1

PARALLEL AND DISTRIBUTED DATABASES

Unit Structure:

A **parallel database system** is one that seeks to improve performance through parallel implementation of various operations such as loading data, building indexes, and evaluating queries.

In **a distributed database system**, data is physically stored across several sites, and each site is typically managed by a DBMS that is capable of running independently of the other sites.

1.1 ARCHITECTURES FOR PARALLEL DATABASES

Three main architectures are proposed for building parallel databases:

1. Shared - memory system, where multiple CPUs are attached to an interconnection network and can access a common region of main memory.

2. Shared - disk system, where each CPU has a private memory and direct access to all disks through an interconnection network.

3. Shared - nothing system, where each CPU has local main memory and disk space, but no two CPUs can access the same storage area; all communication between CPUs is through a network connection.

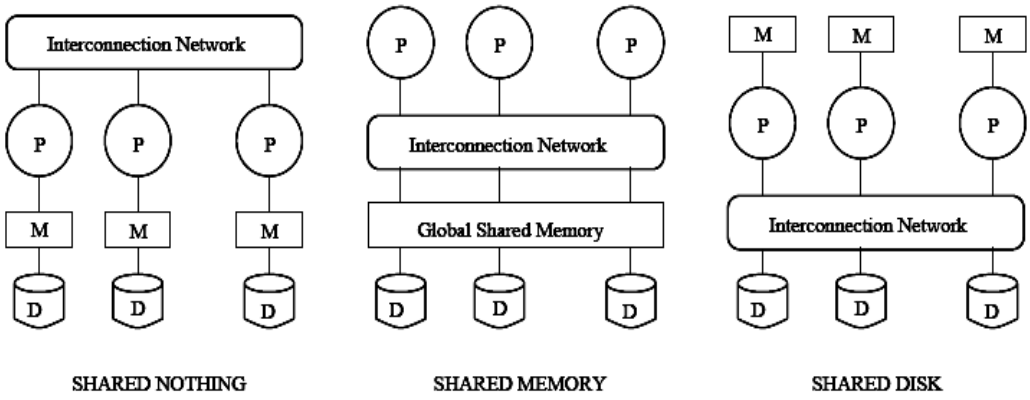

Fig 1.1 Architectures for Parallel Databases

- Scaling the system is an issue with shared memory and shared disk architectures because as more CPUs are added, existing CPUs are slowed down because of the increased contention for memory accesses and network bandwidth.

- **The Shared Nothing Architecture has shown:**
  a) Linear Speed Up**:** the time taken to execute operations decreases in proportion to the increase in the number of CPU's and disks

  b) Linear Scale Up**:** the performance is sustained if the number of CPU's and disks are increased in proportion to the amount of data.

1.2. PARALLEL QUERY EVALUATION

- Parallel evaluation of a relational query in a DBMS with a shared-nothing architecture is discussed. Parallel execution of a single query has been emphasized.

- A relational query execution plan is a graph of relational algebra operators and the operators in a graph can be executed in parallel. If an operator consumes the output of a second operator, we have pipelined parallelism.

- Each individual operator can also be executed in parallel by partitioning the input data and then working on each partition in parallel and then combining the result of each partition. This approach is called Data Partitioned parallel Evaluation.

1.2.1 Data Partitioning:

Here large datasets are partitioned horizontally across several disk, this enables us to exploit the I/O bandwidth of the disks by reading and writing them in parallel. This can be done in the following ways:

a. Round Robin Partitioning

b. Hash Partitioning

c. Range Partitioning

a. Round Robin Partitioning : If there are *n* processors, the i^{th} tuple is assigned to processor *i mod n*

b. Hash Partitioning : A hash function is applied to (selected fields of) a tuple to determine its processor.

Hash partitioning has the additional virtue that it keeps data evenly distributed even if the data grows and shrinks over time.

c. Range Partitioning : Tuples are sorted (conceptually), and *n* ranges are chosen for the sort key values so that each range contains roughly the same number of tuples; tuples in range *i* are assigned to processor *i*.

Range partitioning can lead to data skew**;** that is, partitions with widely varying numbers of tuples across partitions or disks. Skew causes processors dealing with large partitions to become performance bottlenecks.

1.2.2 Parallelizing Sequential Operator Evaluation Code:

Input data streams are divided into parallel data streams. The output of these streams are merged as needed to provide as

inputs for a relational operator, and the output may again be split as needed to parallelize subsequent processing**.**

1.3. PARALLELIZING INDIVIDUAL OPERATIONS

Various operations can be implemented in parallel in a sharednothing architecture.

1.3.1 Bulk Loading and Scanning:

- Pages can be read in parallel while scanning a relation and the retrieved tuples can then be merged, if the relation is partitioned across several disks.

