Distributed Databases
Data Warehousing

CPS 216

Advanced Database Systems

Review

Distributed DBMS

- Top-down approach
  - Data partitioning
  - Query processing
  - Query optimization
  - Concurrency control and recovery
- Bottom-up approach
  - Query processing and optimization

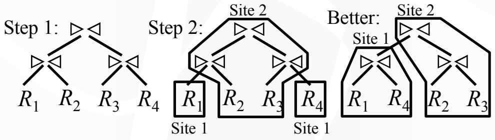
Optimizing distributed queries

What is different from optimizing centralized queries?

- New strategies: parallel joins, semijoins, …
- Plans have a new property: "interesting sites"
- Communication cost is a big factor besides I/O
  - Per-message cost, per-byte cost, CPU cost to pack/unpack data
- Parallelism: response time versus total resource consumption

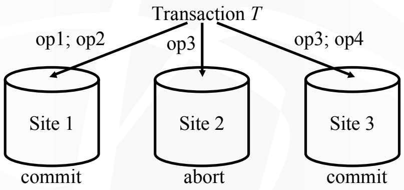
Example: two-step optimization

- Step 1 (compile time): decide the join order, join methods, and access paths
  - Same complexity as in a centralized DBMS
- Step 2 (run time): decide where to execute each operator
  - Can cope with changing load and network characteristics
  - Can use data that has been dynamically allocated to a site (caching or replication)

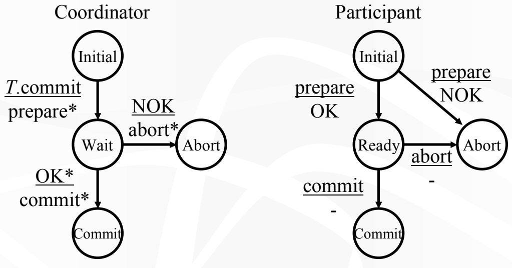


Concurrency control & recovery in Distributed DBMS

- Rich and interesting field
- We will just sample the field by looking at the problem of distributed transaction commit



Two-phase commit



Notation: Incoming message * = everyone

Outgoing message

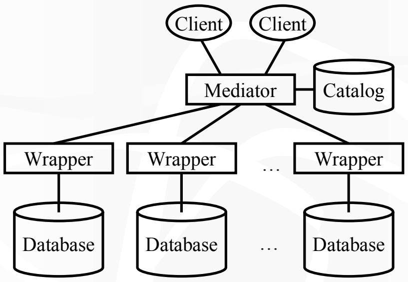
Key points of 2PC

- By sending OK a participant promises the coordinator to commit
  - But it can only commit when instructed to do so by the coordinator
  - The coordinator could tell it to abort instead
- After sending NOK a participant can abort unilaterally
- Coordinator can decide to commit only if all participants have responded OK
- Logging of all messages are required at each site

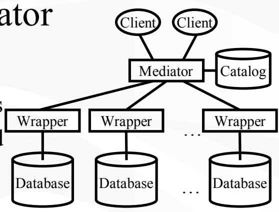
Bottom-up approach to
Building a distributed DBMS

- Data already in various sources
- Build a distributed DBMS to provide global, uniform access to all data
  - How to integrate data?
  - How to deal with heterogeneous and autonomous sources?
  - » Mediation approach

Wrapper/mediator architecture



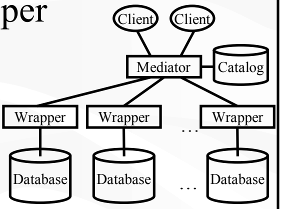
Mediator



- Accept queries from clients
- Rewrite and optimize queries
- Send subplans to be executed by wrappers
- Combine results from wrappers and perform any additional local processing necessary
- Mediator catalog stores global schema and external schema of sources as exported by wrappers

» No source-specific code in a mediator!

Wrapper



- Hide heterogeneity away from mediator
- Translate mediator requests so that they are understood by sources
  - Example: SELECT * FROM Books WHERE title LIKE '%Databases'; → a form-based search request for books with title matching "*Databases"
- Translate results returned by a source so that they are compliant with its external schema
  - Example: result HTML page → Books tuples

Query optimization with wrappers

Basic questions

- Capability: What types of subplans can be handled by a wrapper?
  - How do we enumerate valid plans?
- Cost: What is the cost of executing a subplan by a wrapper?
  - How do we pick the optimal plan?

Example: Garlic query optimization

- Haas et al., VLDB 1997
- Incorporated in DB2
- Rules for generating valid plans
  - Supplied by wrappers and mediator
  - Plugged into the optimizer
- Plans have “interesting properties”
  - Order (as in Selinger)
  - Site (where the output is produced)
  - Columns (in the output)
  - Predicates (that have been applied)
  - Cost, etc.

