Outline

- Introduction & architectural issues
- Data distribution
- Distributed query processing
- Distributed query optimization
- Distributed transactions & concurrency control
- Distributed reliability
- Data replication
- Parallel database systems
- Database integration & querying
  - Query rewriting
  - Optimization issues
- Peer-to-Peer data management
- Stream data management
- MapReduce-based distributed data management
Multidatabase Query Processing

- Mediator/wrapper architecture
- MDB query processing architecture
- Query rewriting using views
- Query optimization and execution
- Query translation and execution
Mediator/Wrapper Architecture

The diagram illustrates the Mediator/Wrapper Architecture for global query processing. It consists of a mediator that processes queries and integrates results from local DBMSs.

- **Query Processing**: The mediator receives a query and processes it.
- **Global view**: Enables a single query interface for users.
- **Result Integration**: The mediator integrates results from different DBMSs.

The architecture supports both same and different interfaces for local DBMSs:

- **Same Interface**:
  - Local Schema
  - Wrapper
  - DBMS

- **Different Interfaces**:
  - Local Schema
  - Wrapper
  - DBMS

This architecture allows for efficient query processing and result integration across different DBMSs.
Advantages of M/W Architecture

- Wrappers encapsulate the details of component DBMS
  - Export schema and cost information
  - Manage communication with Mediator

- Mediator provides a global view to applications and users
  - Single point of access
    - May be itself distributed
  - Can specialize in some application domain
  - Perform query optimization using global knowledge
  - Perform result integration in a single format
Issues in MDB Query Processing

- Component DBMSs are autonomous and may range from full-fledge relational DBMS to flat file systems
  - Different computing capabilities
    - Prevents uniform treatment of queries across DBMSs
  - Different processing cost and optimization capabilities
    - Makes cost modeling difficult
  - Different data models and query languages
    - Makes query translation and result integration difficult
  - Different runtime performance and unpredictable behavior
    - Makes query execution difficult
Mediator Data Model

- Relational model
  - Simple and regular data structures
  - Mandatory schema

- Object model
  - Complex (graphs) and regular data structures
  - Mandatory schema

- Semi-structured (XML) model
  - Complex (trees) and irregular data structures
  - Optional schema (DTD or XSchema)

In this chapter, we use the relational model which is sufficient to explain MDB query processing
MDB Query Processing Architecture

Local/DBMS mappings

Allocation and capabilities

Global/local correspondences

Results
Query Rewriting Using Views

- Views used to describe the correspondences between global and local relations
  - **Global As View**: the global schema is integrated from the local databases and each global relation is a view over the local relations
  - **Local As View**: the global schema is defined independently of the local databases and each local relation is a view over the global relations

- Query rewriting best done with Datalog, a logic-based language
  - More expressive power than relational calculus
  - Inline version of relational domain calculus
Datalog Terminology

- Conjunctive (SPJ) query: a rule of the form
  - \( Q(T) : R_1(T_1), \ldots, R_n(T_n) \)
  - \( Q(T) \): head of the query denoting the result relation
  - \( R_1(T_1), \ldots, R_n(T_n) \): subgoals in the body of the query
  - \( R_1, \ldots, R_n \): predicate names corresponding to relation names
  - \( T_1, \ldots, T_n \): refer to tuples with variables and constants
  - Variables correspond to attributes (as in domain calculus)
  - “-” means unnamed variable

- Disjunctive query = \( n \) conjunctive queries with same head predicate
Datalog Example

With EMP(ENAME,TITLE,CITY) and ASG(ENAME,PNAME,DUR)

```syntahic
SELECT  ENAME, TITLE, PNAME
FROM    EMP, ASG
WHERE   EMP.ENAME = ASG.ENAME
AND     TITLE = "Programmer" OR DUR=24
```

```datalog
Q(ename,title,pname) :- Emp(ename,title,-)
                      Asg(ename,pname,-),
                      title = “Programmer”.

Q(ename,title,pname) :- Emp(ename,title,-)
                      Asg(ename,pname,24).
```
Rewriting in GAV

- **Global schema similar to that of homogeneous DDBMS**
  - Local relations can be fragments
  - But no completeness: a tuple in the global relation may not exist in local relations
    - Yields incomplete answers
  - And no disjointness: the same tuple may exist in different local databases
    - Yields duplicate answers

