Distributed Databases

- Distributed Systems: Performance, Definitions
- Key-Value Stores
- Distributed Protocols
- Consistency of Replicas
- MVCC 2PL and Globally Consistent Timestamps (Spanner)
- Midterm 2

Metrics:
Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is N-times larger.
  - Measured by:
    $$ speedup = \frac{\text{small system elapsed time}}{\text{large system elapsed time}} $$
  - Speedup is **linear** if equation equals N.

- **Scaleup**: increase the size of both the problem and the system
  - N-times larger system used to perform N-times larger job
  - Measured by:
    $$ scaleup = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}} $$
  - Scale up is **linear** if equation equals 1.
Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **Startup/sequential costs**: Cost of starting up multiple processes, and sequential computation before/after parallel computation
  - May dominate computation time, if the degree of parallelism is high
  - Suppose $p$ fraction of computation is sequential
  - **Amdahl’s law**: speedup limited to: $1 / [(1-p)+(p/n)]$
  - **Gustafson’s law**: scaleup limited to: $1 / [n(1-p)+p]$

- **Interference**: Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.

- **Skew**: Increasing the degree of parallelism increases the variance in service times of executing tasks in parallel. Overall execution time determined by slowest of parallel executing tasks.

Parallel and Distributed Data Stores

- Supporting scalable data access
  - **Approach 1**: memcache or other caching mechanisms at application servers, to reduce database access
    - Limited in scalability, poor consistency
  - **Approach 2**: Partition (“shard”) data across multiple separate database servers
  - **Approach 3**: Use existing parallel databases
    - Historically: parallel databases that can scale to large number of machines were designed for decision support not OLTP
  - **Approach 4**: Massively Parallel Key-Value Data Store
    - Partitioning, high availability etc. completely transparent to application

- Sharding systems and key-value stores don’t support many relational features, such as joins, integrity constraints, etc., across partitions.
Sharding

- **Sharding**: partition data across multiple databases
- Partitioning usually done on some *partitioning attributes* (also known as *partitioning keys* or *shard keys* e.g. user ID
  - E.g., records with key values from 1 to 100,000 on database 1, records with key values from 100,001 to 200,000 on database 2, etc.
- Application must track which records are on which database and send queries/updates to that database
- Positives: scales well, easy to implement
- **Drawbacks:**
  - Not transparent: application has to deal with routing of queries, queries that span multiple databases
  - When a database is overloaded, moving part of its load out is not easy
  - Chance of failure more with more databases
    - need to keep replicas to ensure availability, which is more work for application

Parallel and Distributed Databases

- Parallel databases run multiple machines (cluster)
  - Developed in 1980s, well before Big Data
- Parallel databases were designed for smaller scale (10s to 100s of machines)
  - Did not provide easy scalability
- **Replication** used to ensure data availability despite machine failure
  - But typically restart query in event of failure
    - Restarts may be frequent at very large scale
    - Map-reduce systems (Apache Hadoop, Spark) can continue query execution, working around failures
Replication and Consistency

- **Availability** (system can run even if parts have failed) is essential for parallel/distributed databases
  - Via replication, so even if a node has failed, another copy is available
- **Consistency** is important for replicated data
  - All live replicas have same value, and each read sees latest version
  - Often implemented using majority protocols
    - E.g., have 3 replicas, reads/writes must access 2 replicas
      - Details in chapter 23
- **Network partitions** (network can break into two or more parts, each with active systems that can’t talk to other parts)
- Brewer’s CAP “Theorem” that in presence of partitions, cannot guarantee both availability and consistency

CAP “Theorem”

- Three properties of a system
  - **Consistency**
    - an execution of a set of operations (reads and writes) on replicated data is said to be consistent if its result is the same as if the operations were executed on a single node, in a sequential order that is consistent with the ordering of operations issued by each process (transaction)
  - **Availability** (system can run even if parts have failed)
    - Via replication
  - **Partitions** (network can break into two or more parts, each with active systems that can’t talk to other parts)
- You can have at most two of these three properties for any system
- but in a distributed system, partitions happen
- Choose one of consistency or availability
  - Traditional **databases choose consistency**
  - Many **web applications (key-value stores) choose availability**
    - Except for specific parts such as order processing
Distributed Databases

