Segregating Data Within Databases for Performance

Prepared by Bill Hulsizer

When designing databases, segregating data within tables is usually important and sometimes very important. The higher the volume of parallel activity, the greater the need for segregating data the way it is being accessed. This reduces locking conflicts and contention. For example, segregating sales data by the store number where the sales occur reduces locking conflicts between processes that update data for a single store. This allows each process to access a separate, segregated area within the table. The same is also true for inquiry-only processes with no locking. Most databases today will break up a single SQL statement into multiple independent statements for retrieving the data in parallel, then merge the multiple results sets together before returning the full results set. This allows the final results set to be produced faster by running each of the independent, partial SQL statements on a separate processor. All of this needs to be taken into consideration, but the bottom line is the online systems with the highest activity need to have good performance during peak usage times. Other high volume processes follow in priority and the database design has to support all of these processes and more. High volume benchmark testing is always the best way to judge overall performance, but in general, data needs to be segregated the way the highest priority processes need it for good performance and throughput. Data is inserted once and deleted once, but between those times it can be accessed millions of times. Therefore, it needs to be segregated for best overall performance, not just for ease of inserting or deleting data.

To be redundant or not to be...

Sometimes data redundancy is good. Fully normalized databases minimize disk space and maintenance, but they do that at the expense of database performance when running SQL statements. This is especially true in data warehouses where "fat" tables are becoming popular again. Redundantly having descriptions or names for codes in a table eliminates the overhead of accessing separate tables to decode the codes into their descriptions or names. The same is true for segregating data for parallel processing. Consider the store situation again. Many times large companies group stores together by their geographic region and/or brand of store. Processes may run for an entire group of stores at once, rather than just one store. In a normalized database, the store group would not be populated on the sales tables. Instead, a store control table would exist with one row per store. Each store row would have a column populated with the store's associated store group. To run a process for all stores within a store group, the store control table would be accessed to determine which stores are in the group and only those stores would be processed. Doing this allows great ease and flexibility in changing the store group to which a store belongs. However, you can pay a long-term price for this flexibility. For example, transferring merchandise between two stores causes a transaction that updates two stores. If both stores are in the same store group and the data is segregated by store group and then store number, then processes running in parallel by store group will not conflict with each other. Each process will only be accessing data within its store group. If parallel processing like this is being attempted, the processes may not work without segregating the data in this way, due to timeouts or deadlocks caused by locking conflicts.

An alternative to this kind of data segregation is physically separating the data into multiple databases. In the example above, you would have one database or separate tables per store group. Although this works, it causes increased complexity and difficulty in retrieving consolidated information from the

multiple databases. An important aspect to realize is that you must know which database or table to access. In this example, you would always need to know which store group you need to access the correct database or table. If you always know the store group, then it can be specified in every SQL statement. If this is true, then having the store group as the first column of all indexes on the table accomplishes the same thing. Having separate databases for each store group means you don't need to have the store group in the data. You can simply put the store group in the database name, table name or table owner. This does save some disk space. However, having one database or table instead of multiple databases or tables means you only have to do one table backup and reorganization, instead of one per store group. It also means consolidated information can be more easily retrieved. Also, over time the number of store groups may increase or decrease due to a variety of reasons like acquisitions, divestitures or internal corporate reorganizations. With one database or table for all store groups, the disk storage requirements may change, but with one database or table per store group, additional databases and their corresponding maintenance need to be set up. This can be a considerable amount of work compared to the work required for just the one database or table.

Traversing database hierarchies

Sometimes processes access and update multiple tables with information. These tables may be part of a hierarchy. For example, consider a daily sales process running for a group of stores. It may update several sales history tables, one each for daily, weekly and monthly sales totals. In addition it may update the store's inventory by subtracting sales quantities from the on-hand quantities for the items sold. Having the input file and all tables being accessed in the same sequence by store group and then store can improve performance and reduce the process' runtime. All items in a store usually are not sold in a single day. However, by having all of the tables and the sales input file in the same sequence, as the input file is processed the tables are essentially accessed in a "skip-sequential" manner. In other words, they are accessed from beginning to end of the process' subset in one pass through them, but only selected rows are accessed and others are "skipped". In cases like this, DB2 will quickly realize the data is being accessed sequentially and turn on sequential prefetch for all of the tables. This means a majority of the data needed will already be prefetched into bufferpools before the process needs it, virtually eliminating I/O wait time for the process.

