CompSci 516
Database Systems

Lecture 20
Distributed DBMS
NOSQL

Instructor: Sudeepa Roy

Announcements (Tues, 03/22)

- Quiz4 due Thursday
- HW3
- Survey on Ed
- Project report deadline 04/13

Where are we now? (detour)

Relational model, queries, db design
- Relational Model
- Normal Forms, FD
- Query in SQL / RA / RC
- Recursion

DBMS Internals and Query Processing
- Storage
- Index
- Join algo/Sorting
- Execution/ Optimization

Beyond Relational Model
- XML
- NOSQL **JSON/MongoDB**

Transactions
- Basics
- Concurrency Control
- Recovery

(Basic) Big Data Processing
- Map-Reduce/ Spark
- Parallel DBMS
- Distributed DBMS

Other Topics
- Data Mining
- Data Cube

Covered

To be covered

Next

Reading Material

- [RG]
  - Parallel DBMS: Chapter 22.1-22.5
  - Distributed DBMS: Chapter 22.6 – 22.14

- [GUW]
  - Parallel DBMS and map-reduce: Chapter 20.1-20.2
  - Distributed DBMS: Chapter 20.3, 20.4.1-20.4.2, 20.5-20.6

- Other recommended readings:
  - Chapter 2 (Sections 1,2,3) of Mining of Massive Datasets, by Rajaraman and Ullman: http://i.stanford.edu/~ullman/mmds.html
  - Original Google MR paper by Jeff Dean and Sanjay Ghemawat, OSDI' 04: http://research.google.com/archive/mapreduce.html

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Distributed DBMS

Topics in Distributed DBMS

- Architecture
- Data Storage
- Query Execution
- Transactions – updates
- Recovery – Two Phase Commit (2PC)

- A brief overview / examples of all these

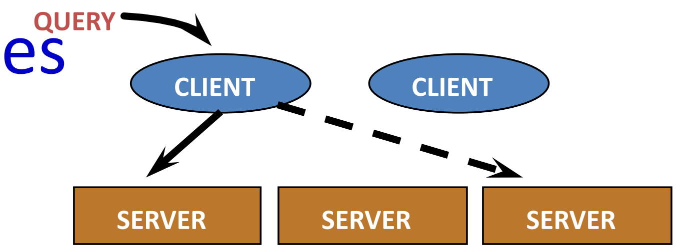
Distributed Data Independence

- Users should not have to know where data is located
  - no need to know the locations of references relations, their copies or fragments (later)
  - extends Physical and Logical Data Independence principles

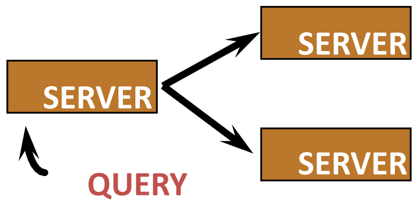
- Queries spanning multiple sites should be optimized in a cost-based manner
  - taking into account communication costs and differences in local computation costs

Distributed DBMS Architectures

- Three alternative approaches



1. Client-Server
   – Client: user interace, server: executes queries
2. Collaborating Server
   – All are of the same status



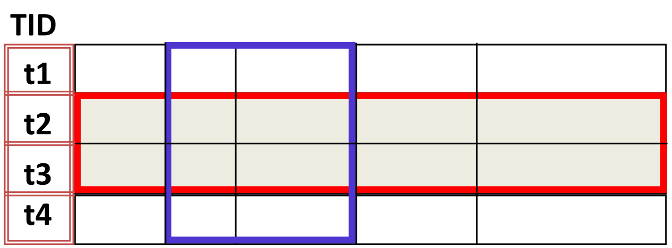
3. Middleware
   – Good for integrating legacy systems, middleware coordinates, individual server executes local queries

Storing Data in a Distributed DBMS

- A single relation may be partitioned or fragmented across several sites
  - typically at sites where they are most often accessed
- The data can be replicated as well
  - when the relation is in high demand or for robustness

- Horizontal:
  - Usually disjoint
  - Can often be identified by a selection query
    - employees in a city – locality of reference
  - To retrieve the full relation, need a union

- Vertical:
  - Identified by projection queries
  - Typically unique TIDs added to each tuple
  - TIDs replicated in each fragments
  - Ensures that we have a Lossless Join



Joins in a Distributed DBMS

- Can be very expensive if relations are stored at different sites

1. Fetch as needed
2. Ship to one site
3. Semi-join
4. Bloom join

Sailors as outer – for each S page, fetch all R pages from Paris
if cached at London, each R page fetched once

Ship Sailor to Paris

Unnecessary shipping
Not all tuples used

LONDON

Sailors (S)