- If a relation has associated indexes, any sorting of data entries required for building the indexes during bulk loading can also be done in parallel.

1.3.2 Sorting:

- Sorting could be done by redistributing all tuples in the relation using range partitioning.

- Ex. Sorting a collection of employee tuples by salary whose values are in a certain range.

- For N processors each processor gets the tuples which lie in range assigned to it. Like processor 1 contains all tuples in range 10 to 20 and so on.

- Each processor has a sorted version of the tuples which can then be combined by traversing and collecting the tuples in the order on the processors (according to the range assigned)

- The problem with range partitioning is data skew which limits the scalability of the parallel sort. One good approach to range partitioning is to obtain a sample of the entire relation by taking samples at each processor that initially contains part of the relation. The (relatively small) sample is sorted and used to identify ranges with equal numbers of tuples. This set of range values, called a splitting vector, is then distributed to all processors and used to range partition the entire relation.

1.3.3 Joins:

- Here we consider how the join operation can be parallelized

- Consider 2 relations A and B to be joined using the age attribute. A and B are initially distributed across several disks in a way that is not useful for join operation

- So we have to decompose the join into a collection of *k* smaller joins by partitioning both *A* and *B* into a collection of *k* logical partitions.

- If same partitioning function is used for both A and B then the union of k smaller joins will compute to the join of A and B.

1.4 DISTRIBUTED DATABASES

- The abstract idea of a distributed database is that the data should be physically stored at different locations but its distribution and access should be transparent to the user.

1.4.1 Introduction to DBMS:

A Distributed Database should exhibit the following properties:

1) **Distributed Data Independence**: - The user should be able to access the database without having the need to know the location of the data.

2) **Distributed Transaction Atomicity: -** The concept of atomicity should be distributed for the operation taking place at the distributed sites.

- **Types of Distributed Databases are:-**

  a) Homegeneous Distributed Database is where the data stored across multiple sites is managed by same DBMS software at all the sites.

  b) Heterogeneous Distributed Database is where multiple sites which may be autonomous are under the control of different DBMS software.

1.4.2 Architecture of DDBs :

There are 3 architectures: -

1.4.2.1Client-Server:

- A Client-Server system has one or more client processes and one or more server processes, and a client process can send a query to any one server process. Clients are responsible for user-interface issues, and servers manage data and execute transactions.
- Thus, a client process could run on a personal computer and send queries to a server running on a mainframe.

- **Advantages: -**
  1. Simple to implement because of the centralized server and separation of functionality.
  2. Expensive server machines are not underutilized with simple user interactions which are now pushed on to inexpensive client machines.
  3. The users can have a familiar and friendly client side user interface rather than unfamiliar and unfriendly server interface

1.4.2.2 Collaborating Server:

- In the client sever architecture a single query cannot be split and executed across multiple servers because the client process would have to be quite complex and intelligent enough to break a query into sub queries to be executed at different sites and then place their results together making the client capabilities overlap with the server. This makes it hard to distinguish between the client and server

- In Collaborating Server **system,** we can have collection of database servers, each capable of running transactions against local data, which cooperatively execute transactions spanning multiple servers.

- When a server receives a query that requires access to data at other servers, it generates appropriate sub queries to be executed by other servers and puts the results together to compute answers to the original query.

1.4.2.3 Middleware:

- Middleware system is as special server, a layer of software that coordinates the execution of queries and transactions across one or more independent database servers.

- The Middleware architecture is designed to allow a single query to span multiple servers, without requiring all database servers to be capable of managing such multi site execution strategies. It is especially attractive when trying to integrate several legacy systems, whose basic capabilities cannot be extended.

- We need just one database server that is capable of managing queries and transactions spanning multiple servers; the remaining servers only need to handle local queries and transactions.

1.5 STORING DATA IN DDBS

Data storage involved 2 concepts

1. Fragmentation
2. Replication

1.5.1 Fragmentation:

- It is the process in which a relation is broken into smaller relations called fragments and possibly stored at different sites.
- It is of 2 types

1. **Horizontal Fragmentation** where the original relation is broken into a number of fragments, where each fragment is a subset of rows.

   The union of the horizontal fragments should reproduce the original relation.

2. **Vertical Fragmentation** where the original relation is broken into a number of fragments, where each fragment consists of a subset of columns.

   The system often assigns a unique tuple id to each tuple in the original relation so that the fragments when joined again should from a lossless join.

   The collection of all vertical fragments should reproduce the original relation.

1.5.2 Replication:

- Replication occurs when we store more than one copy of a relation or its fragment at multiple sites.

- **Advantages:-**
  1. **Increased availability of data:** If a site that contains a replica goes down, we can find the same data at other sites. Similarly, if local copies of remote relations are available, we are less vulnerable to failure of communication links.