- **Rewriting (unfolding)**
  - Similar to query modification
    - Apply view definition rules to the query and produce a union of conjunctive queries, one per rule application
    - Eliminate redundant queries
### GAV Example Schema

#### Global relations

- EMP(ENAME,CITY)
- ASG(ENAME,PNAME,TITLE, DUR)

#### Local relations

- EMP1(ENAME,TITLE,CITY)
- EMP2(ENAME,TITLE,CITY)
- ASG1(ENAME,PNAME,DUR)

\[
\begin{align*}
\text{Emp}(ename, city) & :\quad \text{Emp1}(ename, title, city). & (r_1) \\
\text{Emp}(ename, city) & :\quad \text{Emp2}(ename, title, city). & (r_2) \\
\text{Asg}(ename, pname, title, dur) & :\quad \text{Emp1}(ename, title, city), \quad \text{Asg1}(ename, pname, dur). & (r_3) \\
\text{Asg}(ename, pname, title, dur) & :\quad \text{Emp2}(ename, title, city), \quad \text{Asg1}(ename, pname, dur). & (r_4)
\end{align*}
\]
GAV Example Query

Let $Q$: name and project for employees in Paris

$Q(e,p) :\text{ Emp}(e, \text{"Paris"}), \text{ Asg}(e,p,-,-).$

Unfolding produces $Q'$

$Q'(e,p) :\text{ Emp1}(e,-, \text{"Paris"}), \text{ Asg1}(e,p,-,). \quad (q_1)$

$Q'(e,p) :\text{ Emp2}(e,-, \text{"Paris"}), \text{ Asg1}(e,p,-,). \quad (q_2)$

where

$q_1$ is obtained by applying $r_3$ only or both $r_1$ and $r_3$

In the latter case, there are redundant queries

same for $q_2$ with $r_2$ only or both $r_2$ and $r_4$
Rewriting in LAV

- More difficult than in GAV
  - No direct correspondence between the terms in GS (emp, ename) and those in the views (emp1, emp2, ename)
  - There may be many more views than global relations
  - Views may contain complex predicates to reflect the content of the local relations
    - e.g. a view Emp3 for only programmers

- Often not possible to find an equivalent rewriting
  - Best is to find a \textit{maximally-contained query} which produces a maximum subset of the answer
    - e.g. Emp3 can only return a subset of the employees
Rewriting Algorithms

- The problem to find an equivalent query is NP-complete in the number of views and number of subgoals of the query.
- Thus, algorithms try to reduce the numbers of rewritings to be considered.
- Three main algorithms
  - Bucket
  - Inverse rule
  - MiniCon
LAV Example Schema

Local relations
EMP1(ENAME,TITLE,CITY)
EMP2(ENAME,TITLE,CITY)
ASG1(ENAME,PNAME,DUR)

Global relations
EMP(ENAME,CITY)
ASG(ENAME,PNAME,TITLE, DUR)

\[
\begin{align*}
\text{Emp1}(ename, title, city) & : \quad \text{Emp}(ename, city), \\
& \quad \text{Asg}(ename, -, title, -). \\
\text{Emp2}(ename, title, city) & : \quad \text{Emp}(ename, city), \\
& \quad \text{Asg}(ename, -, title, -). \\
\text{Asg1}(ename, pname, dur) & : \quad \text{Asg}(ename, pname, -, dur) \\
\end{align*}
\]
Bucket Algorithm

- Considers each predicate of the query $Q$ independently to select only the relevant views

**Step 1**
- Build a bucket $b$ for each subgoal $q$ of $Q$ that is not a comparison predicate
- Insert in $b$ the heads of the views which are relevant to answer $q$

**Step 2**
- For each view $V$ of the Cartesian product of the buckets, produce a conjunctive query
  - If it is contained in $Q$, keep it

- The rewritten query is a union of conjunctive queries
Let $Q$ be $Q(e,p) :- \text{Emp}(e, "Paris"), \text{Asg}(e,p,-,-)$.

