Unit 4: Database Architectures

By
M. R. Sanghavi
Unit 4: Syllabus

- Centralized and Client-Server Systems
  - Two, Three Tier Architecture
- Parallel Systems
- Distributed Systems
- Database Connectivity
- Web Technology
- Database Administration and Management
- Connectivity using MongoDB
- Cassandra
Centralized Systems

- Run on a single computer system and do not interact with other computer systems.
- General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory.
- Single-user system (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks; the OS may support only one user.
- Multi-user system: more disks, more memory, multiple CPUs, and a multi-user OS. Serve a large number of users who are connected to the system via terminals. Often called server systems.
A Centralized Computer System
Server systems satisfy requests generated at $m$ client systems, whose general structure is shown below:
Client-Server Systems (Cont.)

- Database functionality can be divided into:
  - **Back-end**: manages access structures, query evaluation and optimization, concurrency control and recovery.
  - **Front-end**: consists of tools such as forms, report-writers, and graphical user interface facilities.
  - The interface between the front-end and the back-end is through SQL or through an application program interface.
Advantages of replacing mainframes with networks of workstations or personal computers connected to back-end server machines:

- better functionality for the cost
- flexibility in locating resources and expanding facilities
- better user interfaces
- easier maintenance
Parallel Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.

- A **coarse-grain parallel** machine consists of a small number of powerful processors.

- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.

- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted
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:
    \[
    \text{speedup} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}
    \]
  - 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:
    \[
    \text{scaleup} = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}}
    \]
  - Scale up is **linear** if equation equals 1.
Speedup

- Linear speedup
- Sublinear speedup

Graph showing the relationship between speed and resources.
Scaleup

\[ \frac{T_S}{T_L} \]

- linear scaleup
- sublinear scaleup

problem size
Parallel Database Architectures

- **Shared memory** -- processors share a common memory
- **Shared disk** -- processors share a common disk
- **Shared nothing** -- processors share neither a common memory nor common disk
- **Hierarchical** -- hybrid of the above architectures
Parallel Database Architectures

(a) shared memory

(b) shared disk

(c) shared nothing

(d) hierarchical
Shared Memory

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.
- Downside – architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck
- Widely used for lower degrees of parallelism (4 to 8).
Shared Disk

- All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - The memory bus is not a bottleneck
  - Architecture provides a degree of fault-tolerance — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.
- Examples: IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early commercial users
- Downside: bottleneck now occurs at interconnection to the disk subsystem.
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.
Shared Nothing

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.

- Examples: Teradata, Tandem, Oracle-n CUBE

- Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.

- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.

- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.
Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
- Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.
- Each node of the system could be a shared-memory system with a few processors.
- Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.
- Reduce the complexity of programming such systems by distributed virtual-memory architectures.
  - Also called non-uniform memory architecture (NUMA)
Distributed Systems

- Data spread over multiple machines (also referred to as sites or nodes).
- Network interconnects the machines
- Data shared by users on multiple machines
Distributed Databases

- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution

- Heterogeneous distributed databases
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality

- Differentiate between local and global transactions
  - A **local transaction** accesses data in the **single** site at which the transaction was initiated.
  - A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.
Trade-offs in Distributed Systems

- Sharing data – users at one site able to access the data residing at some other sites.
- Autonomy – each site is able to retain a degree of control over data stored locally.
- Higher system availability through redundancy — data can be replicated at remote sites, and system can function even if a site fails.
- Disadvantage: added complexity required to ensure proper coordination among sites.
  - Software development cost.
  - Greater potential for bugs.
  - Increased processing overhead.
Common Functions of DBA

- Database Planning
  - Standards, procedures, enforcement
- Requirements Gathering and Conceptual Design
- Logical Design
- Physical Design and Implementation
- Testing and Debugging
- Operations and Maintenance
- Training and Support

(Follows the requirements of the DBLC phases)
DBA’s Managerial Role

- Control and Planning Dimensions of Database Administration
  - Coordinating, Monitoring, and Allocating database administration resources
    - People
    - Data
  - Defining Goals and Formulating Strategic Plans for the Database Administration function
# DBA’s Responsibilities

