Module 3: Vertical Partitioning

Motivation for Vertical Fragmentation

In the previous module, we explored horizontal fragmentation to achieve locality of reference for users and applications that have different needs for different rows of a table based on the predicates of their queries. In this module, we explore vertical fragmentation for users and applications that have different needs for different table columns based on their queries and access patterns.

Vertical fragmentation partitions relation R into fragments R_1, R_2, … . Each fragment contains some attributes of R, plus the primary key (or an alternate identifier) to enable the reconstruction of R from its fragments.

Vertical fragmentation has been explored in single databases to make queries more efficient. Databases structure table data in formatted files. The files are formatted into data blocks of fixed size, each of which stores multiple rows of table records. These pages are stored on disk. During database operations (read, write, update, and delete), the database management system (DBMS) reads the data from disk pages to the applications. The DBMS looks for the disk page that stores the data of interest and, on the page, looks for the row of interest to return to the application.

It is common for applications to read multiple rows that are stored adjacently on one or more disk pages. If an application read multiple rows from disk, it must perform a disk read for each row. Since disk access is relatively slow, reading multiple rows from one or more disk pages will result in poor application performance.

Cache memory is used to speed the process of data access. Cache usually consists of a limited amount of more expensive RAM or Flash memory. It provides faster access than disk. When a row is accessed, the DBMS returns the entire page containing the row into local cache memory. Then, all subsequent reads to data located on that page are read from cache. When an application is finished with that page, it will read the next page from disk into cache.

The number of rows per memory page is dependent on the size of the row. Memory pages that store longer rows (more bytes per row) contain fewer rows per page. Memory pages that store shorter rows (fewer bytes per row) contain more rows per page. Suppose that the row size is such that only 10 rows can be stored per disk page for a given table. This means that an application will require 1 slow disk read for every 10 fast cache reads.

To speed up performance, we would like to find a way to increase the number of cache reads per each disk read. Vertical fragmentation is one method of achieving this performance increase. If we store only the subset of columns we are interested in one vertical fragment, while the subset of columns we are not interested in for this query are stored in a different fragment, we can read from the fragment that contains the columns that we are interested in. Since this fragment contains only a subset of columns of the original table, the number of rows stored per page will increase. That means that the memory page storing rows of this fragment can hold many rows than the original page. Thus, when an application reads the page into

cache, it will require 1 slow disk read for many more than 10 fast cache reads – a significant increase in performance!

Complexity of Vertical Fragmentation

The number of possible vertical fragmentations grows significantly with the number of columns in a table.

(Watch Module 3A video – Complexity of Vertical Fragmentation)

How do we deal with the so many possibilities? It takes too long to find the most optimal solution by exhaustive search through all the possibilities. Instead, we rely on *heuristics* that explore a reasonable subset of solutions to find a reasonable solution in reasonable time.

There are two major classes of fragmentation heuristics. One is a bottom up approach in which each attribute starts with its own fragment and we *group* them together based on some criteria. The other is a top down approach in which we start with a compete relation and we *split* them based on some criteria.

We will examine the top-down solution that splits the complete relation. It is more natural because we expect optimal performance when many attributes are grouped together from the beginning. Furthermore, it is hard to manage many single-column fragments.

Information Requirements for Vertical Fragmentation

To perform vertical fragmentation, it is important to know how applications access the data. Geographically distributed users may access different groups of columns differently. The column groupings give us an understanding of column *affinity* or togetherness.

Determining Attribute Usage

Affinity is hard to determine without careful analysis. It is important to understand the set of queries, the columns they reference, and the frequency of access to identify affinity groups of columns for vertical fragmentation.

We define $Q = \{q_1, q_2, \ldots, q_q\}$ to be the set of queries to relation R. Relation $R = \{A_1, A_2, \ldots, A_n\}$ is composed of a set of attributes A_i. It is this relation R that we want to fragment into smaller partitions; each partition containing a subset of attributes A_i.

