Distributed databases

in brief

Serge Abiteboul

INRIA

April 29, 2010

Introduction

The topic in general

- little theory! “fluffy”?
- technically complex/fun and challenging
- very important because of the Web
- uses for a lot of what you learnt in classical databases

Distributed database systems

company database: provides a unique logical access to all data

company network: allows decentralized processing

contradiction is only apparent:

- centralized access
- to physically distributed data

distributed database systems

Distributed DB: large quantity of structured data residing on several computers (over a network)

Distributed DBMS: large piece of software that allows to have a unique logical access to this data

Warning: centralized database is sometimes the best solution

Two views of distribution

Take a big database and distribute it:

1. put portions on different machines
2. replicate portions
3. more reliability and availability
4. better performance

Take many small databases and integrate them

1. unique entry point to several resources
2. keep them autonomous
3. do not interfere with local operations

Issue in both cases: transparency of data location

Advantages of distribution

- performance
- cost
- reliability
- resource sharing
- load balancing
- autonomy
- modularity

Disadvantages of distribution

- performance
- cost
- complexity
- inconsistency
- security

Architectural issues

Transparency: See only what you should see!

1. data independence
2. network transparency
3. replication transparency
4. fragmentation transparency
5. model/language transparency

3 dimensions

1. distribution of data $\rightarrow$ distributed vs. centralized system
2. distribution of control $\rightarrow$ autonomy
3. heterogeneity of systems $\rightarrow$ hardware, software, network

Ansi/Sparc architecture revisited

Centralized database – 3 level hierarchy

1. external schemaS
2. conceptual schema
3. internal schema

Distributed database – 4 level hierarchy

1. external schemaS
2. global conceptual schema
3. local conceptual schemaS
4. local internal schemaS

Typology: level of autonomy of the local databases

An illustration of a problem

- 8 copies of the same relation on different sites
- updates come from all sites
- sites 3 and 5 decide to add $100 to some entity A
- they send messages to every one
- site 2,4,6,8 reply OK
- for some reason sites 1 and 7 do not reply
- site 5 decides to abort the current transaction
- how do we manage this activity?
- how do we recover from failures?
- transaction, concurrency and recovery in presence of replication

Organization

1. fragmentation and allocation
2. query processing and optimization
3. transaction and concurrency control

Integration, fragmentation and allocation

Bottom-up approach

- Integration of databases

Top-down approach

- design the GCS
- distribute the data to obtain LCS
- relational model: split relations **fragmentation**
- assign fragments to sites: **allocation**

These issues are clearly not independent

Example

- EMPLOYEE RELATION E(enum,name,loc,sal,...)
- CURRENCY RELATION C(country,value,...)
- 12 branches of about same size S1,...,S12
- 6 are in LA, 4 in SF, 2 in SB
- 80% of queries in LA/SF/SB sites refer to EMPLOYEE in LA/SF/SB
- 10% queries in LA/SF/SB refer to CURRENCY
- 3 databases DB-LA, DB-SF, DB-SB
- on each db, the local employees + a copy of C
- if this is too expensive, merge SF and SB sites
- or keep C in SF

Distributed database design

From centralized db design

1. conceptual schema (GCS here)
2. physical schema

New

1. design of fragments
   what should be the fragments
2. physical design for fragments
   where should they go
   storage organization and access paths

Load balancing

- distribute data and processing
- move data to processing or **processing to data**

Fragments: why, how

WHY?

1. same advantage as distribution: performance, availability, reliability, locality (put the right data at the right place)
2. granularity: entire relation is a too large unit of distribution

HOW?