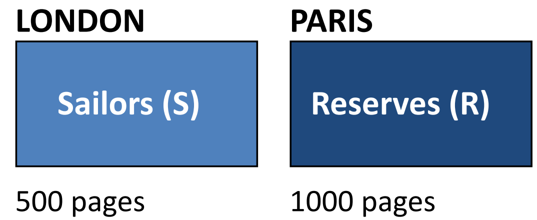
500 pages

PARIS

Reserves (R)

1000 pages

Semijoin



- Suppose want to ship R to London and then do join with S at London. Instead,

1. At London, project S onto join columns and ship this to Paris
   – Here foreign keys, but could be arbitrary join
2. At Paris, join S-projection with R
   – Result is called reduction of Reserves w.r.t. Sailors (only these tuples are needed)
3. Ship reduction of R to back to London
4. At London, join S with reduction of R

- Tradeoff the cost of computing and shipping projection for cost of shipping full R relation
  – Especially useful if there is a selection on Sailors, and answer desired at London

Bloomjoin

LONDON

Sailors (S)

500 pages

PARIS

Reserves (R)

1000 pages

- Similar idea like semi-join
- Suppose want to ship R to London and then do join with S at London (like semijoin)

1. At London, compute a bit-vector of some size k:
   - Hash column values into range 0 to k-1
   - If some tuple hashes to p, set bit p to 1 (p from 0 to k-1)
   - Ship bit-vector to Paris
2. At Paris, hash each tuple of R similarly
   - discard tuples that hash to 0 in S's bit-vector
   - Result is called reduction of R w.r.t S
3. Ship "bit-vector-reduced" R to London
4. At London, join S with reduced R

- Bit-vector cheaper to ship, almost as effective
  - the size of the reduction of R shipped back can be larger. Why?

Distributed Query Optimization

- Similar to centralized optimization, but have differences
  1. Communication costs must be considered
  2. Local site autonomy must be respected
  3. New distributed join methods should be considered

- Query site constructs global plan, with suggested local plans describing processing at each site
  – If a site can improve suggested local plan, free to do so

Review

Announcements (Thurs, 03/24)

- Quiz4 due today noon
- HW3 due 4/5 (Tues) noon
- Survey on Ed due by tonight on project teams dynamics
- More frequent check in for all teams by mentors
- Project report deadline 04/13

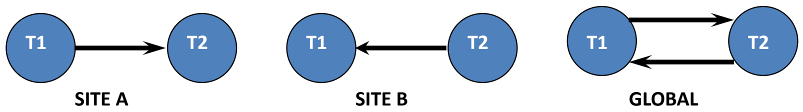
Updating Distributed Data

- Synchronous Replication: All copies of a modified relation (or fragment) must be updated before the modifying transaction commits
  - Always updated but expensive commit protocols (2PC – soon!)
  - By “voting” - e.g., 10 copies; 7 written for update; 4 copies read (why 4?)
  - Read-any Write-all (special case of voting, why not write-any read all?)

- Asynchronous Replication: Copies of a modified relation are only periodically updated; different copies may get out-of-sync in the meantime
  - More efficient – many current products follow this approach
  - Primary site (one master copy) or peer-to-peer (multiple master copies)

Distributed Locking

- How do we manage locks for objects across many sites?



1. Centralized: One site does all locking
   - Vulnerable to single site failure
2. Primary Copy: All locking for an object done at the primary copy site
   - Reading requires access to locking site as well as site where the object copy is stored
3. Fully Distributed: Locking for a copy done at site where the copy is stored
   - Locks at all sites while writing an object (unlike previous two)
   - May lead to “undetected” or “missing” ”global deadlock” due to delay in information propagation
   - Timeout or hierarchical detection
     - e.g. sites (every 10 sec)-> sites in a state (every min)-> sites in a country (every 10 min) -> global waits for graph. Intuition: more deadlocks are likely across closely related sites

Distributed Recovery

- Two new issues:
  - New kinds of failure, e.g., links and remote sites
  - If “sub-transactions” of a transaction execute at different sites, all or none must commit
  - Need a commit protocol to achieve this
  - Most widely used: Two Phase Commit (2PC)
- A log is maintained at each site
  - as in a centralized DBMS
  - commit protocol actions are additionally logged

Two-Phase Commit (2PC)

- Site at which transaction originates is coordinator
- Other sites at which it executes are subordinates
  - w.r.t. coordination of this transaction