<table>
<thead>
<tr>
<th>DBA Activity</th>
<th>DBA Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning</td>
<td>End-User Support</td>
</tr>
<tr>
<td>Organizing</td>
<td>Policies, Procedures, Standards</td>
</tr>
<tr>
<td>Testing</td>
<td>Data Security, Privacy, Integrity</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Data backup and Recovery</td>
</tr>
<tr>
<td>Delivering</td>
<td>Data Distribution and Use</td>
</tr>
</tbody>
</table>
End-User Support

- User Requirements Gathering
  - Understanding of the users’ views and needs
  - Present and Future information needs
- Conflict and Problem Resolution
  - Solutions in one department may cause problems in another
- Finding Solutions to Information Needs
- Ensure Quality and Integrity of Applications and Data
- Build End-User Confidence
- Manage the Training and Support of DBMS users
Policies, Procedures, and Standards

- Policies: General Statements of Direction or action that communicate and support DBA goals
- Procedures: Written Instructions that describe a services of steps to be followed during the performance of a given activity
- Standards: More detailed and specific than policies, and describe the minimum requirements of a DBA activity
  - Rules that are used to evaluate the quality of the activity
Areas of Policies and Procedures

- End-User database requirements gathering
- Database design and modeling
- Documentation and Naming conventions
- Design, coding, and testing of applications
- Database software selection
- Database security and integrity
- Database backup and recovery
- Database maintenance and operation
- End-user training
Data Security, Privacy and Integrity

- User Access Management
  - Define each user to the database
    - Operating System Level
    - Database Level
  - Assign Passwords
  - Define User Groups
  - Assign Access Privileges
    - Read
    - Write
    - Delete
  - Physical Access Control
- View Definitions
  - Protect and Control the Scope of the Data that is accessible to a user
- DBMS utilities access control
  - Limit the use of query and reporting tools
- DBMS usage Monitoring
  - Audit Logs

(More difficult in distributed databases)
Data Backup and Recovery

- Disaster Management
  - Periodic Data and Application Backups
    - Full
    - Incremental
    - Concurrent
  - Proper Backup Identification
  - Convenient and safe backup storage
  - Physical protection of hardware and software
  - Personal Access Control to the software of a database installation
  - Insurance Coverate for the data in the database
Data Backup and Recovery

- Recovery and Contingency plans
  - Tested
  - Evaluated
  - Practiced
- Will Not Recover all components of an IS
  - Establish priorities for the nature and extend of the data recovery process
Data Distribution and Use

- Data is only useful when:
  - Given to the Right User
  - Right Time
  - Right Format

- Programmers Deliver programs to access data
  - Time consuming for DBA

- Data Distribution allows end users to access the database
  - Internet
  - Intranets
  - Queries, Web Front Ends
  - End Users may make improper use of database, data duplication, etc.
DBA’s Technical Role

- DBMS and utilities, evaluation, selection and installation
- Design and implementation of Database
- Testing and Evaluation
- Operation of DBMS, Utilities, and Applications
- Training and Supporting Users
- Maintenance of DBMS, Utilities, and Applications
Today’s Talk

- Developing your first Web Application with MongoDB
- What is MongoDB, Platforms and availability
- Data Modeling, queries and geospatial queries
- Location bases App
- Example uses MongoDB Javascript shell
Why MongoDB

- Intrinsic support for agile development
- Super low latency access to your data
  - Very little CPU overhead
- No Additional caching layer required
- Built in Replication and Horizontal Scaling support
MongoDB

- Document Oriented Database
  - Data is stored in documents, not tables / relations

- MongoDB is Implemented in C++ for best performance
- Platforms 32/64 bit Windows Linux, Mac OS-X, FreeBSD, Solaris

- Language drivers for:
  - Ruby / Ruby-on-Rails
  - Java, C#, JavaScript, C / C++, Erlang, Python, Perl others..... and much more! ..
Design

- Want to build an app where users can check in to a location
- Leave notes or comments about that location
- Iterative Approach:
  - Decide requirements
  - Design documents
  - Rinse, repeat :-)
Requirements

- **Locations**
  - Need to store locations (Offices, Restaurants etc)
    - Want to be able to store name, address and tags
    - Maybe User Generated Content, i.e. tips / small notes ?

