Parallel and Distributed Databases
Based on

• These slides are based on Chapter 20 of: Database Systems: The Complete Book (2nd edition), by Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom, 2008

• Which is an excellent book
Parallel and Distributed Systems

• Parallel system: how to parallelize critical operations
• Distributed systems: how to distribute transactions
• Peer to peer systems
Models of parallelism

- Shared memory machines:
  - Different CPUs share access to a unique main memory, but each has its own cache

- Shared disk machines:
  - Every CPU with its memory, but the disk space is unique

- Shared nothing machines:
  - Every CPU has its own memory and disks

- SN is the most common

- Message overhead: it is important to use few long messages rather than many small messages
Data partitioning

• Tuples are allocated to nodes according to a partitioning strategy
  – Range partitioning: node 1 keeps $\sigma_{k(i)<A<=k(i+1)}(R)$
  – Hash partitioning: we apply a hash function to R.A
  – Random (round-robin) partitioning
  – Block partitioning: round-robin at the level of blocks
  – Co-located partitioning: for each partition $R_i$ of $R$ at node $i$, the semijoin($S,R_i$) are in the same node

• The number of fragments may be fixed or may grow with the nodes
• Every relation may also be vertically partitioned $\pi_{A,B,C} \sigma_{\text{cond}(A)}(R)$
• Every fragment is typically replicated for resilience
Uses of data partitioning

• To execute query working in parallel on different nodes
• To only access relevant nodes when the relation is filtered
• To distribute the load of some maintenance work
• To allocate the crucial fragments on the fastest support
Parallel algorithms for set operators

- **Distinct**: if tuples are distributes using a hash function, Distinct can be executed locally in parallel.

- **Union(R,S), Intersection(R,S), Difference(R,S)**:
  - If R and S are hashed with the same function, can be executed locally.
  - Otherwise, if we have M processors we hash both R and S with a same function in [0, M-1] and send tuple t to processor h(t).
  - We use M buffers in main memory of each processor, and send a buffer to the corresponding machine only when is full.
Parallel algorithms for table operators

• Join(R(X,Y), S(Y,Z)):
  – We distribute tuples of R and S using the same hash function that only depends on Y
  – Join is then performed locally

• GroupBy(R,X,{f1,...,fn}):  
  – Distribute R with a hash function that depends on X
  – GroupBy locally

• Filter and Projection can be performed locally
Join algorithms in detail

- Colocated join: R and S are partitioned in the same way and fragments are co-located: local algorithm

- Directed join: R and S are partitioned in the same way but not co-located: choose one and send it to the corresponding nodes of the other

- Repartitioned join: R and S are not partitioned in the same way: we re-partition one (or both) according to the same approach and use directed join

- Broadcast join: if one table is small, we just send it entire to any node with a piece of S
Performance of parallel algorithms

• Total accesses and total CPU time increase, but we hope to reduce the elapsed time
• A unary operator takes $1/p$ elapsed time if we have $p$ processors operating in parallel
• What about join?
Performance of repartitioned join

• Join:
  1. \((N\text{Pag}(R) + N\text{Pag}(S))/p\) to read and hash the tuples
  2. We must send around \((N\text{Pag}(R) + N\text{Pag}(S))(p-1/p)\) block of data
  3. We need \(2*(N\text{Pag})/p\) at every site to perform a hash join or a sort-merge join (assuming tuple-level pipeline) (ignore the different numbers given in the book)

• Elapsed time is almost the same as sequential-time/p

• Apart from communication time (2) and the fact that one node may get more data and one may get less

• Every node gets \(N\text{Pag}/p\) data: if it fits main memory, we may avoid any I/O!
Distributed databases
Distributed systems vs. shared-noting parallel systems

• In a distributed system:
  – Communication is more expensive than in a parallel system
  – Node failure is independent, which gives better resilience
  – The system may get partitioned in two for a non-negligible amount of time
  – The system may be ‘federated’, that is, it may be managed by different authorities
  – We may have different levels of trust (usually regarded as ‘peer-to-peer’ rather than ‘distributed’)
Data distribution