Example on slides

When a transaction wants to commit – 1/5

1. Coordinator sends prepare message to each subordinate

When a transaction wants to commit – 2/5

2. Subordinate receives the prepare message

a) decides whether to abort or commit its subtransaction

b) force-writes an abort or prepare log record

c) then sends a no or yes message to coordinator

When a transaction wants to commit – 3/5

3. If coordinator gets unanimous yes votes from all subordinates
   a) it force-writes a commit log record
   b) then sends commit message to all subs

Else (if receives a no message or no response from some subordinate),
   a) it force-writes abort log record
   b) then sends abort messages

When a transaction wants to commit – 4/5

4. Subordinates force-write abort/commit log record based on message they get
    a) then send ack message to coordinator
    b) If commit received, commit the subtransaction
    c) write an end record

When a transaction wants to commit – 5/5

5. After the coordinator receives ack from all subordinates,
   - writes end log record

Transaction is officially committed when the coordinator's commit log record reaches the disk

- subsequent failures cannot affect the outcomes

Comments on 2PC

- Two rounds of communication
  - first, voting
  - then, termination
  - Both initiated by coordinator
- Any site (coordinator or subordinate) can unilaterally decide to abort a transaction
  - but unanimity/consensus needed to commit
- Every message reflects a decision by the sender
  - to ensure that this decision survives failures, it is first recorded in the local log and is force-written to disk
- All commit protocol log records for a transaction contain tid and Coordinator-id
  - The coordinator's abort/commit record also includes ids of all subordinates.

Restart After a Failure at a Site – 1/4

- Recovery process is invoked after a sites comes back up after a crash
  - reads the log and executes the commit protocol
  - the coordinator or a subordinate may have a crash
  - one site can be the coordinator some transaction and subordinates for others

Restart After a Failure at a Site – 2/4

- If we have a commit or abort log record for transaction T, but not an end record, must redo/undo T respectively
  - If this site is the coordinator for T (from the log record), keep sending commit/abort messages to subs until acks received
  - then write an end log record for T

Restart After a Failure at a Site – 3/4

- If we have a prepare log record for transaction T, but not commit/abort
  - This site is a subordinate for T
  - Repeatedly contact the coordinator to find status of T
  - Then write commit/abort log record
  - Redo/undo T
  - and write end log record

Restart After a Failure at a Site – 4/4

- If we don't have even a prepare log record for T
  - T was not voted to commit before crash
  - unilaterally abort and undo T
  - write an end record
- No way to determine if this site is the coordinator or subordinate
  - If this site is the coordinator, it might have sent prepare messages
  - then, subs may send yes/no message – coordinator is detected – ask subordinates to abort

Blocking

- If coordinator for transaction T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers.
  - T is blocked
  - Even if all subordinates know each other (extra overhead in prepare message) they are blocked unless one of them voted no
- Note: even if all subs vote yes, the coordinator then can give a no vote, and decide later to abort!