Another good example of hierarchies of tables is the object-oriented approach. As discussed in more detail later, each row can be assigned a unique, random row ID. Each row in each child table in the hierarchy needs to contain the row ID of its immediate parent row to maintain their relationship. This allows you to retrieve all child rows for a parent row. When the hierarchy contains more than two levels, data segregation using redundancy can help performance and throughput. Assume a three-level hierarchy of vendors, vendor orders and order items with each row having a random, unique row ID. To efficiently retrieve all orders for a vendor, the vendor orders table should be in sequence by vendor row ID and then order row ID. To efficiently retrieve all items for an order, the order items table should be in sequence by order row ID and the item row ID. However, to efficiently retrieve *all* orders and *all* items for a vendor, the order items table should be in sequence by vendor row ID, then order row ID and then item row ID. In this scenario, the vendor row ID is redundantly being populated on the order items table. Logically it's not necessary, but for effective data segregation, it needs to be on the table. The impact of segregating data can be very important. If a vendor generally has 10 orders and their items are not segregated as above, the items will be stored in 10 different locations on the disk drive instead of

one, depending on the value of their order row ID on the vendor orders table. Adding a fourth level to the hierarchy increases the problem exponentially. Assume a fourth level is necessary for color and size information for each item ordered. If this table is sequenced by vendor, order, item and then color/size row IDs, then all data for the vendor can be retrieved from this table in one location on the disk drive. This means all data from all four tables can be retrieved from a total of 4 locations on the disk drive. If it is sequenced only by the item and then color/size row IDs, each item could be physically stored in a separate location on the disk drive. If each order has an average of 10 items, each with 10 color/size combinations, then 100 more locations on the disk drive could have to be accessed to retrieve the information. This is a total of 112 locations, instead of four (1 for the vendor, 1 for the vendor orders, 10 for the vendor items and 100 for the item colors/sizes) or 28 times the work necessary simply by segregating the data. This could also continue to a fifth level to handle multiple receipts for each item color/size received from the vendor. If an average of 2 receipts per item color/size are received, another 200 disk locations are added instead of one additional. Processes that walk through a multi-level hierarchy like this can be greatly improved by data segregation and the only real cost is some minor extra disk space. In this case, the vendor row ID needs to be redundantly populated on the order items and item color/size tables and the vendor order row ID needs to be populated on the item color/size table. The disk space to hold this redundant data should be insignificant compared to the total disk space being used by the tables and all of their indexes.

Inter-table vs. Intra-table redundancy

Data redundancy comes in two flavors, inter-table and intra-table redundancy. Inter-table redundancy is what was just described above, where multiple tables are involved. Intra-table redundancy refers to redundancy within a table, specifically within its indexes to achieve index-only access. For example, many applications have a table of customers. Customers are generally assigned a customer identification number or ID. Since these IDs need to be unique, the customer table needs an index on this ID. The data is usually physically stored in this sequence to facilitate accessing other customer-related tables in a hierarchy, like customer invoices and invoice payments. Most customer-oriented applications allow a name search on customer names and have an index on the customer's name to support this. By specifying a range of names or a starting name and a maximum number of customers to retrieve, this index can dramatically improve response time. However, the name search application may also display additional information from the customer table like their phone number or address. If the index on customer name only contains this one column, then the additional information needs to be retrieved from the data, not the index. Since the index is in customer name sequence and the data is probably in customer ID sequence, each customer's data is probably in a different data page. Therefore, to list 15 customers on a panel with their phone number, one or two index pages and 15 data pages will probably need to be accessed. However, if the additional customer information needed by the name search application is redundantly added to the end of the index as additional index columns, then all of the information needed is in the index. The data pages will not need to be accessed to retrieve this information. It will be in the index and can be accessed in one or two index pages. Assuming the worst-case scenario that two index pages are accessed, having this redundant information reduces the work being done by a 17 to 2 ratio. This reduction in cpu processing more than pays for the very minor additional disk space needed for the redundant information in the index.

Identity columns

For several reasons, identity columns should be avoided, if possible. The primary reason is because as rows are inserted into the table, rows get inserted into the index that guarantees their uniqueness. Because the generated number increases sequentially, all of these rows are inserted at the end of this index. Although this is good for reducing page splits, it causes a hot spot of activity at the end of the index. This hot spot quickly becomes more severe as more insert processes are run in parallel. Also, most databases do not guarantee that numbers cannot or will not be skipped when generating the next sequential number. Many applications that use auto-generated numbers really just need a unique number and not necessarily a sequential number. Unique random numbers can be generated by getting the current date and time and reversing the digits. DB2 provides the time to six decimal places after the seconds in the format hh:mm:ss.nnnnnn, so this is definitely a random number when it is reversed. However, two inserts can happen at the exact same time. Several options for handling this are available, including re-retrieving the current time and re-populating the unique number or simply adding 1 to the last digit of the time and reattempting the insert. Another option is to have a one digit number concatenated to the end of the unique number, populate it with zero initially and add one to this number if a duplicate occurs. This method has the added benefit of creating an audit trail of how often duplicates occur. You can simply run a scan for any non-zero number in this extra digit.

