Distributed Data Store

Large-Scale Distributed file system

- Q: What if we have too much data to store in a single machine?

- Q: How can we create one big filesystem over a cluster of machines, whose data is reliable despite individual node failures?
- Basic idea:
  - Split the data into small chunks
  - Distribute the chunks to nodes
  - Replicate each chunk at multiple nodes to deal with failure
  - A master server with the filename -> chunk mapping
- GFS (HDFS)

```
      [ GFS diagram ]

        client

        master

chunkserver chunkserver chunkserver
```

  - Provides a file system (global namespace) over cluster of machines
  - A file is split into (often) 64M chunks
    * each chunk is replicated on multiple chunkservers
  - GFS master + chuck server
  - Master provides
    * file name -> chuck server mapping
    * chunk server monitoring and replacement
  - Chunk server provides actual data chunk

    * best chunkserver is chosen to minimize network bandwidth consumption

  - Optimized for

    * Sequential/random read of large file
    * Append at the end
      - Synchronizing random writes much more costly with failures

  - Q: Is a single master server enough to deal with the huge filesystem?

    * Ex) 1 billion files. 64 byte name per file. 4GB per file. How much space to remember the mapping?

- SAN (Storage Area Network)

  - "Scale up" approach for disks
  - Provides SCSI interface over many hard disks (often thru optical LAN)
  - All nodes in the cluster see one shared big disk
  - Convenient but very expensive

NoSQL datastores

- Limitation of relational databases

  - Relation: very general. But is it the best data representation?
    * e.g., java objects
  - SQL: very expressive query language. But is this expressiveness necessary?
    * e.g., student records. How will it be accessed?
  - The power of SQL also leads to complexity. Do we really need it?
    * Very difficult to reimplement full relational database on a cluster architecture
    * Need for a new system that is designed from the ground up for
      - Ultimate scalability for highly distributed environment
      - Core functionality necessary for new Web apps
  - ACID: very strong data integrity guarantee. But is it necessary?
    * e.g., user's status updates. Is ACID necessary?

- Difficulty of maintaining a shared state among distributed nodes

  - **Byzantine-general problem**
    * Two generals A, B need to attack together to win

        * Messenger may be lost during delivery (say with 90% chance)
        * Q: How can they make sure simultaneous attack on Sunday midnight?
        * Q: Any problem?
            - A sends a messenger to B
            - A attacks after sending a messenger
            - B attacks once he hears from a messenger
        * Q: Any better?
            - A sends a messenger to B
            - B sends ACK messenger back to A
            - A attacks if he gets ACK from B
            - B attacks if he hears from A's messenger
        * Q: Any better?
            - A sends a messenger. B ACKS. A ACKS back to B.
        * Q: How to improve chance of success?
    - Remarks
        * Keeping states on multiple nodes synchronized is often VERY COSTLY.
            - Synchronization overhead outweighs benefit for more than a few nodes
        * If possible, relax "state synchronization guarantee" on multiple nodes
            - Otherwise, the overall performance will be too based compared to centralized processing
    - CAP theorem (by Eric Brewer)
        * *Consistency*: after an update, all readers will see the latest update
        * *Availability*: continuous operation despite node failure
        * *Partition Tolerance*: continuous operation despite network partition
        * Only two of three characteristics can be guaranteed by any system

- Two important questions:
    1. Q: What data model to use?
        - *(Key, value) store*
        - *Column store*
        - *Document store*
    2. Q: How to relax consistency guarantee?
        - MongoDB
            * Atomicity guarantee on individual operation

    - No transaction support on multiple operations
    - The application is responsible for concurrency control
  - ACID vs. BASE
    - *Basically Available*: system works basically all the time
    - *Soft-state*: does not have to be consistent all the time
    - *Eventual consistency*: will be in a consistent state eventually
  - Possible consistency models
    - Read Your Own Writes (RYOW) Consistency
    - Session Consistency
    - Monotonic-Read Consistency: readers will see only newer update in the future
    - . . .

- **(Key, value) store**
  - Example: Amazon SimpleDB, Redis, Memcached, . . .
  - Example scenario
    - Key: student id, Value: (student name, address, email, major, . . .)
    - Q: How to process "find all student info with sid 301"?
    - Q: How to process "find student email with sid 301"?
    - Q: How to process "change the major of sid 301 to CS"?
    - Q: How to process "find student whose email is cho@cs"?
    - Remarks
      - Read can be inefficient for a single-field retrieval
      - Update can be inefficient
      - Data access using a non-primary key is not supported

      - Need to create and maintain a separate index for non-key access

- **Column store**
  - Example: Google's BigTable, Cassandra (Facebook), ...
  - Each data item is a row in a table - Each row has a unique row-key, but may not have all column values
  - Columns are grouped as "column families"
    * Preserves locality of column access and storage
  - *Data is accessed through (row-key, column-name) value*
  - Compared to (key,value) store, more structure of the data is exposed to the system
    * More efficient update and retrieval
  - Q: What about retrieval based on column values only without row-key (like student by email)?
    * Data access is still limited to row-key
- **Document store**
  - Example: MongoDB, Amazon DynamoDB, CouchDB
  - Document consists of multiple fields, which are (key, value) pairs.
  - No preset "schema" for documents
  - Example document:

```
Title: CouchDB
Categories: [Database, NoSQL, Document Database]
Date: Jan 10, 2011
Author: ...
```

  - User specifies the fields on which to build an index
    * Allows data retrieval based on "non-key" fields (by using appropriate indexes)
    * But, this data store will be more complex to build than others and less scalable

Consistent hashing

- Q: How to distribute objects with different keys to k nodes? What about "hash(object) mod k"?

- Q: How to minimize data reorganization when a new node is introduced?
  - *Consistent hashing*
    - Hash BOTH objects and *nodes* to a hash ring
    - Assign objects to the node right behind them
  - Q: What to do when a new node is introduced?

- Q: Non-uniform hash range for each node. How to minimize load imbalance?
  - Hash many virtual nodes to the ring
  - Assign virtual nodes to physical nodes
- Q: How to make data available despite node failure?
  - Replication to the next k nodes

Distributed Transactions

- Q: How can we guarantee atomicity of a transaction on multiple machines? Either everyone commits or no one should commit.

Two-phase commit

- Before commit, ask everyone whether they are ready
  - PREPARE -> VOTE-COMMIT/VOTE-ABORT

  - Note: Anyone who said ready cannot say otherwise later

- If everyone says yes, commit
  - if anyone says no, abort

```
     Coordinator                        Participant

       Initial ------- PREPARE  ------->  Initial
          |                                  |
         Wait                                |
          | <----- VOTE-ABORT ------+- N - ready?
          |                         |        |Y
          | <-------- VOTE-COMMIT --|----- Ready
  +-Y-- any no?                     |        |
  |       |N                        |        |
Abort ----|--- GLOBAL-ABORT --------|------->|
          |                         |        |
        Commit ---- GLOBAL-COMMIT --|------->|
                                    +- N - commit?
                                    |        |Y
                                  Abort    Commit
```

- Q: Any potential problem of two phase commit?

- Q: Where should we add timeout to avoid indefinite wait?

  - Remark: Do not get confused with *two-phase locking*

- Two-phase commit can be used for managing distributed transactions in general
  - Example: A travel site that arranges both flight and hotel
    * User wants to book the flight F and hotel H using credit card C
    * The site needs to coordinate transaction with with banks, hotels, and airlines
    * Q: How should the site handle the booking?

    * Q: What if the credit card authorizations fail? What if the flight is no longer available? What if the hotel is no longer available?

Asynchronous transaction

- Q: What if one participant is very slow?
  - Example: Starbucks. Cashier faster. Barista slower.
- Q: What does two-phase commit mean in this scenario?
- Q: How can we let each participant go ahead without waiting for the slow one?
- Q: What does Starbucks do?
- *Asynchronous transaction*
  - Each participant "commits" whenever he is done and moves ahead
    * Transaction = sequence of smaller transactions by each participant
  - The entire transaction is done when every participant commits
  - No coordinated wait and synchronous commit
- Q: What if the coffee machine breaks down after customer paid?
  - *Compensating transaction*
    * A transaction that "rolls back" a committed transaction
  - The coordinator should keep track of the "dependency" of transactions
    * Together with their compensating transactions
  - If any transaction aborts, run compensating transaction for all committed transactions
- Q: When should we use two-phase commit/asynchronous transaction?
  - Importance of individual commit guarantee

- Duration of individual transaction
- Probability of abort