- Want to be able to find other locations nearby
Requirements

- Locations
  - Need to store locations (Offices, Restaurants etc)
    - Want to be able to store name, address and tags
    - Maybe User Generated Content, i.e. tips / small notes?
  - Want to be able to find other locations nearby

- Checkins
  - User should be able to ‘check in’ to a location
  - Want to be able to generate statistics
<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Mongo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table, View</td>
<td>Collection</td>
</tr>
<tr>
<td>Row(s)</td>
<td>JSON Document</td>
</tr>
<tr>
<td>Index</td>
<td>Index</td>
</tr>
<tr>
<td>Join</td>
<td>Embedded Document</td>
</tr>
<tr>
<td>Partition</td>
<td>Shard</td>
</tr>
<tr>
<td>Partition Key</td>
<td>Shard Key</td>
</tr>
</tbody>
</table>
Collections

Locations
loc1, loc2, loc3

Users
User1, User2
JSON Sample Doc

```json
{ 
  _id: ObjectId("4c4ba5c0672c685e5e8aabf3"),
  author: "roger",
  date: "Sat Jul 24 2010 19:47:11 GMT-0700 (PDT)",
  text: "MongoSF",
  tags: [ "San Francisco", "MongoDB" ]
}
```

Notes:
- `_id` is unique, but can be anything you’d like
BSON

- JSON has powerful, but limited set of datatypes
  - Mongo extends datatypes with Date, Int types, Id, ...

- MongoDB stores data in BSON

- BSON is a binary representation of JSON
  - Optimized for performance and navigational abilities
  - Also compression
  - See bsonspec.org
location1 = {
    name: "10gen East Coast",
    address: "134 5th Avenue 3rd Floor",
    city: "New York",
    zip: "10011"
}
location1 = {
    name: "10gen East Coast",
    address: "134 5th Avenue 3rd Floor",
    city: "New York",
    zip: "10011"
}

db.locations.find({zip:"10011").limit(10)
location1 = {
    name: "10gen East Coast",
    address: "17 West 18th Street 8th Floor",
    city: "New York",
    zip: "10011",
    tags: ["business", "mongodb"]
}
Places v2

location1 = {
    name: "10gen East Coast",
    address: "17 West 18th Street 8th Floor",
    city: "New York",
    zip: "10011",
    tags: ["business", "mongodb"]
}

db.locations.find({zip:"10011", tags:"business"})
Places v3

location1 = {
  name: "10gen East Coast",
  address: "17 West 18th Street 8th Floor",
  city: "New York",
  zip: "10011",
  tags: ["business", "mongodb"],
  latlong: [40.0, 72.0]
}
location1 = {
  name: "10gen East Coast",
  address: "17 West 18th Street 8th Floor",
  city: "New York",
  zip: "10011",

  tags: ["business", "cool place"],

  latlong: [40.0,72.0]
}

db.locations.ensureIndex({latlong: "2d"})
location1 = {
  name: "10gen HQ",
  address: "17 West 18th Street 8th Floor",
  city: "New York",
  zip: "10011",
  tags: ["business", "cool place"],
  latlong: [40.0, 72.0]
}

db.locations.ensureIndex({latlong: "2d"})
db.locations.find({latlong: {$near: [40, 70]}})
location1 = {
    name: "10gen HQ",
    address: "17 West 18th Street 8th Floor",
    city: "New York",
    zip: "10011",
    latlong: [40.0,72.0],
    tags: [“business”, “cool place”],
    tips: [
        {user:"nosh", time:6/26/2010, tip:"stop by for office hours on Wednesdays from 4-6pm"},
        {.....},
    ]
}
Creating your indexes

db.locations.ensureIndex({tags:1})

db.locations.ensureIndex({name:1})

db.locations.ensureIndex({latlong:”2d”})

