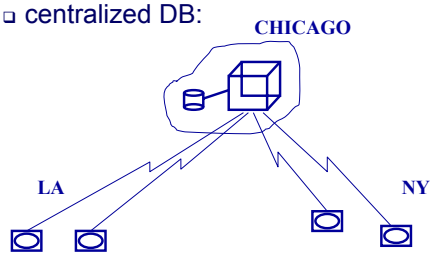
15-721 Database Management Systems

Distributed DBMS and R*

Instructor: Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

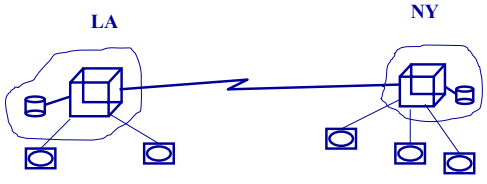
Problem – definition

- centralized DB:



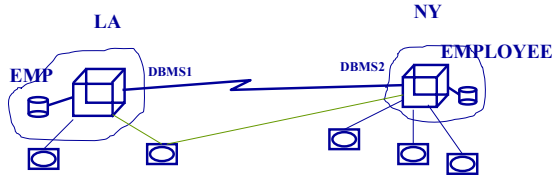
Problem – definition

- Distr. DB:
- DB stored in many places
- ... connected



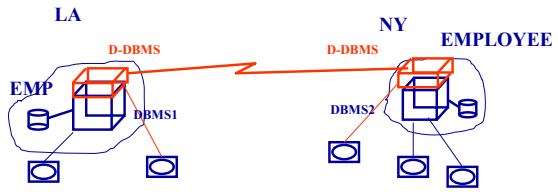
Problem – definition

now: connect to LA; exec sql select * from EMP; ...
connect to NY; exec sql select * from EMPLOYEE; ...



Problem – definition

ideally: connect to distr-LA; exec sql select * from EMPL;



Requirements?

- location transparency
- performance transparency (-> distr. q-opt)
- copy transparency
- transactions transparency
- fragment transparency
- schema transparency
- local dbms transparency
- (no system has all these features)

What's new?

- Query Optimization
  - communication cost
  - larger search space
  - load balance
  - speed, cost, space, time differences on machines

What's new?

- Query Optimization
- Concurrency Control

What's new?

- Query Optimization
- Concurrency Control
  - need distributed algorithms (distr. deadlock)

What's new?

- Query Optimization
- Concurrency Control
- Recovery

What's new?

- Query Optimization
- Concurrency Control
- Recovery
  - much more complex; more parts that can fail; 'two-phase commit'

What's new?

- Query Optimization
- Concurrency Control
- Recovery
- multiple copies; fragments

What's new?

- Query Optimization
- Concurrency Control
- Recovery
- multiple copies; fragments
  - (but, at most 2, in practice)

D-DBMS in practice

Why would one need a D-DBMS?

D-DBMS in practice

Why would one need a D-DBMS?

- geographic distribution / performance
- off-loading mainframes with local processing
- 'sins of the past' - integrating legacy systems

D-DBMS in practice

- there are products (IBM Data Joiner, Oracle*)
- BUT: they are not commercially as successful as we would expect! - why?

D-DBMS - why not?

Speculations:

- data warehouses (copy DBs locally! Sears, Wal-Mart, Kmart)
- D-DBMSs would cut down sales of D/W products
- Distributed query optimization is immature

D-DBMS - other issues?

Integration of data sources: desirable, because of the web - remaining issues:

- semantic consistency (e.g., salaries before/after taxes)
- authentication
- 2-phase-commit on top of legacy databases

Conclusions

D-DBMS research produced great ideas, useful for

- parallel dbms / “active disks” / sensors
- p2p (peer to peer networks)
- ‘middle-ware’

Conclusions

Namely:

- 2 phase commit
- distributed q-opt - semi-joins/bloom-joins
- distributed catalog
- distributed deadlock detection

