SOME CASES

TUNING ACROSS INSTANCES

Overview

- Configuration
  - Disaster-proof systems, interoperability among different languages and different databases.
- Global systems
  - Semantic replication, rotation ownership, chopping batches
- Tuning
  - Clustering, concurrency and hashing, forcing plans

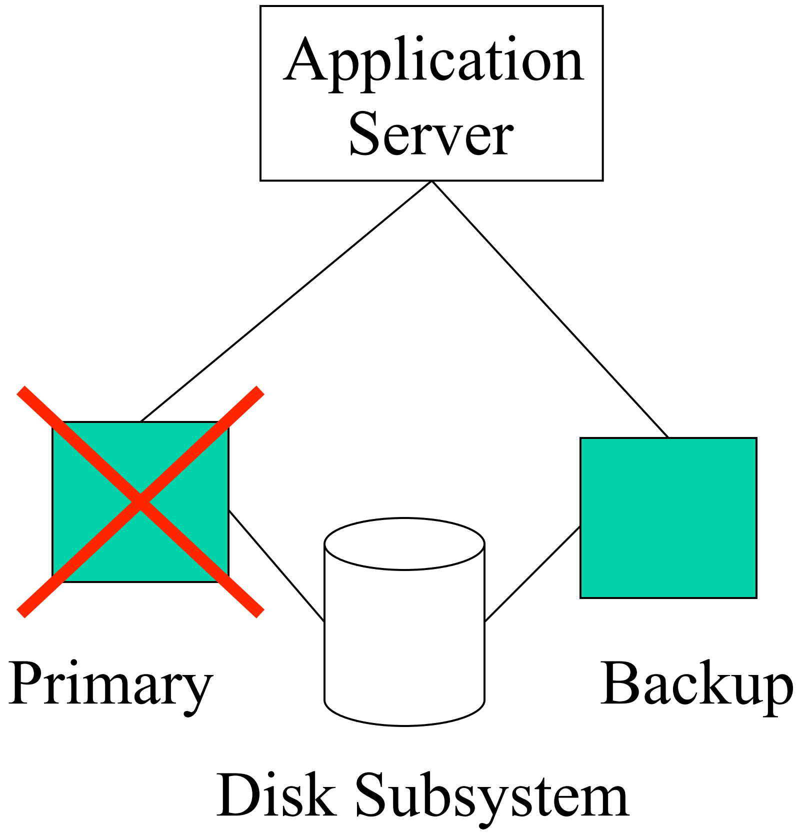
Preparing for Disaster

- Far from the simple model of stable storage that we sometimes teach, though the principle still apply.
- Memory fails, disk fail (in batches), fires happen and power grids fail. If your system is still alive you have a big advantage.
- You can even let your competitors use your facilities … for a price.

Case #1

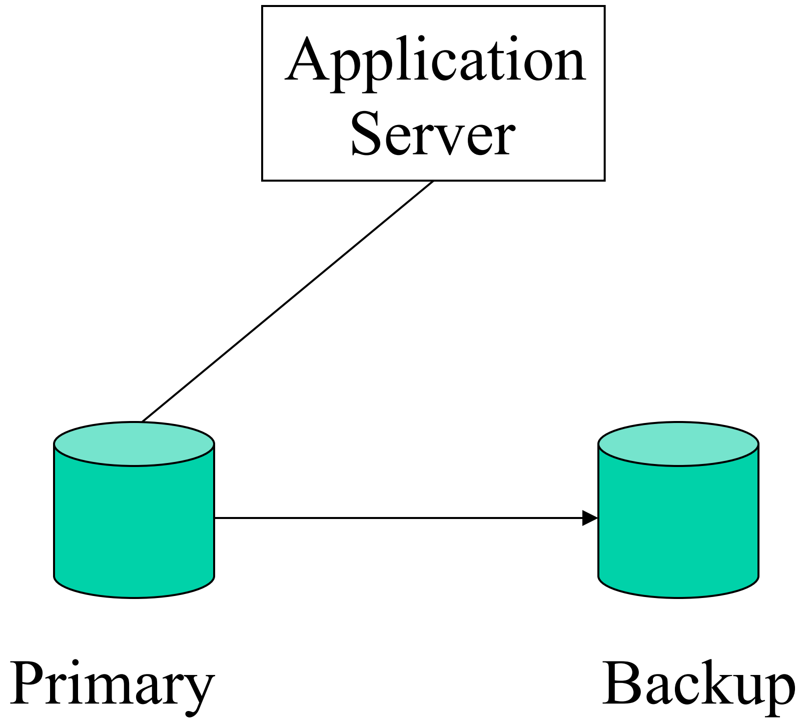
- Application used a few days per month, but with a heavy load (Server for trading bond futures having to do with home mortgages)
- Downtime should be minimal during these few days:
  - Operating System failure
  - Disk failure
  - Power failure
  - Building in fire (Credit Lyonnais, NYSE)

Solution1: Shared Disk High Availability Servers



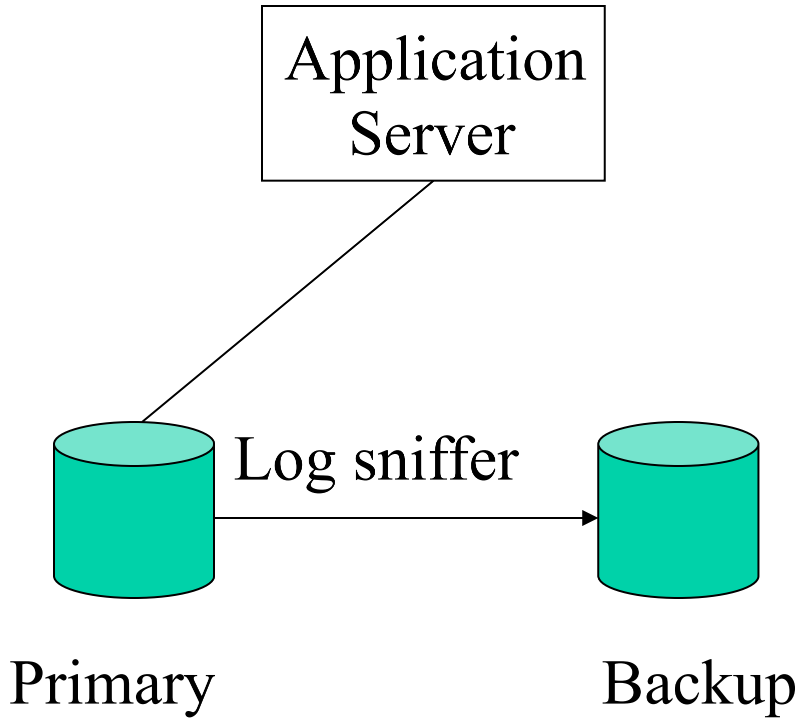
- Primary and backup attached to RAID disks
- If primary fails then backup takes over (warm start from disks)
- If a disk fails, RAID masks it
- Does not survive disasters or correlated failures

