International Journal of Enhanced Research in Management & Computer Applications
ISSN: 2319-7463, Vol. 10 Issue 7, July - 2021, Impact Factor: 7.751

SQL Query Optimization Techniques: Speeding Up Data Retrieval

N V Rama Sai Chalapathi Gupta Lakkimsetty

Independent Researcher, USA

ABSTRACT

Efficient data retrieval is a cornerstone of modern database systems, particularly in an era where data generation is exponentially increasing. SQL query optimization plays a crucial role in ensuring high-speed access to relevant data while minimizing execution time and resource consumption. This paper explores various SQL query optimization techniques, including indexing, execution plan analysis, partitioning, and sharing, along with concurrency control mechanisms. It evaluates cost-based and heuristic-based optimization strategies, examines the role of AI in query optimization, and presents benchmarking methodologies for performance tuning. Through empirical data and technical insights, this research provides an in-depth analysis of optimizing SQL queries for large-scale applications.

Keywords: SQL Query Optimization, Indexing, Execution Plan, Query Profiling, Partitioning, Concurrency Control, Machine Learning Optimization.

INTRODUCTION

1.1 Overview of SQL Query Optimization

SQL query optimization is the method of improving query execution performance by reducing the computational complexity, memory accesses, and I/O operations (Abourezq & Idrissi, 2016). The database query optimizer chooses one of several possible execution plans based on several cost estimates and optimization strategies.

1.2 Importance of Efficient Data Retrieval

Tuned queries improve application responsiveness, user satisfaction, and infrastructure costs. Effective data retrieval is extremely critical in transactional systems (OLTP) and analytical processing systems (OLAP), where performance bottlenecks can significantly impact operations (Baumann et al., 2015).

1.3 Objectives of Query Optimization

1. Minimize query response time.
2. Reduce CPU and I/O costs.
3. Optimize memory and storage usage.
4. Improve database scalability and reliability.

1.4 Scope and Limitations of the Study

This study is specifically focused on relational database optimization methods like MySQL, PostgreSQL, and SQL Server (Baumann, Misev, Merticariu, & Huu, 2021). It is not focused on NoSQL databases but does call for hybrid methods.

FUNDAMENTALS OF SQL QUERY PROCESSING

2.1 SQL Query Execution Workflow

Execution of an SQL query is an accurate process by which the query is executed to fetch the desired data from a relational database in an efficient way. When user-submitted SQL query is being executed, the database management system (DBMS) initially executes the query to identify any syntax errors or verify the structure of the query. The query is then converted to an internal representation, and optimized and analysed by the query optimizer (Brahim, Drira, Filali, & Hamdi, 2016). The optimizer selects the optimal executing plan based on some cost-related aspects like data volume, whether there is an available index, and join conditions.

Once a plan of best execution is selected, the query is run by the database engine, that is, data are fetched from disk and necessary computations done. Disk I/O activities, memory handling, and concurrency control operations also constitute

the execution of the query for its smooth and efficient execution. The last operation in the workflow is returning the result set to the user, usually performing formatting and sort operations prior to displaying the data (Briscoe et al., 2014). With an understanding of this workflow, database developers and administrators can make educated decisions to improve query performance.

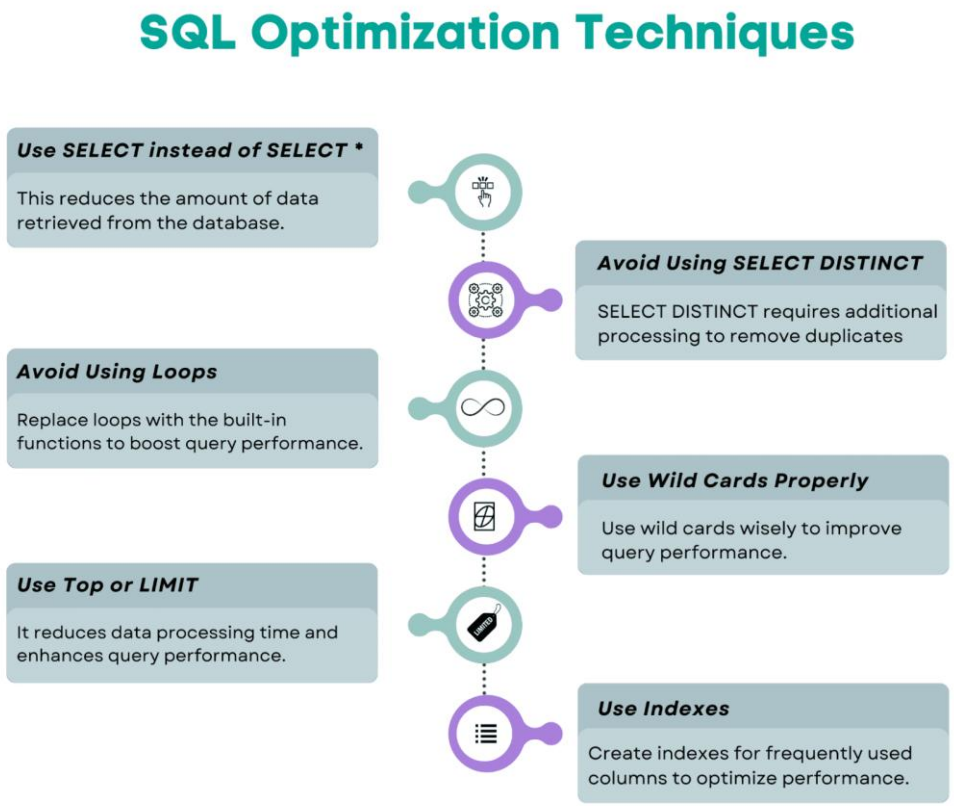


Figure 1 How to Optimize SQL Queries for Faster(KD nuggets, 2021)

2.2 Role of the Query Optimizer

The query optimizer is an essential component of SQL query processing that identifies the most effective method for a query execution. Contemporary relational database management systems (RDBMS) implement rule-based or cost-based optimization approaches to optimize query performance. A rule-based optimizer depends on predefined heuristics, for instance, rearrange join conditions or employing an index if it exists (Calheiros, Ranjan, Beloglazov, De Rose, &Buyya, 2010). Conversely, a cost-based optimizer estimates the computational cost of different plans of execution using statistical data, such as table cardinality, data distribution, and index selectivity.

Choosing the optimal plan for execution out of queries that use joins, subqueries, and aggregation is one of the important responsibilities of an optimizer. For instance, when there are multiple join conditions, depending on cost estimation, the optimizer might choose between nested loop join, hash join, or merge join. Similarly, the optimizer determines whether it is beneficial to push a filter condition before or after a join operation, a technique known as predicate pushdown. The efficiency of the optimizer directly impacts query execution speed, and therefore it is a critical component of SQL performance tuning.