• Partitioning: data communication is expensive, hence we may put data where is most used: horizontal partitioning (e.g.: the database may be distributed nation by nation) or even vertical partitioning (every site keeps the column it uses more)

• Replication: in order to have resilience, every fragment of a relation should be replicated

• Replication makes reading faster and updating slower
Designing data distribution

• The data distribution design:
  – Every relation is divided in horizontal/vertical fragments such as $\pi_{\text{item, date}} \sigma_{\text{nation='Italy'}}(\text{Sales})$
  – Every fragment is mapped to n sites - if we have a primary copy, we must also decide which copy is primary

• How to fragment is the easy part: we may define the smallest possible pieces and then map them to the same site

• Where to put fragments, and specifically how many copies for each fragment, is a difficult optimization problem
Distributed query processing: the distributed join problem

• We have $R(X,Y)$ at site $r$ and $S(Y,Z)$ at site $s$. Communication is the dominating cost. The two simplest possibilities:
  – We send $R$ to $s$
  – We send $S$ to $r$

• We would typically send the smallest one

• There is a third possibility: the semijoin reduction
The semijoin reduction

• The semijoin plan for \text{ioin}(R(X,Y),S(Y,Z))\), assuming that Y is much smaller than X and then Z:
  – Send \(\pi_Y(R)\) to s
  – s computes \(S1(Y,Z) = \text{semijoin}(\pi_Y(R), S(Y,Z))\)
  – Send \(S1(Y,Z)\) to r
  – R computes join\((R(X,Y),S1(Y,Z))\), which is equivalent to join\((R(X,Y),S(Y,Z))\)
• When is this a good idea?
Distributed consistency

• A transaction is now a distributed process that coordinates local transactions
  – How do we manage distributed commit?
  – How do we ensure distributed serializability?

• Consistency of data replication
  – How do we avoid data divergence in case of partitioning?
  – Is there a primary copy or are all copies created equal?
Distributed commit

• A typical distributed transaction in a federated system:
  – A client ‘c’ sends to a merchant ‘m’ and order and the two together send a request to a bank ‘b’ to issue the payment
  – At the end we would like to atomically update the state of the database ‘M’ of ‘m’ and of the database ‘B’ of ‘b’

• In a non-federated system
  – A bank is moving money from accounts in two distinct branches where two halves of its DB are stored. A failure happens. At restart we need a coherent state.
Two-Phase commit

• Assumptions:
  – A many-sites transaction with one site that acts as a coordinator
  – Every site has its local log
  – All messages in the protocol are logged
Fixing a date for a meeting

- We discussed, and 1st of June seems ok
- First phase: I ask everybody ‘is 1st of June ok’?
- People start answering – whoever says ‘yes’ is pre-committed: they MUST put 1st of June as busy in their calendar and cannot change their mind
- Second phase: after everybody has said yes, I tell everybody: ok, it is decided then, it is 1st of June
- I wait the ack of everybody, and if somebody does not ack I will insist until acked
The 2PC: Phase I

• Coordinator C: writes <Prepare,T> on its log
• C: sends to every Participat Pi: \texttt{send(Pi,prepare T)}
• Each Pi must answer, sooner or later, as follows:
  – It cannot commit:
    • writes <don’t commit,T> and \texttt{send(C, don’t commit T)}
  – It wants to commit:
    • Gets ready to redo in case of failure and writes <ready,T> on the log, entering in the pre-committed state: is not a commit, but from now on C and only C has the power to Abort
    • After this: \texttt{send(C, ready T)}
The 2PC: phase II: Abort case

- C decides whether to Commit – which requires that every Pi sended a ready msg – or to abort – which is the only choice if a Pi says ‘no’ or does not answer

- If C decides to Abort:
  - It writes <Abort,T> on its log
  - C: send(Pi, abort T) to every participant
  - Every Pi aborts T and then...
  - ...writes <Abort,T> on its log
The 2PC: phase II: Commit case