Solution2: Dump and Load



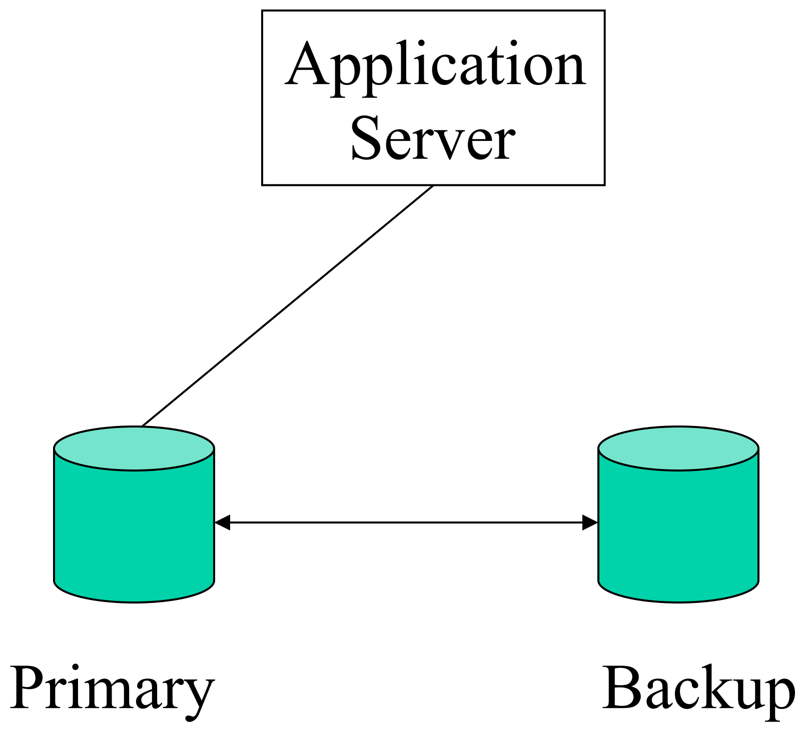
- Full dump from primary to back up at night.
- Incremental dumps every 10 minutes
- If primary fails, backup takeover from last dump.
  - Lose transactions committed on primary since last dump
  - There must be a (manual) record of these transactions
- Back up can be far away

Solution3: Replication Server



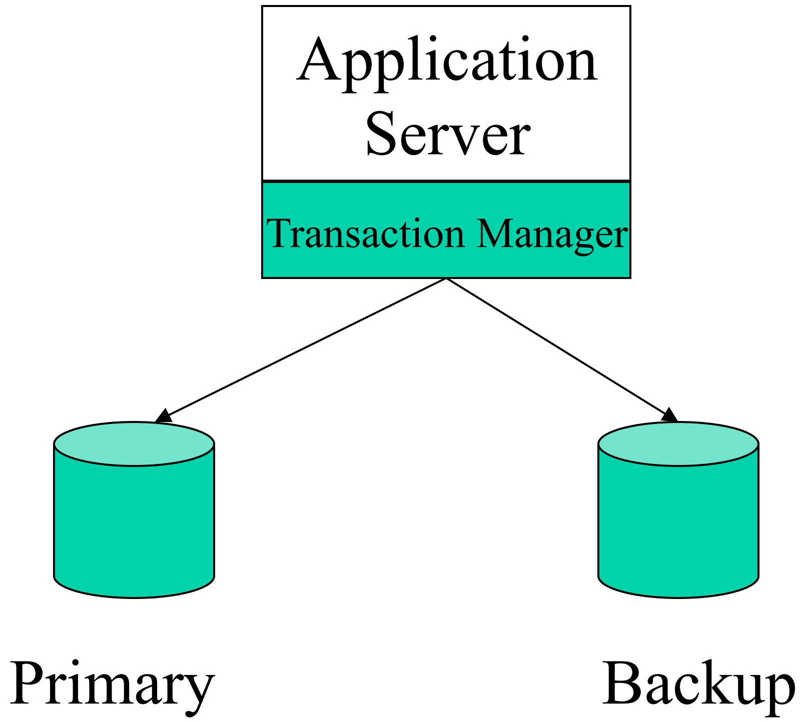
- Full dump of primary transferred to backup at night.
- Each SQL transaction committed on the primary is sent to the backup
- If primary fails, backup takes over from last committed transaction
  - May lose a few seconds of committed transactions
- Administration overhead

Solution4: Remote Mirroring



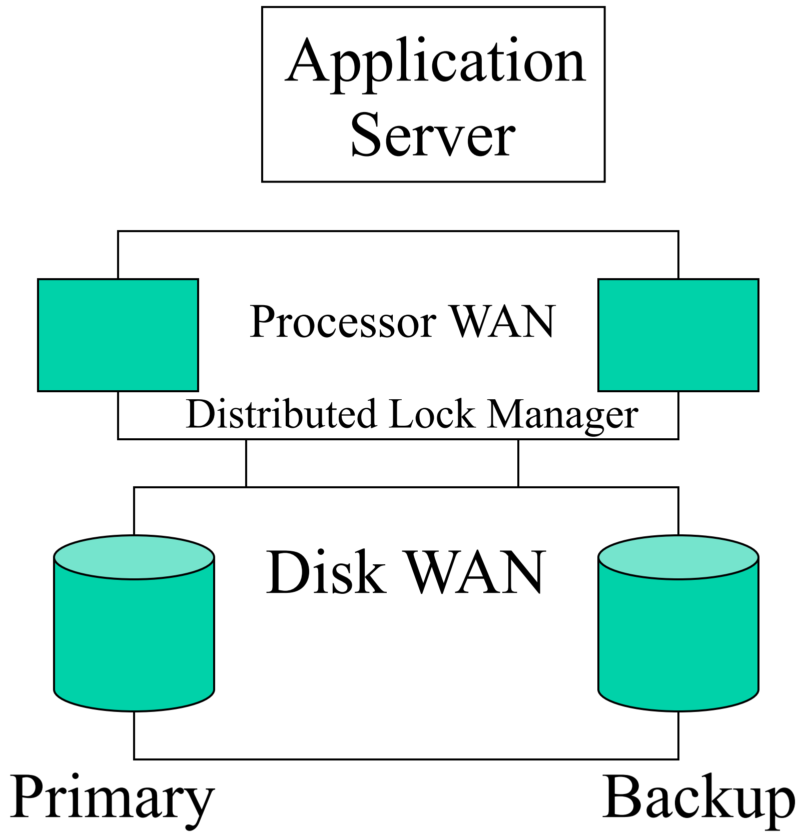
- Writes to local disks are mirrored to disks on a remote site. The commit at the local machine is delayed until the remote disks responds
- Backup problem may cause primary to halt.

Solution5: Two Phase Commit



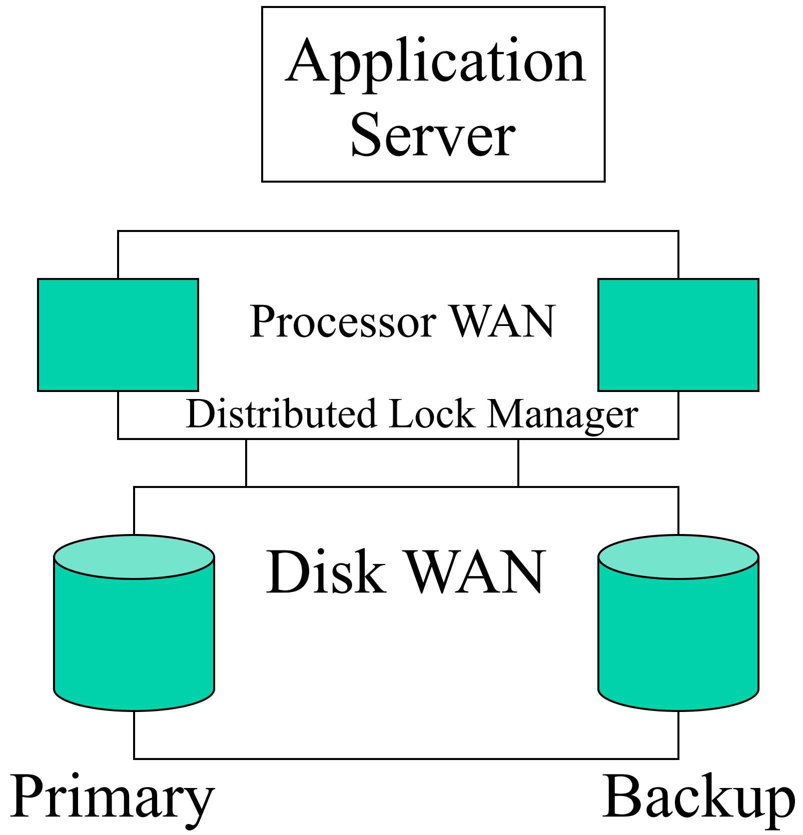
- A transaction manager coordinates the commits between primary and backup
- Blocking occurs if transaction manager fails
- Backup problem may cause primary to halt.

