Performance Criteria

Hi! Database Performance

Database performance focuses on tuning and optimizing the design, parameters, and physical construction of database objects, specifically tables and indexes, and the files in which their data is stored. The actual composition and structure of database objects must be monitored continually and changed accordingly if the database becomes inefficient. **No amount of SQL tweaking or system tuning can optimize the performance of queries run against a poorly designed or disorganized database.**

Techniques for Optimizing Databases

The DBA must be cognizant of the features of the DBMS in order to apply the proper techniques for optimizing the performance of database structures. Most of the major DBMSs support the following techniques although perhaps by different names. Each of the following techniques can be used to tune database performance and will be discussed in subsequent sections.

- *Partitioning* — breaking a single database table into sections stored in multiple files.
- *Raw partitions versus file systems* — choosing whether to store database data in an OS-controlled file or not.
- *Indexing* — choosing the proper indexes and options to enable efficient queries.
- *Denormalization* — varying from the logical design to achieve better query performance.
- *Clustering* — enforcing the physical sequence of data on disk.
- *Interleaving data* — combining data from multiple tables into a single, sequenced file.
- *Free space* — leaving room for data growth.
- *Compression* — algorithmically reducing storage requirements.
- *File placement and allocation* — putting the right files in the right place.
- *Page size* — using the proper page size for efficient data storage and I/O.

- *Reorganization* — removing inefficiencies from the database by realigning and restructuring database objects.

Partitioning

A database table is a logical manifestation of a set of data that physically resides on computerized storage. **One of the decisions that the DBA must make for every table is how to store that data.** Each DBMS provides different mechanisms that accomplish the same thing — mapping physical files to database tables. The DBA must decide from among the following mapping options for each table:

- *Single table to a single file.* This is, by far, the most common choice. The data in the file is formatted such that the DBMS understands the table structure and every row inserted into that table is stored in the same file. However, this setup is not necessarily the most efficient.
- *Single table to multiple files.* This option is used most often for very large tables or tables requiring data to be physically separated at the storage level. Mapping to multiple files is accomplished by using partitioned tablespaces or by implementing segmented disk devices.
- *Multiple tables to a single file.* This type of mapping is used for small tables such as lookup tables and code tables, and can be more efficient from a disk utilization perspective.

Partitioning helps to accomplish parallelism. *Parallelism* is the process of using multiple tasks to access the database in parallel. A parallel request can be invoked to use multiple, simultaneous read engines for a single SQL statement. Parallelism is desirable because it can substantially reduce the elapsed time for database queries.

Multiple types of parallelism are based on the resources that can be invoked in parallel. For example, a single query can be broken down into multiple requests each utilizing a different CPU engine in parallel. In addition, parallelism can be improved by spreading the work across multiple database instances. Each DBMS offers different levels of support for parallel database queries. To optimize database performance, the DBA should

be cognizant of the support offered in each DBMS being managed and exploit the parallel query capabilities.

Raw Partition vs. File System

For a UNIX-based DBMS environment, the DBA must choose between a raw partition and using the UNIX file system to store the data in the database. A *raw partition* is the preferred type of physical device for database storage because writes are cached by the operating system when a file system is utilized. When writes are buffered by the operating system, the DBMS does not know whether the data has been physically copied to disk or not. When the DBMS cache manager attempts to writes the data to disk, the operating system may delay the write until later because the data may still be in the file system cache. If a failure occurs, data in a database using the file system for storage may not be 100% recoverable. This is to be avoided.

If a raw partition is used instead, the data is written directly from the database cache to disk with no intermediate file system or operating system caching, as shown in Figure 11-1. When the DBMS cache manager writes the data to disk, it will physically be written to disk with no intervention. Additionally, when using a raw partition, the DBMS will ensure that enough space is available and write the allocation pages. When using a file system, the operating system will not preallocate space for database usage.

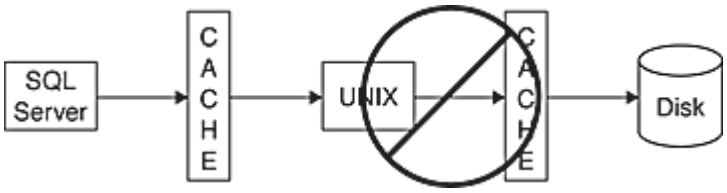

Figure 11-1 *Using raw partitions to avoid file system caching*

From a performance perspective, there is no advantage to having a secondary layer of caching at the file system or operating system level; the DBMS cache is sufficient. Actually, the additional work required to cache the data a second time consumes resources, thereby negatively impacting the overall performance of database operations. **Do not supplement the DBMS cache with any type of additional cache.**

Indexing

Creating the correct indexes on tables in the database is perhaps the single greatest performance tuning technique that a DBA can perform. **Indexes are used to enhance performance.** Indexes are particularly useful for

- Locating rows by value(s) in column(s)
- Making joins more efficient (when the index is defined on the join columns)
- Correlating data across tables
- Aggregating data
- Sorting data to satisfy a query

Without indexes, all access to data in the database would have to be performed by scanning all available rows. Scans are very inefficient for very large tables.

