|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Centralized vs. Distributed Database | | | – Multiple nodes connected using network  – Communication cost can be significant | |  | | |
|  | | Each memory and disk is owned by some processor that acts as a server for that data  – Scales to thousands of servers and beyond Important optimization goal: minimize network data transfer | | |  | | Each processor has a private memory but has direct access to all disks – Scale to tens of servers Example: Network attached storage (NAS) and storage area network (SAN) |
|  | |  | | |  | | |
|  | | All processors share direct access to a common global memory and to all disks – Does not scale beyond a single server Example: multicore processors | | |  | | |
| **Parallel SQL Database**  Most database programs are written in relational language SQL  – Goal: make SQL work on distributed system without rewriting  – Benefits of a high-level programming interface | | | | |  | | |
| **Partitioned Parallelism**  Partition data across nodes – Partition a table using the partition key (e.g., a single or multiple columns) – Each record is mapped to one node A single operator is executed collectively across multiple nodes | | | | | **Partition data across nodes** – Partition a table using the partition key (e.g., a single or multiple columns) – Each record is mapped to one node A single operator is executed collectively across multiple nodes | | |
|  | | | | |  | | |
| Partitioned Parallelism  Query execution  – Merge and Split operators Join execution  – Co-partitioned  – Partition-based  – Broadcast-based | | | | | Distributed Relational Operators  - Split a relation into smaller relations  - Merge multiple small relations into one bigger relation  - Exchange: move relations between distributed nodes  – Broadcast: Send the source relation to all the nodes  – Shuffle: Split relation and send each partition to corresponding node | | |
|  | | | | **Distributed Selection**  Step 1: Each partition is filtered locally  Step 2: Filtered partitions are exchanged and merged in one node | **Distributed Join**  —**Co-Partitioned**    R (A, B), S (B, C)  SELECT \* FROM R, S  WHERE R.B = S.B | Assume R and S are **co-partitioned**  – Partitioned on **join key** using partitioning function    rows in R1 cannot possibly join with rows in S2 | |
|  | | | | |  | | |
|  | | | | |  | | |
| Now both R and S are co-partitioned on B | | | | | Use the co-partitioned join algorithm to finish the query | | |
| **Distributed Join**  —Co-Partitioned    SELECT \* FROM R, S  WHERE R.B = S.B | What if **neither R nor S is partitioned on the join key?**    **Re-partition both R and S on column B using the same partition function!** | | | | **Distributed Join—Broadcast-Based**  If R and S are not co-partitioned, another solution is to broadcast one relation to all nodes | | |
|  | | | | | **Co-partitioned:** Works if the two relations are already co-partitioned on the join key (using the same partitioning function)  **Partition-based**: Partition one or two relations such that both relations are co-partitioned  **Broadcast-based**: Broadcast the smaller relation to all nodes | | |
| Exercise  We execute R⋈S on two nodes with the data layout shown below. Assume S is partitioned on the join key, but R is not. **What is the network IO cost for partition-based join and broadcast-based join, respectively**? (we assume uniform distribution; we assume the join result size is negligible) | | | | |  | | |
|  | | | | | co-partitioned R1xS1, R2xS2 Merge R2, S2 transfer to node N1 | | |
| Distributed Join—Partition-Based | | | | | execute R⋈S distributed on two nodes N1 and N2 where N1 has R1 (5gb) and S1(10Gb) partition and N2 with hold R2 (5Gb) and S2 (10Gb) partitions. Assume S is partitioned on the join key, but R is not. What is the network IO cost for partition-based join and broadcast-based join, respectively? (we assume uniform distribution; we assume the join result size is negligible) | | |
|  | | | | | Broadcasting? | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| * 1. **For this lecture knowledge of 2PL, OCC, MVCC** | | |  | | | |
| **L37** Centralized 2PL Lock table LT  Page 34, tran T1,T2  **Two-phase locking (2PL),**  SH read EX write   * 1. Hold locks through a lock table   2. Lock table is hash table for page 34 hash to LT, H(34) point to backet | Two-phase locking (2PL)  Can have centralized lock table.  Not efficient.e.g Data on First node, Lock table on third   * 1. Each node has a **local LT**   2. A transaction locks a page in the corresponding lock table (A on 1st node)   3. naïve sol. centralized LT not efficient   5. partitioning LT hash tables responsible for pages on that server | | How to handle **deadlock** ? detect **by cycle** in graph. For all nodes maintaining the separate graph is hard. Also maintaining graph on each node.   * 1. **Solution 1**: **Cycle detection in waits-for graph**   2. Challenge 1**:** Hard to perform cycle detection in a **distributed graph**   3. Challenge 2: Maintaining centralized graph creates **a performance bottleneck**   4. **Solution 2:** Deadlock prevention   **NO WAIT**   * 1. - Do not wait for lock deadlock can happen ?   **WAIT DIE**   * 1. Only high-priority txn waits for low-priority txn; low-priority txn self-aborts.   2. E.g., MS Orleans ?   **WOUND WAIT**   * 1. Only low-priority txn waits for high-priority txn; high-priority txn preemptively **aborts** low-priority txns E.g., Google Spanner | | | |