1. horizontal $\sigma_C(R), \sigma_{\neg C}(R)$
2. vertical $\Pi_{AB}(R), \Pi_{AC}(R)$
3. hybrid $\sigma_C(R), \Pi_{AB}(\sigma_{\neg C}(R)), \Pi_{AC}(\sigma_{\neg C}(R))$
4. granularity/degree of fragmentation
   e.g.: too few fragments: little concurrency
   (distributed file systems)
   e.g.: too many fragments: overhead in reconstructing relation

Fragments: where

each fragment on a site: single copy (partitioned db)

replication

- improves query performance
- improved reliability
- cost in updates
- more complex concurrency control
- real systems: often partial replication

Property of fragmentation: reconstructible

reconstructible: no data is lost and one can reconstruct the database using relational algebra

kind	*decomposition*	*reconstruction*
horizontal	σ	$\cup$
vertical	Π	$\bowtie$

simple/complex selection criteria for horizontal fragmentation

What is the data unit?

1. in horizontal: entity is a tuple
   (each t in R is in some fragment)
2. in vertical: entity is a portion of tuple (a property)

Property of fragmentation: disjointness

disjointness facilitates the task: an entity is present in only one fragment

most frequently asked queries: $\sigma_{sal<30}(R), \sigma_{20<sal}(R)$

candidate fragments: $\sigma_{sal<30}(R), \sigma_{20<sal}(R)$ – non disjoint

alternative $\sigma_{sal<=20}(R), \sigma_{20<sal<30}(R), \sigma_{30<=sal}(R)$ – disjoint

disjoint vs. non-disjoint

- disjoint is nice and facilitates updates
- non-disjoint may speed-up some queries
  some form of replication

Fragmentation

How do we get **reconstructible and disjoint**?

- generate these “automatically”
- often done “manually” by the DBA & checked

3 main techniques

1. primary horizontal decomposition
2. derived horizontal decomposition
3. vertical decomposition

Derived horizontal decomposition

- E(enum,name,sal,loc,...)
- J(enum,project)

horizontal decomposition of E: loc=SA and loc=SB
FAQ: given some emp name, list his/her projects

E_1

enum	*name*	*loc*	*sal*
5	*john*	*sa*	10
8	*sally*	*sa*	12
...			

E_2

enum	*name*	*loc*	*sal*
12	*manon*	*sb*	20
4	*bob*	*sb*	12
...			

J

enum	*project*
5	*data bases*
8	*vlsi*
...	
12	*data bases*
4	*www*

J_1

enum	*project*
5	*data bases*
8	*vlsi*
...	

J_2

enum	*project*
12	*data base*
4	*www*
...	

Derived horizontal decomposition

R decomposed to F_1,...,F_n
S decomposed to S $\ltimes$ F_1, ..., S $\ltimes$ F_n
condition for this to work:

$$\begin{array}{ll} \textit{reconstruction} & S = \bigcup (S \ltimes F_i) \\ \textit{disjoint } (i \neq j) & (S \ltimes F_i) \cap (S \ltimes F_j) = \emptyset \end{array}$$

- conceptual modelling
    1. link between R and S
    2. R is the owner of R and S the member
- S has a foreign key X from R
    1. means that X is a key in R
    2. for each tuple t in S, t[X] is in R
    3. sufficient condition for reconstruction and disjoint

Vertical fragmentation

normalization: split relation vertically for semantic reasons
vertical fragment: split more for distribution reasons
Example: E(enum,name,loc,sal)

- E1(enum,name,loc)
- E2(enum,sal)

Reconstruction - lossless join: $R = \bowtie R_i$

1. sufficient condition: key X is repeated in each fragment

Allocation (no replication)

- Where to put the fragments in absence of replication
- Optimization problem
  1. develop a cost/performance model
  2. minimize cost: storage, processing, communication
  3. maximize performance: best response time, largest system throughput
- Very complex problem in general
- If the solution does not meet the requirements (too slow), replicate resources

Replication

- replicate data
- trade-off query (faster) vs. update (slower)
  - actually a query may also become slower since we cannot read a replicate until all updates are performed
- what to replicate and where
- again a complex optimization problem
- use a greedy approach

while not stable do
    for each possible replication of some fragment
        what is the benefit?
        what is the cost?
    replicate one such that
        (benefit $-$ cost) > 0
        (benefit $-$ cost) is maximal