Designing and creating indexes for database tables actually crosses the line between database performance tuning and application performance tuning. Indexes are database objects created by the DBA with database DDL. However, an index is built to make SQL statements in application programs run faster. Indexing as a tuning effort is applied to the database to make applications more efficient when the data access patterns of the application vary from what was anticipated when the database was designed.

Before tuning the database by creating new indexes, be sure to understand the impact of adding an index. The DBA should have an understanding of the access patterns of the table on which the index will be built. Useful information includes the percentage of queries that access rather than update the table, the performance thresholds set within any service level agreements for queries on the table, and the impact of adding a new index to running database utilities such as loads, reorganizations, and recovery.

One of the big unanswered questions of database design is: "How many indexes should be created for a single table?" There is no set answer to this question. The DBA will need to use his expertise to determine the proper number of indexes for each table such that database queries are optimized and the performance of database inserts, updates, and

deletes does not degrade. Determining the proper number of indexes for each table requires in-depth analysis of the database and the applications that access the database.

The general goal of index analysis is to use less I/O to the database to satisfy the queries made against the table. Of course, an index can help some queries and hinder others. Therefore, the DBA must assess the impact of adding an index to all applications and not just tune single queries in a vacuum. This can be an arduous but rewarding task.

An index affects performance positively when fewer I/Os are used to return results to a query. Conversely, an index negatively impacts performance when data is updated and the indexes have to be changed as well. An effective indexing strategy seeks to provide the greatest reduction in I/O with an acceptable level of effort to keep the indexes updated.

Some applications have troublesome queries that require significant tuning to achieve satisfactory performance. Creating an index to support a single query is acceptable if that query is important enough in terms of ROI to the business (or if it is run by your boss or the CEO). If the query is run infrequently, consider creating the index before the process begins and dropping the index when the process is complete.

Whenever you create new indexes, **be sure to thoroughly test the performance of the queries it supports.** Additionally, be sure to test database modification statements to gauge the additional overhead of updating the new indexes. Review the CPU time, elapsed time, and I/O requirements to assure that the indexes help. Keep in mind that tuning is an iterative process, and it may take time and several index tweaks to determine the impact of a change. There are no hard and fast rules for index creation. Experiment with different index combinations and measure the results.

When to Avoid Indexing

There are a few scenarios where indexing may not be a good idea. When tables are very small, say less than ten pages, consider avoiding indexes. Indexed access to a small table can be less efficient than simply scanning all of the rows because reading the index adds I/O requests.

Index I/O notwithstanding, even a small table can sometimes benefit from being indexed — for example, to enforce uniqueness or if most data access retrieves a single row using the primary key.

You may want to avoid indexing variable-length columns if the DBMS in question expands the variable column to the maximum length within the index. Such expansion can cause indexes to consume an inordinate amount of disk space and might be inefficient. However, if variable-length columns are used in SQL WHERE clauses, the cost of disk storage must be compared to the cost of scanning. Buying some extra disk storage is usually cheaper than wasting CPU resources to scan rows. Furthermore, the SQL query might contain alternate predicates that could be indexed instead of the variable-length columns.

Additionally, **avoid indexing any table that is always accessed using a scan**, that is, the SQL issued against the table never supplies a WHERE clause.

Index Overloading

Query performance can be enhanced in certain situations by overloading an index with additional columns. Indexes are typically based on the WHERE clauses of SQL SELECT statements. For example, consider the following SQL statement.

```
select          emp_no,          last_name,          salary
from                                                employee
where             salary               >            15000.00
;
```

Creating an index on the salary column can enhance the performance of this query. However, the DBA can further enhance the performance of the query by overloading the index with the emp_no and last_name columns, as well. With an overloaded index, the DBMS can satisfy the query by using *only* the index. The DBMS need not incur the additional I/O of accessing the table data, since every piece of data that is required by the query exists in the overloaded index.

DBAs should consider overloading indexes to encourage index-only access when multiple queries can benefit from the index or when individual queries are very important.