  2. **Faster query evaluation:** Queries can execute faster by using a local copy of a relation instead of going to a remote site.

1.5.3 Distributed catalog management :

Naming Object

- It's related to the unique identification of each fragment that has been either partitioned or replicated.

- This can be done by using a global name server that can assign globally unique names.
- This can be implemented by using the following two fields:-

1. Local name field – locally assigned name by the site where the relation is created. Two objects at different sites can have same local names.
2. Birth site field – indicates the site at which the relation is created and where information about its fragments and replicas is maintained.

Catalog Structure:

- A centralized system catalog is used to maintain the information about all the transactions in the distributed database but is vulnerable to the failure of the site containing the catalog.
- This could be avoided by maintaining a copy of the global system catalog but it involves broadcast of every change done to a local catalog to all its replicas.
- Another alternative is to maintain a local catalog at every site which keeps track of all the replicas of the relation.

Distributed Data Independence:

- It means that the user should be able to query the database without needing to specify the location of the fragments or replicas of a relation which has to be done by the DBMS
- Users can be enabled to access relations without considering how the relations are distributed as follows: The *local name* of a relation in the system catalog is a combination of a *user name* and a user-defined *relation name*.
- When a query is fired the DBMS adds the user name to the relation name to get a local name, then adds the user's site-id as the (default) birth site to obtain a global relation name. By looking up the global relation name in the local catalog if it is cached there or in the catalog at the birth site the DBMS can locate replicas of the relation.

1.5.4 Distributed query processing:

- In a distributed system several factors complicates the query processing.
- One of the factors is cost of transferring the data over network.
- This data includes the intermediate files that are transferred to other sites for further processing or the final result files that may have to be transferred to the site where the query result is needed.
- Although these cost may not be very high if the sites are connected via a high local n/w but sometime they become quit significant in other types of network.
- Hence, DDBMS query optimization algorithms consider the goal of reducing the amount of data transfer as an optimization criterion in choosing a distributed query execution strategy.
- Consider an EMPLOYEE relation.
- The size of the employee relation is 100 * 10,000=10^6 bytes
- The size of the department relation is 35 * 100=3500 bytes

EMPLOYEE

Fname	Lname	SSN	Bdate	Add	Gender	Salary	Dnum

- 10,000 records
- Each record is 100 bytes
- Fname field is 15 bytes long
- SSN field is 9 bytes long
- Lname field is 15 bytes long
- Dnum field is 4 byte long

DEPARTMENT

Dname	Dnumber	MGRSSN	MgrStartDate

- 100records
- Each record is 35 bytes long
- Dnumber field is 4 bytes long
- Dname field is 10 bytes long
- MGRSSN field is 9 bytes long
- Now consider the following query:

  “For each employee, retrieve the employee name and the name of the department for which the employee works.”

- Using relational algebra this query can be expressed as

  FNAME, LNAME, DNAME

  *(EMPLOYEE * DNO=DNUMBER DEPARTMENT)*

- If we assume that every employee is related to a department then the result of this query will include 10,000 records.
- Now suppose that each record in the query result is 40 bytes long and the query is submitted at a distinct site which is the result site.
- Then there are 3 strategies for executing this distributed query:
  1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the site 3 that is your result site and perform the join at that site. In this case a total of 1,000,000 + 3500 = 1,003,500 bytes must be transferred.
  2. Transfer the EMPLOYEE relation to site 2 (site where u have Department relation) and send the result to site 3. the size of the query result is 40 * 10,000 = 400,000 bytes so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
  3. Transfer the DEPARTEMNT relation to site 1 (site where u have Employee relation) and send the result to site 3. in this case 400,000 + 3500 = 403,500 bytes must be transferred.

1.5.4.1 Nonjoin Queries in a Distributed DBMS:

- Consider the following two relations:
  Sailors (sid: integer, sname:string, rating: integer, age: real)
  Reserves (sid: integer, bid: integer, day: date, rname: string)

- Now consider the following query:

```
SELECT S.age
FROM Sailors S
WHERE S.rating > 3 AND S.rating < 7
```

- Now suppose that sailor relation is horizontally fragmented with all the tuples having a rating less than 5 at Shanghais and all the tuples having a rating greater than 5 at Tokyo.

- The DBMS will answer this query by evaluating it both sites and then taking the union of the answer.

1.5.4.2 Joins in a Distributed DBMS:

- Joins of a relation at different sites can be very expensive so now we will consider the evaluation option that must be considered in a distributed environment.

- Suppose that Sailors relation is stored at London and Reserves relation is stored at Paris. Hence we will consider the following strategies for computing the joins for Sailors and Reserves.
- In the next example the time taken to read one page from disk (or to write one page to disk) is denoted as *td* and the time taken to ship one page (from any site to another site) as *ts*.