System R* architecture

Citations

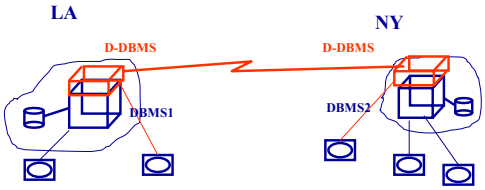
R. Williams, et al., *R* : An Overview of the Architecture*, IBM Research Report RJ3325

Mohan, Lindsay, and Obermark, *Transaction Management in the R* Distributed Database Management System*, TODS 11(4), 1986

Detailed outline

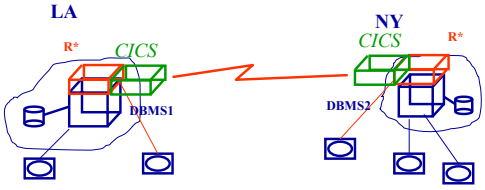
- Environment
- Object naming
- Distributed Catalogs
- Xact management - commit protocols
- Query Optimization
- Concurrency Control and Recovery
- SQL changes

Environment: R*



- Assumptions: unreliable medium, when messages arrive they arrive intact, no duplicates

Environment: CICS



- CICS handles communication, terminal I/O, program and task management

Environment: Relations

- dispersed
- replicated
- Partitioned
  - horizontal/vertical
  - lossless
  - record reconstruction
- snapshots

Object naming

No global naming system (why?)
Instead: System Wide Names (SWN)

- by attaching ‘site’ on user-names
- by attaching ‘birth-site’ on tables

e.g.:
bruce.EMPLOYEE ->
bruce@san-jose.EMPLOYEE@yorktown

Catalog entries

- Object SWN
- Object type and format
- Access paths available
- Mapping if view to lower-level objects
- Statistics for query optimization

Distributed catalogs

- Q: where and how should we store the schema?
- A1: fully replicated (but:....)
- A2: single copy (but:...)
- A3?

Distributed catalogs

- Q: where and how should we store the schema?
- A1: fully replicated (but:....)
- A2: single copy (but:...)
- A3: only birth site keeps moving info - thus

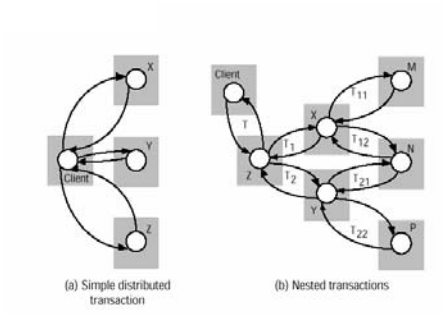
Distributed catalogs

- A3: only birth site keeps moving info - thus each site has
  - local schema +
  - moving info (for items 'born' here) and
  - birth sites of global objects
- thus: <= 2 messages are enough to locate non-local object

Detailed outline

- Environment
- Object naming
- Distributed Catalogs
- Xact management - commit protocols
- Query Optimization
- Query execution
- SQL changes

Distributed Transactions



D-Transaction management

- Q: how to give xact-ids?
- A: site-id & sequence#
  - ordered (to break deadlocks)

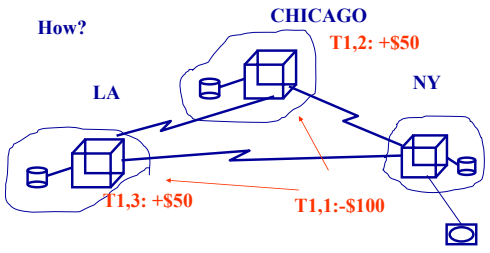
D-Transaction management

- Problem: eg., a transaction moves
  $100 from NY -> $50 to LA, $50 to Chicago
- 3 sub-transactions, on 3 systems, with 3 W.A.L.s
- how to guarantee atomicity (all-or-none)?
- Observation: additional types of failures (links, servers, delays, time-outs ....)