Denormalization

Another way to optimize the performance of database access is to denormalize the tables. In brief, *denormalization*, the opposite of *normalization*, is the process of putting one fact in many places. This speeds data retrieval at the expense of data modification. Denormalizing tables can be a good decision when a completely normalized design does not perform optimally.

The only reason to ever denormalize a relational database design is to enhance performance. As discussed elsewhere in "Database administration," you should consider the following options:

- *Prejoined tables* — when the cost of joining is prohibitive.
- *Report table* — when specialized critical reports are too costly to generate.
- *Mirror table* — when tables are required concurrently by two types of environments.
- *Split tables* — when distinct groups use different parts of a table.
- *Combined tables* — to consolidate one-to-one or one-to-many relationships into a single table.
- *Speed table* — to support hierarchies like bill-of-materials or reporting structures.
- *Physical denormalization* — to take advantage of specific DBMS characteristics.

You might also consider

- Storing *redundant data* in tables to reduce the number of table joins required.
- Storing *repeating groups* in a row to reduce I/O and possibly disk space.
- Storing *derivable data* to eliminate calculations and costly algorithms.

Clustering

A clustered table will store its rows physically on disk in order by a specified column or columns. Clustering usually is enforced by the DBMS with a clustering index. The clustering index forces table rows to be stored in ascending order by the indexed columns. The left-to-right order of the columns as defined in the index, defines the collating sequence for the clustered index. There can be only one clustering sequence per table (because physically the data can be stored in only one sequence).

Figure 11-2 demonstrates the difference between clustered and unclustered data and indexes; the clustered index is on top, the unclustered index is on the bottom. As you can see, the entries on the leaf pages of the top index are in sequential order — in oher words, they are clustered. Clustering enhances the performance of queries that access data sequentially because fewer I/Os need to be issued to retrieve the same data.

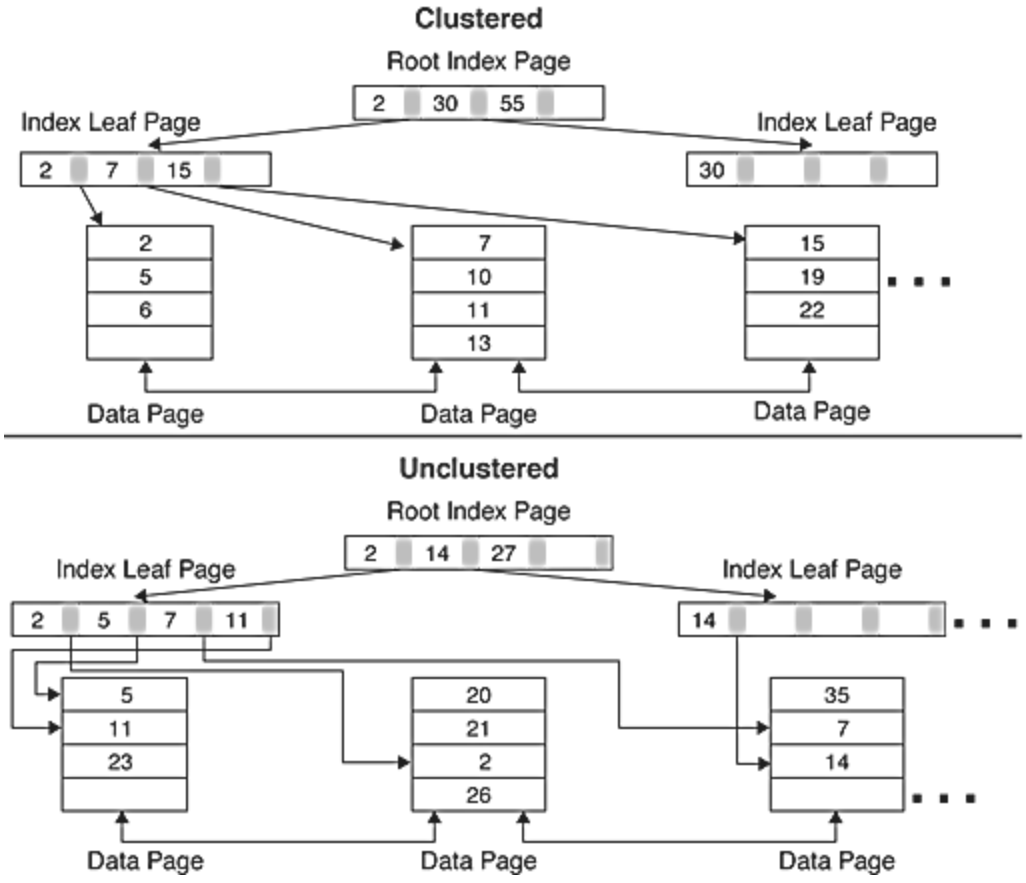

Figure 11-2 *Clustered and unclustered indexes*

Depending on the DBMS, the data may not always be physically maintained in exact clustering sequence. When a clustering sequence has been defined for a table, the DBMS will act in one of two ways to enforce clustering:

1. When new rows are inserted, the DBMS will physically maneuver data rows and pages to fit the new rows into the defined clustering sequence; or
2. When new rows are inserted, the DBMS will try to place the data into the defined clustering sequence, but if space is not available on the required page the data may be placed elsewhere.

The DBA must learn how the DBMS maintains clustering. If the DBMS operates as in the second scenario, data may become unclustered over time and require reorganization. A detailed discussion of database reorganization appears later in this chapter. For now, though, back to our discussion of clustering.

Clustering tables that are accessed sequentially is good practice. In other words, clustered indexes are good for supporting range access, whereas unclustered indexes are better for supporting random access. Be sure to choose the clustering columns wisely. Use clustered indexes for the following situations:

- Join columns, to optimize SQL joins where multiple rows match for one or both tables participating in the join
- Foreign key columns because they are frequently involved in joins and the DBMS accesses foreign key values during declarative referential integrity checking
- Predicates in a WHERE clause
- Range columns
- Columns that do not change often (reduces physically reclustering)
- Columns that are frequently grouped or sorted in SQL statements

In general, the clustering sequence that aids the performance of the most commonly accessed predicates should be used to for clustering. When a table has multiple candidates for clustering, weigh the cost of sorting against the performance gained by clustering for each candidate key. As a rule of thumb, though, if the DBMS supports

clustering, it is usually a good practice to define a clustering index for each table that is created (unless the table is very small).

Clustering is generally not recommended for primary key columns because the primary key is, by definition, unique. However, if ranges of rows frequently are selected and ordered by primary key value, a clustering index may be beneficial.

Page Splitting

When the DBMS has to accommodate inserts, and no space exists, it must create a new page within the database to store the new data. **The process of creating new pages to store inserted data is called *page splitting***. A DBMS can perform two types of page splitting: *normal page splits* and *monotonic page splits*. Some DBMSs support both types of page splitting, while others support only one type. The DBA needs to know how the DBMS implements page splitting in order to optimize the database.

Figure 11-3 depicts a normal page split. To accomplish this, the DBMS performs the following tasks in sequence:

1. Creates a new empty page in between the full page and the next page
2. Takes half of the entries from the full page and moves them to the empty page
3. Adjusts any internal pointers to both pages and inserts the row accordingly

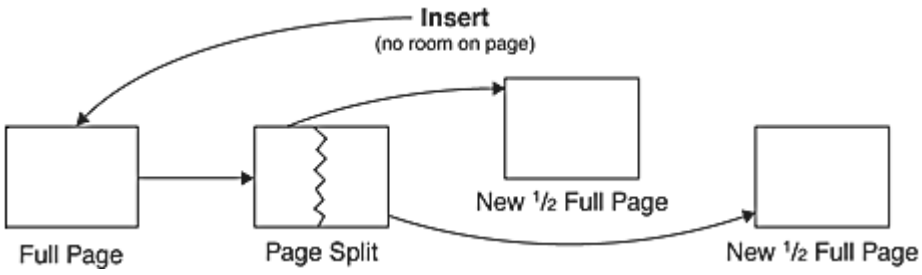

Figure 11-3 *Normal page splitting*

A monotonic page split is a much simpler process, requiring only two steps. The DBMS

- Creates a new page in between the full page and the next page

- Inserts the new values into the fresh page

Monotonic page splits are useful when rows are being inserted in strictly ascending sequence. Typically, a DBMS that supports monotonic page splits will invoke it when a new row is added to the end of a page and the last addition was also to the end of the page.

When ascending rows are inserted and normal page splitting is used, a lot of space can be wasted because the DBMS will be creating half-full pages that never fill up. If the wrong type of page split is performed during database processing, wasted space will ensue, requiring the database object to be reorganized for performance.

