Module 2: Horizontal Partitioning

Distributed Design

The design of distributed database programs involves where to place the database data with respect to the current or anticipated placement of the database programs. We are specifically interested in organizing data. We want to partition the data properly and we want to place the partitions in the right locations. The object of the partitioning and placement is to make data access fast and efficient. The objective is achieved through *locality of reference*, which means that we must place the data that different users will use most often closest to them.

Access Patterns

To understand which users need which data, one must understand user and application access patterns. One must answer the following questions:

- Which types of data do which classes of users need most often?
- How often do different classes of users access each type of data?
- Where are different classes of users located geographically?

Statistics of access patterns of different classes of users are required to answer these questions. Certainly, data access patterns are dynamic and vary over time. The type, volume, and frequency of access vary for each class of user. The classes of users and their needs also vary over time. Furthermore, the frequency of variation can fluctuate. On one end of the fluctuation spectrum is very slow change where statistics about the database are static or nearly static. This is an unusual scenario – access patterns typically change more rapidly. However, understanding distributed database design is greatly simplified if one assumes that the access patterns are static. Once database design is understood for static databases, variability can be introduced to the design process.

Dynamic access patterns are more usual and more realistic. Users do not always have the same needs over time. However, they are more difficult to anticipate. Moreover, design of distributed database systems are greatly complicated by dynamic statistics.

One does not usually have access to all the statistics. Partial knowledge is the norm. Designers have to go with what they know and possibly adjust as they observe usage patterns over time.

In this class, we will address static access patterns with perfect knowledge of the statistics. Designing distributed databases with full knowledge of static access patterns is hard enough. Designing distributed databases with partial knowledge of future changes is exceedingly difficult. In any case, the static approach that we will develop can serve as a basis for more complex dynamic approaches.

Design Strategies

There are two strategies for designing databases: top-down and bottom-up. In the top down strategy, database designers have complete control of the entire database design and the distribution of the data. In the bottom up strategy, database designers have no control over the database design and distribution. Most situations present a hybrid of top-down and bottom-up design. For simplicity, we will discuss top-down strategies.

Top-Down Design

Top-down design captures the conceptual design of the data of the whole enterprise. It is federated by the views of external users and applications. A designer must anticipate new views and new usages of the data. A designer must describe the semantics of the data as used in the entire enterprise.

This is almost identical to typical standard database design. However, we are concerned with the *Distribution Design.* We need to place tables "geographically" on the network and we need to fragment tables.

(Watch Module 2A video – The Top-Down Process)

Bottom-Up Design

Top-down design is the choice when you have the liberty of starting from scratch. Unfortunately, this is not usually the case. Bottom-up design involves integrating independent/semi-independent schemas into a single Global Conceptual Schema (GCS). Since the original schemas were not designed in concert, the integrator must deal with sometimes-complex schema mapping issues as well as heterogeneous integration issues. These are advanced topics that will not be discussed in the course.

Distributed Database Design

To simplify things, we assume that database designers have control over designing the database. Much of the initial design process is identical to the single database design process. However, when evaluating requirements of different users and applications, we are conscious of their needs including location, frequency of access, and the specific data that they will frequently request. Sometimes, it is clear that certain complete tables need to be placed in their entirety at certain locations. Other times, tables may require *fragmentation*!

Why Fragment Tables?

Usually, external user and application views are subsets of relations, both vertical and horizontal subsets. Furthermore, it is frequently the case that applications that require different views are geographically distributed. For the most efficient query processing, we want the data that will be most accessed by an application geographically closest to that application to achieve. This is known as *locality of reference*. Sometimes, fragmenting tables also enable users to experience the benefits of concurrent execution where multiple parts of the query run simultaneously on different fragments!

Fragmentation Replication

Table fragments are replicated to provide locality of reference for multiple users that are geographically distributed. Distributing replicated data can make query access efficient for multiple classes of users that require frequent access to the data. Fragments also provide backup for lost data or lost connections to data. If one site that contains a fragment is unavailable, another site that contains a copy of the fragment can still run enabling users and applications more continuous operations. However, updates to replicated fragments are difficult because data needs to be updated consistently at multiple locations. Because of competing costs and benefits, deciding whether or not to replicate data fragments requires careful analysis of the tradeoffs. Data replication will be discussed in more detail later in the semester.