Solution6: Wide-area Quorum Approach



- Servers are peers and interconnected via a highly redundant wide-area interconnect
- Servers coordinate via a distributed lock manager
- Disks are connected with servers at several points and to one another by a second wide area interconnect

Solution6: Wide-area Quorum Approach



- Heartbeats monitor the connectivity among disks and processors
- If a break is detected, one partition holding a majority of votes continues to execute
- Any single failure of a processor, disk, site o r network is invisible to end-user.

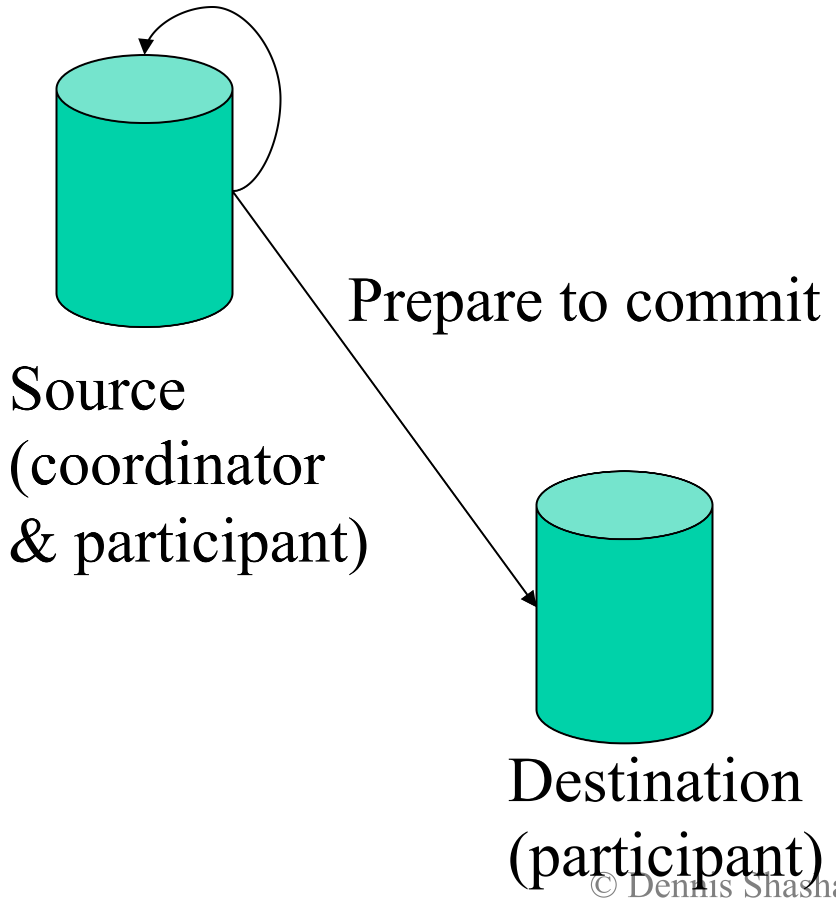
Review Case#1

- Wide-area quorum solution used by traders applications
- Dump and Load or replication server used for back-office applications
- Symmetric approaches (shared disk and two phase commit) are not used in practice for implementing remote backup
- Don't buy batches of disks from one vendor.

Case #2

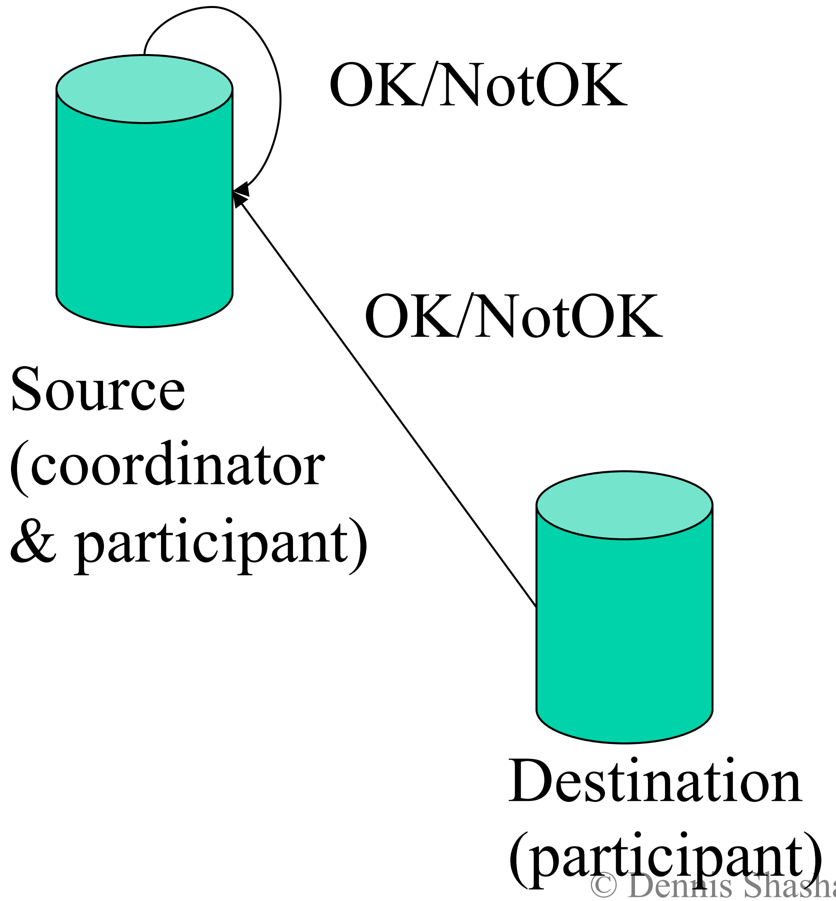
- Front-office system records trade for profit and loss.
  - A trade is a transaction: You sell or buy X stocks of company Y at price Z.
- Trades have to be sent to the back-office for final clearing.
- No trades should be lost!

Solution1: Two-Phase Commit



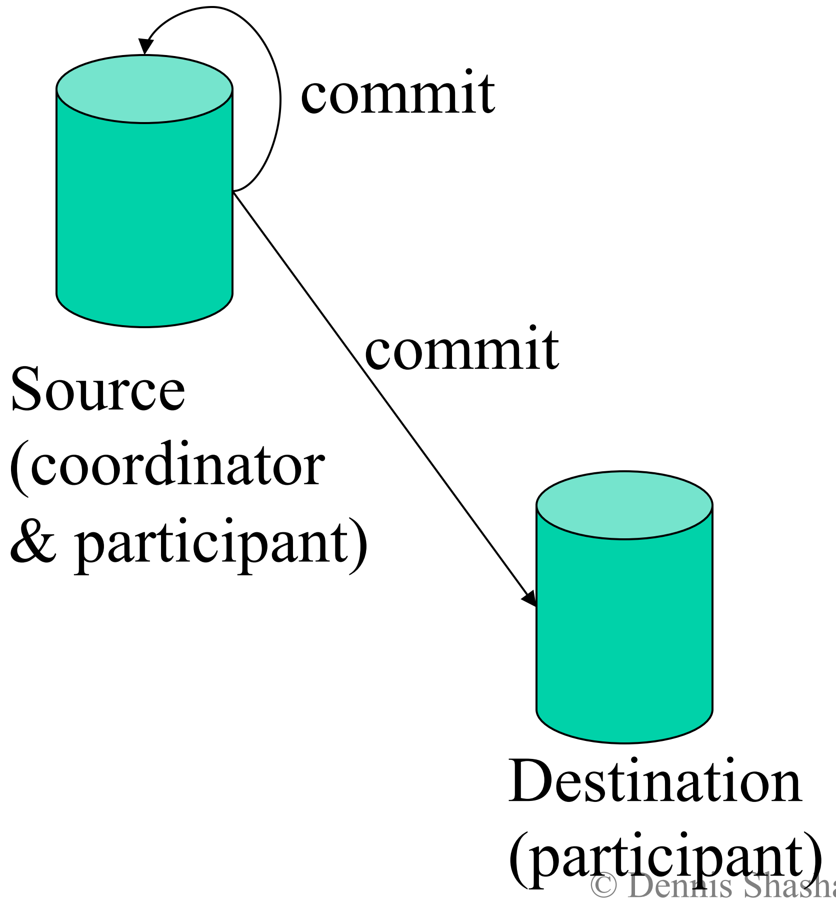
- Phase1: preparation
  - Coordinator sends prepare-to-commit message to all participants