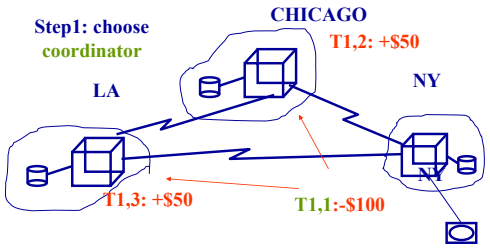
D-Transaction management

- Problem: eg., a transaction moves
  $100 from NY -> $50 to LA, $50 to Chicago

D-Transaction management



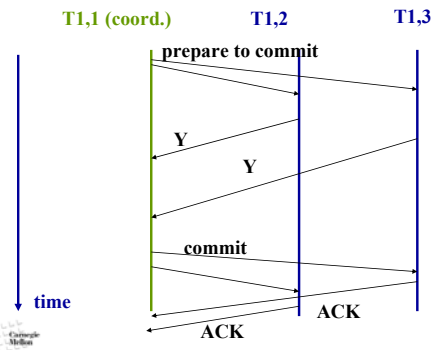
D-Transaction management



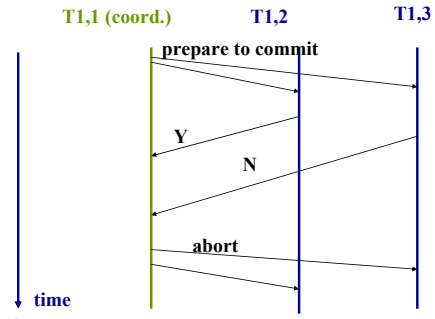
Distributed recovery

- Step 2: execute a protocol, e.g., “2 phase commit”

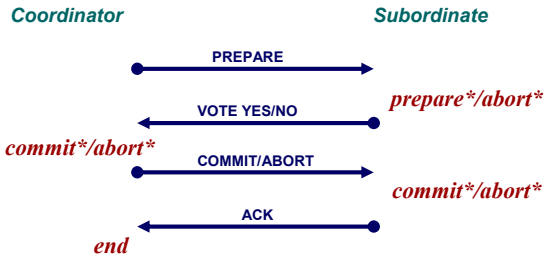
2 phase commit (success)



2 phase commit (failure)



2PC details/logging



2PC principles of operation

- 4 types of messages:
  prepare, vote y/n, commit/abort, ack
- 4 types of log records:
  prepare*, commit*, abort*, end

- Subordinates force-write log records – why?
  (never ask coordinator about that info)
- Why are ACKs sent?
  (to ensure everyone knows final outcome)

Blocking

- Subordinate blocks waiting for message
- Example: sub. has voted Yes, waiting for the decision of the coordinator
  - Subordinate cannot decide unilaterally
  - But, data held cannot be released to other transactions
  - If coordinator failed, sub must wait until it recovers
- Unilateral aborts are OK

2PC logging and traffic

- Committing Transaction:
  - Subordinate
    - Writes 2 records (prepare*, commit*)
    - Sends 2 messages (YES vote and ACK)
  - Coordinator
    - Writes 2 records (commit*, end)
    - Sends 2 msgs to each subord (prepare and commit)
- If everything goes well:
  - 3(N-1) messages.
  - The ACK messages are not counted since the protocol can function correctly without them

2PC and Failures

- Assumptions
  - recovery exists both sides
  - all failed nodes ultimately recover
- What happens if recovery finds node in
  - **prepared** state
    (periodically polls coordinator to find what happened)
  - transaction alive at crash, **no** log information
    (don't know, don't care, undo, write abort record)
  - **commit** or **abort** state
    (periodically send commit or abort to no-ack subords)

2PC and Failures (cont.)

- Coordinator notices subordinate failure.
  - If subordinate **has not sent vote**
    coordinator aborts transaction
  - If subordinate **has not sent ACK**
    coordinator hands Xtion over to recovery process
- Subordinate notices coordinator failure.
  - If subordinate **has not sent vote (not prepared)**
    subordinate *unilaterally* aborts transaction
  - If subordinate **is in prepared state**
    subordinate hands Xtion over to recovery process

2PC and Failures (cont.)