Interleaving Data

When data from two tables is frequently joined, it can make sense to physically interleave the data into the same physical storage structure. **Interleaving can be viewed as a specialized form of clustering**

Free Space

Free space, sometimes called *fill factor*, can be used to leave a portion of a tablespace or index empty and available to store newly added data. The specification of free space in a tablespace or index can reduce the frequency of reorganization, reduce contention, and increase the efficiency of insertion. Each DBMS provides a method of specifying free space for a database object in the CREATE and ALTER statements. A typical parameter is PCTFREE, where the DBA specifies the percentage of each data page that should remain available for future inserts. Another possible parameter is FREEPAGE, where the DBA indicates the specified number of pages after which a completely empty page is available.

Ensuring a proper amount of free space for each database object provides the following benefits:

- Inserts are faster when free space is available.
- As new rows are inserted, they can be properly clustered.
- Variable-length rows and altered rows have room to expand, potentially reducing the number of relocated rows.
- Fewer rows on a page results in better concurrency because less data is unavailable to other users when a page is locked.

However, free space also has several disadvantages.

- Disk storage requirements are greater.
- Scans take longer.
- Fewer rows on a page can require more I/O operations to access the requested information.
- Because the number of rows per page decreases, the efficiency of data caching can decrease because fewer rows are retrieved per I/O.

The DBA should monitor free space and ensure that the appropriate amount is defined for each database object. The correct amount of free space must be based on

- Frequency of inserts and modifications
- Amount of sequential versus random access
- Impact of accessing unclustered data
- Type of processing
- Likelihood of row chaining, row migration, and page splits

Don't define a static table with free space — it will not need room in which to expand.

The remaining topics will be discussed in the next lecture.

Compression

Compression can be used to shrink the size of a database. By compressing data, the database requires less disk storage. Some DBMSs provide internal DDL options to compress database files; third-party software is available for those that do not provide such features.

When compression is specified, data is algorithmically compressed upon insertion into the database and decompressed when it is read. Reading and writing compressed data consumes more CPU resources than reading and writing uncompressed data: The DBMS must execute code to compress and decompress the data as users insert, update, and read the data.

So why compress data? Consider an uncompressed table with a row size of 800 bytes. Five of this table's rows would fit in a 4K data page (or block). Now what happens if the data is compressed? Assume that the compression routine achieves 30% compression on average (a very conservative estimate). In that case, the 800-byte row will consume only 560 bytes (800 x 0.30 = 560). After compressing the data, seven rows will fit on a 4K page. Because I/O occurs at the page level, a single I/O will retrieve more data, which will optimize the performance of sequential data scans and increase the likelihood of data residing in the cache because more rows fit on a physical page.

Of course, **compression always requires a trade-off** that the DBA must analyze. On the positive side, we have disk savings and the potential for reducing I/O cost. On the negative side, we have the additional CPU cost required to compress and decompress the data.

However, compression is not an option for every database index or table. For smaller amounts of data, it is possible that a compressed file will be larger than an uncompressed file. This is so because some DBMSs and compression algorithms require an internal dictionary to manage the compression. The dictionary contains statistics about the composition of the data that is being compressed. For a trivial amount of data, the size of the dictionary may be greater than the amount of storage saved by compression.

File Placement and Allocation

The location of the files containing the data for the database can have a significant impact on performance. A database is very I/O intensive, and **the DBA must make every effort to minimize the cost of physical disk reading and writing.**

This discipline entails

- Understanding the access patterns associated with each piece of data in the system
- Placing the data on physical disk devices in such a way as to optimize performance

The first consideration for file placement on disk is to separate the indexes from the data, if possible. Database queries are frequently required to access data from both the table and an index on that table. If both of these files reside on the same disk device, performance degradation is likely. To retrieve data from disk, an arm moves over the surface of the disk to read physical blocks of data on the disk. If a single operation is accessing data from files on the same disk device, latency will occur; reads from one file will have to wait until reads from the other file are processed. Of course, if the DBMS combines the index with the data in the same file, this technique cannot be used.

Another rule for file placement is to analyze the access patterns of your applications and separate the files for tables that are frequently accessed together. The DBA should do this for the same reason he should separate index files from table files.

A final consideration for placing files on separate disk devices occurs when a single table is stored in multiple files (partitioning). It is wise in this case to place each file on a separate disk device to encourage and optimize parallel database operations. If the DBMS can break apart a query to run it in parallel, placing multiple files for partitioned tables on separate disk devices will minimize disk latency.

Database Log Placement

Placing the transaction log on a separate disk device from the actual data allows the DBA to back up the transaction log independently from the database. It also minimizes dual

writes to the same disk. Writing data to two files on the same disk drive at the same time will degrade performance even more than reading data from two files on the same disk drive at the same time. Remember, too, every database modification (write) is recorded on the database transaction log.

