23]	Proceedings of the ACM SIGMOD International Conference on Management of Data	Penelitian ini menggunakan *range partitioning* untuk *database* Oracle dan PostgreSQL.	Hasil menunjukkan bahwa tabel berpartisi mempercepat eksekusi *query SELECT* hingga 40% dibanding tabel tanpa partisi, namun terdapat penurunan kecil pada performa *INSERT* akibat overhead pemeliharaan partisi. Penelitian menegaskan bahwa *partitioning* efektif untuk *read-heavy workload*.
16	*MySQL VS PostgreSQL: A Comparative Analysis of RelationalDatabase Management Systems (RDBMS) Technologies ResponseTime in Web-based E-commerce* Penulis: Jannette G, 2024 [24]	Tarlac State University	Melakukan eksperimen dengan melakukan operasi *INSERT*, *UPDATE*, *DELETE* dengan scenario data yang digunakan adalah 10,000 dan 100,000 *records*	Untuk data kecil, hasilnya bervariasi. Namun untuk data yang lebih besar (100k), PostgreSQL menunjukkan skalabilitas dan performa yang lebih baik di semua operasi (*INSERT*, *UPDATE*, *DELETE*)
17	*A Performance Benchmark for the PostgreSQL and MySQL Database*s Penulis:	MDPI	Metode ini mengevaluasi dan membanding kan PostgreSQL dan MySQL dalam skenario sistem	PostgreSQL secara konsisten menunjukkan performa yang lebih unggul dan stabil dibandingkan MySQL, terutama dalam skenario pembacaan data (*SELECT*) dan dalam lingkungan dengan beban kerja yang kompleks.

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
	Sanket V, 2024 [25]		autentikasi pengguna berkelanjutan (continuous user authentication).	
18	*PERFORMANCE COMPARISON BETWEEN TWO RELATIONAL DATABASE MANAGEMENT SYSTEMS B-tree indexing in PostgreSQL and MySQL* Penulis: Simon L, 2024 [26]	UMEÅ University	Penelitian ini membandingkan performa PostgreSQL dan MySQL dengan melakukan eksperimen untuk mengukur kecepatan operasi CRUD (*INSERT*, *SELECT* , *UPDATE*, *REMOVE*) dan alokasi memori (*memory usage*)	PostgreSQL tanpa *index* memberikan *insert* tercepat namun lambat untuk *select*, *update*, dan *delete*, sedangkan MySQL tanpa *index* paling efisien dalam penggunaan ruang tetapi paling lambat untuk operasi *query*. Dengan *index* B-Tree, PostgreSQL unggul pada operasi *select*, *update*, dan *delete* untuk dataset kecil, sementara MySQL dengan *index* lebih cepat pada dataset besar namun memiliki performa *insert* paling lambat. Studi menyimpulkan bahwa tidak ada DBMS yang sepenuhnya unggul; pemilihan bergantung pada kebutuhan spesifik dan karakteristik *workload*.
19	Perbandingan Performa Optimasi Struktur Penulisan *Query* SQL *Database* MySQL, PostgreSQL, dan Microsoft	Universitas Multimedia Nusantara	Penelitian ini menggunakan metodologi *Database* System Development Lifecycle (DSDLC) dengan pendekatan bottom-up	Pilihan *database* terbaik antara Microsoft SQL Server dan PostgreSQL sangat bergantung pada beban kerja (jumlah thread) dan upaya optimasi *query* 1. Performa Awal (Sebelum Optimasi): Microsoft SQL Server adalah pilihan dengan

No	Judul Penelitian, Penulis, & Tahun	Jurnal / Publisher	Metode	Hasil dan Temuan Utama
	SQL Server Menggunakan Apache JMeter Penulis: Tanujaya G [27]		dengan mengukur response time, throughput, dan error rate pada PostgreSQL, SQL Server dan MySQL	performa terbaik "langsung dari kotak" (*out-of-the-box*). 2. Dampak Optimasi: PostgreSQL adalah *database* yang mendapatkan manfaat paling signifikan dari adanya optimasi *query*.