We further define an attribute usage function **use**$(q_i, A_j) = 1$ if A_j is used in query q_i. The set of attributes for which the **use**(q_i, A_j) function equals 1 corresponds to the attributes of the SELECT list and of the WHERE clause predicates of q_i.

Suppose we are given a relation **proj** that contains the attributes **pid, pno**, **budget**, **pname**, and **loc**. The relation is represented as: proj = {pid, pno, pname, budget, loc}.

Furthermore, suppose we have the following four queries defined on table **proj**:

```
q1:   SELECT   budget
      FROM     proj
      WHERE    pno = value

q2:   SELECT   pname, budget
      FROM     proj

q3:   SELECT   pname
      FROM     proj
      WHERE    loc = value

q4:   SELECT   sum(budget)
      FROM     proj
      WHERE    loc = value
```

For each query, we see that:

- use(q_1, pno) = 1, use(q_1, pname) = 0, use(q_1, budget) = 1, use(q_1, loc) = 0
- use(q_2, pno) = 0, use(q_2, pname) = 1, use(q_2, budget) = 1, use(q_2, loc) = 0
- use(q_3, pno) = 0, use(q_3, pname) = 1, use(q_3, budget) = 0, use(q_3, loc) = 1
- use(q_4, pno) = 0, use(q_4, pname) = 0, use(q_4, budget) = 1, use(q_4, loc) = 1

Let us represent this in an **Attribute Usage Matrix** by just transferring the values of the above use equations into a matrix:

	A1 (pno)	A2 (pname)	A3 (budget)	A4 (loc)
q_1	1	0	1	0
q_2	0	1	1	0
q_3	0	1	0	1
q_4	0	0	1	1

Important => In general, we assume that there is a column that serves as the primary key of the table. In this case, the column is **pid**. We do not include the primary key of the table in the Attribute Usage Matrix because the vertical fragmentation algorithm places the primary key in each fragment to ensure that we can rejoin the fragments. Thus, when performing vertical fragmentation, we do not consider primary keys in our calculations.

Determining Affinity

The attribute use matrix does not help us yet. We cannot determine the affinity of the attributes because we don't know the access frequency of the attribute groups by different users. We need this to calculate attribute affinity - **aff**(A_i, A_j). the affinity is

how often A_i and A_j are accessed together by individual queries. It is dependent upon the frequency of queries that request attributes A_i and A_j simultaneously.

Let $\mathbf{acc_j(q_i)}$ be the number of times per hour that site j runs query i.

- for q_1: $acc_1(q_1) = 15$, $acc_2(q_1) = 20$, $acc_3(q_1) = 10$, $acc_{total}(q_1) = 45$
- for q_2: $acc_1(q_2) = 5$, $acc_2(q_2) = 0$, $acc_3(q_2) = 0$, $acc_{total}(q_2) = 5$
- for q_3: $acc_1(q_3) = 25$, $acc_2(q_3) = 25$, $acc_3(q_3) = 25$, $acc_{total}(q_3) = 75$
- for q_4: $acc_1(q_4) = 3$, $acc_2(q_4) = 0$, $acc_3(q_4) = 0$, $acc_{total}(q_4) = 3$

This is represented in the following matrix:

	acc_1	acc_2	acc_3	acc_{total}
q_1	15	20	10	45
q_2	5	0	0	5
q_3	25	25	25	75
q_4	3	0	0	3

Now, we will create the **Attribute Affinity Matrix** by looking at all attributes pairwise and determining the rate of access per site per pair. The rate of access per pair is how many times per some period of time (e.g., hour) each pair is accessed together for some site. Then, we sum all the rates together from each site. For example, attributes A_2 and A_2 (the same attribute) are accessed together in only q_2 and q_3 (from the Attribute Usage Matrix) with rates of 5 and 75 times per hour, respectively (from the matrix above). Thus, the total affinity of A_2 and A_2 is 80 times per hour. Similarly, attributes A_3 and A_4 are accessed together in only q_4 (from the Attribute Usage Matrix) with a rate of 3 times per hour (from the matrix above). Calculating all these attribute affinities in the same manner, we get the following Attribute Affinity Matrix:

	A1 (pno)	A2 (pname)	A3 (budget)	A4 (loc)
A_1	45	0	45	0
A_2	0	80	5	75
A_3	45	5	53	3
A_4	0	75	3	78

Clustering Algorithm

In its raw form, the Attribute Affinity Matrix does not seem to be helpful to us. We really want to reorder the attributes to see which ones cluster well together. This clustering will drive the vertical fragmentation process. If we reorder both columns and rows A_2 and A_3, we have the following Clustered Affinity Matrix:

	A1 (pno)	A3 (budget)	A2 (pname)	A4 (loc)
A_1	45	45	0	0
A_3	45	53	5	3
A_2	0	5	80	75
A_4	0	3	75	78

Notice that attributes A_1 and A_3 as well as A_2 and A_4 cluster well together. This is because there are blocks of relatively high numbers of accesses per hour that are clustered together. The other blocks have a relatively low number of accesses per hour. Examining this by eye may be sufficient for small numbers of columns. We need an automated algorithm to identify clusters for large numbers of columns.

Automating the Process

We want an automatic algorithm for clustering an affinity matrix. The algorithm should cluster columns according to similar usage and should be computable in $O(n^2)$, where n is number of columns. The following describes such an algorithm:

(Watch Module 3B video – Automating the Clustered Affinity Matrix Calculation)

The algorithm is still computationally complex because it operates in $O(n^3)$. The good news is that the algorithm operates in polynomial time, which is far better than the exponential time as hinted at by Bell's algorithm above. However, we can get the algorithm down to $O(n^2)$ as described in the following:

(Watch Module 3C video – Simplifying the Bond Energy Algorithm)

An Example

Given our Attribute Affinity Matrix above, let us run through an example of how we fragment a table using the Bond Energy Algorithm. Let us start with the first two columns and place them into the matrix.

	A1 (pno)	A2 (pname)		
A_1	45	0		
A_2	0	80		
A_3	45	5		
A_4	0	75		

Let us calculate the Bond Energy (BE) value for the affinity of column A_1 to A_2 and the affinity of A_2 to A_1 (which are the same).

$$BE = 2* ((45*0) + (0*80) + (45*5) + (0*75)) = 450$$

Now, let us place column A_3 into the matrix. We will try A_3 before column A_1, between columns A_1 and A_2, and after column A_2. The placement that yields the greatest positive change in Bond Energy wins.

Before A_1:

	A3 (budget)	A1 (pno)	A2 (pname)	
A_1	45	45	0	
A_2	5	0	80	

A_3	53	45	5	
A_4	3	0	75	

Δ BE = 2 * ((45*45) + (5*0) + (53*45) + (3*0)) = 8820 // BE between A_3 // and A_1

Between A_1 and A_2:

	A1 (pno)	A3 (budget)	A2 (pname)	
A_1	45	45	0	
A_2	0	5	80	
A_3	45	53	5	
A_4	0	3	75	

Δ BE = 2 * (((45*45) + (0*5) + (45*53) + (0*3)) // BE between A_1 and A_3

+((45*0) + (5*80) + (53*5) + (3*75)) // BE between A_3 and A_2

- ((45*0) + (0*80) + (45*5) + (0*75))) // BE broken between // A_1 and A_2

= 2*(4410 + 890 – 225) = 10150

After A_2:

	A1 (pno)	A2 (pname)	A3 (budget)	
A_1	45	0	45	
A_2	0	80	5	
A_3	45	5	53	
A_4	0	75	3	

Δ BE = 2*((0*45) + (80*5) + (5*53) + (75*3)) = 1780 // BE between A_2 // and A_3

In this case, placing A_3 between A_1 and A_2 yields the highest Δ BE.