2.5.1 Fetch as needed:

- We can do a page oriented nested loops joins in London with Sailors as the outer join and for each Sailors page we will fetch all Reserves pages form Paris.
- If we cache the fetched Reserves pages in London until the join is complete , pages are fetched only once, but lets assume that Reserves pages are not cached, just to see how bad result we can get:
- To scan Sailors the cost is 500td for each Sailors page, plus the cost of scanning and shipping all of Reserves is 1000(td+ts). Therefore the total cost is 500td+500000(td+ts).
- In addition if the query was not submitted at the London site then we must add the cost of shipping the result to the query site and this cost depends on the size of the result.
- Because *sid* a key for the Sailors. So, the number of tuples in the result is 100,000 (which is the number of tuples in Reserves) and each tuple is 40+50=90 bytes long.
- Thus (4000 is the size of the result) 4000/90=44 result tuples fit on a page and the result size is 100000/44=2273 pages.

1.5.4.3 Ship to one site:

- There are three possibilities to compute the result at one site:
  - Ship the Sailors from London to Paris and carry out the join.
  - Ship the Reserves form Paris to London and carry out the join.
  - Ship both i.e. Sailors and Reserves to the site where the query was posed and compute the join.
- And the cost will be:
  - The cost of scanning and shipping Sailors form London to Paris and doing the join at Paris is 500(2td+ ts) + 4500td.
  - The cost shipping Reserves form Paris to London and then doing the join at London is 1000 (2td + ts) + 4500td.

2.5.2 Semi joins and bloomjoins:

- Consider the strategy of shipping Reserves from Paris to London and computing the joins at London.
- It may happen that some tuples in Reserves do not join with any tuple in the Sailors, so we could somehow identify the tuples that are guaranteed not to join with any Sailors tuples and we could avoid shipping them.

- **Semijoins**:
  - The basic idea of Semijoins can be proceed in three steps:
    1) At London compute the projection of Sailors onto the join columns, and ship this projection to Paris.
    2) At Paris, compute the natural join of the projection received from the first site with the Reserves relation. The result of this join is called the reduction of Reserves with respect to Sailors because only those Reserves tuples in the reduction will join with tuples in the Sailors relation. Therefore, ship the reduction of Reserves to London, rather than the entire Reserves relation.
    3) At London, compute the join of the reduction of Reserves with Sailors.
- Computing the cost of Sailors and Reserves using Semijoin:
  - Assume we have a projection based on first scanning Sailors and creating a temporary relation with tuples that have only an *sid* field, then sorting the temporary and scanning the sorted temporary to eliminate duplicates.
  - If we assume that the size of the *sid* field is 10 bytes, then the cost of projection is 500td for scanning Sailors, plus 100td for creating the temporary, plus 400td for sorting it, plus 100td for the final scan, plus 100td for writing the result into another temporary relation, that is total is 1200td.

    500td + 100td + 400td + 100td + 100td = 1200td.
  - The cost of computing the projection and shipping them it to Paris is 1200td + 100ts.
  - The cost of computing the reduction of Reserves is 3 * (100+1000)=3300td.
- **But what is the size of Reduction?**
  - If every sailor holds at least one reservation then the reduction includes every tuple of Reserves and the effort invested in shipping the projection and reducing Reserves is a total waste.

- So because of this observation we can say that Semijoin is especially useful in conjunction with a selection on one of the relations.
- For example if we want to compute the join of Sailors tuples with a rating>8 of the Reserves relation, then the size of the projection on *sid* for tuples that satisfy the selection would be just 20% of the original projection that is 20 pages.

- **Bloomjoin**:
  - Bloomjoin is quit similar to semijoins.
  - The steps of Bloomjoins are:

  Step 1:

  The main difference is that a bit-vector is shipped in first step, instead of the projection of Sailors.

  A bit-vector (some chosen tuple) of size k is computed by hashing each tuple of Sailors into the range 0 to k-1 and setting bit I to 1 if some tuple hashes to I and 0 otherwise.

  Step 2:

  The reduction of Reserves is computed by hashing each tuple of Reserves (using the *sid* field) into the range 0 to k-1, using the same hash function which is used to construct the bit-vector and discard the tuples whose hash values corresponds to 0 bit.

  Because no Sailors tuples hash to such an i and no Sailors tuples can join with any Reserves tuple that is not in the reduction.

- Thus the cost of shipping a bit-vector and reducing Reserves using the vector are less than the corresponding costs is Semijoins.

1.5.4.4 Cost-Based Query Optimization:

A query involves several operations and optimizing a query in a distributed database poses some challenges:

- Communication cost must be considered. If we have several copies of a real time then we will have to decide which copy to use.
- If the individual sites are run under the control of different DBMS then the autonomy of each site must be respected while doing global query planning.