Solution 1: Two-Phase Commit



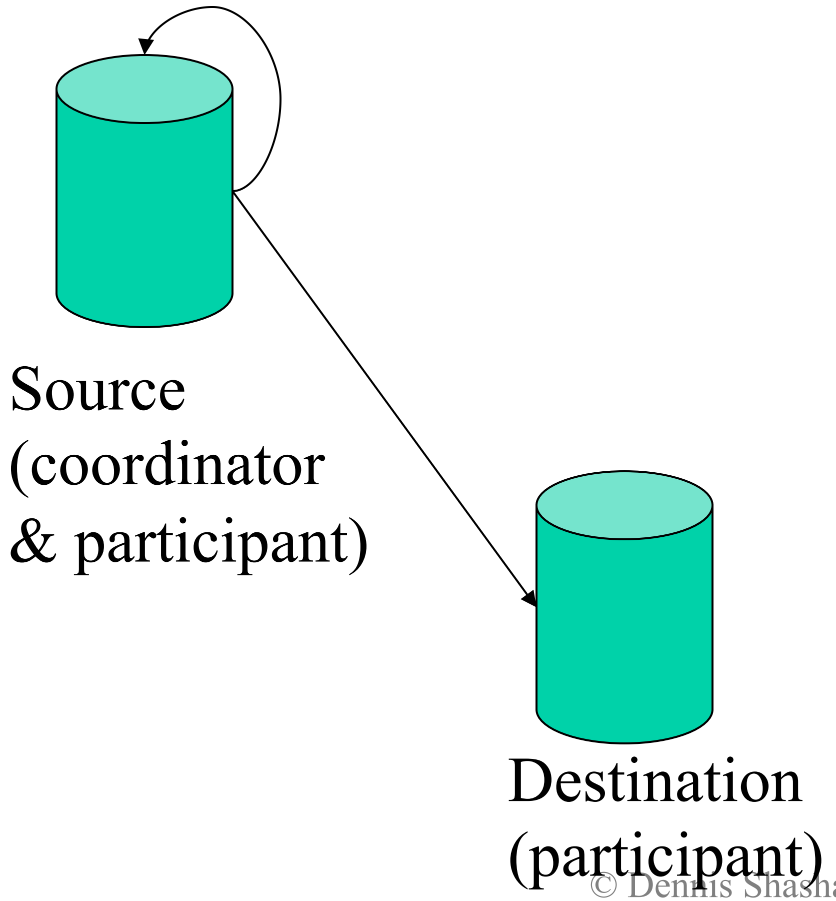
- Phase1: preparation
  - Coordinator sends prepare-to-commit message to all participants
  - All participants send a notification to the coordinator (OK/NotOK)

Solution 1: Two-Phase Commit



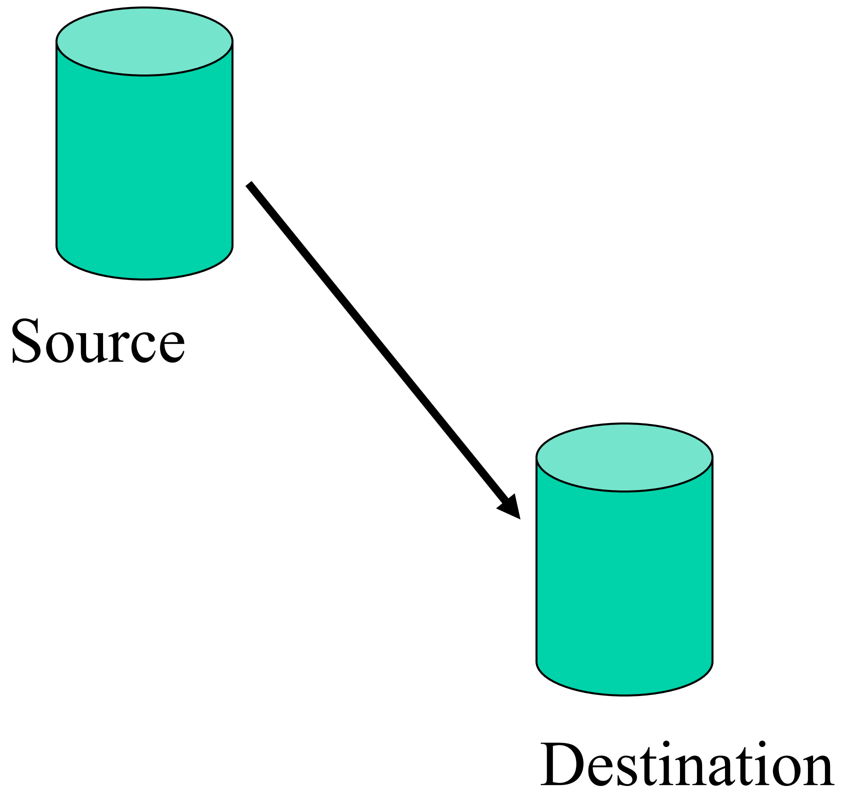
- Phase1: preparation
  - Coordinator sends prepare-to-commit message to all participants
  - All participants send a notification to the coordinator (OK/NotOK)
- Phase2: commit/abort
  - If all participants return OK then coordinator sends a commit message to all participants, else it sends an abort message.

Solution 1: Two-Phase Commit



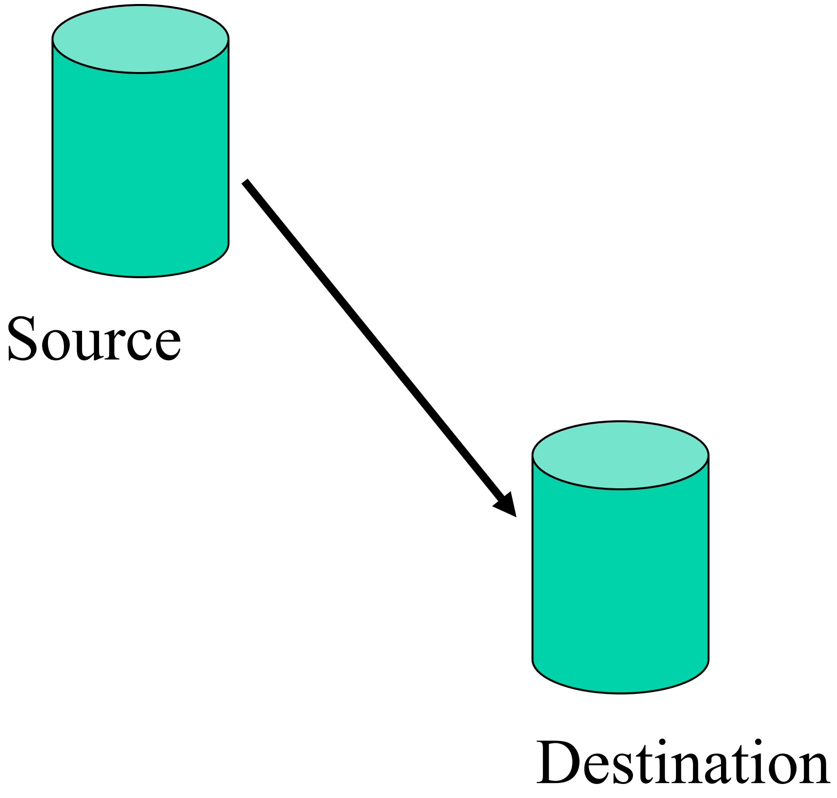
+ Commits are coordinated between the source and the destination
- If one participant fails then blocking can occur
- Not all db systems support prepare-to-commit interface