- Recovery receives inquiry from prepared subord
  - If **there is final information** about Xtion
    sends abort or commit accordingly
  - If **no information is found** about Xtion
    abort

Hierarchical 2PC

- Hierarchical 2PC: simple extension
- How save messages/communication/blocking?
- “Presumed abort” and “Presumed Commit”

Presumed Abort

- Obs.1: it is safe to "forget" a Xtion after deciding to abort
  - ⇒ Abort need not be *, no ACKs are needed for aborts
  - ⇒ No end record after writing an abort record
  - ⇒ Subord failure => no need to do anything
- Obs.2: Partially read-only Xtion at a sub node
  - ⇒ Subordinate only needs "forget" Xtion
  - ⇒ Messages: PREPARE, READ VOTE, COMMIT (only to writers)

PA Protocol Overhead

- Completely read-only Xtion:
  - Noone writes log
  - Each nonleaf sends *prepare* to each subordinate
  - Each nonroot sends *READ vote*
- Partially read-only Xtion:
  - Each nonleaf
    - sends *prepare* and *commit* to update subords
    - sends *prepare* to read-only subords
    - Writes prepare*, commit*, end (if it updates) log records
    - If nonroot, sends *YES vote* and *ACK* to cooordinate
  - Read leaf = read-only PA
  - Update leaf = subordinate in regular 2PC

Presumed Commit

- Observation: Transactions usually commit ☺
- Cheaper to:
  - Require ACKs for Aborts
  - Eliminate ACKs for commits
- Force only abort*, no information means commit!
  - Is this going to work?
    (No! Commit after crash after sending out "prepare"!)
- Record subord names before prepared state
  - ⇒ Subordinates as in PA; coord writes collecting*
  - ⇒ Read-only optimizations apply here

PC Protocol Overhead

- Completely read-only Xtion:
  - Each nonleaf writes collecting* and commit records
  - Each nonleaf sends *prepare* to each subordinate
  - Each nonroot sends *READ vote*
- Partially read-only Xtion:
  - Root
    - sends *prepare* and *commit* to subords that sent YES vote
    - sends *prepare* to read-only subords
    - If root, collecting*, commit* log records
  - Each nonleaf, nonroot
    - sends *prepare* and *commit* to subords that sent YES vote
    - sends *prepare* to read-only subords
    - sends YES vote to coordinator if update subtree
    - writes collecting*, prepared*, commit
  - Read leaf = read-only PA
  - Update leaf = sends YES, writes prepare* and commit

Distributed recovery

- Many, many additional details (what if the coordinator fails? what if a link fails? etc)
- and many other solutions (e.g., 3-phase commit)

Detailed outline

- Environment
- Object naming
- Distributed Catalogs
- Xact management - commit protocols
- Query Optimization
- Query execution
- SQL changes

Distributed Q-opt

- Steps:
  - parse
  - resolve names
  - authorization
  - compilation + plan generation
    - binding? (e.g., an access path may be dropped mid-flight!)

Distributed Q-opt

- Q: how to do binding?
  - A1: at a chosen site (-> ~ centralized)
  - A2: at the originating site (but: needs much info, which may be out-dated)
  - A3: distributed binding

Distributed binding

- master site decides inter-site issues + high level binding
- local sites do low-level decisions

Local optimality: OK

global optimality: NOK

Solution: Master site sends global plan; local sites complain, if things changed

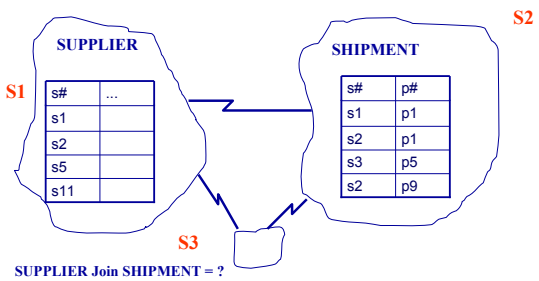
Distributed q-opt

- cost to minimize?

Distributed q-opt

- cost to minimize?
- cost = CPU + I/O + communication
  - comm. cost =
    msg-cost * #messages +
    byte-cost * #bytes
- (could minimize elapsed time, instead...)