Distributed Data Placement

The goal of data placement is to optimize access by reducing contention on physical devices. Within a client/server environment, this goal can be expanded to encompass the optimization of application performance by reducing network transmission costs.

Data should reside at the database server where it is most likely, or most often, to be accessed. For example, Chicago data should reside at the Chicago database server, Los Angeles-specific data should reside at the Los Angeles database server, and so on. If the decision is not so clear-cut (e.g., San Francisco data, if there is no database server in San Francisco), place the data on the database server that is geographically closest to where it will be most frequently accessed (in the case of San Francisco, L.A., not Chicago).

Be sure to take fragmentation, replication, and snapshot tables into account when deciding upon the placement of data in your distributed network.

Disk Allocation

The DBMS may require disk devices to be allocated for database usage. If this is the case, the DBMS will provide commands to initialize physical disk devices. The disk initialization command will associate a logical name for a physical disk partition or OS file. After the disk has been initialized, it is stored in the system catalog and can be used for storing table data.

Before initializing a disk, verify that sufficient space is available on the physical disk device. Likewise, make sure that the device is not already initialized.

Use meaningful device names to facilitate more efficient usage and management of disk devices. For example, it is difficult to misinterpret the usage of a device named DUMP_DEV1 or TEST_DEV7. However, names such as XYZ or A193 are not

particularly useful. Additionally, maintain documentation on initialized devices by saving script files containing the actual initialization commands and diagrams indicating the space allocated by device.

Page Size (Block Size)

Most DBMSs provide the ability to specify a page, or block, size. The *page size* is used to store table rows (or more accurately, records that contain the row contents plus any overhead) on disk. For example, consider a table requiring rows that are 125 bytes in length with 6 additional bytes of overhead. This makes each record 131 bytes long. To store 25 records on a page, the page size would have to be at least 3275 bytes. However, each DBMS requires some amount of page overhead as well, so the practical size will be larger. If page overhead is 20 bytes, then the page size would be 3295 — that is, 3275 + 20 bytes of overhead.

This discussion, however, is simplistic. In general practice, most tablespaces will require some amount of free space to accommodate new data. Therefore, some percentage of free space will need to be factored into the above equation.

To complicate matters, many DBMSs limit the page sizes that can be chosen. For example, DB2 for OS/390 limits page size to 4K, 8K, 16K, or 32K. In this case, the DBA will need to calculate the best page size based on row size, the number of rows per page, and free space requirements.

Consider this question: "In DB2 for OS/390, what page size should be chosen if 0% free space is required and the record size is 2500 bytes?"

The simplistic answer is 4K, but it might not be the best answer. A 4K page would hold one 2500-byte record per page, but an 8K page would hold three 2500-byte records. The 8K page would provide for more efficient I/O, because reading 8K of data would return three rows, whereas reading 8K of data using two 4K pages would return only two rows.

Choosing the proper page size is an important DBA task for optimizing database I/O performance.

Database Reorganization

Relational technology and SQL make data modification easy. Just issue an INSERT, UPDATE, or DELETE statement with the appropriate WHERE clause, and the DBMS takes care of the actual data navigation and modification. In order to provide this level of abstraction, the DBMS handles the physical placement and movement of data on disk. Theoretically, this makes everyone happy. The programmer's interface is simplified, and the RDBMS takes care of the hard part — manipulating the actual placement of data. However, things are not quite that simple. The manner in which the DBMS physically manages data can cause subsequent performance problems.

Every DBA has encountered the situation where a query or application that used to perform well slows down after it has been in production for a while. **These slowdowns have many potential causes** — perhaps the number of transactions issued has increased, or the volume of data has expanded. However, the performance problem might be due to database disorganization. Database disorganization occurs when a database's logical and physical storage allocations contain many scattered areas of storage that are too small, not physically contiguous, or too disorganized to be used productively. Let's review the primary culprits.

- The first possibility is *unclustered data*. If the DBMS does not strictly enforce clustering, a clustered table or index can become unclustered as data is added and changed. If the data becomes significantly unclustered, the DBMS cannot rely on the clustering sequence. Because the data is no longer clustered, queries that were optimized to access data cannot take advantage of the clustering sequence. In this case, the performance of queries run against the unclustered table will suffer.
- *Fragmentation* is a condition in which there are many scattered areas of storage in a database that are too small to be used productively. It results in wasted space, which can hinder performance because additional I/Os are required to retrieve the same data.
- *Row chaining* or *row migration* occurs when updated data does not fit in the space it currently occupies, and the DBMS must find space for the row. With row chaining, the DBMS moves a part of the new, larger row to a location within the

tablespace where free space exists. With row migrations, the full row is placed elsewhere in the tablespace. In each case, a pointer is used to locate either the rest of the row or the full row. Both row chaining and row migration will result in the issuance of multiple I/Os to read a single row. Performance will suffer because multiple I/Os are more expensive than a single I/O.