Replication in materialized views

- instead of replicating a relation, materialize a view
- frequent in distributed environment
  - make data available locally (local copy)
- Update propagation
  - update db: propagate to materialized views
  - update view: propagate/translate to a database update

Integrity control in distributed contexts

- intra fragment: like in centralized case
- inter fragment: requires messages – expensive
- Example: G and J on two different sites
  - G(eno,jno,resp,duration), J(jno,jname,budget)
  - constraint: $\forall g \in G\ (\exists j \in J\ (g.jno = j.jno))$
  - trigger on insert-in-G(42,32,“programmer”,12)

Query processing

Query processing

what is the problem?

- a query arrives at site i and uses data from sites j and k

query on the GCS $\Rightarrow$ program on the local physical schemas

Example

G(eno,jno,resp,duration), E(eno,ename,title)

- $E_1@3 = \sigma_{eno<=45}(E)$, $E_2@4 = \sigma_{eno>45}(E)$
- $G_1@1 = \sigma_{eno<=45}(G)$, $G_2@2 = \sigma_{eno>45}(G)$

query at site 5:

select ename
from E, G
where E.eno = G.eno
and resp = “manager”

Exemple - continued

$\Pi_{ename}(\sigma_{resp="manager" \wedge E.eno=G.eno}(E \times G))$
$\Pi_{ename}(E \bowtie_{eno} (\sigma_{resp="manager"}(G)))$

strategy1: send all to site 5 and compute

strategy2: proj/sel in G_1 then send to site 3 compute join in site 3

same thing for G_2 and site 4

send both results to site 5 and compute union

Goal: minimize costs

rough idea – assume

$$\text{CPU} << \text{I/O} << \text{communication}$$

approach

- minimize communication cost only
- reduce to problem of centralized db
  then minimize local processing and I/O

problem: this is based on slow communication

- e.g., kilobytes per second
- LAN : bandwidth same order of magnitude as the disk

A standard possible architecture

Layers

- decomposition
  SQL on GCS $\Rightarrow$ algebraic query on GCS
- localization
  algebraic query on GCS $\Rightarrow$ algebraic query on LCS's
- global optimization (focus on communication)
  optimize communication
- local optimization (I/O and processing)
  generate query plans for the local queries

Query processing

- Like in centralized query processing
- use reducers, access path, join ordering as before
- goal is reduction of CPU + IO + communications
- size of temporary results is critical if I have to ship them
- response time vs. total time
- search space is even larger because you have the choice on where to evaluate an operation
- new technique: semi-join

Importance of join ordering

- Decide **where** to perform joins
- Determine data transfer
- Ex: E@S1 $\bowtie_{eno}$ G@S2 $\bowtie_{jno}$ J@S3
- 5 alternatives:
  1. E $\rightarrow$ S2; join; temporary result $\rightarrow$ S3
  2. G $\rightarrow$ S1; join; TR $\rightarrow$ S3
  3. G $\rightarrow$ S3; join; TR $\rightarrow$ S1
  4. J $\rightarrow$ S2; join; TR $\rightarrow$ S1
  5. E,J $\rightarrow$ S2; join
- to choose: need to know sizes of E,G,J, E $\bowtie$ G, J $\bowtie$ G,
- we discarded: E $\rightarrow$ S3 (not as good as last)

Semi-join

- important technique for distributed databases
- R(U) and S(V)
- definition: $R \ltimes S = \Pi_U(R \bowtie S)$
- key observation

  $R \bowtie S = (R \ltimes S) \bowtie S$

  $(R \ltimes S) \bowtie (S \ltimes R)$
- Semi-join algorith for computating join
  - send $\Pi_{U \cap V}(S)$ to site 1
  - compute $R \ltimes S$ and send it to site 2
  - compute result
  - communication cost: $size(\Pi_U(S)) + size(R \ltimes S)$
  - communication for join: size(R)