Using random numbers spreads the inserts out randomly within the index that enforces their uniqueness, eliminating the hot spot of activity during inserts. If rows are being deleted as frequently as they're being inserted, the freespace left from deleting rows will provide adequate space for the newly inserted rows. If the table is growing in size, then freespace will need to be allocated, but this is evenly spread throughout the index. As for all tables that are growing in size, table and index reorganizations will need to be run at appropriate intervals. If sequential numbers really are required, hopefully there can be, for example, a unique number for each store group or some other control group like store, cost center, etc.. This information should be stored in a control table used just for this purpose, to reduce locking conflicts caused by updates to other columns. If these numbers are updated frequently, locking conflicts can occur between parallel processes. For example, if sequential control numbers are maintained for each store and 100 stores exist, then all 100 stores could exist within the same data and index pages. Page level locking has much less overhead than row level locking, so row level locking should always be avoided. To eliminate the potential for locking conflicts, this table and all of its indexes should each have 99% freespace allocated within them. If the rows are so small that multiple rows are still stored within each page, add a column to each table and index that is large enough to force one row per page in the data and all index pages. This will completely eliminate all locking conflicts between processes updating different rows. However, locking conflicts can still occur between multiple processes attempting to update the same row. For this reason, updating a control number like this should be done in its own logical unit of work. The program doing this update should first issue a commit to release all currently held locks (if any), select the current sequential number, add one to it, update the row and then issue another commit to release the locked data and index rows. This minimizes the amount of time the row is locked, allowing much greater concurrency and reduced locking conflicts. In version 8 of DB2 for z/OS, "sequences" are introduced for this exact purpose. After creating a sequence, it can be incremented by referring to its next or previous value when populating data in columns. This isolates control of the next number to within DB2, which can further reduce locking conflicts.

Joining from a control table

Another very good reason for segregating data is when a control table is used to join to another table. For example, an SQL statement can specify a store group, access a store control table to determine the stores in the group and then join to a second table to retrieve more detailed information for just the necessary stores. Databases do this by retrieving a list of stores from the store control table for the store group. They then process each store in the list separately by joining to the table with detailed information once per store. If the data in this table is segregated by store number, then just that store's data is accessed during this join. If the stores are intermingled in the data, then the data for all stores will processed each time the join is done. If a store group contains 10 stores, then data for all 10 stores will be searched while retrieving data for each store. This means 10 times the work is being done. The more stores in the store group, the more unnecessary work is being done. If 100 stores are in the store group, then 100 times the work is unnecessarily being done.

Mass inserts and deletes

Many times data gets segregated to make it easier to do mass inserts and deletes of data. Yes, it's easier to do one large load of daily data, rather than split the input file the way the database is segregated and do multiple loads into the table in parallel. But in the big picture, is that what you really want to do? Let's use a banking application that lists transactions for a checking account as an example. New data arrives every day and gets loaded to the database in mass using a load utility. To make it easier to load new data and delete the oldest data, the table is partitioned by transaction date. Because of the way the table is partitioned, multiple loads cannot be run in parallel because they would all try to access the same data area at the same time. This means very little can be done to decrease the time it takes to load the data, even if loading the data becomes a bottleneck process. When users list the transactions for an account, they are listed in date sequence. Assuming an average of 22 business days per month and one transaction every other day, an average of 11 transactions will be listed. If these transactions all happened on a different day, then one page from 11 different data and index partitions needs to be accessed to list the transactions. If the data were segregated the way it is needed for this application, only one page from one data and one index partition would have to be accessed on average. This is an 11 to 1 reduction in the amount of work being done. Obviously this is a better way to segregate the data for this application. If this is the highest priority application that uses the data and loading and deleting data, as well as all other processes, can still be accomplished within reasonable timeframes, then this is the way the data should be segregated. However, this not only allows, but requires parallel loads and deletes to meet a reasonable timeframe. It also requires a multi-processing environment with more than just a few processors available. A highly tuned process can run even the fastest mainframe processor at close to 100% capacity if enough disk activity can be eliminated. Although loading data is a disk intensive process, everything is buffered very well. Generally, no more than two loads should be run in parallel per processor available, but benchmarking will give the best final answer. Similarly, no more than two delete processes should be run in parallel on a processor and these processes need to commit the deletes frequently to reduce the total locks maintained by the lock manager. For parallel loads and deletes, each process needs to be accessing its own independent disk area, to avoid disk or lock contention. In a mature table with as many rows being deleted as there are being inserted, the space from deleted rows should be adequate to fulfill the space needed by rows being inserted. In tables that are growing, some freespace will need to be allocated in the data and indexes. Parallel table

reorganizations will need to be implemented, too. In mainframe environments, the number of processors available can be restrictive, especially if only two or three processors are available on the system. In distributed systems where 64, 128 or more processors can be available on a system, massive parallel processing can be done, as long as the processes remain independent of each other.

Summary

Data normalization can minimize disk space and maintenance for a database, but many times good performance and throughput require denormalizing to some degree. Segregation of data is a good reason for denormalizing, if it accomplishes the performance and throughput desired with minimal adverse long-term impact. Effective segregation can require specific SQL coding requirements, so segregation issues not only need to be addressed, but done so very early in the system design phase to avoid rework having to change and retest functioning SQL to gain the improvements segregation can provide.