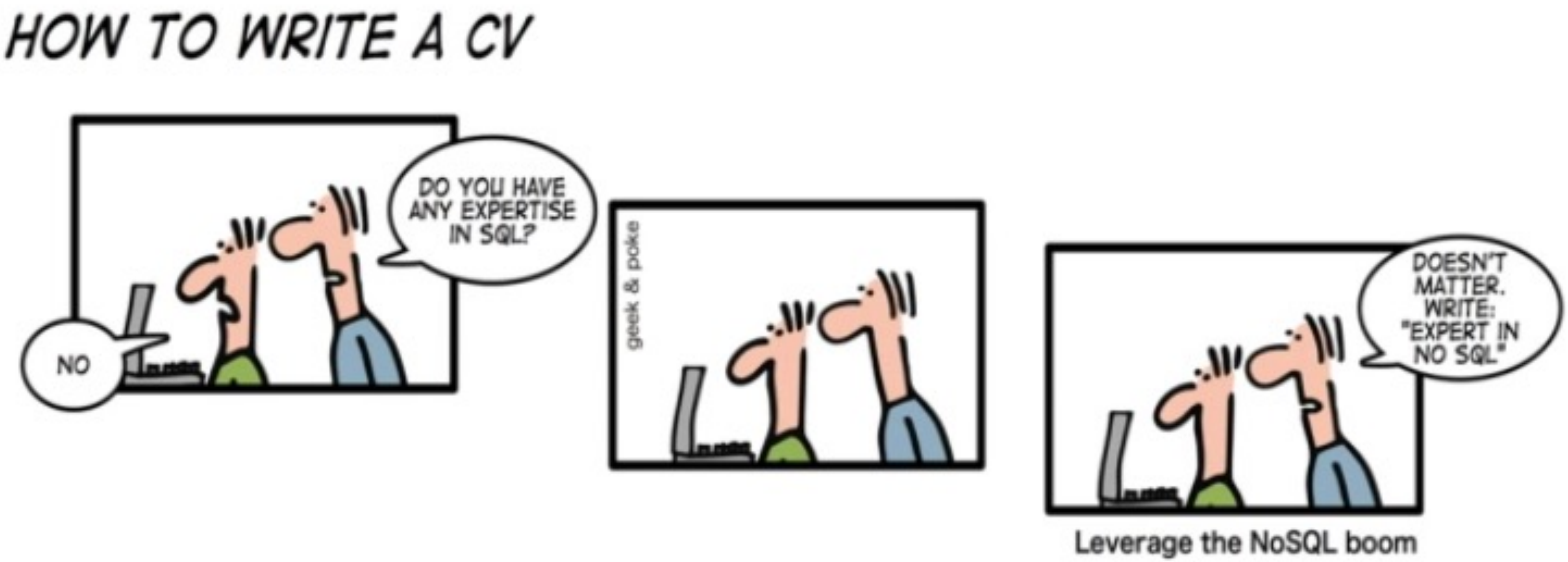
Link and Remote Site Failures

- If a remote site does not respond during the commit protocol for transaction T, either because the site failed or the link failed:
  - If the current site is the coordinator for T, should abort T
  - If the current site is a subordinate, and has not yet voted yes, it should abort T
  - If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds
  - needs to periodically contact the coordinator until receives a reply

Observations on 2PC

- Ack messages used to let coordinator know when it can “forget” a transaction; until it receives all acks, it must keep T in the transaction Table
- If coordinator fails after sending prepare messages but before writing commit/abort log records, when it recovers, it aborts the transaction
- If a subtransaction does no updates, its commit or abort status is irrelevant

NoSQL



- Optional reading:
  - Cattell's paper (2010-11)
  - Warning! some info will be outdated
  - see webpage http://cattell.net/datastores/ for updates and more pointers

NOSQL

- Many of the new systems are referred to as "NoSQL" data stores
  - MongoDB, CouchDB, VoltDB, Dynamo, Membase, ...
- NoSQL stands for "Not Only SQL" or "Not Relational"
  - not entirely agreed upon
- NoSQL = "new" database systems
  - not typically RDBMS
  - relax on some requirements, gain efficiency and scalability
- New systems choose to use/not use several concepts we learnt so far
  - You may find systems that use multi-version Concurrency Control (MVCC) or, asynchronous replication

OLTP (Online Transaction Processing)	Data Warehousing/OLAP (On Line Analytical Processing)
Mostly updates	Mostly reads
Applications: Order entry, sales update, banking transactions	Applications: Decision support in industry/organization
Detailed, up-to-date data	Summarized, historical data (from multiple operational db, grows over time)
Structured, repetitive, short tasks	Query intensive, ad hoc, complex queries
Each transaction reads/updates only a few tuples (tens of)	Each query can access many records, and perform many joins, scans, aggregates
MB-GB data	GB-TB data
Typically clerical users	Decision makers, analysts as users
Important: Consistency, recoverability, Maximizing tr. throughput	Important: Query throughput Response times

Applications of New Systems

- Designed to scale simple "OLTP"-style application loads
  - to do updates as well as reads
  - in contrast to traditional DBMSs and data warehouses
  - to provide good horizontal scalability for simple read/write database operations distributed over many servers

- Originally motivated by Web 2.0 applications
  - these systems are designed to scale to thousands or millions of users

NoSQL: Key Features

1. the ability to horizontally scale “simple operations” throughput over many servers
2. the ability to replicate and to distribute (partition) data over many servers
3. a weaker concurrency model than the ACID transactions of most relational (SQL) database systems
4. efficient use of distributed indexes and RAM for data storage
5. the ability to dynamically add new attributes to data records

BASE (not ACID ☺)

- Recall ACID for RDBMS desired properties of transactions:
  - Atomicity, Consistency, Isolation, and Durability

- NOSQL systems typically do not provide ACID

- Basically Available
- Soft state
- Eventually consistent

ACID vs. BASE

- The idea is that by giving up ACID constraints, one can achieve much higher performance and scalability

- The systems differ in how much they give up
  - e.g., most of the systems call themselves "eventually consistent", meaning that updates are eventually propagated to all nodes
  - but many of them provide mechanisms for some degree of consistency, such as multi-version concurrency control (MVCC)

“CAP” “Theorem”

- Often Eric Brewer’s CAP theorem cited for NoSQL

- A system can have only two out of three of the following properties:
  - Consistency
    - Every read receives the most recent write or an error
  - Availability
    - Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - Partition-tolerance
    - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