Is it useful?

size(R)<size(S), R on site 1, S on site 2

size($\Pi_{U \cap V}$(S)) + size(R $\ltimes$ S) vs. size(R)

always more processing

sometimes less communication

Bit vector filtering - Based on Bloom Filter

a technique to compute semi-join

R@1 $\ltimes$ S@2, semi-join on attributes W = $U \cap V$

hash function F: tup(W) $\rightarrow$ [1..N]

compute F(π_W(S)) (subset of [1..N])

send it as a bit vector to site 1

compute R1 = { r in R | F(r) in F(π_W(S)) }

Key observation: R $\ltimes$ S $\subseteq$ R1 $\subseteq$ R

send R1 to site 2 and compute result there

false positive: R1 - (R $\ltimes$ S)

Bit vector filtering - Based on Bloom Filter

- advantage: less communications
- disadvantage: more I/O (e.g., 2 scans of S)
- disadvantage vs. semijoin: false positive
- possibly large saving in communications if size of projected tuple is large
- variations
  - compress the bit vector (does not work much)
  - send bit vectors back and forth (more semi-joins) - rarely effective
  - use several hash functions with the same bit vector (important saving)

Details of join algorithms

suppose you want to perform the following algorithm

for each r in R, compute r $\bowtie$ S

- R is the external (site 1)
- S is the internal relation (site 2)

ship-whole vs. fetch-as-needed

4 strategies:

1. ship-whole-external
   send R to site 2: join can be performed as soon as tuples start arriving
2. ship-all-internal
   send S to site 1: we have to wait until S entirely arrived to process first r in R
3. fetch-as-needed – semi-join

   for each tuple r in R do
       send Π_V(r) to site 2
       send S $\bowtie$ Π_V(r) to site 1
       done

   possibly very bad in term of communications
4. send both to a third site

if the relations are sorted by the join attributes, we can proceed in a pipeline manner - send pages of data

Exemple

G(eno,jno,resp,duration), J(jno,jname,budget)
external J $\bowtie$ internal G on JNO
index on JNO in G

1 ship J	good use of index in G
2 ship G	better than 1 if size(G) $<<$ size(J)
	local processing may be expensive
3 semi-join	better than 1 if size(G$\rtimes$J) $<<$ size(J)
	good use of index in G
4 ship-both	always bad

- If G is much larger and communication is expensive: choose 2
- if J is small or if many tuples match, choose strategy 1
- otherwise, choose 3

Distributed sorting

Deadlock problems in query processing
R is fragmented in 2 “producers”

p1: 1, 3, ..., 999, 1002, 1004, ..., 2000
p2: 2, 4, ..., 1000, 1001, 1003, ..., 1999

scenario with 2 consumers
p1 and p2: sort, then send odd to c1 and even to c2
c1 and c2: merge

problem: c1 needs to see 1001 to output 1
deadlock if buffers are too small
possible fix: p1 and p2 send dummies regularly to let each site know about their state

Transaction and concurrency control

Transaction as in the centralized case

actions: r[x], w[x]

partial order on the operations

Note: each write is an arbitrary function of all previous reads of the transaction

conflicting operations

$read_1[x]$	$write_2[x]$
$write_1[x]$	$read_2[x]$
$write_1[x]$	$write_2[x]$

schedule: indicates how a set of transactions was executed

serial schedule: one transaction runs first, then another one...

serializable/correct schedule: equivalent to a serial schedule

Schedule is serializable iff its graph is acyclic

As before

two main techniques

1. 2 Phase Locking
   1. a transaction need a read/write lock before reading/writing an entity
   2. once a transaction released a lock, it cannot acquire more locks
   3. 2PL can produce deadlocks (abort transaction)
2. Timestamping
   1. Put your timestamp on entities you update
   2. If you access an entity with a younger timestamp than you, abort