Solution 2: Replication Server



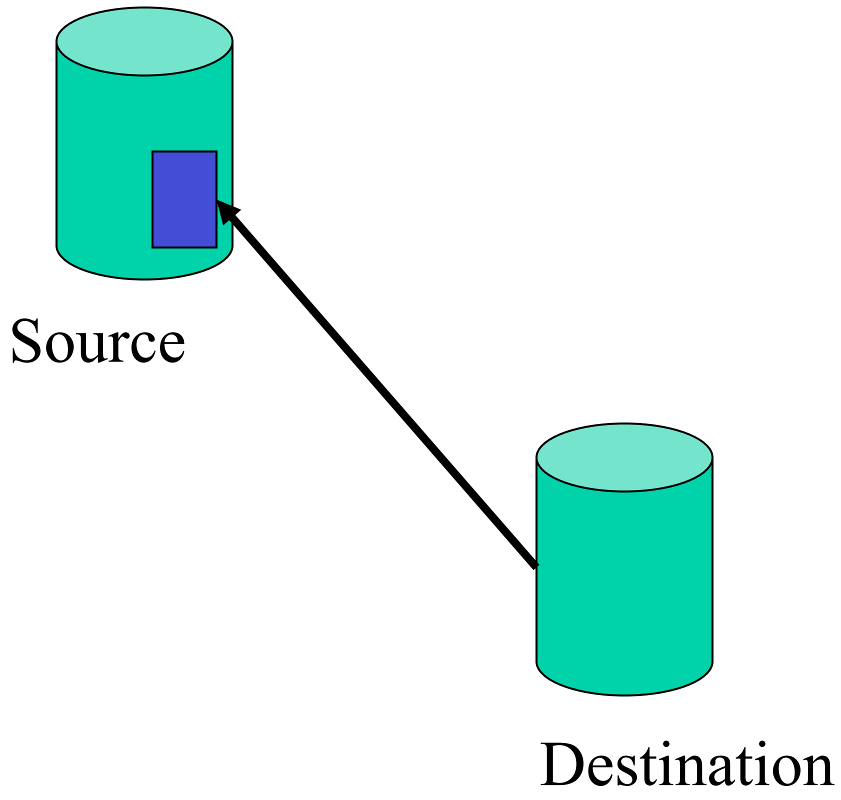
- At night, full dump of source db is transferred to destination db
- During the day, all SQL transactions committed on the source db are sent to the destination

Solution 2: Replication Server



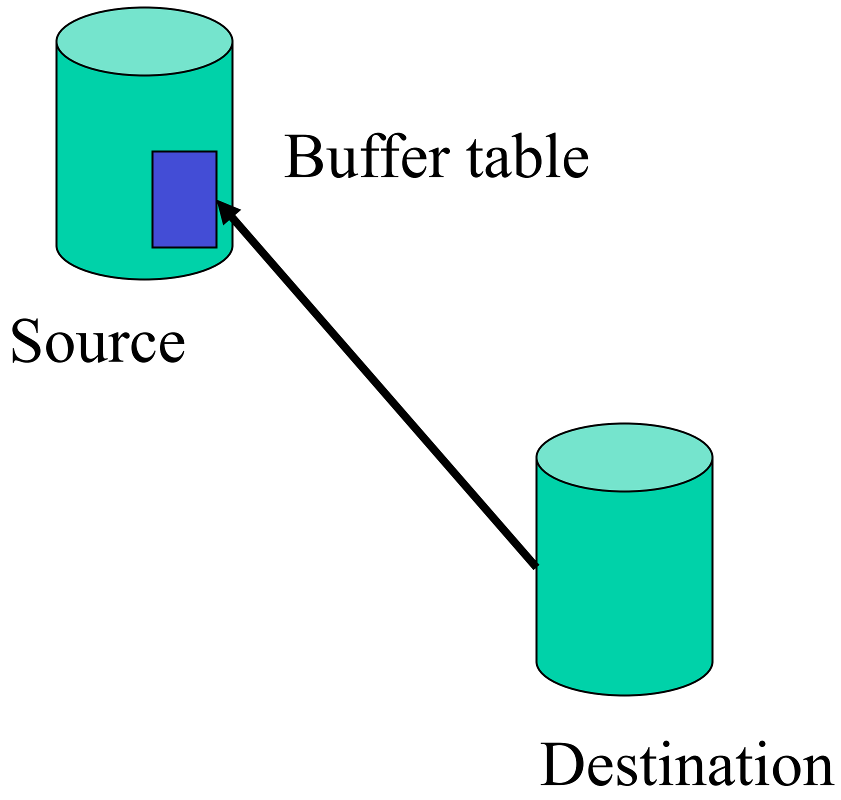
+ Destination within a few seconds of being up-to-date
+ Decision support queries can be asked on destination db
- Administrator is needed when network connection breaks!

Solution3: Buffer Tables



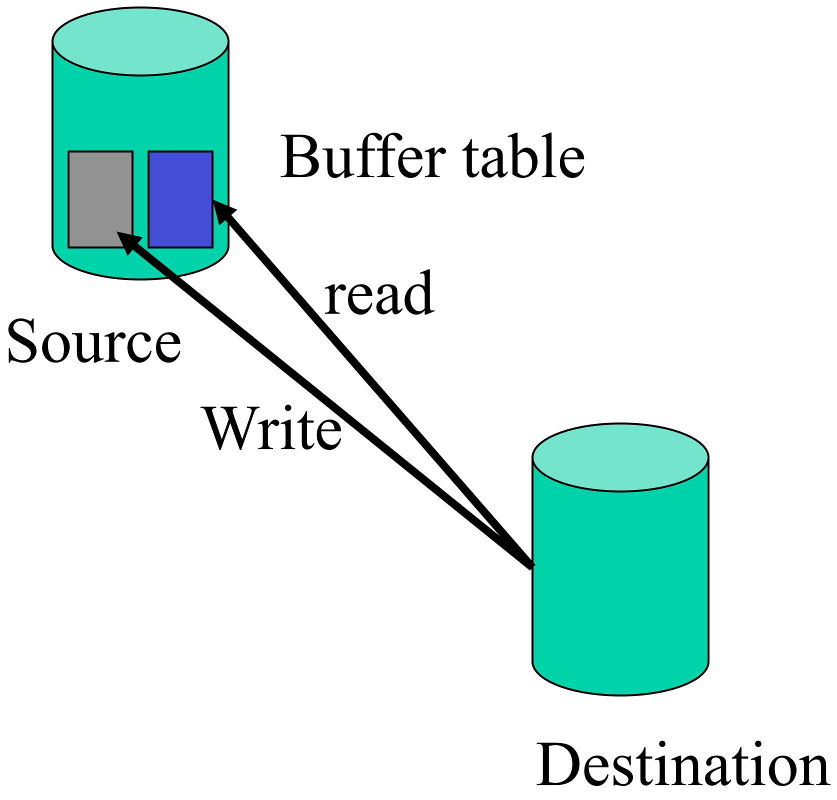
- A buffer table is created on the source db
- Whenever a trade transaction is committed a tuple is inserted in the buffer table
- The destination reads new tuple t from buffer table and performs transaction locally
- After processing t, the destination database deletes t

Solution3: Buffer Tables



+ No specific mechanism is necessary at source or destination
- Blocking can occur!
  - Problem: Source and destination update the buffer table (insert/ delete)

Solution3: Buffer Tables



+ No specific mechanism is necessary at source or destination
- Blocking can occur!
  - Problem: Source and destination update the buffer table (insert/delete)
  - Solution: Introduce a new table TuplesToBeDeleted. Destination updates TuplesToBeDeleted and only reads from Buffer.