Why Not Fragment Data?

There are some reasons not to fragment data. Updates are difficult because multiple fragments may potentially be updated and managing that process can be complex. In addition, performance is sometimes degraded because operations require the coordination of multiple distributed sites. During the coordination, data is generally not available to queries and applications because the fragments need to maintain consistency. In addition, fragmentation often induces costly cross-platform joins across multiple fragments. Furthermore, checking the semantic integrity of operations, such as "do not delete a course unless all the students dropped it," requires costly cross platform operations. Finally, if one site that contains a fragment is down, then the distributed database management system must ensure that updates to fragments during that time are not lost. This means when the site comes back up, the site must update its fragments to ensure that the data in the fragments reflect all the transactions that occurred white the site was down and be consistent with the rest of the database. Maintaining fragment consistency will be addressed in the Reliability Protocols module.

(Watch Module 2B video – Example Fragmentation)

Achieving Correct Fragmentation

While different fragmentation strategies can be performed, one must be able to ensure that the fragmentation preserves the data and does not duplicate data. Fragmentation strategies must be provably correct.

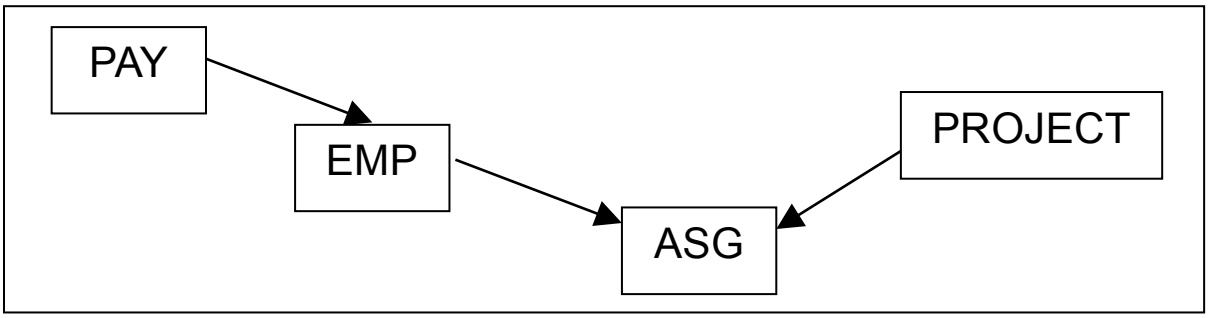
(Watch Module 2C – Achieving Correct Fragmentation)

Primary vs. Derived Fragmentation

Primary fragmentation is achieved by applying certain heuristics to the table to be fragmented. We will discuss these shortly. *Derived* fragmentation is derived according to the referencing table's fragmentation.

For example, the following figure depicts four tables (PAY, EMPLOYEE, ASIGNMENT, and PROJECT) represented by rectangles and three one-to-many relationships represented by arrows. If PAY is fragmented via primary fragmentation, then EMP would be fragmented via derived fragmentation to match PAY's fragmentation.

Similarly, if EMP is fragmented via primary (or even derived) fragmentation, then ASG would be fragmented via derived fragmentation to match EMP's fragmentation. Derived fragmentation will be discussed in more detail later in this Module.



Heuristics for Horizontal Fragmentation

Horizontal fragmentation is a complex process. The number of potential fragmentation possibilities is exponential with respect to the predicates. Predicates are constraint phrases that are typically found in the WHERE clause of a query and are discussed in the next section. If the number of predicates is *n*, then the number of potential horizontal fragments is on the order of 2^n, or in Big-O notation, $O(2^n)$. (See http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/ and http://web.mit.edu/16.070/www/lecture/big_o.pdf for more on Big-O notation.)

Predicates

Query predicates are a key factor in determining horizontal fragmentation. Predicates appear in the WHERE clause (selection) of a query. For example, let $R(A_1, A_2, \ldots, A_n)$ be a relation where each column A_i is defined over a domain D_i. We say $p_{Ai,j}$ is a simple predicate on column A_i if it is of the form:

$$p_{Ai,j} = A_i \; \Theta \; \text{Value} \qquad \text{where } \Theta \text{ is a comparator} \in \{ =, <, >, \leq, \geq, <> \}$$

Examples:

- $p_{PNAME,1}$ = 'CAD/CAM'
- $p_{PNAME,2}$ = 'Database Development'
- $p_{BUDGET,1}$ > 200000
- $p_{BUDGET,2}$ <= 200000

Usually, multiple predicates are necessary to describe a selection of rows in a relation.