- *Page splits* can cause disorganized databases, too. If the DBMS performs monotonic page splits when it should perform normal page splits, or vice versa, space may be wasted. When space is wasted, fewer rows exist on each page, causing the DBMS to issue more I/O requests to retrieve data. Therefore, once again, performance suffers.
- *File extents* can negatively impact performance. An *extent* is an additional file that is tied to the original file and can be used only in conjunction with the original file. When the file used by a tablespace runs out of space, an extent is added for the file to expand. However, file extents are not stored contiguously with the original file. As additional extents are added, data requests will need to track the data from extent to extent, and the additional code this requires is unneeded overhead. Resetting the database space requirements and reorganizing can clean up file extents.

Let's take a look at a disorganized tablespace by comparing Figures 11-4 and 11-5. Assume that a tablespace consists of three tables across multiple blocks, such as the tablespace and tables depicted in Figure 11-4. Each box represents a data page.

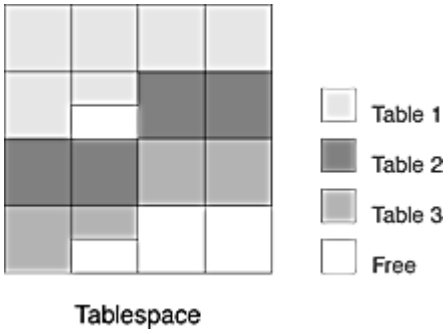

Figure 11-4 *organized tablespace*

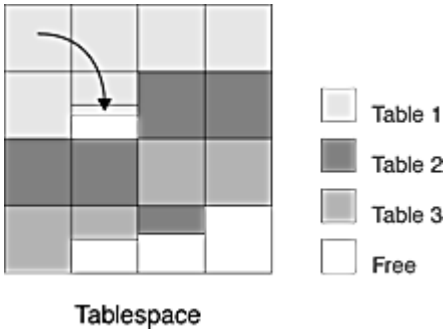

Figure 11-5 *disorganized tablespace*

Now, let's make a couple of changes to the data in these tables. First, we'll add six rows to the second table. However, no free space exists into which these new rows can be stored. How can the rows be added? The DBMS requires an additional extent to be taken into which the new rows can be placed. This result in fragmentation: The new rows have been placed in a noncontiguous space. For the second change, let's update a row in the first table to change a variable-length column; for example, let's change the value in a LASTNAME column from WATSON to BEAUCHAMP. Issuing this update results in an expanded row size because the value for LASTNAME is longer in the new row: "BEAUCHAMP" contains 9 characters whereas "WATSON" only consists of 6. This action results in row chaining. The resultant tablespace shown in Figure 11-5 depicts both the fragmentation and the row chaining.

Depending on the DBMS, there may be additional causes of disorganization. For example, if multiple tables are defined within a tablespace, and one of the tables is dropped, the tablespace may need to be reorganized to reclaim the space.

To correct disorganized database structures, the DBA can run a database or tablespace reorganization utility, or REORG, to force the DBMS to restructure the database object, thus removing problems such as unclustered data, fragmentation, and row chaining. The primary benefit of reorganization is the resulting speed and efficiency of database functions because the data is organized in a more optimal fashion on disk. **In short, reorganization maximizes availability and reliability for databases.**

Tablespaces and indexes both can be reorganized. How the DBA runs a REORG utility depends on the DBMS. Some DBMS products ship with a built-in reorganization utility. Others require the customer to purchase the utility. Still others claim that the customer will not need the utility at all when using their DBMS. I have found the last claim to be untrue. Every DBMS incurs some degree of disorganization as data is added and modified.

Of course, DBAs can manually reorganize a database by completely rebuilding it. However, accomplishing such a reorganization requires a complex series of steps. Figure 11-6 depicts the steps entailed by a manual reorganization.

