Chapter 18: Parallel Databases

Introduction

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
  - Recent desktop computers feature multiple processors and this trend is projected to accelerate
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis.
  - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.
- Different queries can be run in parallel with each other. Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning
- The relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = *n*):

Round-robin:

Send the *I* th tuple inserted in the relation to disk *i* mod *n*.

Hash partitioning:

- Choose one or more attributes as the partitioning attributes.
- Choose hash function *h* with range 0…*n* - 1
- Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to disk *i*.

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- Large relations are preferably partitioned across all the available disks.
- If a relation consists of *m* disk blocks and there are *n* disks available in the system, then the relation should be allocated **min**(*m,n*) disks.

Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction throughput; used primarily to scale up a transaction processing system to support a larger number of transactions per second.
- Easiest form of parallelism to support, particularly in a shared-memory parallel database, because even sequential database systems support concurrent processing.
- More complicated to implement on shared-disk or shared-nothing architectures
  - Locking and logging must be coordinated by passing messages between processors.
  - Data in a local buffer may have been updated at another processor.
  - **Cache-coherency** has to be maintained — reads and writes of data in buffer must find latest version of data.

Cache Coherency Protocol

- Example of a cache coherency protocol for shared disk systems:
  - Before reading/writing to a page, the page must be locked in shared/exclusive mode.
  - On locking a page, the page must be read from disk
  - Before unlocking a page, the page must be written to disk if it was modified.
- More complex protocols with fewer disk reads/writes exist.
- Cache coherency protocols for shared-nothing systems are similar. Each database page is assigned a *home* processor. Requests to fetch the page or write it to disk are sent to the home processor.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

the first form scales better with increasing parallelism because
the number of tuples processed by each operation is typically more than the number of operations in a query.

Parallel Sort

Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner).
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:
  - The sorted partitions at each processor P_i are range-partitioned across the processors P_0, ..., P_{m-1}.
  - Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on processors P_0,..., P_{m-1} are concatenated to get the final result.

Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.
- When **scheduling** execution tree in parallel system, must decide:
  - How to parallelize each operation and how many processors to use for it.
- Determining the amount of resources to allocate for each operation is a problem.
  - E.g., allocating more processors than optimal can result in high communication overhead.
- Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

Design of Parallel Systems

Some issues in the design of parallel systems:

- Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite of failure.
  - Redundancy achieved by storing extra copy of every data item at another processor.

Design of Parallel Systems (Cont.)

- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
    - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and “catches up” on changes at the end.
- Also need support for on-line repartitioning and schema changes (executed concurrently with other processing).

Chapter 19: Distributed Databases

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites

Homogeneous Distributed Databases

- In a homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

Distributed Data Storage

- Assume relational data model
- Replication
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.

Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- **Full replication** of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.

Data Replication (Cont.)

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* is replicas exist.
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel.
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*.
- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r.
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.

Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.

Data Transparency

- **Data transparency**: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - **Fragmentation transparency**
  - **Replication transparency**
  - **Location transparency**

Naming of Data Items - Criteria

1. Every data item must have a system-wide unique name.
2. It should be possible to find the location of data items efficiently.
3. It should be possible to change the location of data items transparently.
4. Each site should be able to create new data items autonomously.

Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local **transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a **transaction coordinator**, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

- Coordinator asks all participants to ***prepare*** to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms
- Data models may differ (hierarchical, relational, etc.)
- Transaction commit protocols may be incompatible
- Concurrency control may be based on different techniques (locking, timestamping, etc.)
- System-level details almost certainly are totally incompatible.
- A **multidatabase system** is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - Creates an illusion of logical database integration without any physical database integration

Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - Technical difficulties and cost of conversion
    - Organizational/political difficulties
      - Organizations do not want to give up control on their data
      - Local databases wish to retain a great deal of **autonomy**

Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g., data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- Agreement on units of measure
- Variations in names
  - E.g., Köln vs Cologne, Mumbai vs Bombay

Query Processing

- Several issues in query processing in a heterogeneous database
- Schema translation
  - Write a **wrapper** for each data source to translate data to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g., web forms, flat file data sources
  - Queries have to be broken up and processed partly at the source and partly at a different site
- Removal of duplicate information when sites have overlapping information
  - Decide which sites to execute query
- Global query optimization

Directory Access Protocols

- Most commonly used directory access protocol:
  - LDAP (Lightweight Directory Access Protocol)
  - Simplified from earlier X.500 protocol
- Question: Why not use database protocols like ODBC/JDBC?
- Answer:
  - Simplified protocols for a limited type of data access, evolved parallel to ODBC/JDBC
  - Provide a nice hierarchical naming mechanism similar to file system directories
    - Data can be partitioned amongst multiple servers for different parts of the hierarchy, yet give a single view to user
      - E.g., different servers for Bell Labs Murray Hill and Bell Labs Bangalore
  - Directories may use databases as storage mechanism

LDAP: Lightweight Directory Access Protocol

- LDAP Data Model
- Data Manipulation
- Distributed Directory Trees

Real examples

- Vertica
- TeraData
- BigTable
- (aside on MapReduce)

http://portal.acm.org/citation.cfm?id=129894 – De Witt, Gray. CACM 1992

http://portal.acm.org/citation.cfm?id=1629197 - Stonebreaker et al. CACM 2010

http://portal.acm.org/citation.cfm?id=1629198 – Dean and Gemwhat. CACM 2010