Example rules for a DBMS source

- wrap_access(*table*, *columns*, *predicates*) =
  SCAN$_{\text{DBMS}}$(*table*, *columns*, *predicates*)
  - Condition: *table* is at my site
  - I can handle any projection and selection (by converting them to a single-table SELECT-FROM-WHERE SQL statement)
- wrap_join(*subplan*$_1$, *subplan*$_2$, *predicates*) =
  JOIN$_{\text{DBMS}}$(*subplan*$_1$, *subplan*$_2$, *predicates*)
  - Condition: *subplan*$_1$.site = *subplan*$_2$.site = my site
  - I can handle any local join (by converting it to a multi-table SELECT-FROM-WHERE SQL statement)

Example rules for a Web source

- wrap_access(*table*, *columns*, *predicates*) =
  FETCH$_{\text{Web}}$(Books, title LIKE *string*)
  - Condition: *table* = Books, (title LIKE *string*) ∈ *predicates*
  - I can search books by title (with wildcards); no projection
- wrap_access(*table*, *columns*, *predicates*) =
  FETCH$_{\text{Web}}$(Books, author = string)
  - Condition: *table* = Books, (author = *string*) ∈ *predicates*
  - I can search books by exact author names; no projection
  - I cannot search books by title and author at the same time
- No wrap_join rule
  - I cannot handle process joins

Example rules for the mediator

- med_pushdown(*subplan*) = RECEIVE(SEND(*subplan*))
  - Condition: *subplan*.site ≠ mediator
- med_pushdown(*subplan*) = *subplan*
  - Condition: *subplan*.site = mediator
- med_access(*table*, *columns*, *predicates*) =
  ∀ *plan* ∈ wrap_access(*table*, *columns*, *predicates*):
  FILTER$_{\text{med}}$(med_pushdown(*plan*)),
  predicates – *plan*.predicates)
  - I can get the result of a single-table scan from a wrapper and then evaluate remaining selection predicates

More rules for the mediator

- med_join(*subplan*$_1$, *subplan*$_2$, *predicates*) =
  ∀ *plan* ∈ wrap_join(*subplan*$_1$, *subplan*$_2$, *predicates*):
  med_pushdown(plan)
  - Condition: *subplan*$_1$.site = *subplan*$_2$.site ≠ mediator
  - I can push down a join to a wrapper
- med_join(*subplan*$_1$, *subplan*$_2$, *predicates*) =
  JOIN$_{\text{med}}$(med_pushdown(*subplan*$_1$),
  med_pushdown(*subplan*$_2$), *predicates*)
  - I also can handle a join locally
- And more…

Plan enumeration

- Call all wrap_access and med_access rules to generate single-table access plans
- Repeatedly call all wrap_join and med_join rules to generate multi-table join plans
- Example final plans
  - FILTER$_{\text{med}}$(
    RECEIVE(SEND(FETCH$_{\text{Web}}$(Books, title LIKE *string*))),
    author = *string*), versus
    FILTER$_{\text{med}}$(
    RECEIVE(SEND(FETCH$_{\text{Web}}$(Books, author = *string*))),
    title LIKE *string*)
  - RECEIVE(SEND(JOIN$_{\text{DBMS}}$(R, S))), versus
    JOIN$_{\text{med}}$(RECEIVE(SEND(R)), RECEIVE(SEND(S)))

Costing

- Wrapper-supplied cost model
  - – Lots of work for wrapper developers
- Calibration
  - – Define a generic cost model with parameters for all wrappers
    - Example: cost = $c \cdot$ (# of tuples)
  - – Run test queries to measure the parameters for each wrapper
- Learning curve
  - – Use recent statistics to adjust cost estimates
    - Example: cost = average over last three runs

Summary of wrapper/mediator

Not all sources are created equal!

- What's in a source?
  - – Wrapper: source schema ↔ external schema
  - – Mediator: external schema ↔ global schema
- What can it do?
  - – Wrappers and mediators supply rules describing their query processing capabilities
- How much does it cost?
  - – Wrappers supply cost model, or
  - – Mediator calibrates or learns the cost model

Data warehousing

- Data resides in many distributed, heterogeneous OLTP (On-Line Transaction Processing) sources
  - – Sales, inventory, customer, …
  - – NC branch, NY branch, CA branch, …
- Need to support OLAP (On-Line Analytical Processing) over an integrated view of the data

» Store the integrated data at a central repository called the data warehouse

OLTP versus OLAP

OLTP	OLAP
• Mostly updates	• Mostly reads
• Short, simple transactions	• Long, complex queries
• Clerical users	• Analysts, decision makers
• Goal: ACID, transaction throughput	• Goal: fast queries