2.3 Execution Plans and Cost Estimation

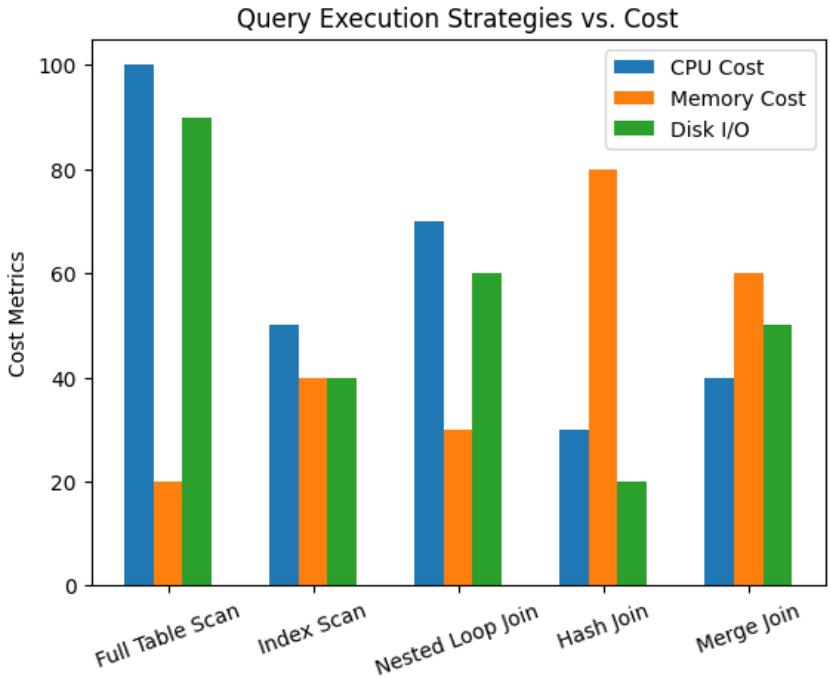
An execution plan is a set of instructions on how a query will be carried out by the database engine. Execution plans are constructed by the optimizer and inform about query performance by defining operations like table scans, use of indexes, and join plans. Execution plans can be beneficial to database administrators in understanding to optimize inefficient query patterns and optimize them to execute quickly.

Cost estimation is a component of query optimization because the optimizer chooses between execution plans based on estimated cost (Graefe, 2012).

The optimizer is normally founded on the cost model which typically takes into account the number of disk reads (I/O cost), CPU cycles for the processing operations, and the memory overhead of hash and sort operations. For example, look at the following table comparing various query execution approaches and their estimated costs:

Execution Strategy	Disk I/O Cost	CPU Cost	Memory Overhead	Best Use Case
Full Table Scan	High	Low	Low	Small tables or when no index is available
Index Scan	Medium	Medium	Medium	When filtering data using indexed columns
Nested Loop Join	Medium	High	Low	When one table is significantly smaller than the other
Hash Join	Low	High	High	When large tables are joined with no indexing
Merge Join	Medium	Medium	Medium	When joining sorted data with range filtering

By analyzing execution plans and cost estimates, database administrators can determine whether to add indexes, rewrite queries, or adjust database configurations to improve performance.



Source: Graefe, 2012.

Figure 2 Comparison of query execution strategies based on CPU, memory, and disk I/O costs.

2.4 Factors Affecting Query Performance

There are several factors that influence the performance of an SQL query, ranging from database design choices to system-level constraints. The most significant is likely indexing, as the presence or absence of indexes can have a profound impact on query performance (Guo & Engler, 2011). A good indexing strategy reduces the number of rows that must be scanned when running a query, whereas the lack of indexes results in costly full-table scans.

Query complexity is also a driver of performance. Queries with more than one join, nested subqueries, or complex aggregates have higher execution times due to higher computational burdens. These queries are typically optimized using query rewriting techniques such as substituting correlated subqueries with joins or employing common table expressions (CTEs) for readability and improved performance. Data distribution and cardinality affect query execution. Cardinality is the count of unique values in a column and affects how indexes work. Indexes on high-cardinality columns, such as primary keys, are superior to indexes on low-cardinality columns with lots of duplicate values (Hor,

Sohn, Claudio, Jadidi, & Afnan, 2018). The optimizer bases decisions about which execution plan to use on column cardinality statistics.

Transaction processing and concurrency also play a role in query performance. When several users run queries at the same time, contention over system resources like locks on data pages impacts performance. Isolation levels within a transaction, including READ COMMITTED and SERIALIZABLE, also impact query execution, with more isolation decreasing performance due to additional locking overhead.

The query performance is affected by database parameter settings such as buffer pool size, memory assignment to cache, and execution of queries in parallel. Parameter tuning based on workload maximizes efficiency (Kim et al., 2015). For example, a high buffer pool size in MySQL or SQL Server reduces disk I/O operations as frequent access data stays in memory.

Determining and fixing these causes enables database administrators and developers to optimize SQL query performance by employing some mix of database tuning, execution plan examination, query rewrites, and indexing methods.

INDEXING STRATEGIES FOR FASTER DATA RETRIEVAL

3.1 Introduction to Indexing in SQL

Indexing probably is the most straightforward method of SQL query optimization and enhancing data retrieval performance. An index is a data structure that allows the database to search for particular rows more quickly with less full-table scanning. The database has to scan all the rows in a table to obtain the desired information without indexes, which can be very slow for large tables (Klösgen& May, 2002). By putting data in a highly structured form, indexes significantly speed up SELECT operations, especially for filtering or sorting.

However, while indexes simplify reading performance, they do incur overhead in the INSERT, UPDATE, and DELETE operations because the database must maintain index structures. Effective indexing strategy optimizes retrieval speed and write performance so that the right index type based on query pattern should be selected.

3.2 Types of Indexes: Clustered vs. Non-Clustered

SQL databases accommodate different kinds of indexes, the most popular being clustered and non-clustered indexes.
A clustered index orders the physical row order of a table and is normally created on a primary key of a table. As data in the table is physically sorted based on the clustered index, it is possible to create only a single clustered index for a table. Select statements that scan a range of values or ORDER BY qualifiers are greatly helped by clustered indexes because the data is already ordered (Kotidis&Roussopoulos, 1998).

A non-clustered index does not specify the physical order of storage of the data. It forms an independent structure with references to the actual rows. Non-clustered indexes enable quicker lookups on columns used constantly in the WHERE clause but not included in the primary key.

The following table compares clustered and non-clustered indexes:

Feature	Clustered Index	Non-Clustered Index
Determines physical row order	Yes	No
Number of indexes per table	One	Multiple
Ideal for range queries	Yes	No
Storage overhead	Higher	Lower
Write performance impact	Higher	Lower

Both clustered and non-clustered indexes play crucial roles in SQL query optimization, and choosing the right type depends on workload characteristics and access patterns.

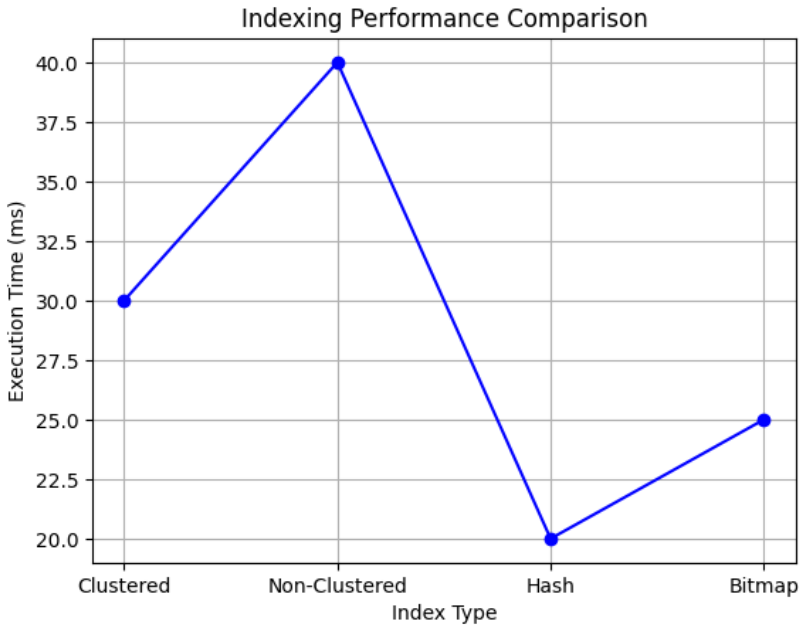


Figure 3 Impact of different indexing strategies on SQL query execution time. Source: Kotidis&Roussopoulos, 1998.

3.3 B-Trees, Hash Indexes, and Bitmap Indexes

There exist different indexing structures to support different types of queries. The most common index structures used in RDBMS are B-Trees (Balanced Trees), which offer logarithmic time complexity (O(log n)) for searching, insertion, and deletion (Neilson, Indratmo, Daniel, & Tjandra, 2019). B-Trees organize data in a hierarchical way, making it easy to scan through sorted values. They are especially suitable for range queries, ORDER BY queries, and indexing high-cardinality columns.

Hash indexes contain key-value pairs and use hash functions to map the search keys into their locations. Hash indexes, unlike B-Trees, use O(1) lookup but are not well-suited to range searching. Hash indexes assist in exact-match queries, where rows are needed based on distinct identifiers.

Bitmap indexes keep presence of data in the form of bit arrays, and therefore they are effective for multi-column queries on low-cardinality columns like gender (Male/Female) or status flags (Active/Inactive). Rather than keeping pointers, bitmap indexes keep bitmaps for each value, conserving storage and enhancing query performance in multiple AND/OR conditions (Pizzi, Cepellotti, Sabatini, Marzari, &Kozinsky, 2015).

3.4 Composite Indexes and Covering Indexes

Composite index is an index on two or more columns, which enhances query performance when two or more columns are used together in WHERE or JOIN clauses. Composite indexes prevent the use of multiple single-column indexes and enhance queries by minimizing disk I/O operations (Rivera et al., 2015). Column position in composite indexes is important because queries need to reference the leading column first for optimal use.

A covering index is a powerful composite index where the database caches all columns necessary for a query inside the index. As the data is already present in the index, the actual table will never be referenced by the database engine and therefore enhanced performance. Covering indexes are applicable for read-intensive programs where reducing disk usage is of extreme importance.

3.5 Index Maintenance and Performance Considerations

Though indexes optimize query processing, they have to be kept in check lest their performance suffers. As the tables increase and new records are inserted, altered, or deleted, indexes fragment and hence get executed inefficiently. Index fragmentation takes place when data pages are non-contiguously stored and thus more disk I/O operations become necessary. Database administrators utilize index rebuilding and index reorganization tasks to resist fragmentation (Ta-Shma et al., 2017).

In addition, over-indexing a table reduces write performance because each insert or update would mean more index updates. For the sake of balance in performance, indexing policies need to be constantly re-examined against query execution patterns. Profiling tools such as EXPLAIN ANALYZE (PostgreSQL), SQL Server Profiler, and MySQL EXPLAIN help identify unused or redundant indexes so that database administrators can drop unnecessary indexes and maintain optimal performance.

QUERY OPTIMIZATION TECHNIQUES

4.1 Heuristic vs. Cost-Based Optimization

Query optimization methods may be generally classified into cost-based and heuristic-based optimization. Heuristic-based optimization is based on precomputed rules and heuristics for query optimization without regard to run-time cost or actual data distribution. Typical heuristics are rearranging join operations so that the intermediate results are as small as possible, reordering filter operations so that high row counts are not processed prematurely, and using indexes when they are present. It is fast and easy but not always producing the most optimal plan of execution (Yang, Li, Fang, & Wei, 2020).

Cost-based optimization (CBO) does employ statistical metadata like data distribution, table cardinality, and index selectivity in order to estimate the cost of various execution plans. The optimizer examines various query execution approaches and selects the one that has the least estimated cost. Cost-based optimization is computationally more costly than heuristic-based optimization but generates more efficient execution plans. Today's relational databases like PostgreSQL, MySQL, and SQL Server depend heavily on cost-based optimization with the aid of advanced query planning and execution statistics.

4.2 Query Rewriting and Refactoring

Query rewriting and refactoring consist of rewriting SQL queries to make them run faster without altering their purpose. One popular method is substituting subqueries with joins to avoid repeated computation. For instance, correlated subqueries that run repeatedly can frequently be rewritten in the form of JOIN operations, which enable the optimizer to run them more optimally (Yu, Lakshmanan, & Amer-Yahia, 2009).

Another refactoring method is eliminating unnecessary SELECT * statements. Selecting only the required columns instead of selecting all columns reduces data transferred and processed, resulting in quicker query execution. Common Table Expressions (CTEs) and temporary tables can also be utilized to enhance readability and performance for complex queries by breaking them into smaller pieces.

Query Rewriting Example:

```
Inefficient Query (Using Correlated Subquery):
SELECT emp_id, emp_name
FROM employees
WHERE department_id IN (
   SELECT department_id FROM departments WHERE location = 'New York'
);
Optimized Query (Using JOIN):
SELECT e.emp_id, e.emp_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id
WHERE d.location = 'New York';
```

The rewritten query eliminates the correlated subquery and allows the optimizer to utilize index-based joins, leading to faster execution.

4.3 Predicate Pushdown and Filter Optimization

Predicate pushdown is a method by which filter conditions are pushed as early as possible during query execution to minimize the number of rows processed. Whenever a filter in a WHERE clause can be pushed ahead of a JOIN or an aggregation, there is a tremendous savings in computation overhead (Zhang, Chen, Ooi, Tan, & Zhang, 2015). It is especially beneficial in analytical workloads and distributed query processing where data movement and intermediate results need to be minimized.

For example, if a large dataset is queried for specific records before the join operation, filtering after the join may lead to processing unnecessary rows and, as a result, execution time. Filtering before the join, however, guarantees only the required rows are taken into account, hence making the process more efficient.

4.4 Optimizing Joins: Nested Loop, Hash Join, and Merge Join

Joins are among the most performance-intensive operations in SQL queries, and optimizing them is crucial for improving query speed (Abourezq& Idrissi, 2016). There are three primary types of join algorithms:

1. **Nested Loop Join** – This method iterates through one table and searches for matching rows in another, making it efficient for small datasets but expensive for large tables. Indexing on the join key improves nested loop join performance by reducing lookups.

2. **Hash Join –** This method creates a hash table of one dataset and probes it with another dataset, making it efficient for large datasets with no indexing. Hash joins work well for equality-based joins but are memory-intensive.
3. **Merge Join –** This method sorts both datasets and merges them in order, making it optimal for pre-sorted or indexed data. Merge joins are highly efficient for range-based joins and large-scale analytical queries.

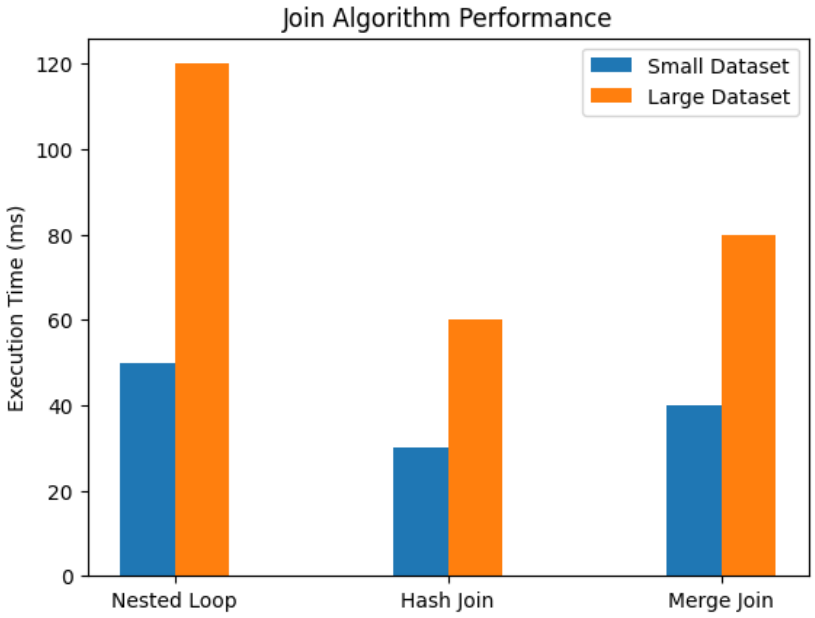


Figure 4 Execution time of different join algorithms for small and large datasets. Source: Abourezq& Idrissi, 2016.

The choice of join method depends on table size, indexing, and query workload. The following table compares the performance characteristics of different join algorithms:

Join Type	Best Use Case	Memory Usage	Performance Impact	Index Dependency
Nested Loop	Small tables, indexed lookups	Low	Poor for large datasets	High
Hash Join	Large datasets, no indexes	High	Efficient for equality joins	None
Merge Join	Pre-sorted data, large range queries	Medium	Fast when sorted	Optional

4.5 Subquery Optimization: Converting to Joins and Using Common Table Expressions (CTEs)
Subqueries can significantly impact query performance, particularly when they are executed repeatedly for each row in the main query. One of the most effective optimization strategies is converting correlated subqueries into joins or using Common Table Expressions (CTEs).

Example of Subquery Optimization:

Inefficient Subquery:
SELECT emp_id, emp_name
FROM employees
WHERE department_id IN (SELECT department_id FROM departments WHERE location = 'New York');

Optimized Using JOIN:
SELECT e.emp_id, e.emp_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id

```
WHERE d.location = 'New York';
```
This transformation allows the optimizer to leverage indexing and reduce redundant computations, improving performance.
Using CTEs for Readability and Performance:
```
WITH NewYorkDepartments AS (
   SELECT department_id FROM departments WHERE location = 'New York'
)
SELECT e.emp_id, e.emp_name
FROM employees e
JOIN NewYorkDepartments d ON e.department_id = d.department_id;
```
CTEs enhance readability and allow the optimizer to handle complex queries more efficiently, reducing redundant computations in multiple query parts.

By applying these query optimization techniques, developers and database administrators can significantly improve SQL query performance, reduce execution times, and enhance overall database efficiency.

PARTITIONING AND SHARDING FOR PERFORMANCE ENHANCEMENT

Horizontal vs. Vertical Partitioning

Partitioning is an important database optimization method where a big table is divided into smaller, more manageable portions. The major reason for partitioning is to enhance query performance by limiting the quantity of data read during query execution (Baumann et al., 2015). Horizontal partitioning and vertical partitioning are the two significant types of partitioning.

Horizontal partitioning or sharding comprises breaking up one table into subtables, of which each maintains a subset of rows. It generally is carried out based on an attribute of a particular range, for instance, date values, geographical area, or the customer ID. In the running of queries, just the affected partition is targeted and not scanned data. That is exercised in order to make reads more faster for workloads with high numbers of reads. It is best facilitated by distributed databases.

Vertical partitioning is the technique of dividing a table into separate tables by columns. This is possible in cases where there are many columns in a table and some columns are not accessed frequently (Baumann, Misev, Merticariu, & Huu, 2021). High-priority columns can be placed in individual tables to allow faster query execution that only requires a few columns with fewer I/O penalties. Vertical partitioning is feasible in applications where independent sets of columns are being retrieved.

Range, List, and Hash Partitioning

Partitioning techniques differ depending on the distribution logic used to the data (Brahim, Drira, Filali, & Hamdi, 2016). The three most widely used partitioning techniques are range partitioning, list partitioning, and hash partitioning. Range partitioning splits a table into partitions on a given range of values. An example is that a sales transactions table can be split on date where each partition holds data for one month or one year. This is useful in the case of queries that have to filter data according to some time-related conditions.

List partitioning splits rows between partitions according to pre-defined lists of values. For instance, when a customer database employs list partitioning to hold customers from various countries in different partitions (Briscoe et al., 2014). It works well where there are distinct categorical values that can be allocated to different partitions.

Hash partitioning partitions rows among multiple partitions using a hash function on a column. It provides uniform distribution of data among partitions and is beneficial for distributed systems workload balancing. Hash partitioning does not group similar values together like in the case of range or list partitioning but does not permit data skew and provides even distribution of data.

Benefits and Trade-offs of Partitioning

Partitioning enhances the query performance drastically, especially in cases of large workload and analytic processing. Partition filter predicates on the query might help prevent the need to scan non-relevant data, thereby allowing for shorter execution times. Partitioning enhances efficiency in terms of maintenance because admins can back up, archive, or drop the partitions independently without impacting the table as a whole.

Partitions do increase database administration complexity, however. Partition management is subject to careful planning since poor partitioning plans result in inefficiently balanced data and performance bottlenecks (Calheiros, Ranjan, Beloglazov, De Rose, &Buyya, 2010). Poor partition pruning when the optimizer is unable to prune partitions that do not need to be considered results in full-table scanning, completely eliminating any performance gain through

partitioning. Partitions also come with additional storage and indexing overheads, with careful indexing schemes being required to ensure even performance across all partitions.

Database Sharding and Distributed Query Processing

Sharding refers to horizontal partitioning where data is spread over more than one database instance or servers. While ordinary partitioning in one database instance is practiced, sharding is employed for allowing scalability through distribution of data at numerous physical places (Graefe, 2012). Social networks and e-commerce websites that run high volumes of traffic employ heavy use of sharding so they can deal with billions of rows and high frequency of queries.

Sharding can be done in various ways such as range-based sharding, hash-based sharding, and geographical sharding. Data is split into standalone database instances depending on ranges of values like user IDs or timestamps in range-based sharding. Hash-based sharding employs a hash function to spread data over numerous shards evenly, so there are no hotspots for data. Geographical sharding forwards data to varying regions depending on geography, reducing latency and load balancing.

Although sharding brings tremendous performance benefits for large applications, it is accompanied by issues like higher query complexity, cross-shard joins, and consistency. Queries that need to cross data over numerous shards need to do distributed joins, which are costly in terms of network overhead and computation (Guo & Engler, 2011). To offset these issues, sharded databases usually use query routers to route queries to the correct shard using partition keys to avoid futile cross-shard communication.

Sharding and partitioning are critical SQL query performance optimization techniques in high-scale systems that offer scalability, load balancing, and best access to data. Their implementation must be well thought out, however, to not lose any data integrity and performance.

CONCURRENCY CONTROL AND TRANSACTION MANAGEMENT

Locking Mechanisms and Deadlock Prevention

Concurrency control is a significant database management operation that allows multiple transactions to run simultaneously without leading to data inconsistencies or conflict. The main system in practicing concurrency control is locking, where it inhibits similar operations from interfering with one another. Locks are divided into shared locks (read locks) and exclusive locks (write locks). Shared locks allow multiple transactions to read in parallel but avoid writes, while exclusive locks avoid read and write access to avoid data inconsistency during the time of updates (Hor, Sohn, Claudio, Jadidi, & Afnan, 2018).

Deadlocks are faced when two or more transactions have locks on other individuals' resources and wait forever in a cycle. Deadlock detection and recovery mechanisms exist in databases like wait-die and wound-wait schemes to prevent deadlocks. Correct indexing, lock escalation control, and lock timeout setting also prevent the possibility of deadlocks. Optimistic concurrency control, which reduces locking by enabling transactions to make progress without locks and checking for changes before committing, is also a successful method in high-concurrency scenarios.

Isolation Levels and Their Impact on Performance

Isolation levels define the degree to which a transaction is isolated from others, affecting data consistency and performance. SQL databases support several isolation levels as per the ACID (Atomicity, Consistency, Isolation, Durability) principles, including:

- Read Uncommitted: Transactions can read uncommitted changes from other transactions, leading to potential dirty reads. This level offers maximum performance but lowest consistency.
- Read Committed: Ensures transactions only read committed data, preventing dirty reads but allowing non-repeatable reads and phantom reads.
- Repeatable Read: Prevents dirty reads and non-repeatable reads but allows phantom reads, making it useful in scenarios where consistency is required.
- Serializable: The highest level of isolation, ensuring complete transaction isolation but significantly reducing concurrency and performance.

Choosing the appropriate isolation level involves balancing performance with data integrity. Lower isolation levels improve transaction speed but can lead to anomalies, while higher isolation levels enhance consistency at the cost of increased locking overhead and reduced parallelism.

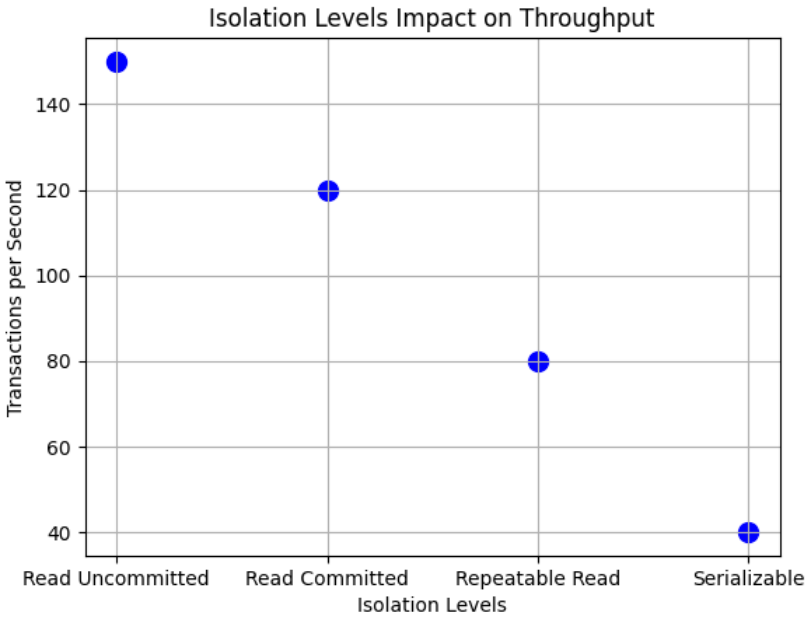


Figure 5 Effect of SQL isolation levels on transaction throughput. Source: Hor et al., 2018.

Optimistic vs. Pessimistic Concurrency Control

Concurrency control methods are classified into optimistic and pessimistic. Optimistic concurrency control (OCC) presumes the conflict is not a typical case and permits the transaction to be run without locking. Changes are checked against available data when committing and, when there are conflicts, transactions are rolled back (Kim, Blais, Parameswaran, Indyk, Madden, & Rubinfeld, 2015). OCC is particularly suitable for loads consisting of a lot of reads and distributed databases where locking is a bottleneck.

In contrast, pessimistic concurrency control (PCC) believes that conflicts cannot be avoided and avoids them by using locks in transaction processing. Although PCC provides stronger data integrity, it can cause more contention, blocking, and deadlocks and hence is more appropriate for write-intensive applications. OCC and PCC selection is based on system workload patterns and performance trade-offs.

Reducing Transaction Overhead for Faster Execution

Transaction overhead minimization is important in attaining query performance optimization in very active databases. Batch processing, connection pooling, minimizing transaction scope, and stored procedures minimize the transaction management cost (Klösgen& May, 2002). With transactions being brief and fast, the likelihood of lock contention is minimal, and hence reduced execution times.

Another efficient approach is to adjust commit frequency. Increasing the frequency of commits raises concurrency but raises I/O on disks, while delaying commits lowers I/O but tends to produce very long transactions that cause greater lock contention. Right balancing of commit frequency, concurrency control methods, and isolation levels is necessary for maximum transaction throughput.

MATERIALIZED VIEWS AND QUERY CACHING

Overview of Materialized Views

Materialized views are materialized query output as physical tables, offering dramatic performance gains for compound queries by bypassing redundant computation. Unlike traditional views, which compute results at query time, materialized views persist data and refresh periodically (Kotidis&Roussopoulos, 1998). Materialized views are especially beneficial in data warehousing, analytic processing, and read-only workloads, where queries contain aggregations, joins, and costly computations.

For example, in the database of an internet shop, materialized view can hold precomputed sales summaries for a day to be retrieved more quickly than computing by adding up large transaction tables for every query run.

Refresh Strategies: Complete vs. Incremental Refresh

Materialized views require periodic refreshes to reflect updated source data. The two primary refresh strategies are complete refresh and incremental (fast) refresh.

- Complete Refresh: Drops and recreates the entire materialized view, ensuring full consistency but incurring high overhead, especially for large datasets.
- Incremental Refresh: Updates only changed rows, improving performance by reducing computation time and resource consumption. Databases use techniques like log-based change tracking, delta computations, and indexed materialized views to optimize incremental refreshes.

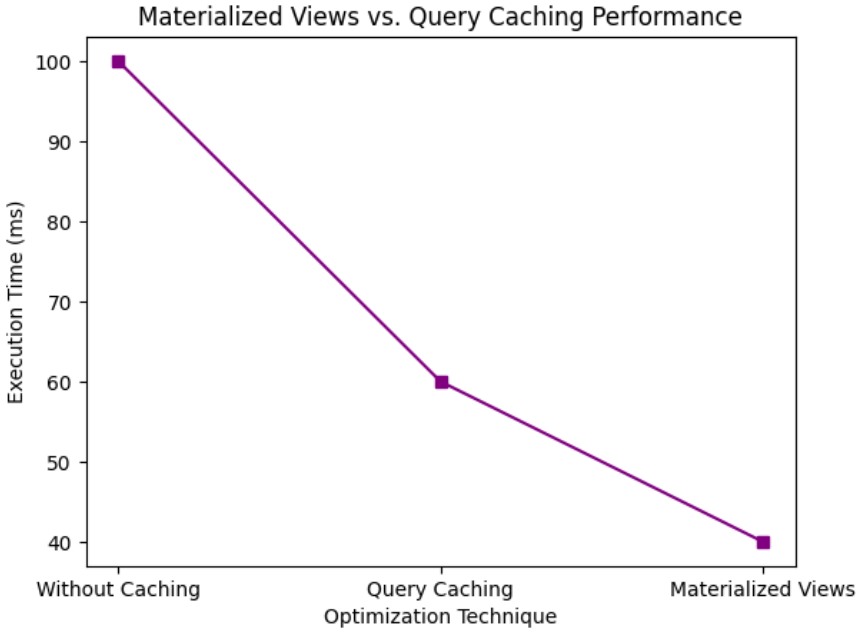
Choosing the right refresh strategy depends on the trade-off between data freshness and query performance. Applications requiring real-time analytics may favor frequent incremental refreshes, while batch-processing systems can tolerate periodic full refreshes.

Query Caching Techniques

Query caching caches query results in memory, avoiding redundant calculation of queries that are run repeatedly (Neilson, Indratmo, Daniel, & Tjandra, 2019). Application-level caching, distributed caching systems (such as Redis, Memcached), and database query caching all reduce database loading and enhance response time.

Database cache mechanisms, including SQL Server Plan Cache, MySQL Query Cache (deprecated), and PostgreSQL Prepared Statements, cache result sets and execution plans for queries that are executed more than once. Application-level caching, employing libraries like Hibernate Query Cache and Spring Cache, caches query results at the application level to minimize database round-trips.

Caching is problematic with cache invalidation, stale data, and memory overhead (Pizzi, Cepellotti, Sabatini, Marzari, &Kozinsky, 2015). Using smart cache expiry policies like time-based expiration, write-through caching, and event-driven cache invalidation ensures that the cache is current without compromising performance.



Source: Pizzi et al., 2015.

Figure 6 Comparison of execution times for queries using caching and materialized views.

7.4 Using Database Buffers and Result Set Caching

Database engines improve query performance through buffer pools, shared memory, and result set caching. Buffer pool caches frequently accessed disk pages in memory and minimizes disk I/O. SQL databases use Least Recently Used (LRU) page replacement policies to efficiently cache buffer cache (Rivera, Verrelst, Gómez-Dans, Muñoz-Marí, Moreno, & Camps-Valls, 2015).

Result set caching caches query results to be run again subsequently, saving on computation time. Oracle Result Cache, SQL Server Result Set Cache, and MySQL Query Cache (prior to deprecation) use result caching to improve query performance for reporting and analytics queries.

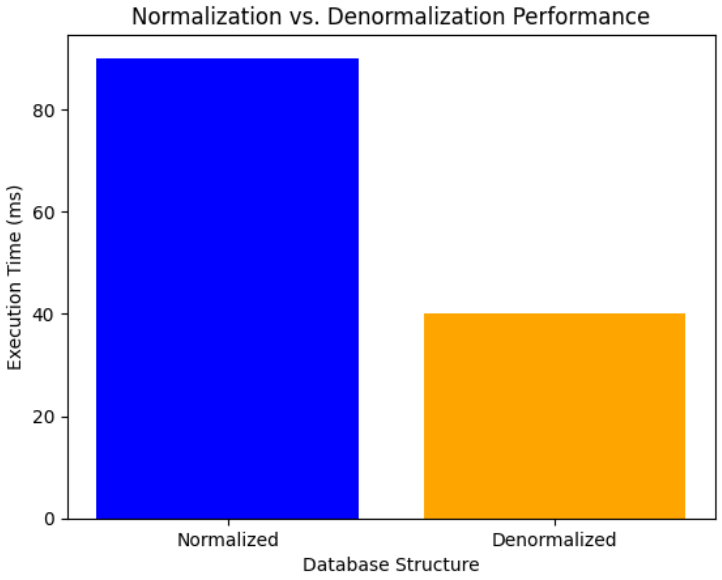
Materialized view and cache optimization needs to be achieved in order to get fast response times for queries, decrease computational cost, and scale high-traffic databases effectively.

IMPACT OF NORMALIZATION AND DENORMALIZATION ON PERFORMANCE

8.1 Normalization and Query Optimization Trade-offs

Normalization is a method of database design that minimizes redundancy by storing data in numerous related tables (Ta-Shma, Akbar, Gerson-Golan, Hadash, Carrez, & Moessner, 2017). Normalization ensures data consistency and conserves storage costs but typically makes queries more complex since multiple joins are involved. SQL databases typically follow the third normal form (3NF) or Boyce-Codd normal form (BCNF) in order to ensure the highest data integrity.

But normalized databases do create performance issues on read-oriented applications, because data is normally accessed by costly join operations. For instance, a very normalized shopping cart database may split customer, order, and product information into distinct tables, with high-cost joins being needed to read user order history.



Source: Ta-Shma et al., 2017.

Figure 7 Execution time differences between normalized and denormalized database structures

8.2 When to Use Denormalization for Performance Gains

Denormalization is to decrease the joins by replication of data, giving good read performance but at higher storage and possible data inconsistency expense.

Denormalization is commonly employed in analytics use and data warehouses, where query speed exceeds storage optimality (Yang, Li, Fang, & Wei, 2020).

One particular example is keeping precomputed overall sales in an orders table rather than calculating aggregates on the fly. Denormalization techniques involve storing duplicate foreign keys, applying summary tables, and joining the often-joined tables to themselves to reduce the execution times of queries.

8.3 Handling Redundant Data and Reducing Joins

Minimizing joins is essential in high-performance environments (Yu, Lakshmanan, & Amer-Yahia, 2009). Methods like precomputed joins in materialized views, indexing on foreign keys, and partitioned algorithms for join reduce the response time of the query at the cost of low redundancy levels.

8.4 Best Practices for Structuring Relational Databases

Denormalization best practices as compared to normalization include caching, materialized view maintenance, optimization of partitioning strategy, and foreign key indexing. The choice would rely upon the application read-write ratio, available storage space, and performance requirement from query executions (Zhang, Chen, Ooi, Tan, & Zhang, 2015).

To maintain the best possible balance between storage space efficiency, data integrity, and query performance, SQL databases function well with mixed workloads.

ADVANCED OPTIMIZATION TECHNIQUES

9.1 Adaptive Query Optimization (AQO)

Adaptive Query Optimization (AQO) is a very sophisticated method enabling contemporary database engines to dynamically fine-tune execution plans in accordance with runtime performance feedback (Abourezq& Idrissi, 2016). Classical SQL query optimization depends on fixed cost estimate models, but AQO brings runtime tuning by looking at execution statistics and tuning upcoming queries. The method is most valuable for complex queries with random data distributions, e.g., queries with skewed joins, parameter-sensitive queries, or changing workload patterns.

A number of databases, such as Oracle (Adaptive Query Optimization), PostgreSQL (Adaptive Parallel Query Execution), and SQL Server (Intelligent Query Processing), have introduced AQO capabilities. For instance, Oracle's AQO has capabilities such as adaptive joins, dynamic collection of statistics, and background re-optimization, where queries adaptively modify execution plans during the course of execution. SQL Server's Batch Mode Adaptive Joins and Memory Grant Feedback also adaptively modify execution parameters in real-time for improved efficiency.

One of the difficulties of AQO is achieving an optimal balance between adaptiveness, overhead, and predictability (Baumann et al., 2015). While AQO improves performance considerably, excessive dynamic adaptation causes unstable fluctuations in the execution plan with a resultant increase in resource contention. Therefore, AQO would be most effective if combined with stable indexing, error-free statistics collection, and tuning policies with workloads taken into account.

9.2 Machine Learning-Based Query Optimization

The integration of machine learning (ML) into query optimization has opened new avenues for improving database performance. Traditional cost-based optimizers rely on heuristic-driven approaches that may not always generalize well across diverse workloads (Baumann, Misev, Merticariu, & Huu, 2021). ML-based query optimization leverages historical execution data, reinforcement learning, and predictive modeling to select optimal execution plans.

Several techniques are being explored in research and industry, including:

- Neural Cost Models: Google’s research on ML-enhanced optimizers proposes deep learning-based cost models that predict query execution times more accurately than traditional estimations.
- AutoIndexing and AutoTuning: SQL Server’s Autonomous Database Tuning and PostgreSQL’s pg_autoindex use ML to recommend index structures dynamically.
- Reinforcement Learning-Based Plan Selection: IBM’s Db2 has experimented with reinforcement learning to adjust execution plans by learning from past query performance.

Despite its advantages, ML-based query optimization faces challenges related to model interpretability, training data availability, and computational overhead. However, with continued advancements, AI-driven optimizers are expected to become standard in next-generation database systems.

9.3 Query Optimization in NoSQL and Hybrid Databases

NoSQL databases, including MongoDB, Cassandra, and DynamoDB, will most probably need alternative optimization techniques from those used by relational databases. Unlike SQL databases, which are very index-dependent, plan-dependent, and cost-estimate-dependent, NoSQL query performance is data model-denormalized dependent, horizontally scaled dependent, and document storage space-efficient dependent (Brahim, Drira, Filali, & Hamdi, 2016).

For instance, MongoDB's Aggregation Framework pipelines queries through optimized stages in order to minimize unnecessary scanning of documents. Likewise, Cassandra utilizes partition-based retrieval, with queries being performed within pre-partitioned partitions for quick lookups. Hybrid databases supporting both SQL and NoSQL paradigms need adaptive optimization strategies that balance relational consistency with scalability offered by NoSQL.

The most important optimization techniques in NoSQL databases are query performance schema design (Briscoe et al., 2014). In contrast to SQL, where applying normalization maintains redundancy low, NoSQL performance is about maintaining related data in documents to maintain cross-collection queries low. Caching, secondary indexes, and query profiling also contribute to optimizing response time.

9.4 Role of AI in Next-Generation Query Optimization

The use of AI in SQL query optimization is rapidly evolving, and AI-powered systems are most likely to optimize and automate multiple parts of database performance tuning (Calheiros, Ranjan, Beloglazov, De Rose, &Buyya, 2010). AI can handle query patterns, usage of indexes, workload shifts, and constraints on hardware resources to create self-optimizing plans for execution. Recent research is also exploring the creation of fully autonomous databases, where AI is learned from past executions history and system telemetry and automatically optimizes. Oracle Autonomous

Database and AWS Aurora Machine Learning are two such cloud-based AI-based systems, which utilize predictive analytics and self-tuning algorithms for dynamically optimizing query execution.
The future of AI-based query optimization is deep reinforcement learning-based optimizers, workload-aware caching, and real-time anomaly detection that allows databases to self-heal, self-optimize, and scale automatically without tuning (Calheiros, Ranjan, Beloglazov, De Rose, &Buyya, 2010).

PERFORMANCE BENCHMARKING AND BEST PRACTICES

10.1 Establishing Query Performance Metrics

Measuring query performance requires establishing standardized performance metrics to evaluate efficiency across different workloads. Key performance indicators (KPIs) include:

- Query Execution Time: Measures the total time taken from query submission to result retrieval.
- CPU and Memory Utilization: Evaluates resource consumption for queries.
- Disk I/O Operations: Tracks the number of read/write operations, impacting query speed.
- Index Usage Rate: Analyzes the effectiveness of indexing strategies.
- Throughput (Queries Per Second - QPS): Measures system capacity for handling concurrent queries.

These metrics are essential for performance benchmarking, query tuning, and workload optimization.

10.2 Benchmarking Tools and Techniques

Several tools are available for benchmarking SQL query performance. Some commonly used benchmarking frameworks include:

Tool	Database Support	Key Features
TPC-H / TPC-C	SQL Databases (MySQL, PostgreSQL, Oracle)	Industry-standard benchmarks for transaction processing and analytical queries.
pg_stat_statements	PostgreSQL	Tracks query execution statistics for optimization.
SQL Server Query Store	SQL Server	Stores query history and performance data for tuning.
MySQL Performance Schema	MySQL	Provides real-time query profiling insights.
Apache JMeter	Multi-Database Support	Simulates concurrent query execution for performance testing.

Benchmarking involves running standardized query sets, analyzing execution plans, and identifying bottlenecks. Regular benchmarking ensures database performance remains optimized under changing workloads (Graefe, 2012).

10.3 Common Pitfalls and Mistakes in Query Optimization

Despite best practices, several common mistakes can degrade SQL query performance, including:

- Lack of Proper Indexing: Missing or redundant indexes can lead to full table scans.
- Unoptimized Joins: Poor join strategies result in high computational costs.
- Ignoring Execution Plans: Failing to analyze query plans leads to inefficient queries.
- **Excessive Use of SELECT ***: Retrieving unnecessary columns increases data transfer time.
- Unoptimized Subqueries: Using correlated subqueries instead of joins leads to performance degradation.

Avoiding these pitfalls requires regular query analysis, execution plan monitoring, and workload-based tuning.

10.4 Future Trends in SQL Query Performance Tuning

The future of SQL performance tuning is shifting toward automation, AI-driven optimizations, and hybrid database architectures (Guo & Engler, 2011). Trends include:

- Self-Tuning Databases: Autonomous databases that optimize queries dynamically.
- AI-Powered Indexing: Machine learning-based index recommendation engines.
- Hybrid SQL-NoSQL Query Optimization: Enhancing query efficiency in mixed database environments.
- Graph-Based Query Execution: Optimizing SQL queries using graph models for complex relationships.

As data volumes continue to grow, next-generation query optimizers will rely on real-time analytics, AI, and distributed computing techniques to achieve ultra-fast data retrieval.

CONCLUSION AND FUTURE WORK

11.1 Summary of Key Findings

This study touched on the various SQL query optimization methods utilized towards better data retrieving performance. From indexing algorithms and partitioning techniques to AI optimized and adaptive question answering, numerous methods decrease latency from queries and improve database performance.

11.2 Challenges and Open Research Areas

Even with the development in SQL optimization, problems like dynamic workload tuning, predictability of query run time, and distributed database performance tuning are still topics for research. Real-time query adaptation and hybrid execution engines are future research areas of interest.

11.3 Future Directions in SQL Query Optimization

Next-generation SQL query optimization will revolve around self-tuning database engines, AI-driven workload balancing, and real-time query adaptation. As workloads become increasingly large, automation and smart optimization will be the hallmark of the next generation of database performance tuning.

REFERENCES

[1]. Abourezq, M., & Idrissi, A. (2016). Database-as-a-Service for Big Data: An Overview. *International Journal of Advanced Computer Science and Applications*, *7*(1). https://doi.org/10.14569/ijacsa.2016.070124

[2]. Baumann, P., Mazzetti, P., Ungar, J., Barbera, R., Barboni, D., Beccati, A., Bigagli, L., Boldrini, E., Bruno, R., Calanducci, A., Campalani, P., Clements, O., Dumitru, A., Grant, M., Herzig, P., Kakaletris, G., Laxton, J., Koltsida, P., Lipskoch, K., . . . Wagner, S. (2015). Big Data Analytics for Earth Sciences: the EarthServer approach. *International Journal of Digital Earth*, *9*(1), 3–29. https://doi.org/10.1080/17538947.2014.1003106

[3]. Baumann, P., Misev, D., Merticariu, V., & Huu, B. P. (2021). Array databases: concepts, standards, implementations. *Journal of Big Data*, *8*(1). https://doi.org/10.1186/s40537-020-00399-2

[4]. SathishkumarChintala, Sandeep Reddy Narani, Madan Mohan Tito Ayyalasomayajula. (2018). Exploring Serverless Security: Identifying Security Risks and Implementing Best Practices. International Journal of Communication Networks and Information Security (IJCNIS), 10(3). Retrieved from https://ijcnis.org/index.php/ijcnis/article/view/7543

[5]. Brahim, M. B., Drira, W., Filali, F., & Hamdi, N. (2016). Spatial data extension for Cassandra NoSQL database. *Journal of Big Data*, *3*(1). https://doi.org/10.1186/s40537-016-0045-4

[6]. Briscoe, B., Brunstrom, A., Petlund, A., Hayes, D., Ros, D., Tsang, I., Gjessing, S., Fairhurst, G., Griwodz, C., & Welzl, M. (2014). Reducing Internet Latency: A survey of techniques and their merits. *IEEE Communications Surveys & Tutorials*, *18*(3), 2149–2196. https://doi.org/10.1109/comst.2014.2375213

[7]. Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. a. F., &Buyya, R. (2010). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software Practice and Experience*, *41*(1), 23–50. https://doi.org/10.1002/spe.995

[8]. Graefe, G. (2012). A survey of B-tree logging and recovery techniques. *ACM Transactions on Database Systems*, *37*(1), 1–35. https://doi.org/10.1145/2109196.2109197

[9]. Sandeep Reddy Narani , Madan Mohan Tito Ayyalasomayajula , SathishkumarChintala, "Strategies For Migrating Large, Mission-Critical Database Workloads To The Cloud", Webology (ISSN: 1735-188X), Volume 15, Number 1, 2018. Available at: https://www.webology.org/data-cms/articles/20240927073200pmWEBOLOBY%2015%20(1)%20-%2026.pdf

[10]. Guo, P. J., & Engler, D. (2011). Using automatic persistent memoization to facilitate data analysis scripting. *SQL Query Optimization Techniques: Speeding up Data Retrieval*. https://doi.org/10.1145/2001420.2001455

[11]. Hor, A., Sohn, G., Claudio, P., Jadidi, M., & Afnan, A. (2018). A SEMANTIC GRAPH DATABASE FOR BIM-GIS INTEGRATED INFORMATION MODEL FOR AN INTELLIGENT URBAN MOBILITY WEB APPLICATION. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *IV–4*, 89–96. https://doi.org/10.5194/isprs-annals-iv-4-89-2018

[12]. Kim, A., Blais, E., Parameswaran, A., Indyk, P., Madden, S., & Rubinfeld, R. (2015). Rapid sampling for visualizations with ordering guarantees. *Proceedings of the VLDB Endowment*, *8*(5), 521–532. https://doi.org/10.14778/2735479.2735485

[13]. Sravan Kumar Pala, "Synthesis, characterization and wound healing imitation of Fe3O4 magnetic nanoparticle grafted by natural products", Texas A&M University - Kingsville ProQuest Dissertations Publishing, 2014. 1572860.Available online at: https://www.proquest.com/openview/636d984c6e4a07d16be2960caa1f30c2/1?pq-origsite=gscholar&cbl=18750

[14]. Credit Risk Modeling with Big Data Analytics: Regulatory Compliance and Data Analytics in Credit Risk Modeling. (2016). International Journal of Transcontinental Discoveries, ISSN: 3006-628X, 3(1), 33-39.Available online at: https://internationaljournals.org/index.php/ijtd/article/view/97

[15]. Klösgen, W., & May, M. (2002). Spatial subgroup mining integrated in an Object-Relational spatial database. In *Lecture notes in computer science* (pp. 275–286). https://doi.org/10.1007/3-540-45681-3_23

[16]. Kotidis, Y., &Roussopoulos, N. (1998). An alternative storage organization for ROLAP aggregate views based on cubetrees. *SQL Query Optimization Techniques: Speeding up Data Retrieval*, 249–258. https://doi.org/10.1145/276304.276327

[17]. Neilson, A., Indratmo, N., Daniel, B., & Tjandra, S. (2019). Systematic Review of the literature on big data in the transportation domain: Concepts and applications. *Big Data Research*, *17*, 35–44. https://doi.org/10.1016/j.bdr.2019.03.001

[18]. Sravan Kumar Pala, "Advance Analytics for Reporting and Creating Dashboards with Tools like SSIS, Visual Analytics and Tableau", *IJOPE*, vol. 5, no. 2, pp. 34–39, Jul. 2017. Available: https://ijope.com/index.php/home/article/view/109

[19]. Pizzi, G., Cepellotti, A., Sabatini, R., Marzari, N., &Kozinsky, B. (2015). AiiDA: automated interactive infrastructure and database for computational science. *Computational Materials Science*, *111*, 218–230. https://doi.org/10.1016/j.commatsci.2015.09.013

[20]. Rivera, J., Verrelst, J., Gómez-Dans, J., Muñoz-Marí, J., Moreno, J., & Camps-Valls, G. (2015). An Emulator Toolbox to Approximate Radiative Transfer Models with Statistical Learning. *Remote Sensing*, *7*(7), 9347–9370. https://doi.org/10.3390/rs70709347

[21]. Goswami, MaloyJyoti. "Utilizing AI for Automated Vulnerability Assessment and Patch Management." EDUZONE,Volume 8, Issue 2, July-December 2019, Available online at: www.eduzonejournal.com

[22]. Ta-Shma, P., Akbar, A., Gerson-Golan, G., Hadash, G., Carrez, F., & Moessner, K. (2017). An ingestion and analytics architecture for IoT applied to smart city use cases. *IEEE Internet of Things Journal*, *5*(2), 765–774. https://doi.org/10.1109/jiot.2017.2722378

[23]. Yang, W., Li, T., Fang, G., & Wei, H. (2020). PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. *SQL Query Optimization Techniques: Speeding up Data Retrieval*. https://doi.org/10.1145/3318464.3386131

[24]. Goswami, MaloyJyoti. "Leveraging AI for Cost Efficiency and Optimized Cloud Resource Management." International Journal of New Media Studies: International Peer Reviewed Scholarly Indexed Journal 7.1 (2020): 21-27.

[25]. Yu, C., Lakshmanan, L., & Amer-Yahia, S. (2009). It takes variety to make a world. *SQL Query Optimization Techniques: Speeding up Data Retrieval*. https://doi.org/10.1145/1516360.1516404

[26]. Zhang, H., Chen, G., Ooi, B. C., Tan, K., & Zhang, M. (2015). In-Memory Big Data Management and Processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *27*(7), 1920–1948. https://doi.org/10.1109/tkde.2015.2427795