| **Distributed OCC**  Optimistic concurrency control | Each read returns the **value** and a **version number (seuste vo memorija)**  Each **committed write update** both the value and the version number  During validation, check whether any record in the read-set has been modified since the earlier read, by comparing the version numbers | | | | | |
| **Distributed MVCC**  Multi-version concurrency control  Need to generate monotonically increasing timestamp   * 1. -Each transaction is assigned a timestamp (*ts*) when the transaction starts   2. -Each record version has a write timestamp (wts) and a read timestamp (rts)   **Solution 1:** **centralized timestamp allocator**   * 1. Every txn must contact the timestamp allocator for a timestamp | | | **Solution 2: synchronized clocks**   * 1. -Use atomic clock and GPS to synchronize clocks across all servers   (Google)  Accordingly execution according timestamps covered by synh. clocks | | | |
| **INDEX**  **Distributed Index**  Index must be distributed across servers  Typically partition data based on primary key | | | **Distributed Primary Index** Each node maintains local index for its local partition of data  Index search on PK   * 1. - Uses partition function to find the right partition   2. - Search the local index in that partition to locate record | | | |
| **Distributed Secondary Index**  Each node maintains a local partition of the secondary index  Index search on Secondary Key (SK)   * 1. - Uses SK partition function to find the right partition   2. - Search the local SK index in that partition to find PK   3. - Search PK index to locate record | | | result SC 23 has 3 dif. PK-> 3 diff. nodes index, and than related records | | | |
| **Atomic commit protocol ACP**:  all partitions reach the same commit or abort decision of a transaction  The two updates must commit or abort atomically  If have centralized log it is not well scalable  **Atomicity needed**, commit/abort on all machines on transaction  (have’nt good sol) | | | A naïve approach: all nodes log and commit independently  Node 2 crashes before logging   * 1. - Transaction T commits in node 1 but not in node 2 | | | |
| **Two-Phase Commit (2PC)**  Key idea: let the coordinator log the final commit/abort decision  We must have consensus for commit/abort | | |  | | | Coordinator can be one of the nodes |
|  | | Coordinator log the final commit/abort decision  Phase 1: prepare phase  Coordinator promise (c/a) and ask for agreement of all subordinates    Coordinator commit if both yes  Subordinates copy decision |  | | Phase 2: commit phase   * Coordinator logs the decision * Coordinator sends the decision to subordinates   Coordinator forgets the transaction after receiving ACKs  What to do (commit/abort)  Coordinator commits, subordinates record decision | |
| 2PC – Abort Example  Subordinate returns VOTE NO if the transaction is aborted   * Subordinate can release locks and forget the transaction * Coordinator **do abort** and send info to all nodes   notif. only second subordinate | | | 2PC – All Subordinates Abort’  Skip the second phase entirely if the  transaction aborts at all the subordinates | | | |
| 2PC – Failures | | Use timeout to detect failures  Subordinate timeout   * 1. -Waiting for PREPARE: self abort | **Use timeout to detect failures**  Coordinator timeout   * 1. Waiting for vote: self-abort | Subordinate timeout   * 1. - Waiting for decision: contact coordinator or peer subordinates (**may block until the coordinator recovers**) | | |
|  | | Use timeout to detect failures  Coordinator timeout   * 1. - Waiting for ACK: contact subordinates | **2PC – Alternative Designs?** | Subordinate returns vote to coordinator before logging prepare?  **Problem**: subordinate may crash before the log record is written to disk. The log record is thus lost but the coordinator already committed the transaction | | |
| **2PC – Alternative Designs?** | | Coordinator sends decision to subordinates before logging the decision?  **Problem**: coordinator crashes before logging the decision and decides to abort after restart |  |  | | |
| **Primary Backup Replication**  **(Active-Passive)**  Replication (Active-Passive)  **Data replication** | | 1. -The primary node ships log to the backup node 2. -The backup node replays the log 3. -If the primary node crashes, the backup node is promoted to be the new primary | **Active-Active Replication** | 1. Same sequence of transactions are sent to all the active replicas 2. Each replica executes the transactions deterministically, such that all replicas produce the same results | | |