Warehousing versus mediation

Warehousing	Mediation
• Eager "integration"	• Lazy "integration"
– In advance: before queries	– On demand: at query time
– Answer could be stale	– Answer is more up-to-date
• Copy data from sources	• Leave data at sources
– Need to maintain consistency	– No need to maintain consistency
– Query processing is local to the warehouse	– Sources participate in query processing
• Faster	
• Can operate when sources are unavailable	

Maintaining a data warehouse

Buzz word: the "ETL" process

- Extraction: extract relevant data and/or changes from sources
- Transformation: transform data to match the warehouse schema
- Loading: integrate data/changes into the warehouse

» Can still use a wrapper/mediator architecture

Warehouse data = materialized views

- If the transformation process can be captured by SQL, then warehouse data can be seen as a view

  -
    ```
    CREATE VIEW warehouse_table AS
      SELECT …
      FROM source_table1, source_table2, …
      WHERE …;
    ```
- Except the view is materialized
  - That is, the result is stored
  - And needs to be maintained when source data changes

Maintaining materialized views

$V_{\text{old}} = Q(R_{\text{old}}, \ldots)$

Change detected: $R_{\text{new}} \leftarrow R_{\text{old}} - \nabla R \cup \Delta R$

What is V_{new}?

- Recomputation: $V_{\text{new}} \leftarrow Q(R_{\text{new}}, \ldots)$
  - Done periodically, e.g., every "night"
  - What if there is no "night," e.g., an international organization?
  - What if recomputation takes longer than a day?
- Incremental maintenance
  - Compute only the changes to V: ∇V and ΔV
  - Apply the changes to V_{old}: $V_{\text{new}} \leftarrow V_{\text{old}} - \nabla V \cup \Delta V$
    - » Potentially much faster if changes are small

Incremental maintenance

Example: $V = \sigma_p R$

- Change: $R_{\text{new}} \leftarrow R_{\text{old}} - \nabla R$
  - $V_{\text{new}} = \sigma_p R_{\text{new}} = \boxed{\sigma_p (R_{\text{old}} - \nabla R) = \sigma_p R_{\text{old}} - \sigma_p \nabla R}$
    $= V_{\text{old}} - \nabla V$
- Change: $R_{\text{new}} \leftarrow R_{\text{old}} \cup \Delta R$
  - $V_{\text{new}} = \sigma_p R_{\text{new}} = \boxed{\sigma_p (R_{\text{old}} \cup \Delta R) = \sigma_p R_{\text{old}} \cup \sigma_p \Delta R}$
    $= V_{\text{old}} \cup \Delta V$

Change propagation equations

Change propagation

- More change propagation equations
  - $(R \cup \Delta R) \bowtie S =$
    $(R \bowtie S) \cup (\Delta R \bowtie S)$
  - $(R - \nabla R) \bowtie S =$
    $(R \bowtie S) - (\nabla R \bowtie S)$
- Repeatedly apply change propagation equations to "factor out" changes
  - $(\sigma_{pr} (R \cup \Delta R)) \bowtie_{prs} \sigma_{ps} S =$
    $(\sigma_{pr} R \cup \sigma_{pr} \Delta R) \bowtie_{prs} \sigma_{ps} S =$
    $(\sigma_{pr} R \bowtie_{prs} \sigma_{ps} S) \cup (\sigma_{pr} \Delta R \bowtie_{prs} \sigma_{ps} S)$

Self-maintainability

- A warehouse is self-maintainable if it can be maintained without accessing the sources
- Self-maintainable: $V = \sigma_p R$
- Not self-maintainable: $V = R \bowtie S$
  - ΔR and ∇R need to be joined with S
  - ΔS and ∇S need to be joined with R
  - Problem: need to query the source for maintenance
    - What if the source/network is slow?
    - What if the source/network is down?
    - What if the source has been updated again?

Making warehouse self-maintainable

- Add auxiliary views

Example: Order $\bowtie_{O.\text{OID} = L.\text{OID AND } O.\text{month} = \text{'nov' AND } L.\text{product} = \text{'book'}}$ Lineitem

- Naïve approach: add base tables O and L
- Better approach: push selections down and then add selection views $\sigma_{\text{month} = \text{'nov'}}\, O$ and $\sigma_{\text{product} = \text{'book'}}\, L$
- Use constraints
  - The join is a foreign-key join (L.OID references O.OID), so only $\sigma_{\text{month} = \text{'nov'}}\, O$ is needed
  - If we only insert matching orders and lineitems together, then no auxiliary view is needed

Next time

- Warehouse design
- Data cube
- ROLAP versus MOLAP