Review Case#2

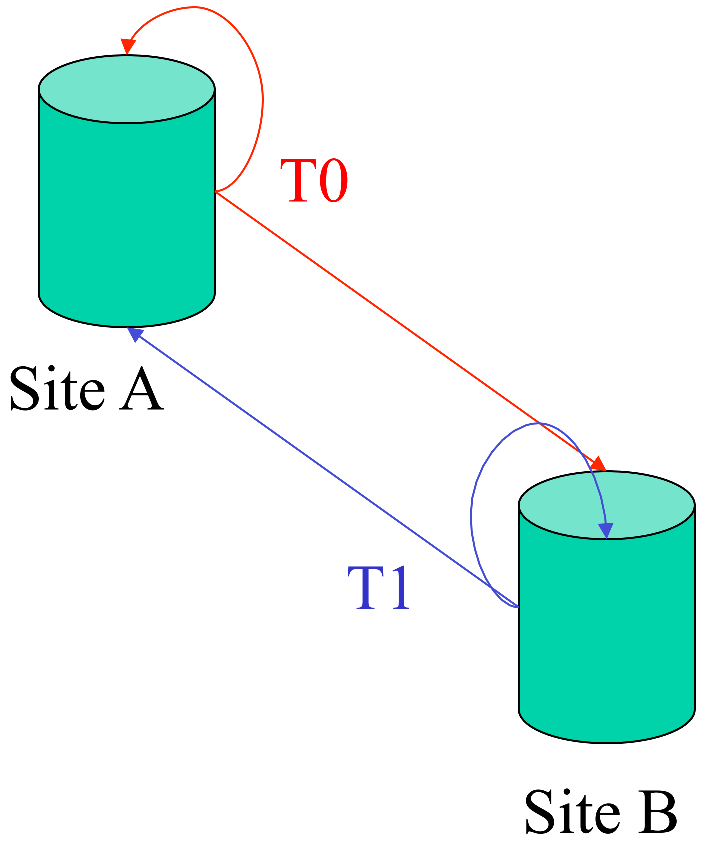
- Two Phase Commit for the coordination of commits (aborts) on distributed databases
  - 1: preparation
  - 2: commit (abort)
- In practice Two Phase Commit is not used for data replication because it introduces a dependency between source and destination
- The ‘Buffer table’ solution does not require any specific system support to replicate data between a source and a destination

Backup at Scale at Facebook

Case #3

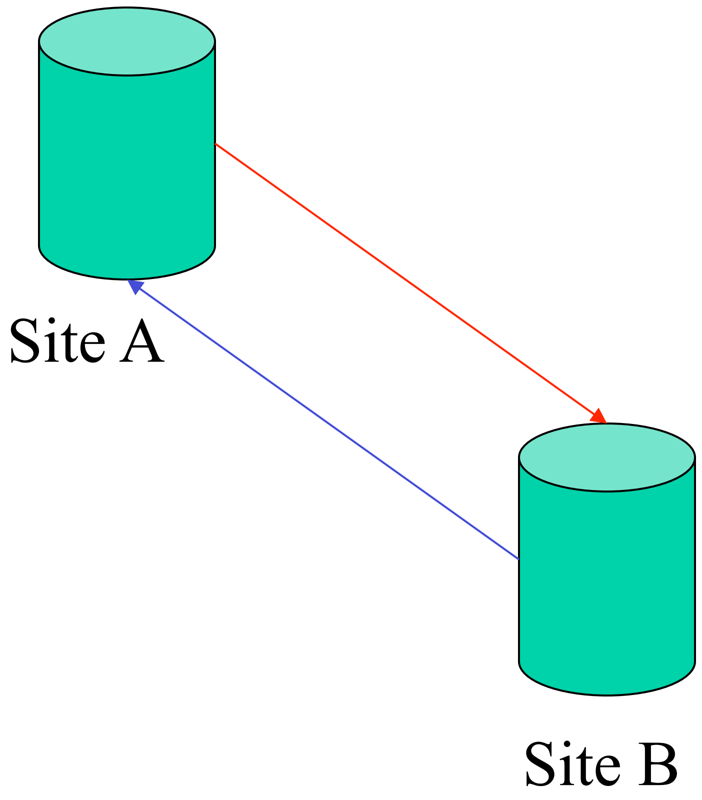
- Accounts data are replicated on 2 sites
- Updates can take place on both sites
- Accounts data have to be kept consistent

Solution1: Eager Replication



- All replicas are exactly synchronized at all nodes by updating all replicas as part of one atomic transaction
- Two phase commit ensures serial execution of T0 and T1
  - Delays or aborts transactions if committing would violate serialization
- Low update performance

Solution2: Lazy Replication



- Replication server ensures that replica updates are propagated to other nodes after the updating transaction commits
- Improved response time
- Problem: lost updates

Lost updates

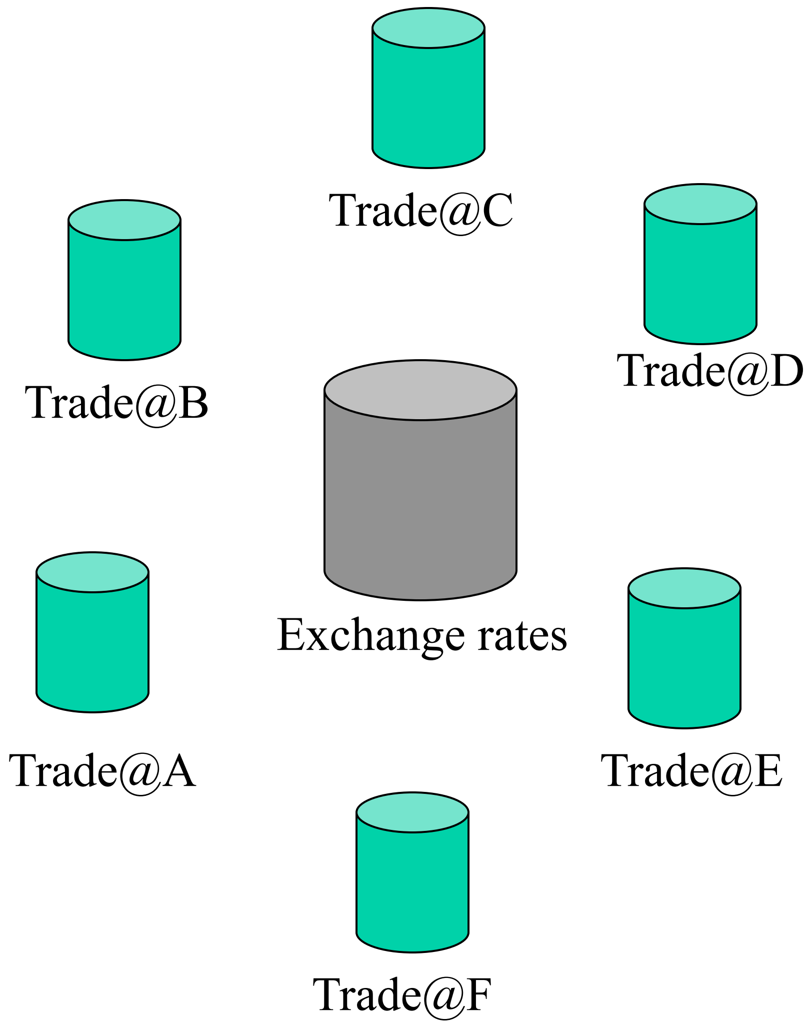
- Time 0:
  - Site A has cash position of 10M$ on Account X
  - Site B has cash position of 10M$ on Account X
- Time 1:
  - T0: Site A increases position of account X by 2M$
  - T1: Site B increases position of account X by 1M$
  - Replication server: Update at both sites are translated into a delete followed by an insert which are sent to the other site
- Time 2:
  - Site A writes the value obtained from B
  - Site B writes the value obtained from A
- Time 3:
  - Site A has cash position of 11M$ on Account X
  - Site B has cash position of 12M$ on Account X