Most Boolean combinations of predicates can be translated into *conjunctive normal form*: p_1 ^ p_2 ^ …^ p_k , where the symbol ^ means AND. Many queries contain WHERE clauses with OR conjunctions as well. But for simplicity, let us just consider AND conjunctions.

We attempt to fragment tables according to WHERE clause patterns. A combination of predicates in conjunctive normal form is a *minterm.* Let the set of all predicates on a relation be Pr = {p_1, p_2, …}.

Let the set of *minterm* predicates M = { m_1, m_2, …, m_z } where each m_i is some conjunction of predicates of Pr. Sometimes different predicates can be equal to each other. For example:

- For equality: !(attr = val) = (attr <> val)
- For inequality: !(attr > val) = (attr ≤ val)

It is not necessary to duplicate predicates in minterms. One of the expression is sufficient.

Some example predicates are:

- p_1: LOC = 'Montreal'
- p_2: LOC = 'New York'
- p_3: LOC = 'Paris'
- p_4: BUDGET > 200000
- p_5: BUDGET <= 200000

Some example minterms are:

- m_1: LOC = 'New York' ^ BUDGET > 200000
- m_2: LOC = 'New York' ^ BUDGET <= 200000
- m_3: LOC = 'Paris' ^ BUDGET > 200000
- m_4: LOC = 'Paris' ^ BUDGET <= 200000
- m_5: LOC = 'Montreal' ^ BUDGET > 200000
- m_6: LOC = 'Montreal ' ^ BUDGET <= 200000

Factors in Determining Horizontal Fragmentation

The first factor in determining horizontal fragmentation is minterm selectivity. Minterm selectivity, sel(m_i), is defined as the number of rows of a table returned when the minterm is placed in the WHERE clause of a query.

The second factor is query access frequency by users and applications. If the set of queries of a table are Q = {q_1, q_2, …}, then acc(q_i) = the number of times q_i is run on the database per some unit time.

It is up to the database designer to choose the right fragmentation design based on an understanding of the predicates of each application query and of the corresponding access frequencies. Each fragment should have the property that there is equal probability of access of every tuple by every application! Tuples from different fragments should have different probabilities of access by different applications. We want to ensure that all current and future tuples will have a fragment to reside in and we do not want to overdo fragmentation.

A Heuristic for Horizontal Fragmentation

(Watch Module 2D video – Working Through Horizontal Fragmentation)

Issues exist with modifying the fragmentation based on new (dynamic) information. For example, what if we added a new project row with BUDGET = 600K? Should we create a new fragment or extend a current fragment? If we create a new fragment, what are its boundaries? The answers to these questions will differ based on knowledge of how the data will vary over time. We will address dynamic data at this time.

An Example of Horizontal Fragmentation

The following example is based on the knowledge of only the predicates that are used by different classes of users. We leave out the access frequencies for now.

Step 1. Suppose we identify the following predicates:

1. p_1: LOC = 'Montreal'
2. p_2: LOC = 'New York'
3. p_3: LOC = 'Paris'
4. p_4: BUDGET > 200000
5. p_5: BUDGET <= 200000

Step 2. We define the full minterm set from the predicates:

1. m_1: LOC = 'Montreal'
2. m_2: LOC = 'New York'
3. m_3: LOC = 'Paris'
4. m_4: BUDGET > 200000
5. m_5: BUDGET <= 200000
6. m_6: LOC = 'Montreal' ^ LOC = 'New York'
7. m_7: LOC = 'Montreal' ^ LOC = 'Paris'
8. m_8: LOC = 'Montreal' ^ BUDGET > 200000
9. m_9: LOC = 'Montreal' ^ BUDGET <= 200000
10. m_{10}: LOC = 'New York' ^ LOC = 'Paris'
11. m_{11}: LOC = 'New York' ^ BUDGET > 200000
12. m_{12}: LOC = 'New York' ^ BUDGET <= 200000
13. m_{13}: LOC = 'Paris' ^ BUDGET > 200000
14. m_{14}: LOC = 'Paris' ^ BUDGET <= 200000
15. m_{15}: BUDGET > 200000 ^ BUDGET <= 200000
16. m_{16}: LOC = 'Montreal' ^ LOC = 'New York' ^ LOC = 'Paris'
17. m_{17}: LOC = 'Montreal' ^ LOC = 'New York' ^ BUDGET > 200000
18. m_{18}: LOC = 'Montreal' ^ LOC = 'New York' ^ BUDGET <= 200000
19. m_{19}: LOC = 'Montreal' ^ LOC = 'Paris' ^ BUDGET > 200000
20. m_{20}: LOC = 'Montreal' ^ LOC = 'Paris' ^ BUDGET <= 200000
21. m_{21}: LOC = 'Montreal' ^ BUDGET > 200000 ^ BUDGET <= 200000
22. m_{22}: LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET > 200000
23. m_{23}: LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET <= 200000
24. m_{24}: LOC = 'New York' ^ BUDGET > 200000 ^ BUDGET <= 200000
25. m_{25}: LOC = 'Paris' ^ BUDGET > 200000 ^ BUDGET <= 200000
26. m_{26}: LOC = 'Montreal' ^ LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET > 200000

27. m_{27}: LOC = 'Montreal' ^ LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET <= 200000
28. m_{28}: LOC = 'Montreal' ^ LOC = 'New York' ^ BUDGET > 200000 ^ BUDGET <= 200000
29. m_{29}: LOC = 'Montreal' ^ LOC = 'Paris' ^ BUDGET > 200000 ^ BUDGET <= 200000
30. m_{30}: LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET > 200000 ^ BUDGET <= 200000
31. m_{31}: LOC = 'Montreal' ^ LOC = 'New York' ^ LOC = 'Paris' ^ BUDGET > 200000 ^ BUDGET <= 200000

Step 3. Define the inferences from the predicates:

1. p_1 => not p_2
2. p_1 => not p_3
3. p_2 => not p_1
4. p_2 => not p_3
5. p_3 => not p_1
6. p_3 => not p_2
7. p_4 => not p_5
8. p_5 => not p_4

Step 4. Based on the inferences, eliminate the minterms with contradictions. We are left with:

1. m_1: LOC = 'Montreal'
2. m_2: LOC = 'New York'
3. m_3: LOC = 'Paris'
4. m_4: BUDGET > 200000
5. m_5: BUDGET <= 200000
6. m_8: LOC = 'Montreal' ^ BUDGET > 200000
7. m_9: LOC = 'Montreal' ^ BUDGET <= 200000
8. m_{11}: LOC = 'New York' ^ BUDGET > 200000
9. m_{12}: LOC = 'New York' ^ BUDGET <= 200000
10. m_{13}: LOC = 'Paris' ^ BUDGET > 200000
11. m_{14}: LOC = 'Paris' ^ BUDGET <= 200000

Step 5. After subsumption of minterms, only the following minterms remain:

1. m_8: LOC = 'Montreal' ^ BUDGET > 200000
2. m_9: LOC = 'Montreal' ^ BUDGET <= 200000
3. m_{11}: LOC = 'New York' ^ BUDGET > 200000
4. m_{12}: LOC = 'New York' ^ BUDGET <= 200000
5. m_{13}: LOC = 'Paris' ^ BUDGET > 200000
6. m_{14}: LOC = 'Paris' ^ BUDGET <= 200000

Given the initial PROJECT table:

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	130000	New York

P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

The minterm set yields the following fragments:

m_8 = null

m_9 =

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

m_{11} =

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York

m_{12} =

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	130000	New York

m_{13} =

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

m_{14} = null

(Watch Module 2E video – Derived Horizontal Fragmentation)

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*.

It is often a good idea to fragment tables that are related to and joined with fragmented tables. Suppose there is a one-to-many relationship between tables R_1 and R_2. The *primary fragmentation* of table R_1 drives *derived fragmentation* of table R_2. In the process of determining the derived fragmentation, we must ensure that the derived fragmentation we chose for table R_2 is provably correct.