Chapters 21-23 : Distributed Databases

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

Distributed Databases

- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - The goal is to provide a view of a single database, hiding details of distribution
  - Done for improving (local) efficiency, improving availability, …
- Heterogeneous distributed databases
  - Different software/schema on different sites
  - The goal is to integrate existing databases to provide useful functionality
  - The various databases may already exist.
- In distributed databases two types of transactions exist:
  - A **local transaction** accesses data in the *single* site at which the transaction was initiated.
  - A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

Distributed Data Storage

- Data Storage can be distributed by replicating data or be fragmenting data.
- **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.

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
- Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Data Replication

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* if 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
  - The original relation is obtained by the **union** of the fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schema 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
  - The original relation is obtained by the **join** of the fragments
- Examples:
  - Horizontal fragmentation of an account relation, by branches
  - Vertical fragmentation of an employer relation, to separate the data for e.g. salaries, functions, etc

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
  - An examples is to horizontally fragment an account relation by branches, and vertically fragment it to *hide* balances

Distributed Query Processing

Data Integration From Multiple Sources

- Many database applications require data from multiple databases
- A **federated database 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
  - Each database has its **local schema**
  - **Global schema** integrates all the local schema
    - **Schema integration**
  - Queries can be issued against global schema, and translated to queries on local schemas
    - Databases that support common schema and queries, but not updates, are referred to as **mediator** systems

Data Integration From Multiple Sources

- **Data virtualization**
  - Allows data access from multiple databases, but without a common schema
- **External data** approach allows database to treat external data as a database relation (**foreign tables**)
  - Many databases today allow a local table to be defined as a view on external data
  - SQL Management of External Data (SQL MED) standard
- **Wrapper** for a data source is a view that translates data from local to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema

Data Warehouses and Data Lakes

- **Data warehouse** is an alternative to data integration
  - Migrates data to a common schema, avoiding run-time overhead
  - Cost of translating schema/data to a common warehouse schema can be significant
- **Data lake**: architecture where data is stored in multiple data storage systems, in different storage formats, but which can be queried from a single system.

Schema and Data Integration

- **Schema integration**: creating a unified conceptual schema
  - Requires creation of **global schema**, integrating several **local schema**
- **Global-as-view approach**
  - At each site, create a view of local data, mapping it to the global schema
  - Union of local views is the global view
  - Good for queries, but not for updates
    - E.g., which local database should an insert go to?
- **Local-as-view approach**
  - Create a view defining contents of local data as a view of global data
    - Site stores local data as before, the view is for update processing
  - Updates on global schema are mapped to updates to the local views

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

Unified View of Data (Cont.)

- Variations in names
  - E.g., Köln vs Cologne, Mumbai vs Bombay
- One approach: globally unique naming system
  - E.g., GeoNames database (www.geonames.org)
- Another approach: specification of name equivalences
  - E.g., used in the Linked Data project supporting integration of a large number of databases storing data in RDF data

Query Processing Across Data Sources

- Several issues in query processing across multiple sources
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g., web forms, flat file data sources
  - Queries must 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

Join Locations and Join Ordering

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $r1 \bowtie r2 \bowtie r3$

- $r1$ is stored at site S_1
- $r2$ at S_2
- $r3$ at S_3
- For a query issued at site S_l, the system needs to produce the result at site S_l

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site $S_{I.}$
  - Ship a copy of the *r1* relation to site S_2 and compute $temp_1 = r1 \bowtie r2$ *at* S_2.
  - Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie r3$ at S_3
  - Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stores at site S_1

  Let r_2 be a relation with schema R_2 stores at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1.
  1. Compute $temp_1 \leftarrow \prod_{R1 \cap R2}$ (r1) at $S1$.
  2. Ship $temp_1$ from S_1 to S_2.
  3. Compute $temp_2 \leftarrow r_2 \bowtie$ temp1 at S_2
  4. Ship $temp_2$ from S_2 to S_1.
  5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$.

Semijoin Reduction

- The **semijoin** of r_1 with r_2, is denoted by:

$$r_1 \ltimes r_2 \qquad \Pi_{R1}\,(r_1 \bowtie r_2)$$

- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 above, $temp_2 = r_2 \ltimes r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.
- Semijoin can be computed approximately by using a Bloom filter
  - For each tuple of r_2 compute hash value on join attribute; if hash value is i, and set bit i of the bitmap
  - Send bitmap to site containing r_1
  - Fetch only tuples of r_1 whose join attribute value hashes to a bit that is set to 1 in the bitmap
  - Bloom filter is an optimized bitmap filter structure

Distributed Query Optimization

- New physical property for each relation: location of data
- Operators also need to be annotated with the site where they are executed
  - Operators typically operate only on local data
  - Remote data is typically fetched locally before operator is executed
- Optimizer needs to find best plan taking data location and operator execution location into account.