• C gets a ‘ready’ from every Pi and decides to Commit:
  – It writes <Commit,T> on its log
  – C: send(Pi, commit T) to every participant
  – Every Pi commits, which implies that it writes <Commit,T> on its log
Recovering after a crash

• The basic idea is very simple. The only difficult thing is proving that:
  – If there is a failure at any moment, we can always recover
  – If every site is guaranteed to eventually restart, then the protocol is guaranteed to eventually terminate
Messages and failures

- Every message may be duplicated, the second copy is just ignored; message send is ‘idempotent’
- Every message may be lost, when an answer does not arrive:
  - We first reiterate the request, with some policy (this is not even specified in the protocols)
  - We eventually assume that the partner is down
- Restart is always log-guided: I read the log and restart ‘from there’
Recovering Pi after a crash

• Last log record for T was:
  – <Commit,T> or <Abort,T>: easy, do as in the non-distributed case
  – <Don’t commit,T>, or is a <Write,T>: perform a local abort
  – <Ready,T>: contact the coordinator and the other sites to discover which was the decision; until an answer is obtained, the transaction is in the pre-committed condition and can neither be aborted nor be committed
Recovering C after a crash

• Last log record for T was:
  – <Prepare,T>: may send(Pi, Abort T), which is always allowed before the (Pi, Commit T), or do nothing
  – <Abort,T>: may (re)send(Pi, Abort T), or do nothing
  – <Commit T>: may (re)send(Pi, Commit T), or do nothing

• Are there other possibilities?

• If C receives a status request from some Pi that just recovered, for a transaction T, it consults the log:
  – Last record is <Commit T>: the transaction is committed
  – Otherwise, is Aborted
Recovering C by doing nothing

• <Prepare,T>:
  – Some site may be waiting a I phase or II phase msg from C; in this case, they will solicit C, which will answer ‘abort’

• <Abort,T>:
  – Some site may be waiting a II phase msg from C; in this case, they will solicit C, which will answer ‘abort’

• <Commit T>:
  – Some site may be waiting a II phase msg from C; in this case, they will solicit C, which will answer ‘commit’
When messages get lost

• C: send(P, prepare)
  – If lost: Pi may solicit but may also safely assume Abort

• Pi: send(C, ready/don’t commit)
  – If lost: Pi may solicit but may also safely decide to Abort

• C: send(Pi, abort/commit)
  – If lost: Pi MUST solicit or get information by the peers
  – Until the decision is known, Pi must remain in the very uncomfortable ‘pre-committed’ state
  – What if C is down ‘forever’?

• The third case is the problem of the 2PC protocol
Distributed locking: the centralized solution

• We can either lock the many physical copies – one by one – of a piece of data, or we may get a logical lock on the logical data: both solutions work

• The centralized solution: we have a centralized lock server which manages lock on the logical data

• The usual problems of centralized solutions:
  – Bottleneck for performance
  – Single point of failure
Distributed locking: the primary copy

• One copy of the data item is primary, and every lock should be taken there
• We still have a bottleneck and a single point of failure
Distributed locking: the distributed solution

• Every transaction just gets S/X locks on the local copies that it reads or writes

• Consistency problem: one transaction may read a copy while another is writing a different copy

• Two solutions:
  – Write-locks-all: in order to write, a transaction must get an X lock on all copies; in order to read, one lock is enough
  – Majority locking: in order to write, I need \((n+1)/2\) X locks, in order to read, I need \((n+1)/2\) S locks
Distributed locking: the quorum

• The quorum: we have an s quorum and an x quorum such that
  – $x+x>n$ and $s+x>n$ (n: number of copies)
  – In order to read, I need S on s copies; in order to write I need X lock on x copies
  – by $x+x>n$ and $s+x>n$ no two transactions may be able to take enough conflicting locks at the same time

• Some typical cases
  – $x=s=(n+1)/2$
  – $x=n$, $s=1$
  – $x=n-1$, $s=2$
Distributed deadlock

• Every centralized solution may be used – the waits-for graph, the timeout, the prevention
• In practice, we opt for timeout