Distributed Q-opt –joins



Distributed q-opt - join plans

Joins: join order + join method + LOCATION

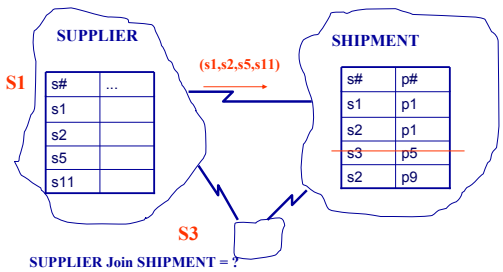
Distributed q-opt - join plans

SEVERAL choices - R* chooses one of 5:

- (a) ship inner to S1; join there
- (b) ship outer to S2, tuple-at-a-time
- (c) ('semi-join'): reduce inner; ship that to S1
- (d) ship both tables to a third site
- (e) ship outer to a third site; do (c)

Semijoins

- Idea: reduce the tables before shipping



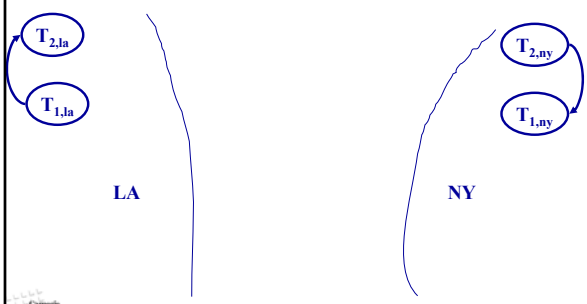
Semijoins

- Formally:
- SHIPMENT' = SHIPMENT $\triangleright\!\!<$ SUPPLIER

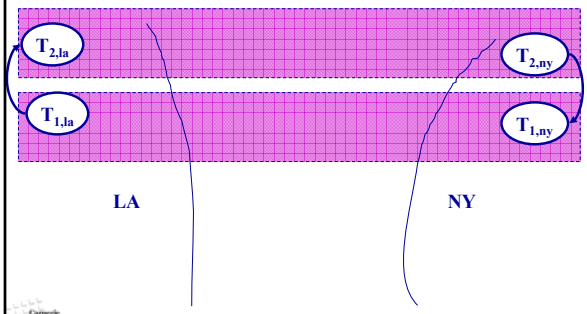
Detailed outline

- Environment
- Object naming
- Distributed Catalogs
- Xact management - commit protocols
- Q-opt
- Query execution
- SQL changes

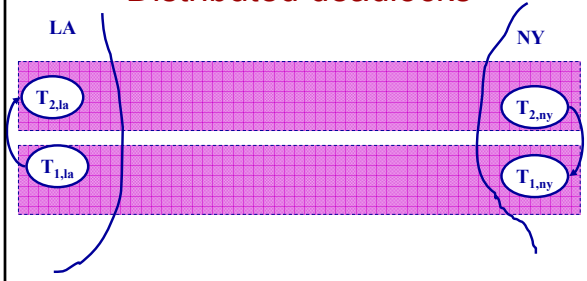
Distributed deadlocks



Distributed deadlocks



Distributed deadlocks



- cites need to exchange wait-for graphs
- clever algorithms, to reduce # messages

Distributed deadlocks

- cites need to exchange wait-for graphs
- clever algorithms, to reduce # messages
  - naively: each site ships its wait-for strings, until all have all
  - anything better?

Distributed deadlocks

- anything better?
- A: each site ships ONLY the strings where 'first-xact-id' < 'last-xact-id'
  - (any other ordering, is fine!)
- Eg: LA: T1-> T2; NY T2->T1
  - only NY will send

Detailed outline

- Environment
- Object naming
- Distributed Catalogs
- Xact management - commit protocols
- Q-opt
- Query execution
- SQL changes

SQL extensions

- DEFINE SYNONYM <rel-name> AS <SWN>
- DISTRIBUTE TABLE <t-name> HORIZONTALLY | VERTICALLY | REPLICATED ...
- DEFINE SNAPSHOT ...
- REFRESH SNAPSHOT
- MIGRATE TABLE ...

Conclusions

- 2 phase commit
- distributed q-opt; distr. deadlock detection
- distributed catalog