Finding places:

db.locations.find({latlong:{$near:[40,70]}})

With regular expressions:

db.locations.find({name: /^typeaheadstring/})

By tag:

db.locations.find({tags: “business”})
Inserting and updating locations

Initial data load:
`db.locations.insert(place1)`

Using update to Add tips:
`db.locations.update({name:"10gen HQ"},
    {$push :{tips:
        {user:"nosh", time:6/26/2010,
          tip:"stop by for office hours on
          Wednesdays from 4-6"}}})`
Requirements

- Locations
  - Need to store locations (Offices, Restaurants etc)
    - Want to be able to store name, address and tags
    - Maybe User Generated Content, i.e. tips / small notes?
  - Want to be able to find other locations nearby

- Checkins
  - User should be able to ‘check in’ to a location
  - Want to be able to generate statistics
Users

user1 = {
    name: “nosh”
    email: “nosh@10gen.com”,
    ...
    ...
    checkins: [{
        location: “10gen HQ”,
        ts: 9/20/2010 10:12:00,
        ...
    },
    ...
}
db.users.find({'checkins.location': "10gen HQ"})

db.checkins.find({'checkins.location': "10gen HQ"})
  .sort({ts:-1}).limit(10)

db.checkins.find({'checkins.location': "10gen HQ",
  ts: {$gt: midnight}}).count()
user1 = {
    name: "nosh",
    email: "nosh@10gen.com",
    checkins: [4b97e62bf1d8c7152c9ccb74, 5a20e62bf1d8c736ab]
}

checkins [] = ObjectId reference to locations collection
Check-in = 2 ops
  read location to obtain location id
  Update ($push) location id to user object

Queries: find all locations where a user checked in:
  checkin_array = db.users.find( .. ,
    {checkins:true}).checkins
  
  db.location.find( { _id: { $in: checkin_array } } )
Unsharded Deployment

- Configure as a replica set for automated failover
- Async replication between nodes
- Add more secondaries to scale reads
Sharded Deployment

- Autosharding distributes data among two or more replica sets
- Mongo Config Server(s) handles distribution & balancing
- Transparent to applications
Use Cases

- RDBMS replacement for high-traffic web applications
- Content Management-type applications
- Real-time analytics
- High-speed data logging

Web 2.0, Media, SaaS, Gaming, Finance, Telecom, Healthcare
Cassandra

• Introduction
• Data Model
• System Architecture
• CAP Theorem
• API
What is Cassandra?

- NoSQL column family implementation (more about it later)
- Highly scalable, available and no Single Point of Failure.
- Very high write throughput and good read throughput. It is pretty fast.
- SQL like query language (from 0.8) and support search through secondary indexes (well no JOINs, Group By etc...).
- Tunable consistency and support replication
- Flexible Schema
Data Model

keyspace

settings (eg, partitioner )

column family

settings (eg, comparator, type [Std])

column

name
value
clock
keyspace

• ~ = database
• typically one per application
• some settings are configurable only per keyspace
column family

• group records of similar kind
• not same kind, because CFs are sparse tables
• ex:
  ◦ User
  ◦ Address
  ◦ Tweet
  ◦ PointOfInterest
  ◦ HotelRoom
think of cassandra as row-oriented

• each row is uniquely identifiable by key
• rows group columns and super columns
column family

Key1
23

Key4
56

user=eben

nickname=The Situation

user=alison

icon=

n=42

The Situation
like notation

User {
  123 : { email: alison@foo.com, icon: },
  456 : { email: eben@bar.com, location: The Danger Zone}
}

a column has 3 parts

1. name
   - byte[]
   - determines sort order
   - used in queries
   - indexed

2. value
   - byte[]
   - you don’t query on column values

3. timestamp
   - long (clock)
   - last write wins conflict resolution
super columns group columns under a common name
super column family

**Central Park**
- Address: 10017
- Description: Fun to walk in.

**Empire State Bldg**
- Address: 212.555.11212
- Description: Great view from 102nd floor!

**Phoenix Zoo**
- Address: 85255
- Description: Fun to walk in.
System Architecture

- **Partitioning**
  How data is partitioned across nodes

- **Replication**
  How data is duplicated across nodes

- **Cluster Membership**
  How nodes are added, deleted to the cluster
Partitioning

• Nodes are *logically* structured in Ring Topology.
• Hashed value of key associated with data partition is used to assign it to a node in the ring.
• Hashing rounds off after certain value to support ring structure.
• Lightly loaded nodes moves position to alleviate highly loaded nodes.
Replication

• Each data item is replicated at N (replication factor) nodes.

• Different Replication Policies
  ◦ Rack Unaware – replicate data at N-1 successive nodes after its coordinator
  ◦ Rack Aware – uses ‘Zookeeper’ to choose a leader which tells nodes the range they are replicas for
  ◦ Datacenter Aware – similar to Rack Aware but leader is chosen at Datacenter level instead of Rack level.
cap theorem

- **Consistency**
  - all clients have same view of data

- **Availability**
  - writeable in the face of node failure

- **Partition tolerance**
  - processing can continue in the face of network failure (crashed router, broken network)
Who’s Using Cassandra?
Cluster Membership and Failure Detection

• Gossip protocol is used for cluster membership.
• Super lightweight with mathematically provable properties.
• State disseminated in $O(\log N)$ rounds where $N$ is the number of nodes in the cluster.
• Every $T$ seconds each member increments its heartbeat counter and selects one other member to send its list to.
• A member merges the list with its own list.
Gossip
Gossip

Round 2: 4
Gossip

Round 3: 7
Cassandra: How does it work?

- Nodes are arranged in a circle according to a key space (P2P network and uses consistent hashing).
- Each node owns the next clockwise address space.
- If replicated, each node owns next two clockwise address spaces.
- Any node can accept any request and route it to the correct node.
Cassandra: How does it work? (Contd.)

• Writes are written to enough nodes, and Cassandra repairs data while reading. (As you would guess, that is how writes are fast.)
• Data is updated in the memory, and it keeps an append only commit log to recover from failures. (This avoid rotational latency at the disk). Can do about 80-360MB/sec per node.
• When ever a read happens, Cassandra will sync all the nodes having replicas (read repair).
**read api**

**get() : Column**
- get the Col or SC at given ColPath
  
  ```java
  COSC cosc = client.get(key, path, CL);
  ```

**get_slice() : List<ColumnOrSuperColumn>**
- get Cols in one row, specified by SlicePredicate:
  
  ```java
  List<ColumnOrSuperColumn> results =
  client.get_slice(key, parent, predicate, CL);
  ```

**multiget_slice() : Map<key, List<CoSC>>**
- get slices for list of keys, based on SlicePredicate
  
  ```java
  Map<byte[],List<ColumnOrSuperColumn>> results =
  client.multiget_slice(rowKeys, parent, predicate, CL);
  ```

**get_range_slices() : List<KeySlice>**
- returns multiple Cols according to a range
  
  ```java
  parent, predicate, keyRange, CL);
  ```
write api

client.insert(userKeyBytes, parent,
new Column("band".getBytes(UTF8),
"Funkadelic".getBytes(), clock), CL);

batch_mutate
  ◦ void batch_mutate(
  ◦  map<byte[], map<String, List<Mutation>>> , CL)

remove
  ◦ void remove(byte[],
  ◦ ColumnPath column_path, Clock, CL)
Advantages

• Easy to scale
• No single point of failure
• High write-through
• Handles lots of data
• Durable
• No more SQL injection

Disadvantages

• No joins
• Index & sort keys only
• Not good for large blobs
• Rows must fit in memory
• Built on Thrift
Building applications with MongoDB
– An introduction

Roger Bodamer
roger@10gen.com
@rogerb

http://mongodb.org
http://10gen.com