- The NoSQL systems generally give up consistency
  - However, the trade-offs are complex

https://en.wikipedia.org/wiki/CAP_theorem

What is different in NOSQL systems

- When you study a new NOSQL system, notice how it differs from RDBMS in terms of

1. Concurrency Control
2. Data Storage Medium
3. Replication
4. Transactions

Choices in NOSQL systems: 1. Concurrency Control

a) Locks
- some systems provide one-user-at-a-time read or update locks
- MongoDB provides locking at a field level

b) MVCC

c) None
- do not provide atomicity
- multiple users can edit in parallel
- no guarantee which version you will read

d) ACID
- pre-analyze transactions to avoid conflicts
- no deadlocks and no waits on locks

Choices in NOSQL systems: 2. Data Storage Medium

a) Storage in RAM
   – snapshots or replication to disk
   – poor performance when overflows RAM
b) Disk storage
   – caching in RAM

Choices in NOSQL systems: 3. Replication

- whether mirror copies are always in sync

a) Synchronous

b) Asynchronous
   - faster, but updates may be lost in a crash

c) Both
   - local copies synchronously, remote copies asynchronously

Choices in NOSQL systems: 4. Transaction Mechanisms

a) support
b) do not support
c) in between
   - support local transactions only within a single object or “shard”
   - shard = a horizontal partition of data in a database

FYI only –Optional slide

Comparison from Cattell’s paper (2011)

System	Conc Contol	Data Storage	Repli-cation	Tx
Redis	Locks	RAM	Async	N
Scalaris	Locks	RAM	Sync	L
Tokyo	Locks	RAM or disk	Async	L
Voldemort	MVCC	RAM or BDB	Async	N
Riak	MVCC	Plug-in	Async	N
Membrain	Locks	Flash + Disk	Sync	L
Membase	Locks	Disk	Sync	L
Dynamo	MVCC	Plug-in	Async	N
SimpleDB	None	S3	Async	N
MongoDB	Locks	Disk	Async	N
Couch DB	MVCC	Disk	Async	N
Terrastore	Locks	RAM+	Sync	L
HBase	Locks	Hadoop	Async	L
HyperTable	Locks	Files	Sync	L
Cassandra	MVCC	Disk	Async	L
BigTable	Locks+stamps	GFS	Sync+ Async	L
PNUTs	MVCC	Disk	Async	L
MySQL Cluster	ACID	Disk	Sync	Y
VoltDB	ACID, no lock	RAM	Sync	Y
Clustrix	ACID, no lock	Disk	Sync	Y
ScaleDB	ACID	Disk	Sync	Y
ScaleBase	ACID	Disk	Async	Y
NimbusDB	ACID, no lock	Disk	Sync	Y

FYI only –Optional slide

Data Store Categories

- The data stores are grouped according to their data model
- Key-value Stores:
  - store values and an index to find them based on a programmer- defined key
  - e.g., Project Voldemort, Riak, Redis, Scalaris, Tokyo Cabinet, Memcached/Membrain/Membase
- Document Stores:
  - store documents, which are indexed, with a simple query mechanism
  - e.g., Amazon SimpleDB, CouchDB, MongoDB, Terrastore
- Extensible Record Stores:
  - store extensible records that can be partitioned vertically and horizontally across nodes ("wide column stores")
  - e.g., Hbase, HyperTable, Cassandra, Yahoo's PNUTS
- "New" Relational Databases:
  - store (and index and query) tuples, e.g., the new RDBMSs that provide horizontal scaling
  - e.g., MySQL Cluster, VoltDB, Clustrix, ScaleDB, ScaleBase, NimbusDB, Google Megastore (a layer on BigTable)

RDBMS benefits

- Relational DBMSs have taken and retained majority market share over other competitors in the past 30 years
- While no "one size fits all" in the SQL products themselves, there is a common interface with SQL, transactions, and relational schema that give advantages in training, continuity, and data interchange
- Successful relational DBMSs have been built to handle other specific application loads in the past:
  - read-only or read-mostly data warehousing, OLTP on multi-core multi-disk CPUs, in-memory databases, distributed databases, and now horizontally scaled databases

NoSQL benefits

- We haven't yet seen good benchmarks showing that RDBMSs can achieve scaling comparable with NoSQL systems like Google's BigTable
- If you only require a lookup of objects based on a single key, then a key-value/document store may be adequate and probably easier to understand than a relational DBMS
- Some applications require a flexible schema
- A relational DBMS makes "expensive" (multi-node multi-table) operations "too easy"
  - NoSQL systems make them impossible or obviously expensive for programmers
- The new systems are slowly gaining market shares too