Need for reconciliation

Review Case#3

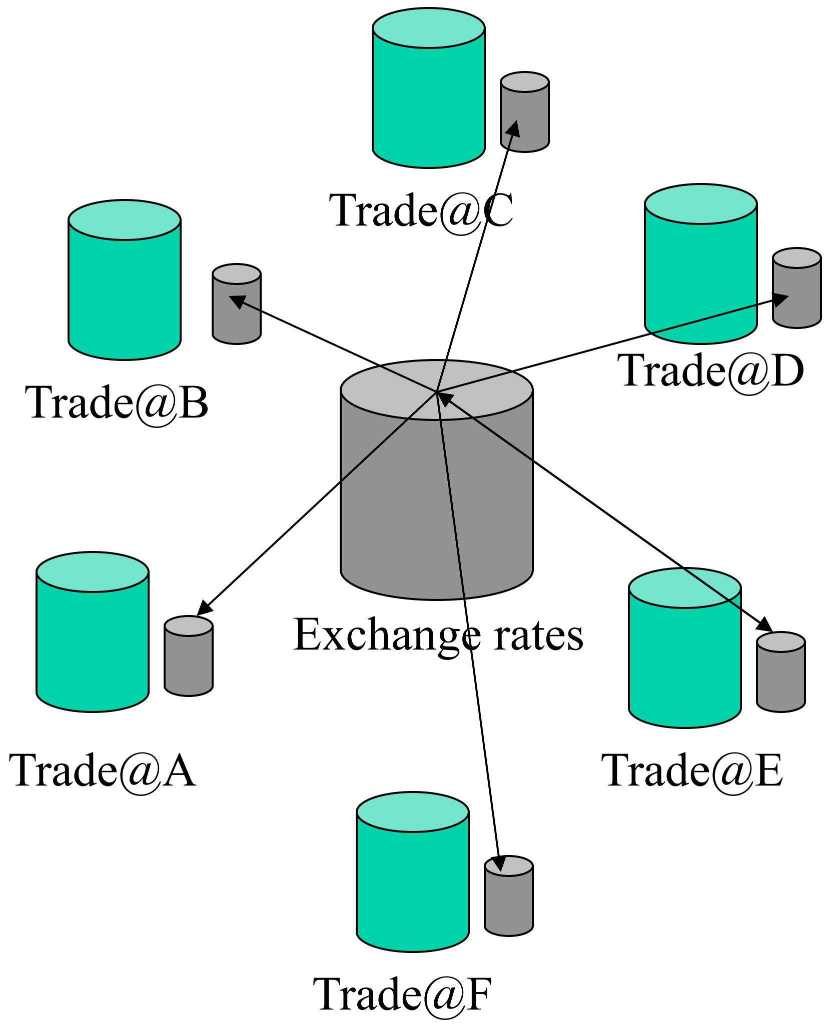
- Eager replication vs. Lazy replication for synchronizing replicated data
  - Eager replication
    - + No concurrency anomalies
    - - Poor update performances
  - Lazy replication
    - + High throughput
    - - Need for reconciliation to avoid lost updates

Case #4



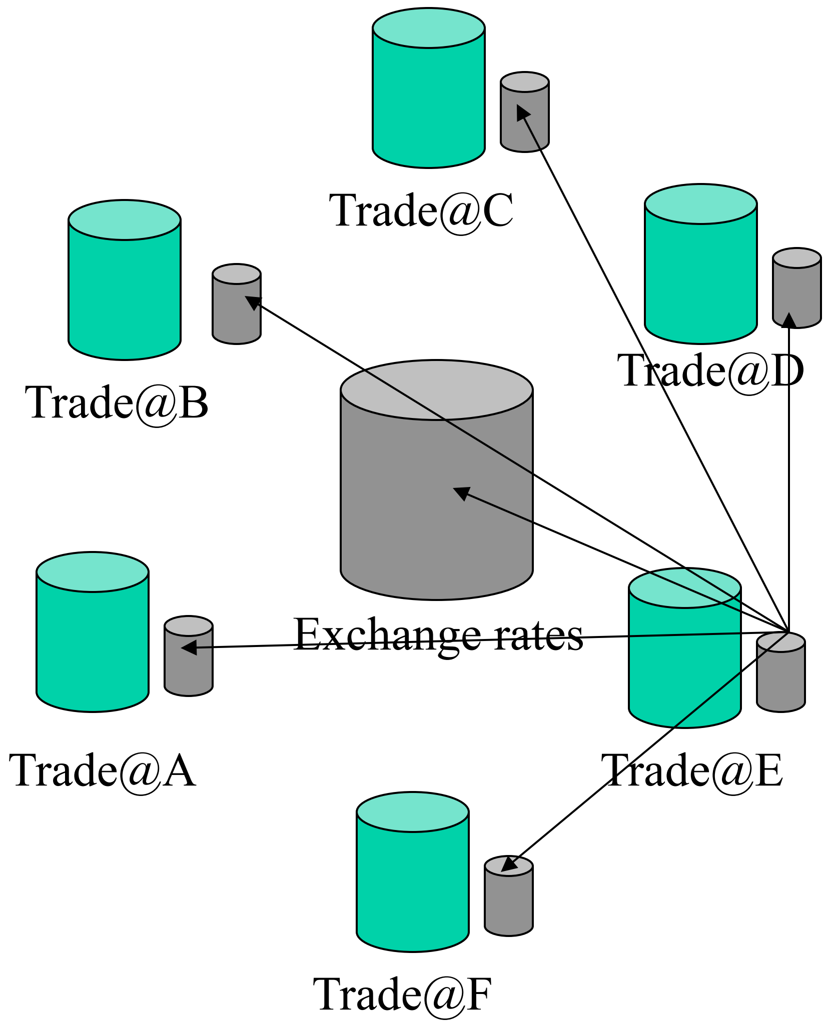
- Each trader accesses local trade information as well as global exchange rate information.
- When traders run a calculation they expect the most up-to-date exchange rate
- Current solution: trades are stored locally and exchange rates centrally.
- Problem: Calculations are slow!