Step 1: we obtain 2 buckets (one for each subgoal of $Q$)

$b_1 = \text{Emp1}(ename, title', city), \text{Emp2}(ename, title', city)$

$b_2 = \text{Asg1}(ename, pname, dur')$

(the prime variables (title’ and dur’) are not useful)

Step 2: produces

$Q'(e,p) :- \text{Emp1}(e, -, "Paris"), \text{Asg1}(e,p,-,).$ \hspace{1cm} (q_1)$

$Q'(e,p) :- \text{Emp2}(e, -, "Paris"), \text{Asg1}(e,p,-,).$ \hspace{1cm} (q_2)$
Query Optimization and Execution

- Takes a query expressed on local relations and produces a distributed QEP to be executed by the wrappers and mediator

- Three main problems
  - Heterogeneous cost modeling
    - To produce a global cost model from component DBMS
  - Heterogeneous query optimization
    - To deal with different query computing capabilities
  - Adaptive query processing
    - To deal with strong variations in the execution environment
Heterogeneous Cost Modeling

- Goal: determine the cost of executing the subqueries at component DBMS

- Three approaches
  - Black-box: treats each component DBMS as a black-box and determines costs by running test queries
  - Customized: customizes an initial cost model
  - Dynamic: monitors the run-time behavior of the component DBMS and dynamically collect cost information
Black-box Approach

- Define a logical cost expression
  - \( \text{Cost} = \text{init cost} + \text{cost to find qualifying tuples} + \text{cost to process selected tuples} \)
  - The terms will differ much with different DBMS

- Run probing queries on component DBMS to compute cost coefficients
  - Count the numbers of tuples, measure cost, etc.
  - Special case: sample queries for each class of important queries
    - Use of classification to identify the classes

- Problems
  - The instantiated cost model (by probing or sampling) may change over time
  - The logical cost function may not capture important details of component DBMS
Customized Approach

- Relies on the wrapper (i.e. developer) to provide cost information to the mediator

- Two solutions
  - Wrapper provides the logic to compute cost estimates
    - $\text{Access\_cost} = \text{reset} + (\text{card} - 1) \times \text{advance}$
    - $\text{reset} =$ time to initiate the query and receive a first tuple
    - $\text{advance} =$ time to get the next tuple (advance)
    - $\text{card} =$ result cardinality
  - Hierarchical cost model
    - Each node associates a query pattern with a cost function
    - The wrapper developer can give cost information at various levels of details, depending on knowledge of the component DBMS
Hierarchical Cost Model

Default-scope rules

Wrapper-scope rules

Collection scope rules

Predicate-scope rules

Query specific rules

select (Collection, Predicate)
  CountObject = ...
  TotalSize = ...
  TotalTime = ...
  etc...

Source 1:
  select (Collection, Predicate)
  TotalSize = ...

Source 2:
  select (Collection, Predicate)
  TotalSize = ...

select(PROJ, Predicate)
  TotalSize = ...

select(EMP, Predicate)
  TotalTime = ...

select(EMP, TITLE = value)
  TotalTime = ...

select(EMP, ENAME = Value)
  TotalTime = ...
Dynamic Approach

- Deals with execution environment factors which may change
  - Frequently: load, throughput, network contention, etc.
  - Slowly: physical data organization, DB schemas, etc.
- Two main solutions
  - Extend the sampling method to consider some new queries as samples and correct the cost model on a regular basis
  - Use adaptive query processing which computes cost during query execution to make optimization decisions
Heterogeneous Query Optimization

- Deals with heterogeneous capabilities of component DBMS
  - One DBMS may support complex SQL queries while another only simple select on one fixed attribute

- Two approaches, depending on the M/W interface level
  - Query-based
    - All wrappers support the same query-based interface (e.g. ODBC or SQL/MED) so they appear homogeneous to the mediator
    - Capabilities not provided by the DBMS must be supported by the wrappers
  - Operator-based
    - Wrappers export capabilities as compositions of operators
    - Specific capabilities are available to mediator
Query-based Approach

- We can use 2-step query optimization with a heterogeneous cost model
  - But centralized query optimizers produce left-linear join trees whereas in MDB, we want to push as much processing in the wrappers, i.e. exploit bushy trees

- Solution: convert a left-linear join tree into a bushy tree such that
  - The initial total cost of the QEP is maintained
  - The response time is improved

- Algorithm
  - Iterative improvement of the initial left-linear tree by moving down subtrees while response time is improved
Left Linear vs Bushy Join Tree

(a) Left Linear Join Tree

(b) Bushy Join Tree
Operator-based Approach

- M/W communication in terms of subplans
- Use of planning functions (Garlic)
  - Extension of cost-based centralized optimizer with new operators
    - Create temporary relations
    - Retrieve locally stored data
    - Push down operators in wrappers
    - accessPlan and joinPlan rules
  - Operator nodes annotated with
    - Location of operands, materialization, etc.
Planning Functions Example