2.1.1. Sintesis Literatur Penelitian Terdahulu

Berdasarkan sintesis terhadap sembilan belas penelitian terdahulu yang dirangkum pada Tabel 2.1, dapat disimpulkan bahwa optimasi performa database relasional merupakan aspek krusial yang dipengaruhi oleh penerapan teknik indexing, partitioning, tuning parameter sistem, optimasi query, serta pemilihan DBMS yang sesuai dengan karakteristik workload. Sejumlah penelitian menegaskan bahwa indexing berperan penting dalam meningkatkan efisiensi eksekusi query dengan mengurangi sequential scan dan mempercepat proses pencarian data, terutama pada operasi SELECT dan UPDATE. Studi oleh Halfpap [11], Inersjö [20], Saidu [19], serta Thallapally [18] menunjukkan bahwa pemilihan struktur index yang tepat, seperti B-Tree, Bitmap, dan Hash Index, mampu menurunkan waktu eksekusi query secara signifikan, meskipun efektivitasnya sangat bergantung pada pola akses data dan jenis workload. Penelitian Hershcovitch [21] memperluas kajian indexing melalui konsep elastic indexing yang menyeimbangkan efisiensi ruang dan performa query pada database in-memory, sementara Martins [14], Samidi & Hariyanto [22], serta Malli & Jayaprakash [16] menegaskan bahwa optimasi index yang dikombinasikan dengan cost-based optimizer dan tuning parameter sistem mampu meningkatkan performa database hingga lebih dari 50%.

Selain indexing, berbagai penelitian menekankan pentingnya teknik partitioning dalam mengelola data berskala besar. Penelitian oleh Salgova & Matiasko [8], Khan [12], Avula [13], Boymau [15], serta Bandle dkk. [23] menunjukkan bahwa penerapan range partitioning secara konsisten meningkatkan performa query SELECT melalui mekanisme partition pruning, dengan peningkatan performa yang dilaporkan berkisar antara 35% hingga 60%. Penelitian Soltanmohammadi [22] bahkan menunjukkan bahwa pendekatan tailored partitioning berbasis dua lapis mampu meningkatkan efisiensi query hingga 2,5 kali lipat dan mendukung distribusi data yang lebih seimbang. Namun demikian, beberapa penelitian juga menyoroti adanya trade-off, di mana operasi INSERT cenderung mengalami penurunan performa akibat overhead pemeliharaan partisi dan index, sebagaimana dilaporkan oleh Salgova & Matiasko [8], Boymau [15], serta Bandle dkk. [23].

Lebih lanjut, sejumlah penelitian mengkaji kombinasi berbagai teknik optimasi database. Studi oleh Basavegowda [17], Malli & Jayaprakash [16], Khan [12], serta Tanujaya [27] menunjukkan bahwa integrasi indexing, partitioning, optimasi query, dan tuning parameter sistem menghasilkan peningkatan performa yang lebih signifikan dibandingkan penerapan teknik secara terpisah. Di sisi lain, beberapa penelitian melakukan perbandingan performa antar DBMS, seperti PostgreSQL, MySQL, dan Microsoft SQL Server. Penelitian oleh Jannette [24], Sanket [25], Simon [26], serta Tanujaya [27] menunjukkan bahwa PostgreSQL secara umum memiliki skalabilitas dan stabilitas performa yang lebih baik pada dataset besar dan workload kompleks, khususnya untuk operasi SELECT setelah dilakukan optimasi, meskipun tidak ada satu DBMS yang sepenuhnya unggul untuk semua skenario. Secara keseluruhan, meskipun efektivitas masing-masing teknik optimasi telah banyak dibuktikan, sebagian besar penelitian masih dilakukan dalam lingkungan eksperimen atau simulasi dengan data sintetis dan belum secara spesifik mengevaluasi dampak penerapan

indexing dan partitioning secara terintegrasi terhadap performa query SELECT pada PostgreSQL dalam konteks sistem operasional nyata, sehingga membuka peluang penelitian lebih lanjut pada bidang tersebut.