Solution1: Master replication



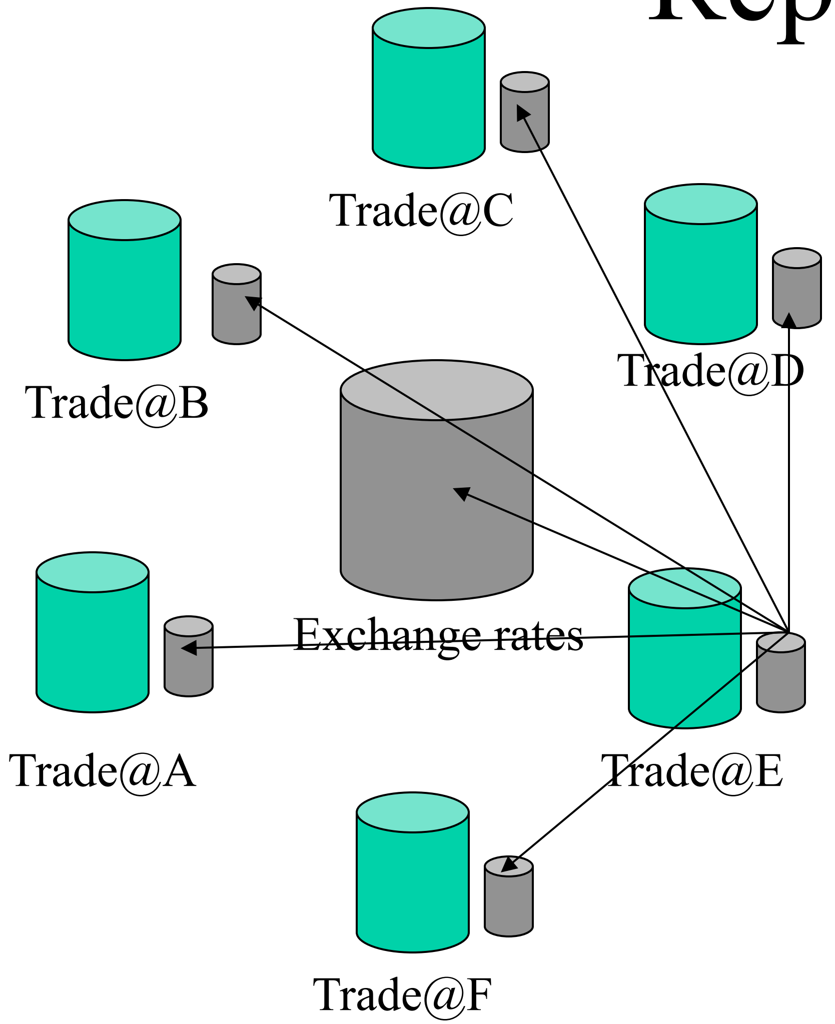
- Update central site and replicate updates on all local sites
- Problem: Calculations are still slow.
  - If a trader changes a rate and runs a calculation, the calculation should reflect the new rate

Solution2: Group Replication



- Update on a site are replicated to all other sites
- Problem: Two sites can update the exchange rates.
  - All sites should agree on a new exchange rate after a short time

Solution2: Clock-based Replication



- Synchronize clocks at the different sites.
- Attach a timestamp to each update of an exchange rate.
- An update is accepted locally if the timestamp of the update is greater than the timestamp of the exchange rate in that database.

Review Case#4

- Lazy replication schemes for reconciliation
  - Master replication
    - Works well for any kind of updates
    - Poor performance
  - Clock-based replication:
    - Works well for write only transactions

Case #5

- When the trading day is over, there are many operations that must be done to move trades to the backoffice, to clear out positions that have fallen to zero, and so on. Call it “rollover”.
- Straightforward provided no trades are hitting the database at the same time.
- In a global trading situation however, rollover in New York may interfere with trading in Tokyo.

Solution

- Chop the rollover transaction into smaller ones.
- New trades do not conflict with rollover. Lock conflicts are due to the fact that rollover uses scans.

Data Integration

- Fact table

  Sales(id, itemid, customerid, storeid, amount, qty)
- The itemid field must be checked against the item table, the customerid field must be checked against the customer table and the storeid field must be checked against the store table.
- This can be done in linear time.

Circumventing Superlinearity

- Create an empty table successfulsales
- Then issue the query:

```
insert into successfulsales
select      sales.id, sales.itemid, sales.customerid,
            sales.storeid, sales.amount, sales.qty
from sales, item, store, customer
where itemid in (select itemid from item)
   and customerid in (select customerid from item)
   and storeid in (select storeid from store)
```

Circumventing Superlinearity

- Now comes the difficult part: create a table with the sales that did not make it into successfulsales.
- One way is to initialize the unsuccessfulsales table with all sales and then to delete the successful ones:

```
insert into unsuccessfulsales
select * from sales;
delete from unsuccessfulsales
where id in (select id from successfulsales);
```

- If neither successfulsales nor successfulsales is indexed then response time is proportional to the square of the number of sales (most sales are successful).

Circumventing Superlinearity

- A first solution is to index the tables successfulsales and unsuccessfulsales
- A second solution is to insert the data in small batches (response time is then proportional to the size of those batches – not to the size of the complete sales table).

Circumventing Superlinearity

- A third solution is to use an outer join when inserting data into the successfulsales table.

```
insert into successfulsales
select     sales.id, item.id, store.id, customer.id,
           sales.amount, sales.qty
from (((sales left outer join item
                     on sales.itemid =  item.id)
                          left outer join store
                          on sales.storeid = store.id)
                               left outer join customer
                               on sales.customerid =
                                    customer.id);
```

Circumventing Superlinearity

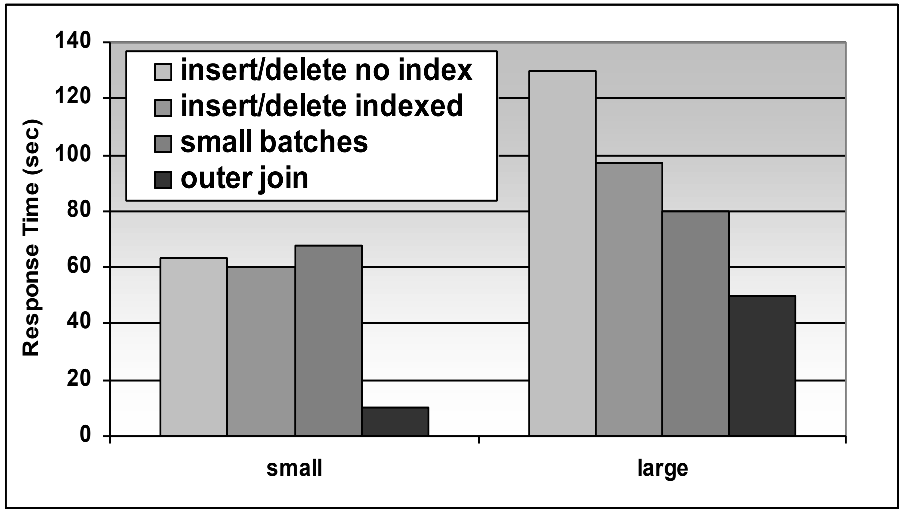
- Unsuccessful sales are obtained as follows:

```
insert into unsuccessfulsales
select * from successfulsales
where itemid is null
    or customerid is null
    or storeid is null;
```

- Successful sales are obtained as follows:

```
delete from successfulsales
where itemid is null
      or customerid is null
      or storeid is null;
```

Circumventing Superlinearity



- Small: 500000 sales
  Large: 1000000 sales
- SQL Server 2000
- Outer join achieves the best response time.
- Small batches do not provide benefits for the small database. The overhead of crossing the application interface is higher than the benefit of joining with smaller tables.

Tuning in the Cloud

- The cloud is Utility Computing
  - Packaging of computing resources, such as computation, storage and services, as a metered service. This model has the advantage of a low or no initial cost to acquire computer resources; instead, computational resources are essentially rented

What is the Cloud

- The cloud is many things
  - Daas: Data as a Service
    - Application-dependant: e.g., Urban Mapping
  - Saas: Software as a Service
    - RDBMS functionalities: e.g., MS SQL Azure, Amazon RDS, SimpleDB, Google Cloud SQL
  - Paas: Platform as a Service
    - Virtualized servers E.g., Amazon EC2

Provisioning

- AWS RDS allows to access Oracle, MySQL or SQL Server on the AWS infrastructure
  - Specially interesting if you are developing a server to be deployed on an EC2 instance
- Provisioning in terms of capacity and IOPS
  - 3 TB and 30000 IOPS
  - EC2 instances with CPU provisioning
  - Note that there is NO network provisioning

Tuning in the Cloud

- Virtualized Environment
  - No HW tuning
  - No HW information; just provisioning
- Cyclical/Unstable environment
  - Network fluctuates
  - Workload fluctuates
  - HW resources actually available fluctuate
  - Tuning target is moving!
- Cost depends on traffic
  - Tuning how the application interface is crossed is a key issue!