Conversation

Submitted by alok on Tue, 06/30/2015 - 00:42

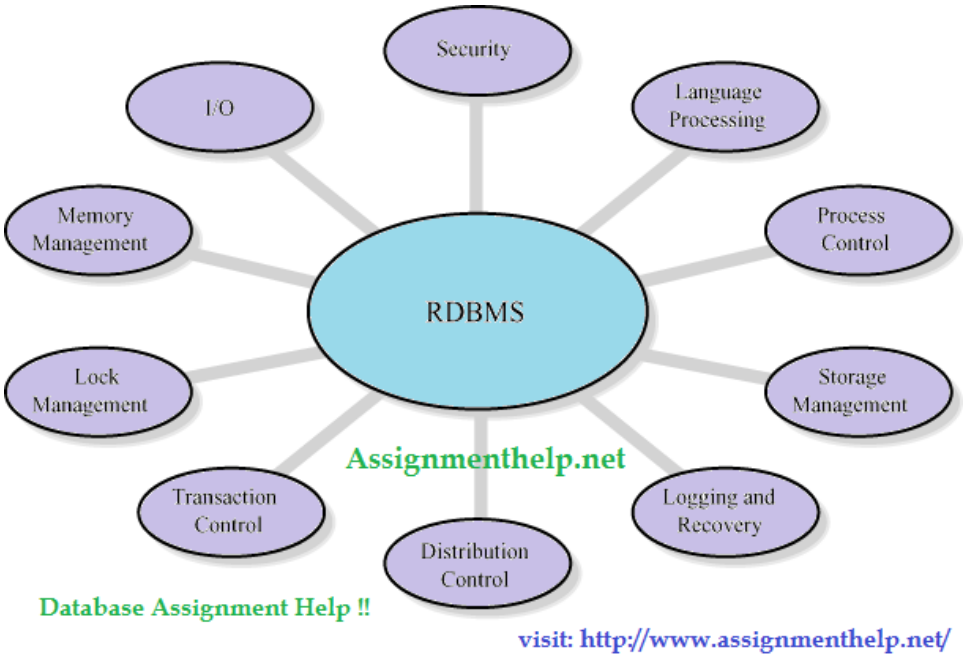
Background[edit]

A relational database management system provides data that represents a two-dimensional table, of coluns and rows. For example, a database might have this table:

RowId	EmpId	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	55000

This simple table includes an employee identifier (EmpId), name fields (Lastname and Firstname) and a salary (Salary). This two-dimensional format exists only in theory, in practice, storage hardware requires the data to be serialized into one form or another.

The most expensive operations involving hard disks are seeks. In order to improve overall performance, related data should be stored in a fashion to minimize the number of seeks. This is known as locality of reference, and the basic concept appears in a number of different contexts. Hard disks are organized into a series of blocks of a fixed size, typically enough to store several rows of the table. By organizing the data so rows fit within the blocks, and related rows are grouped together, the number of blocks that need to be read or sought is minimized.



Row-oriented systems[edit]

The common solution to the storage problem is to serialize each row of data, like this;

```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```

As data is inserted into the table, it is assigned an internal ID, the `rowid` that is used internally in the system to refer to data. In this case the records have sequential rowids independent of the user-assigned empid. In this example, the DBMS uses short integers to store rowids, in practice larger numbers, 64-bit or 128-bit, are normally used.

Row-based systems are designed to efficiently return data for an entire row, or record, in as few operations as possible. This matches the common use-case where the system is attempting to retrieve information about a particular object, say the contact information for a user in a rolodex system, or product information for an online shopping system. By storing the record's data in a single block on the disk, along with related records, the system can quickly retrieve records with a minimum of disk operations.

Row-based systems are not efficient at performing operations that apply to the entire data set, as opposed to a specific record. For instance, in order to find all the records in the example table that have salaries between 40,000 and 50,000, the DBMS would have to seek through the entire data set looking for matching records. While the example table shown above will likely fit in a single disk block, a table with even a few hundred rows would not, and multiple disk operations would be needed to retrieve the data and examine it.

To improve the performance of these sorts of operations, most DBMSs support the use of database indexes, which store all the values from a set of columns along with pointers back into the original rowid. An index on the salary column would look something like this:

```
001:40000;
002:50000;
003:44000;
004:55000;
```

As they store only single pieces of data, rather than entire rows, indexes are generally much smaller than the main table stores. By scanning smaller sets of data the number of disk operations is reduced. If the index is heavily used, it can provide dramatic time savings for common operations. However, maintaining indexes adds overhead to the system, especially when new data is written to the database. In this case not only is the record stored in the main table, but any attached indexes have to be updated as well.

Database indexes on one or more columns are typically sorted by value, which makes operations like range queries fast.

There are a number of row-oriented databases that are designed to fit entirely in RAM, an in-memory database. These systems do not depend on disk operations, and have equal-time access to the entire dataset. This reduces the need for indexes, as it requires the same amount of operations to full scan the original data as a complete index for typical aggregation purposes. Such systems may be therefore simpler

and smaller, but can only manage databases that will fit in memory.

Column-oriented systems[edit]

A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. For our example table, the data would be stored in this fashion:

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

In this layout, any one of the columns more closely matches the structure of an index in a row-based system. This may cause confusion that can lead to the mistaken belief a column-oriented store "is really just" a row-store with an index on every column. However, it is the mapping of the data that differs dramatically. In a row-oriented indexed system, the primary key is the rowid that is mapped to indexed data. In the column-oriented system, the primary key is the data, mapping back to rowids.[3] This may seem subtle, but the difference can be seen in this common modification to the same store:

```
…;Smith:001,Jones:002,004;Johnson:003;…
```

As two of the records store the same value, "Jones", it is possible to store this only once in the column store, along with pointers to all of the rows that match it. For many common searches, like "find all the people with the last name Jones", the answer is retrieved in a single operation. Other operations, like counting the number of matching records or performing math over a set of data, can be greatly improved through this organization.

Whether or not a column-oriented system will be more efficient in operation depends heavily on the workload being automated. It would appear that operations that retrieve data for objects would be slower, requiring numerous disk operations to collect data from multiple columns to build up the record. However, these whole-row operations are generally rare. In the majority of cases, only a limited subset of data is retrieved. In a rolodex application, for instance, operations collecting the first and last names from many rows in order to build a list of contacts is far more common than operations reading the data for any single address. This is even more true for writing data into the database, especially if the data tends to be "sparse" with many optional columns. For this reason, column stores have demonstrated excellent real-world performance in spite of any theoretical disadvantages.[4]

This is a simplification. Moreover, partitioning, indexing, caching, views, OLAP cubes, and transactional systems such as write ahead logging or multiversion concurrency controlall dramatically affect the physical organization of either system. That said, online transaction processing (OLTP)-focused RDBMS systems are more row-oriented, while online analytical processing (OLAP)-focused systems are a balance of row-oriented and column-oriented.

Benefits[edit]

Comparisons between row-oriented and column-oriented data layouts are typically concerned with the efficiency of hard-disk access for a given workload, as seek time is incredibly long compared to the other delays in computers. Sometimes, reading a megabyte of sequentially stored data takes no more time than

one random access.[5] Further, because seek time is improving much more slowly than CPU power (see Moore's Law), this focus will likely continue on systems that rely on hard disks for storage. Following is a set of oversimplified observations which attempt to paint a picture of the trade-offs between column- and row-oriented organizations. Unless, of course, the application can be reasonably assured to fit most/all data into memory, in which case huge optimizations are available from in-memory database systems.

1. Column-oriented organizations are more efficient when an aggregate needs to be computed over many rows but only for a notably smaller subset of all columns of data, because reading that smaller subset of data can be faster than reading all data.
2. Column-oriented organizations are more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows.
3. Row-oriented organizations are more efficient when many columns of a single row are required at the same time, and when row-size is relatively small, as the entire row can be retrieved with a single disk seek.
4. Row-oriented organizations are more efficient when writing a new row if all of the row data is supplied at the same time, as the entire row can be written with a single disk seek.

In practice, row-oriented storage layouts are well-suited for OLTP-like workloads which are more heavily loaded with interactive transactions. Column-oriented storage layouts are well-suited for OLAP-like workloads (e.g., data warehouses) which typically involve a smaller number of highly complex queries over all data (possibly terabytes).

Compression[edit]

Column data is of uniform type; therefore, there are some opportunities for storage size optimizations available in column-oriented data that are not available in row-oriented data. For example, many popular modern compression schemes, such as LZW or run-length encoding, make use of the similarity of adjacent data to compress. While the same techniques may be used on row-oriented data, a typical implementation will achieve less effective results.[6][7]

To improve compression, sorting rows can also help. For example, using bitmap indexes, sorting can improve compression by an order of magnitude.[8] To maximize the compression benefits of the lexicographical order with respect to run-length encoding, it is best to use low-cardinality columns as the first sort keys.[9] For example, given a table with columns sex, age, name, it would be best to sort first on the value sex (cardinality of two), then age (cardinality of <150), then name.

Columnar compression achieves a reduction in disk space at the expense of efficiency of retrieval. Retrieving all data from a single row is more efficient when that data is located in a single location, such as in a row-oriented architecture. Further, the greater adjacent compression achieved, the more difficult random-access may become, as data might need to be uncompressed to be read. Therefore, column-oriented architectures are sometimes enriched by additional mechanisms aimed at minimizing the need for access to compressed data.[10]