Applying the same algorithm to the placement of A_4 and reordering the rows to match the columns yields the following Clustered Affinity Matrix:

	A1 (pno)	A3 (budget)	A2 (pname)	A4 (loc)
A_1	45	45	0	0
A_3	45	53	5	3
A_2	0	5	80	75
A_4	0	3	75	78

Partitioning Algorithm

We are not quite done yet. Once we have a Clustered Affinity Matrix, we still need an algorithm to automatically partition it. We would like to find the partitioning between columns A_{j-1} and A_j such that it organizes the columns such that the Top Attributes (TA) and the Bottom Attributes (BA) have a relatively high clustering:

	$A_1A_2...A_{j-1}$	$A_jA_{j+1}...A_k$
$A_1A_2...A_{j-1}$	TA	
$A_jA_{j+1}...A_k$		BA

We look at the set of queries Q = $\{q_1, q_2, \ldots, q_q\}$. We want to set the partition to maximize the number of queries that access either one or the other partition, but not both! We want to minimize the number of queries that access both partitions. This is because queries that access both partitions must perform expensive joins between both partitions. If we are faced with many expensive joins, it is better not to fragment the table to avoid so many joins.

- $AQ(q_i) = \{ A_j \mid use(q_i, A_j) = 1\}$ // All Queries
- $TQ = \{ q_i \mid AQ(q_i) \subseteq TA \}$ // Top (left) quarter Queries where all attributes of q_i in TQ)
- $BQ = \{ q_i \mid AQ(q_i) \subseteq BA \}$ // Bottom (right) quarter Queries where all attributes of q_i in BQ
- OQ = AQ - { TQ+BQ } // Other Queries whose attributes span both quarters
- AQ = TQ + BQ + OQ

Now count the frequency of access of the queries. For each query in either all queries, top queries, bottom queries, or other queries, determine the number of accesses per hour for that query over all the sites.

- $CQ = \sum (q_i \text{ in } Q) \sum(\text{sites } j) \text{ acc}_j(q_i)$ // Count of all Queries
- $CTQ = \sum (q_i \text{ in } TQ) \sum(\text{sites } j) \text{ acc}_j(q_i)$ // Count of TQ queries
- $CBQ = \sum (q_i \text{ in } BQ) \sum(\text{sites } S_j) \text{ acc}_j(q_i)$ // Count of BQ queries
- $COQ = \sum (q_i \text{ in } OQ) \sum(\text{sites } S_j) \text{ acc}_j(q_i)$ // Count of OQ queries
- CQ = CTQ + CBQ + COQ

Now determine the best cut-off point *z*. There are n-1 choices of cut-off points, so you want to find a cut-off point that maximizes: $z = CTQ*CBQ - COQ^2$

(Watch Module 3D video – The Partitioning Algorithm)

Proof of Correctness

Remember, we need to prove completeness, reconstruction, and disjointness.

- Completeness

All attributes are contained in at least one fragment R_i. The primary key is contained in each fragment.

- Reconstruction

  R is the product of joining all fragments R_i on the primary key in each fragment.

- Distjointness

  All attributes are contained it at most one fragment R_i. The exception is that the primary key must be contained in each fragment to enable joins between fragments.

Conclusion

Distributed DB design depends on

- The enterprise conceptual model
- The predicates of the user and application queries
- The application access frequency and location

Aside from the standard schema design of single relational databases, the distributed database system designer must place data tables near the applications that access the data most frequently.

Table fragmentation is useful in distributed database systems when applications and users have different data access requirements of the same table. Fragments that are used often by an application will be placed closer to that application to achieve faster access to that data through *locality of reference*. In the process of determining the derived fragmentation, we must ensure that the fragmentation we chose for table is provably correct.