2.1.2. Research Gap dan Posisi Kontribusi Penelitian

Berdasarkan sintesis literatur yang telah dilakukan, dapat diidentifikasi beberapa celah penelitian (*research gap*) yang masih terbuka. Pertama, sebagian besar penelitian terdahulu masih menguji teknik indexing dan partitioning secara terpisah, sehingga interaksi dan dampak kombinasi kedua teknik tersebut terhadap performa database belum banyak dikaji secara mendalam. Penelitian yang mengombinasikan indexing dan partitioning secara simultan dalam satu skenario pengujian yang terintegrasi pada PostgreSQL masih relatif terbatas.

Kedua, banyak penelitian terdahulu dilakukan pada lingkungan simulasi atau menggunakan data sintetis, seperti pada penelitian Simon (2024), sehingga belum sepenuhnya merepresentasikan karakteristik beban kerja database operasional yang nyata [26]. Padahal, performa database sangat dipengaruhi oleh pola akses data, volume transaksi, serta kompleksitas relasi antar tabel yang umumnya lebih tinggi pada sistem produksi perusahaan.

Ketiga, meskipun beberapa penelitian telah membahas peningkatan performa database secara umum, belum banyak penelitian yang secara spesifik memfokuskan analisis pada query SELECT dengan karakteristik SELECT-heavy workload, terutama pada PostgreSQL. Padahal, pada banyak sistem informasi perusahaan, operasi SELECT merupakan operasi yang paling dominan dan memiliki dampak langsung terhadap waktu respon aplikasi.

Keempat, hingga saat ini belum ditemukan penelitian yang mengimplementasikan dan mengevaluasi kombinasi teknik indexing dan partitioning pada lingkungan database PostgreSQL milik PT ABC, yang

memiliki karakteristik data dan permasalahan bottleneck performa yang berbeda dibandingkan studi terdahulu.

Berdasarkan celah penelitian tersebut, penelitian ini memposisikan diri untuk mengisi gap dengan mengimplementasikan dan mengevaluasi kombinasi teknik indexing dan partitioning secara terintegrasi pada PostgreSQL dalam konteks studi kasus operasional nyata PT ABC, dengan fokus pada peningkatan performa query SELECT. Kontribusi utama penelitian ini adalah memberikan bukti empiris mengenai dampak penerapan kombinasi kedua teknik tersebut terhadap performa database sebelum dan sesudah optimasi, sehingga diharapkan dapat menjadi referensi praktis bagi implementasi optimasi database PostgreSQL pada lingkungan perusahaan dengan karakteristik beban kerja serupa.

2.2 Teori yang berkaitan

2.2.1 *Database* Performance *Tuning*

Database performance tuning merupakan proses sistematis yang bertujuan untuk mengoptimalkan berbagai komponen dalam sistem *database*, seperti *query*, *index*, partisi, serta parameter konfigurasi, agar sistem dapat mencapai tingkat efisiensi maksimum [14]. Proses ini dilakukan melalui beberapa tahapan penting yang saling berkaitan. Tahap pertama adalah analisis *query execution plan* menggunakan perintah *EXPLAIN ANALYZE* untuk mengetahui jalur eksekusi *query* serta mendeteksi bagian yang menjadi *bottleneck*. Selanjutnya, dilakukan perancangan *index* yang sesuai dengan pola kueri, di mana kolom dengan tingkat akses tinggi atau sering digunakan dalam klausa *WHERE* menjadi prioritas utama untuk di*index*. Tahap berikutnya adalah pembagian tabel besar menjadi beberapa bagian kecil menggunakan teknik *partitioning*, yang bertujuan untuk mempermudah pemrosesan data dan mempercepat waktu eksekusi *query*. Dengan demikian, kombinasi antara optimasi *query*, desain *index* yang tepat, penerapan partisi, dan konfigurasi sistem

yang efisien menjadi kunci utama dalam meningkatkan performa keseluruhan *database* relasional.