- Distributed Systems: Performance, Definitions
- Key-Value Stores
- Distributed Protocols
- Consistency of Replicas
- MVCC 2PL and Globally Consistent Timestamps (Spanner)

Key Value Storage Systems

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are partitioned across multiple machines and
- Queries are routed by the system to appropriate machine
- Records are also replicated across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are consistent
Key Value Storage Systems

- Key-value stores may store
  - uninterpreted bytes, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - Wide-table (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
- JSON
  - MongoDB, CouchDB (document model)
- Document stores store semi-structured data, typically JSON
- Some key-value stores support multiple versions of data, with timestamps/version numbers

Data Representation

- An example of a JSON object is:

```json
{
  "ID": "22222",
  "name": {
    "firstname": "Albert",
    "lastname": "Einstein"
  },
  "deptname": "Physics",
  "children": [
    { "firstname": "Hans", "lastname": "Einstein" },
    { "firstname": "Eduard", "lastname": "Einstein" }
  ]
}
```
Key Value Storage Systems

- Key-value stores support
  - put(key, value): used to store values with an associated key,
  - get(key): which retrieves the stored value associated with the specified key
  - delete(key) -- Remove the key and its associated value
- Some systems also support range queries on key values
- Document stores also support queries on non-key attributes
  - See book for MongoDB queries
- Key value stores are not full database systems
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems
  - Also called NoSQL systems

Architecture of a Key-Value Store (modeled after Yahoo! PNUTS)
Index Structures in Key-Value Stores

- Similar to relational DBs:
  - B+-tree file organization works well for range queries
  - Write optimized trees, especially LSM trees (Section 24.2) work well for updates as well as for range queries
    - Used in BigTable, HBase and many other key-value stores
  - Some key-value stores organize records on each node by hashing, or just build a hash index on the records

Transactions in Key-Value Stores

- Most key-value stores don’t support full-fledged transactions
  - But treat each update as a transaction, to ensure integrity of internal data structure
  - Some key-value stores allow multiple updates to one data item to be committed as a single transaction
  - Without support for transactions, secondary indices cannot be maintained consistently
    - Some key-value stores do not support secondary indices at all
    - Some key-value stores support asynchronous maintenance of secondary indices
  - Some key-value stores support ACID transactions across multiple data items along with two-phase commit across nodes
    - Google MegaStore and Spanner
  - More details in Chapter 23
Transactions in Key-Value Stores

- Some key-value stores support concurrency control via locking and snapshots
- Some support *atomic test-and-set* and *increment* on data items
  - Others do not support concurrency control
- Key-value stores implement recovery protocols based on logging to ensure durability
  - Log must be replicated, to ensure availability in spite of failures
- Distributed file systems are used to store log and data files in some key-value stores such as BigTable, HBase
  - But distributed file systems do not support (atomic) updates of files except for appends
  - LSM trees provide a nice way to index data without requiring updates of files
- Some systems use persistent messaging to manage logs
- Details in Chapter 23

Querying and Performance Optimizations

Many key-value stores do not provide a declarative query language!

- Applications must manage joins, aggregates, etc on their own
- Some applications avoid computing joins at run-time by creating (what is in effect) materialized views
  - Application code maintains materialized views
  - E.g., If a user makes a post, the application may add a summary of the post to the data items representing all the friends of the user
- Many key-value stores allow related data items to be stored together
  - Related data items form an *entity-group*
  - e.g., user data item along with all posts of that user
  - Makes joining the related tuples very cheap
- Other functionality includes
  - Stored procedures executed at the nodes storing the data
  - Versioning of data, along with automated deletion of old versions
Distributed Databases

- Distributed Systems: Performance, Definitions
- Key-Value Stores
- Distributed Protocols
- Consistency of Replicas
- MVCC 2PL and Globally Consistent Timestamps (Spanner)

Distributed Protocols

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
  - Latency of replication across geographically distributed data centers much higher than within data center
    - Some key-value stores support synchronous replication
    - Others support asynchronous replication
      - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
      - Must deal with small risk of data loss if data center fails.
Distributed Transactions

- Local transactions
  - All clients access/update data at a single database
- Global transactions
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local transaction manager
  - Each site has a transaction coordinator
    - Global transactions submitted to any transaction coordinator
Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - transaction must be committed at all sites or aborted at all sites.
- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the execution and commit/abort of the transactions executing at that site.