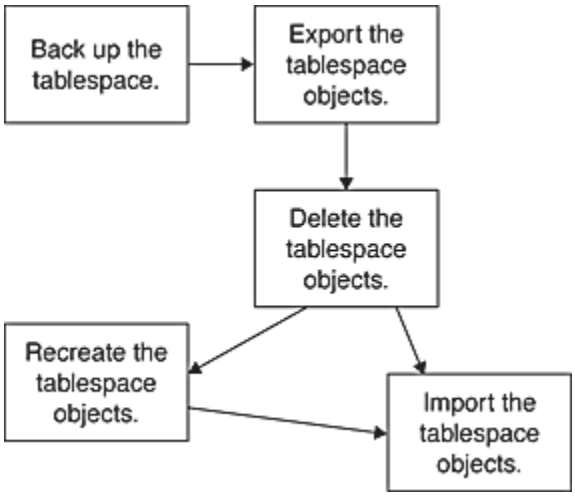

Figure 11-6 *Typical steps for a manual reorganization*

If a utility is available for reorganizing, either from the DBMS vendor or a third=party vendor, the process is greatly simplified. Sometimes the utility is as simple as issuing a simple command such as

```
REORG TABLESPACE TSNAME
```

A traditional reorganization requires the database to be down. The high cost of downtime creates pressures both to perform and to delay preventive maintenance — a no-win situation familiar to most DBAs. Some REORG utilities are available that perform the reorganization while the database is online. Such a reorganization is accomplished by making a copy of the data. The online REORG utility reorganizes the copy while the

original data remains online. When the copied data has been reorganized, an online REORG uses the database log to "catch up" by applying to the copy any data changes that occurred during the process. When the copy has caught up to the original, the online REORG switches the production tablespace from the original to the copy. Performing an online reorganization requires additional disk storage and a slow transaction window. If a large number of transactions occur during the online reorganization, REORG may have a hard time catching up.

Determining When to Reorganize

System catalog statistics can help to determine when to reorganize a database object. Each DBMS provides a method of reading through the contents of the database and recording statistical information about each database object. Depending on the DBMS, this statistical information is stored either in the system catalog or in special pages within the database object itself.

One statistic that can help a DBA determine when to reorganize is *cluster ratio*. **Cluster ratio is the percentage of rows in a table that are actually stored in a clustering sequence.** The closer the cluster ratio is to 100%, the more closely the actual ordering of the rows on the data pages matches the clustering sequence. A low cluster ratio indicates bad clustering, and a reorganization may be required. A low cluster ratio, however, may not be a performance hindrance if the majority of queries access data randomly instead of sequentially.

Tracking down the other causes of disorganization can sometimes be difficult. Some DBMSs gather statistics on fragmentation, row chaining, row migration, space dedicated to dropped objects, and page splits; others do not. Oracle provides a plethora of statistics in dynamic performance tables that can be queried. Refer to the sidebar "Oracle Dynamic Performance Tables" for more details.

Oracle Dynamic Performance Tables

Oracle stores vital performance statistics about the database system in a series of dynamic performance tables. These tables are sometimes referred to as the "V$ tables" because the table names are prefixed with the characters V$.

The V$ tables are used by the built-in Oracle performance monitoring facilities and can be queried by the DBA for insight into the well-being and performance of an Oracle instance. Examples of some of the statistics that can be found in the V$ tables include

- Free space available
- Chained rows
- Rollback segment contention
- Memory usage
- Disk activity

Of course, there is quite a lot of additional performance information to be found in these tables. Oracle DBAs should investigate the V$ tables and query these tables regularly to analyze the performance of the Oracle system, its databases, and applications.

Tablespaces are not the only database objects that can be reorganized. Indexes, too, can benefit from reorganization. As table data is added and modified, the index too must be changed. Such changes can cause the index to become disorganized.

A vital index statistic to monitor is the number of levels. Recall from Chapter 4 that most relational indexes are b-tree structures. As data is added to the index, the number of levels of the b-tree will grow. When more levels exist in the b-tree, more I/O requests are required to move from the top of the index structure to the actual data that must be accessed. Reorganizing an index can cause the index to be better structured and require fewer levels.

Another index statistic to analyze to determine if reorganization is required is the distance between the index leaf pages, or *leaf distance*. Leaf distance is an estimate of the average number of pages between successive leaf pages in the index. Gaps between leaf pages can develop as data is deleted from an index or as a result of page splitting. Of course, the best value for leaf distance is zero, but achieving a leaf distance of zero in practice is not

realistic. In general, the lower this value, the better. Review the value over time to determine a high-water mark for leaf distance that will indicate when indexes should be reorganized.

Review Question

1. Define Techniques for Optimizing Databases

References

http://www.microsoft-accesssolutions.co.uk

http://www.cs.sfu.ca/CC/354