2.2.2 *Indexing*

Dalam konteks PostgreSQL, *index* memegang peranan penting dalam mempercepat proses pencarian dan pengambilan data. Halfpap (2024) menegaskan bahwa pemilihan jenis *index* yang tepat sangat memengaruhi kinerja *query* karena *index* berfungsi sebagai struktur data yang memperpendek waktu akses terhadap baris tertentu dalam tabel [11]. PostgreSQL mendukung berbagai jenis *index*, antara lain B-Tree, GIN (*Generalized Inverted Index*), dan BRIN (*Block Range Index*). *Index* B-Tree merupakan tipe *index* default yang paling umum digunakan karena efisien untuk operasi pencarian berbasis kesetaraan (*equality*) maupun rentang nilai (*range query*). GIN biasanya diterapkan untuk tipe data teks atau JSONB karena mendukung pencarian kata secara penuh (*full-text search*), sedangkan BRIN digunakan untuk tabel dengan ukuran sangat besar dan data yang memiliki urutan alami seperti data temporal (*timestamp*). Isah (2024) menemukan bahwa B-Tree memberikan performa paling stabil pada sebagian besar beban kerja, sedangkan Bitmap *Index* lebih cocok untuk kolom dengan jumlah nilai unik yang rendah (*low cardinality*), dan Hash *Index* unggul pada beban kerja dengan banyak operasi *INSERT* [19]. Namun, seperti dijelaskan oleh Basavegowda (2023), penambahan *index* yang terlalu banyak dapat meningkatkan *maintenance overhead*, terutama pada operasi tulis seperti *INSERT*, *UPDATE*, dan *DELETE*, sehingga jumlah dan jenis *index* perlu disesuaikan dengan kebutuhan aktual sistem [17].

2.2.3 *Partitioning*

Partitioning merupakan salah satu teknik utama dalam upaya peningkatan performa *database*, dengan proses membagi satu tabel besar menjadi beberapa bagian kecil (partisi) berdasarkan nilai tertentu pada kolom kunci [22]. PostgreSQL menyediakan tiga jenis utama *partitioning*, yaitu *range partitioning*, *list partitioning*, dan hash

partitioning. *Range partitioning* digunakan untuk membagi data berdasarkan interval nilai, seperti tanggal transaksi; *List partitioning* digunakan untuk mengelompokkan data berdasarkan nilai diskrit tertentu, seperti status, kategori, atau wilayah. Teknik ini efektif apabila nilai pada kolom partisi bersifat terbatas dan jarang berubah. Namun, apabila terjadi penambahan nilai baru pada kolom partisi, maka *administrator database* perlu menambahkan partisi baru secara manual. Kondisi ini dapat menambah kompleksitas pengelolaan partisi dan berpotensi mempengaruhi skalabilitas sistem apabila tidak dirancang dengan baik. Oleh karena itu, penggunaan list partitioning lebih sesuai untuk data dengan domain nilai yang relatif stabil dan terkontrol; sedangkan hash *partitioning* digunakan untuk mendistribusikan data secara merata berdasarkan hasil fungsi hash. Keuntungan utama dari penggunaan partisi adalah terjadinya pengurangan *full table scan* karena *query* hanya akan menelusuri partisi yang relevan melalui mekanisme *partition* pruning. Menurut penelitian Khan (2022) menunjukkan bahwa dengan menerapkan *range partitioning* pada kolom waktu transaksi, kinerja *query* dapat meningkat hingga 50 persen karena sistem hanya membaca sebagian kecil data yang relevan [12]. Namun, jumlah partisi yang berlebihan dapat memperlambat operasi tulis akibat meningkatnya jumlah metadata dan *index* yang harus diperbarui [15].