System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of communication links
    - Handled by network protocols, by routing messages via alternative links
  - Network partition
    - A network is said to be partitioned when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
  - Network partitioning and site failures are generally indistinguishable.
Commit Protocols

- **Commit protocols** are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - cannot have transaction committed at one site and aborted at another
- **Two-phase commit** (2PC) protocol is widely used
- Three-phase commit (3PC) protocol avoids some drawbacks of 2PC, but is more complex
- **Consensus protocols** solve a more general problem, but can be used for atomic commit
  - More on these later in the chapter
- The protocols we study all assume *fail-stop* model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
  - Protocols that can tolerate some number of malicious sites discussed in bibliographic notes online

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let $T$ be a transaction initiated at site $S_i$, and let the transaction coordinator at $S_i$ be $C_i$
Phase 1: Obtaining a Decision

- Coordinator asks all participants to prepare to commit transaction $T_i$
  - $C_i$ adds the records $\langle\text{prepare } T_i\rangle$ to the log and forces log to stable storage
  - sends prepare $T$ messages to all sites at which $T_i$ executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record $\langle\text{no } T_i\rangle$ to the log and send abort $T_i$ message to $C_i$
  - if the transaction can be committed, then:
    - add the record $\langle\text{ready } T_i\rangle$ to the log
    - force all records for $T_i$ to stable storage
    - send ready $T_i$ message to $C_i$

Transaction is now in ready state at the site

Phase 2: Recording the Decision

- $T_i$ can be committed of $C_i$ received a ready $T_i$ message from all the participating sites: otherwise $T$ must be aborted.
- Coordinator adds a decision record, $\langle\text{commit } T_i\rangle$ or $\langle\text{abort } T_i\rangle$, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.
Two-Phase Commit Protocol

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use a consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- Consensus is also used to ensure consistency of replicas of a data item
  - Details later in the chapter
Replication

- *High availability* is a key goal in a distributed database
  - *Robustness*: the ability to continue function despite failures
- Replication is key to robustness
- Replication decisions can be made at level of data items, or at the level of partitions

Distributed Databases

- Distributed Systems: Performance, Definitions
- Key-Value Stores
- Distributed Protocols
- Consistency of Replicas
- MVCC 2PL and Globally Consistent Timestamps (Spanner)
Consistency of Replicas

- **Consistency** of replicas
  - Ideally: all replicas should have the same value \( \rightarrow \) updates performed at all replicas
    - But what if a replica is not available (disconnected, or failed)?
  - Suffices if reads get correct value, even if some replica is out of date
  - Above idea formalized by **linearizability**: given a set of read and write operations on a (replicated) data item
    - There must be a linear ordering of operations such that each read sees the value written by the most recent preceding write
    - If \( o_1 \) finishes before \( o_2 \) begins (based on external time), then \( o_1 \) must precede \( o_2 \) in the linear order
  - Note that linearizability only addresses a single (replicated) data item; serializability is orthogonal

Consistency with Geo-Replication

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
  - Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - Update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.
Concurrency Control With Replicas

- Focus here on concurrency control with locking
  - Failures addressed later
  - Ideas described here can be extended to other protocols
- **Primary copy**
  - one replica is chosen as primary copy for each data item
    - Node containing primary replica is called primary node
  - concurrency control decisions made at the primary copy only
- Benefit: Low overhead
- Drawback: primary copy failure results in loss of lock information and non-availability of data item, even if other replicas are available
  - Extensions to allow backup server to take over possible, but vulnerable to problems on network partition

Concurrency Control With Replicas (Cont.)

- **Majority protocol**:
  - Transaction requests locks at multiple/all replicas
  - Lock is successfully acquired on the data item only if lock obtained at a majority of replicas
- Benefit: resilient to node failures and node failures
  - Processing can continue as long as at least a majority of replicas are accessible
- Overheads
  - Higher cost due to multiple messages
  - Possibility of deadlock even when locking single item
    - How can you avoid such deadlocks?
Concurrency Control With Replicas (Cont.)

- **Read-Biased protocol**
  - Shared lock can be obtained on any replica
    - Reduces overhead on reads
  - Exclusive lock must be obtained on all replicas
    - Blocking if any replica is unavailable

Quorum Consensus Protocol

*Quorum consensus* protocol for locking

- Each site is assigned a weight; let \( S \) be the total of all site weights
- Choose two values **read quorum** \( Q_r \) and **write quorum** \( Q_w \)
  - Such that \( Q_r + Q_w > S \) and \( 2 * Q_w > S \)
- Each read must lock enough replicas that the sum of the site weights is \( \geq Q_r \)
- Each write must lock enough replicas that the sum of the site weights is \( \geq Q_w \)
- Can choose \( Q_r \) and \( Q_w \) to tune relative overheads on reads and writes
  - Suitable choices result in majority and read-biased protocols.
    - What are they?
Dealing with Failures

- **Read-one-write-all-copies** assumes all copies are available
  - Will block if any site is not available
- **Read one write all available** (ignoring failed sites) is attractive, but wrong
  - Failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an incorrect value
  - With network partitioning, sites in each partition may update same item concurrently
    - believing sites in other partitions have all failed

Handling Failures with Majority Protocol

- The majority protocol with **version numbers**:
  - Each replica of each item has a version number
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations:
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
  - Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task
Distributed Databases

- Distributed Systems: Performance, Definitions
- Key-Value Stores
- Distributed Protocols
- Consistency of Replicas
- MVCC 2PL and Globally Consistent Timestamps (Spanner)

Multiversion 2PL and Globally Consistent Timestamps

- Recall multiversion 2PL protocol:
  - Read only transactions get timestamp at start
    - $T_i$ reads latest committed version of data items with $TS < startTS(T_i)$
  - Update transactions perform 2PL, and also get timestamp at commit
  - Serialization order defined by timestamp
- Question: can we use MV2PL in a distributed system
- Answer: yes, but a lot of conditions apply
  - If commits are serialized at central coordinator, timestamps can be given based on counter
  - But if commits are distributed, how to give timestamps in a consistent manner?
    - Clocks may not be in sync, later commit may get lower timestamp
    - Out of order timestamp issue may result in serialization order not matching timestamp order
Multiversion 2PL and Globally Consistent Timestamps

- Centralized coordinator to assign consistent timestamps
  - Can be done, but becomes bottleneck
- Google Spanner ideas:
  - In an ideal world, clocks are synchronized, and can be used to assign commit timestamps to transactions
  - In reality, clocks are out of sync
  - Key ideas:
    - Use atomic clocks, GPS etc to periodically get precise time
    - Derive bound on how out-of-sync a node’s clock \( t' \) can be w.r.t. to actual time \( t \):
      \[ t' - \varepsilon \leq t \leq t' + \varepsilon \]
      - Means maximal skew between any two nodes is \( 2\varepsilon \)
  - So how do we assign commit timestamps in distributed fashion?
    - Commit wait:
      - assign commit time \( t_c \) as \( t' + \varepsilon \)
      - wait, holding locks, until \( t_c + \varepsilon \) (which is \( t' + 2\varepsilon \))
      - If transaction \( T_i \) holds lock this long, it is impossible for any \( T_j \), on a maximally skewed other node, to:
        - get locks after \( T_i \) and
        - have commit time before \( T_i \)

Multiversion 2PL and Globally Consistent Timestamps (cont):

- Google Spanner ideas (cont):
  - If version of \( x \) has timestamp \( t_s \), then \( x \) definitely had that value at time \( t_s \)
  - System can generate transactionally consistent snapshot as of actual time \( t_s \) (external consistency)
  - Commit processing can still take time
    - With 2PC status of transaction may not be known for a while
      - Reads may have to wait till status of transaction is known
    - But read-only transactions can use a snapshot timestamp \( t_s \) such that all transactions before that timestamp have been committed or aborted
      - Read can proceed without waiting
      - But perhaps with older versions of data
  - Throughput unaffected
  - Somewhat longer latency
Other Concurrency Control Techniques

- Distributed snapshot isolation
  - Running Snapshot Isolation separately on each node may result in different serialization orders at different nodes
  - Extensions to SI to ensure consistent ordering have been proposed
- Concurrency control in federated databases
  - Local transactions
  - Global transactions
  - Local serializability may not guarantee global serializability unless all nodes use 2PL
  - Use idea of tickets to create conflicts that will ensure serializability

Exam 2

- This evening:
  - answer sheet will be posted
  - regrade requests will be opened until Saturday, midnight
- Next week:
  - Consensus!, go over quizzes, prep for final