Distributed concurrency control

non-replicated databases

- notion of serializability extends easily
- techniques such as 2PL and TS
- deadlock management is harder

replicated databases: more complicated scheduling

- one-copy-serializable
- Read-One-Write-All ROWA

CC without replication

one local scheduler at each site

global scheduler = union of local schedulers

local locks

serializability theory extends to this context

2PL guarantees serialiazability/correctness

Problem 1: deadlock management

Technique 1: Prevention

- e.g., use a predefined ordering of resources (impractical)
- e.g., analysis of code: difficult to know which data will be used
- safe, no redo or rollback necessary

Technique 2: Avoidance

- e.g., time-out
- e.g., priorities (e.g., timestamp)
- T_j locks A, T_i request A
- wait-die

  if T_i higher priority then T_i waits
  else T_i aborts

- wound-wait

  if T_i higher priority then T_j aborts
  else T_i waits

Technique 3: Detection

most used kind

detect cycles and break them by aborting some transaction

main tool: Maintain the distributed Wait-for-graph

- cycle $\Rightarrow$ deadlock

abort to break cycles

- issue as in centralized case: choose the victim

Cycle detection

difficulty: the graph is distributed and dynamic

Centralyzed cycle detection

- one site receives local wait-for-graphs
- construct global wait-for-graph and detect cycles

Distributed deadlock detection

wait(i): the process that is blocking process i

message: probe(i,j,k) send by process j to process k to let it know that process i is blocked by k

algorithm

- when i requests a resource that is used by j
  wait(i) := j
  probe(i,i,j)
- when k receives probe(i,j,k) (from j)

  if k is waiting then if k = i then deadlock detected
  else probe(i,k,wait(k))

more complicated: processes should be "released"

possibility of false alarm: the deadlock is not real but the release did not arrive in time

make sure the releases have been treated *before* sending a probe

Problem 2: replicated data

serializable does not work anymore
x duplicated at site 1 and 2
two transactions:

- T1: read(x); x:=x+5; write(x); commit
- T2: read(x); x:=x*10; write(x); commit

2 local schedules:

- S1: R1(x),W1(x),C1,R2(x),W2(x),C2
- S2: R2(x),W2(x),C2,R1(x),W1(x),C1

each is serial
suppose that x = 1 before
after x@s1 is 60 and x@s2 is 15
there should be some consistency between the two schedules

One-copy serializable

Definition of correctness

schedule should be equivalent to a serial schedule
on a database with a single copy

implies: two conflicting operations should be in the same relative order in all local schedules where they appear together

Read-once/write-all ROWA

a read(x) operation is translated to read(x_i) for some copy of x
a write(x) is transalated to
{ write(x_i) | for all copies of x }

One-copy serializable - continued

ideal world: consider all write to be simultaneous

guarantees one-copy serializable

reality: some write may fail (one copy is not available) $\rightarrow$ block the transaction

alternative: write-all-available

when a site recovers, it should update its data before serving data (otherwise, it may serve out-of-date data)

CC with replicated data: centralized 2PL

Centralized 2PL

- one site keeps all the lock tables and
  is responsible for granting locks
- advantage: simple and works OK
- disadvantage: the central LM is a potential bottleneck
  if it fails $\Rightarrow$ everything stops

CC with replicated data: primary copy 2PL

Primary copy 2PL

- each entity is assigned a primary site
- lock is managed there
- reduces the bottleneck of the centralized 2PL

CC with replicated data: distributed 2PL

Distributed 2PL

- each site has a lock manager and locks for data item it stores
- ROWA replica protocol
- lock request
  $\rightarrow$ involved lock managers
  $\rightarrow$ participating processors
- advantage for reads:
  to read local data, need only a local lock
- disadvantage: to write, need to obtain locks from all copies
- need to maintain a catalog of all copies

Nested transactions - autonomous systems

transaction on the global database

subtransactions in local databases

problem: no control on the TM of local databases

- problems with serializability
- problems with deadlock detection
- problems with failure recovery