Consider 3 component databases with 2 wrappers:

- $w_1.db_1$: EMP(ENO, ENAME, CITY)
- $w_1.db_2$: ASG(ENO, PNAME, DUR)
- $w_2.db_3$: EMPASG(ENAME, CITY, PNAME, DUR)

Planning functions of $w_1$

- AccessPlan $(R$: rel, $A$: attlist, $P$: pred) = scan($R$, $A$, $P$, db($R$))
- JoinPlan $(R_1$, $R_2$: rel, $A$: attlist, $P$: joinpred) = join($R_1$, $R_2$, $A$, $P$)
  - condition: db($R_1$) ≠ db($R_2$)
  - implemented by $w_1$

Planning functions of $w_2$

- AccessPlan $(R$: rel, $A$: attlist, $P$: pred) = fetch(city=c)
  - condition: (city=c) included in $P$
- AccessPlan $(R$: rel, $A$: attlist, $P$: pred) = scan($R$, $A$, $P$, db($R$))
  - implemented by $w_2$
**Heterogenous QEP**

\[
\text{SELECT ENAME, PNAME, DUR} \\
\text{FROM EMPASG} \\
\text{WHERE CITY = "Paris" AND DUR > 24}
\]
Adaptive Query Processing - Motivations

- Assumptions underlying heterogeneous query optimization
  - The optimizer has sufficient knowledge about runtime
    - Cost information
  - Runtime conditions remain stable during query execution
- Appropriate for MDB systems with few data sources in a controlled environment
- Inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions
Example: QEP with Blocked Operator

- Assume ASG, EMP, PROJ and PAY each at a different site
- If ASG site is down, the entire pipeline is blocked
- However, with some reorganization, the join of EMP and PAY could be done while waiting for ASG
Adaptive Query Processing – Definition

- A query processing is adaptive if it receives information from the execution environment and determines its behavior accordingly
  - Feed-back loop between optimizer and runtime environment
  - Communication of runtime information between mediator, wrappers and component DBMS
  - Hard to obtain with legacy databases

- Additional components
  - Monitoring, assessment, reaction
  - Embedded in control operators of QEP

- Tradeoff between reactiveness and overhead of adaptation
Adaptive Components

- Monitoring parameters (collected by sensors in QEP)
  - Memory size
  - Data arrival rates
  - Actual statistics
  - Operator execution cost
  - Network throughput

- Adaptive reactions
  - Change schedule
  - Replace an operator by an equivalent one
  - Modify the behavior of an operator
  - Data repartitioning
Eddy Approach

- Query compilation: produces a tuple \( \langle D, P, C, \text{Eddy} \rangle \)
  - \( D \): set of data sources (e.g. relations)
  - \( P \): set of predicates
  - \( C \): ordering constraints to be followed at runtime
  - \( \text{Eddy} \): \( n \)-ary operator between \( D \) and \( P \)

- Query execution: operator ordering on a tuple basis using \( \text{Eddy} \)
  - On-the-fly tuple routing to operators based on cost and selectivity
  - Change of join ordering during execution
    - Requires symmetric join algorithms such as Ripple joins
QEP with Eddy

- $D = \{R, S, T\}$
- $P = \{\sigma_P(R), R \Join JN_1 S, S \Join JN_2 T\}$
- $C = \{S < T\}$ where < imposes $S$ tuples to probe $T$ tuples using an index on join attribute
  - Access to $T$ is wrapped by JN

Result tuples
Query Translation and Execution

- Performed by wrappers using the component DBMS
  - Conversion between common interface of mediator and DBMS-dependent interface
    - Query translation from wrapper to DBMS
    - Result format translation from DBMS to wrapper
  - Wrapper has the local schema exported to the mediator (in common interface) and the mapping to the DBMS schema
  - Common interface can be query-based (e.g. ODBC or SQL/MED) or operator-based

- In addition, wrappers can implement operators not supported by the component DBMS, e.g. join
Wrapper Placement

- Depends on the level of autonomy of component DB
  - Cooperative DB
    - May place wrapper at component DBMS site
    - Efficient wrapper-DBMS com.
  - Uncooperative DB
    - May place wrapper at mediator
    - Efficient mediator-wrapper com.
- Impact on cost functions