2.3 Algoritma yang digunakan

2.3.1. B-Tree

B-Tree merupakan salah satu algoritma struktur data yang paling umum digunakan untuk sistem peng*index*an pada *database* relasional, termasuk PostgreSQL [28]. Algoritma ini dirancang untuk menyimpan data dalam bentuk pohon yang seimbang (*balanced tree*), di mana setiap simpul (*node*) dapat memiliki lebih dari dua anak [29]. Karakteristik tersebut memungkinkan B-Tree untuk menjaga kedalaman pohon tetap rendah, sehingga proses pencarian (*search*), penyisipan (*insertion*), dan

penghapusan (*deletion*) data dapat dilakukan secara efisien. Berikut formula b-tree:

$$O(\log(n))$$

Dalam konteks PostgreSQL, B-Tree berfungsi sebagai mekanisme utama dalam peng*index*an data untuk mempercepat proses eksekusi *query* [30]. Ketika pengguna menjalankan *query* yang melibatkan operasi pencarian atau penyortiran berdasarkan nilai kolom tertentu, PostgreSQL dapat memanfaatkan *index* B-Tree untuk menemukan data yang relevan tanpa harus melakukan pemindaian penuh terhadap seluruh tabel (*sequential scan*). Dengan demikian, waktu respon sistem dapat dikurangi secara signifikan, terutama pada tabel dengan jumlah baris yang sangat besar.

Menurut penelitian yang dilakukan oleh Martins (2021), penerapan *index* berbasis kolom seperti B-Tree mampu meningkatkan efisiensi kinerja PostgreSQL hingga mencapai 91% dalam eksekusi *query* yang bersifat kompleks. Peningkatan tersebut terjadi karena struktur B-Tree memungkinkan sistem untuk mengakses data secara langsung ke lokasi penyimpanan yang relevan, sehingga mengurangi jumlah operasi baca (I/O operations) yang dibutuhkan [14].

2.4 Tools/software yang digunakan

2.4.1 PGAdmin4

PgAdmin4 merupakan *Graphical User Interface (GUI)* resmi untuk PostgreSQL yang digunakan untuk mempermudah proses pengelolaan *database*, pembuatan tabel, serta penerapan strategi *indexing* dan *partitioning*. Melalui pgAdmin, dapat menjalankan perintah SQL, memantau *query execution plan* menggunakan fitur *EXPLAIN ANALYZE*, serta menganalisis performa *database* setelah penerapan optimasi. Selain itu, pgAdmin menyediakan tampilan grafis yang memudahkan pengamatan hasil *tuning*, seperti perubahan waktu eksekusi, penggunaan *index*, dan proses *partition pruning* pada tabel yang besar [13]. Dengan antarmuka

yang interaktif, pgAdmin juga mendukung fungsi pemantauan terhadap *query performance statistics*, sehingga pengguna dapat mengidentifikasi kueri yang paling sering dieksekusi atau kueri dengan waktu respon tertinggi untuk kemudian dilakukan optimasi lebih lanjut.

2.4.2 PostgreSQL

PostgreSQL merupakan *Relational Database Management System* (RDBMS) utama yang digunakan dalam penelitian ini. Sistem ini dipilih karena bersifat open source, memiliki stabilitas tinggi, serta mendukung berbagai teknik optimasi modern seperti *declarative partitioning*, *index-only scan*, dan beragam jenis *index* (B-Tree, GIN, dan BRIN) [13]. PostgreSQL juga memiliki *cost-based optimizer* yang canggih dan mampu memilih rencana eksekusi dengan biaya terendah berdasarkan statistik internal sistem [11]. Selain itu, PostgreSQL mendukung fitur *EXPLAIN ANALYZE* yang sangat berguna untuk menganalisis *query* execution plan dan mengidentifikasi potensi bottleneck pada *query* tertentu. Dengan kemampuan tersebut, PostgreSQL menjadi platform ideal untuk menguji dampak penerapan *indexing* dan *partitioning* terhadap peningkatan performa *query* dalam penelitian ini.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA