Database Optimization

Introduction

- Database Optimization consists of a set of activities and procedures designed to reduce response time of database systems.
- When database is not performing optimally:
    - Check network to assess if it's a traffic issue
    - Check Server resources: Memory, CPU, disk I/O
    - Check database connections
    - Check locks – contention on certain objects
    - Query performance – the focus of rest of this session
- The data on disk is in random order, depending on disk fragmentation, page allocation algorithms, page sizes, DBMS space allocation technique, and other factors.
- The most optimum arrangement of data on disk depends on the manner by which the data is most frequently accessed. Access patterns vary by departments: Finance, payroll, marketing, etc.
    - Recent data may be frequently updated, additional data added on daily basis while older data may not be accessed frequently or accessed as read-only strictly, i.e. organization cannot change history.
- Databases keep in memory most recently accessed data to reduce the need to go to disk. Memory space is limited.
- Reducing disk access operations speeds up the database performance. Less records retrieved from disk to be read and analyzed in order to satisfy a query, translate into better performance.
- Databases create a user process (aka database connection) to manage the client session.
    - Client may be an application or a user running queries on the database.

Locking

- A common case of sudden database performance deterioration
- DBMS applies locks in the process of performing DDL and DML operations.
- Contention results in long wait times for a resource (db object) to be available.
- Deadlocks could occur in certain situations
- The nolock table hint should be provided to the optimizer to refrain from locking the table on reads.
- Situations where the select result may not reliable:
    - **Dirty Read:** Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.
      Dirty Read Example:-
      Transaction A begins.
      UPDATE EMPLOYEE SET SALARY = 10000 WHERE EMP_ID= '123';

Transaction B begins.
SELECT * FROM EMPLOYEE;
(Transaction B sees data which is updated by transaction A. But, those updates have not yet been committed.)

- **Non-Repeatable Read:** Non Repeatable Reads happen when in a same transaction same query yields to a different result. This occurs when one transaction repeatedly retrieves the data, while a difference transactions alters the underlying data. This causes the different or non-repeatable results to be read by the first transaction.
  Non-Repeatable Example:-
  Transaction A begins.
  SELECT * FROM EMPLOYEE WHERE EMP_ID= '123';
  Transaction B begins.
  UPDATE EMPLOYEE SET SALARY = 20000 WHERE EMP_ID= '123';
  (Transaction B updates rows viewed by the transaction A before transaction A commits.)
  If Transaction A issues the same SELECT statement, the results will be different.
- **Phantom Read:** Phantom read occurs where in a transaction execute same query more than once, and the second transaction result set includes rows that were not visible in the first result set. This is caused by another transaction inserting new rows between the execution of the two queries. This is similar to a non-repeatable read, except that the number of rows is changed either by insertion or by deletion.
  Phantom Read Example:-
  Transaction A begins.
  SELECT * FROM EMPLOYEE WHERE SALARY > 10000 ;
  Transaction B begins.
  INSERT INTO EMPLOYEE (EMP_ID, FIRST_NAME, DEPT_ID, SALARY) VALUES ('111', 'Jamie', 10, 35000);
  Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

Indexing

- A separate structure (database object, file) that contains direct pointers to the table rows arranged in a manner to improve the speed of data retrieval when the data is queried by the column(s) on which the index is built.
- Unique / Non unique. A unique index has one pointer to a data row for each index entry while a non-unique index may have multiple points to various data rows for each index entry.
- The primary objective is to enhance the query performance. A unique index may also be used to enforce unique constraints. For example, a unique index on the primary key ensures that a process does not insert a record with a duplicate primary key.
- Indexes may be on a single column or multi-columns (composite index). For example, creating a unique index on the StudentID and TestScore states that one pointer only shall exist in every index entry therefore if a process attempts to insert a second test score for the same student, the transaction will fail due to an index violation.
- Primary Index is the one built on the primary key; therefore, only one exists. Could have many secondary indexes.

- Clustered index is a special type of a Primary index whereby the records of the table are physically ordered by the primary key on which the index is built.
- Overhead and Cost of Indexes:
    - Indexes slow down write operations due to the need to update the index whenever the table is updated.
        - A common strategy is to drop the index before a bulk load and recreate it afterwards if the cost of doing so is less than the speed of inserts with an index in place.
    - Indexes consume disk space.
    - Too many indexes may impact performance as the optimizer has to do more work.
    - Indices have to be rebuilt during the database maintenance window which adds more time to the maintenance processes and uses it more CPU and memory.
- Problem with NULL values
    - A Null value is an undefined attribute/columns value. It's different than blank.
    - Handling varies by DBMS vendor.
    - Null values are typically ignored by the index therefore the entire row is left out.
    - Some DBMS vendors provide special handling, e.g. create index but substitute all NULLs for 0.
    - Avoid allowing NULL values unless it makes sense, e.g. a date field.
- A Covering Index is an index that includes additional columns at the leaf nodes. When include columns exist, the database engine doesn't have to go back to the clustered index (or the tables pages) to retrieve the needed data..

How to choose indexes

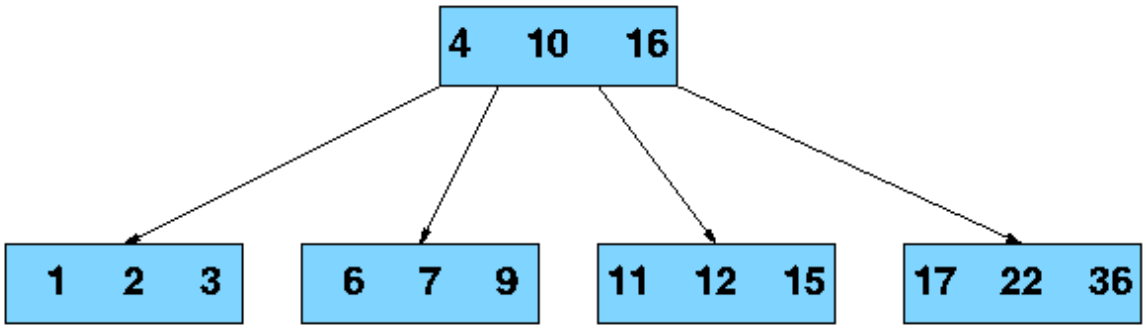
- Do not index small tables, table scans are more efficient for small tables.
- Create an index for the primary key (unless you have a clustered index on the primary key). Some DBMS do that automatically.
- Create an index on frequently used foreign keys. Some DBMS request the identification of foreign keys in the DDL and creates indexes automatically on the foreign keys.
- Create an index on any heavily used secondary key
- Create an index on attributes/columns that are frequently used in GROUP BY and DISTINCT clauses
- Create an index on attributes used in aggregate or built-in functions, e.g. AVG. You may have to build a composite index. For example, if the typical query is to average the salaries by department, it may make more sense to create a composite index on (department, salary)
- Avoid indexing an attribute that is frequently updated as it slows down all insert/update statements.
- Avoid indexing attributes if the most frequent queries will return significant number of the rows, e.g. if the typical query using that attribute as a criteria returns 25% or more of the row then a table scan will be as efficient and cheaper from the standpoint of maintenance.
- Avoid indexes on attributes that consist of long character strings.
- Analyzing effectiveness of indexes may be done using Query Execution Plan (QEP).
- Database stats are updated to assist the optimizer in choosing the most optimum execution plan.

Indexing Algorithms

- Different types: Bitmap, Heap, Hash, Indexed Sequential Access Method (ISAM), B-tree, Cluster.
  - Bitmap index may be logically represented as a grid with 0 or 1 where it's applicable, used for low selectability columns, e.g. M/F gender. This index doesn't consume a lot of space.
  - Hash index uses a hash function that takes the key value as input and produces a hash key as an output value. The hash key value is a bucket that contains a pointer to the data. If multiple keys generate the same hash key value (collision), a list is created inside the bucket of all data pointers. This index is very fast for equality type lookups, e.g. where name = "Scott".
- It's possible to combine different indexing techniques.
- B-tree Index

B-Tree

- The starting point is called the root.
- The ending points are called the leaves. Pointers to the data are stored in leaves.
- Order: max number of children per node (4 for the example below meaning 3 keys per node)
- Depth: The number of levels in the tree.



- Balanced b-tree: Left and right are balanced and depth is the same for all leaf nodes.
- Great fit for database indexes because it can be modeled to reduce disk reads. The height is kept low and the number of keys per node is maximized. Typically, the node is loaded with keys to fill a disk block size (page).

Index density and fill factor

- Density represent how selective is the index, i.e. how many distinct unique values exist for every index entry. For example, an index on student id is less dense than an index on Gender (M/F). A less dense index is a better index.
- Fill Factor determines the percent of the pages at the leaf level to fill leaving the rest of the page for growth. For example, a 75% fill factor means that the pages at the leaf nodes should be filled only to 75% when rebuilding the index.
  - The fill factor setting applies only when the index is created or rebuilt.
  - A well chosen Fill Factor will keep the index from being scattered on disk.
  - Research required to figure out whether most inserts occur at the end of the table or in the middle of the table (in relationship to the index column) in order to determine a good fill factor.

- As inserts and deletes take place, the index may get Skewed which may be determined by the Skew Index that you will find in the metadata tables.
    - A skewed index means parts of the tree are used heavier than other parts.
    - Rebuilding the index takes care of all these problems. A skew index of more than .5 (50%) means index needs to be rebuilt.

Query Processing & Optimization

- Query Processing is the process of transforming a query written in a high level language (e.g. SQL) into an execution strategy expressed in a low level language then executing the strategy to retrieve the data.
- Query processing consists of:
    1) Query decomposition 2) Query optimization 3) Code generation 4) Runtime query execution. The last step is the creation and returning the query result set to the requestor.

    Query Decomposition: 1) transform query into relational algebra query, 2) validate query for syntax and semantics. Is syntax correct? Are the objects valid? Are the operations valid on the identified objects, e.g. where Position = "Manager" AND Position = "Assistant"? the English like SQL query is transformed into a tokenized representation that is more compact and conducive for the next step.

    Query Optimization is the activity of choosing an efficient execution strategy that minimizes resource usage. It's the planning stage for the execution strategy.

        - There are lots of indexes to consider and table scans can be more efficient to satisfy parts of the query. The optimizer decides which parts of the decomposed query should be done using an index and which ones using a table scan.
        - Different techniques:
            - Using heuristics, i.e. rule based, For example, a table scan cost 10 pts, and index scan costs 8 pts, etc.
            - Cost Based Optimization that rely on database statistics. Stats consist of table stats: number of rows, row length, number of columns, number of distinct values in each column, columns with indexes, …; index stats: number and name of columns in the index key, number of key values in the index, number of distinct key values, disk pages used by the index,…; Environment resources: logical and physical disk block size, location and size of data files, … Stats are not always up to date, typically updated by DBA during maintenance window, e.g. runstat in Oracle.
        - Dynamic or Static:
            - Dynamic involves parsing, validating, and optimizing every time the query is processed. Takes longer but ensures most optimum approach.
            - Static – parsing, validation, and optimization occurs once which makes query processing faster and allows for more time to be spent on evaluating best execution strategy. Faster but does not ensure most optimum plan.

    Code generation. The actual code that will be issued to the database to retrieve the data for each portion of the plan.

Runtime Query Execution. The execution of the code on the database and the retrieval of the data to be returned to the client.

- Optimizer hints can be provided to the optimizer on the SQL statement if the user have insights into the data that the optimizer can use to create a better execution plan, i.e. overrides to the execution plan.
  - Query hints. For example, FIRST_ROWS means optimize to get the first few rows as fast as possible while ALL_ROWS means optimized to get all rows as fast as possible.
  - Table hints. For example, NOLOCK means don't lock the table for just reading
  - Index hints, i.e. specify a particular index name to use.
- DBMS have tools to show the execution plan so that it may be adjusted with hints or by adding indexes strategically. In MS SQL Server, it is called "Display Estimated Execution Plan" in the Management studio.
  - Table Scan – means that no indexes are used, table is scanned row by row (slow)
  - Index access – means that an index was used to identify the exact row in question (fast)

Server Side Database Optimization

Parameter settings for:

- Data cache
- SQL cache
- Sort cache
- Optimizer mode

Management of Physical Storage

- Use RAID (Redundant Array of Independent Disks) to provide balance between performance and fault tolerance
- Minimize disk contention. Separate different aspects of the database functions on different drives, i.e. System table space, user data table space, index table space, temp table space, rollback table space.
- Put high-usage tables in their own table spaces
- Assign separate data files in separate storage volumes for indexes, system, and high-usage tables
- Take advantage of table storage organizations in database. Some databases store data row wise and some column wise.
- Partition tables based on usage