CLOUD COMPUTING

Partitioning and Consistent Hashing

Dr. Prafullata Kiran Auradkar

Department of Computer Science and Engineering

Acknowledgements:

Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. H.L. Phalachandra** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

CLOUD COMPUTING

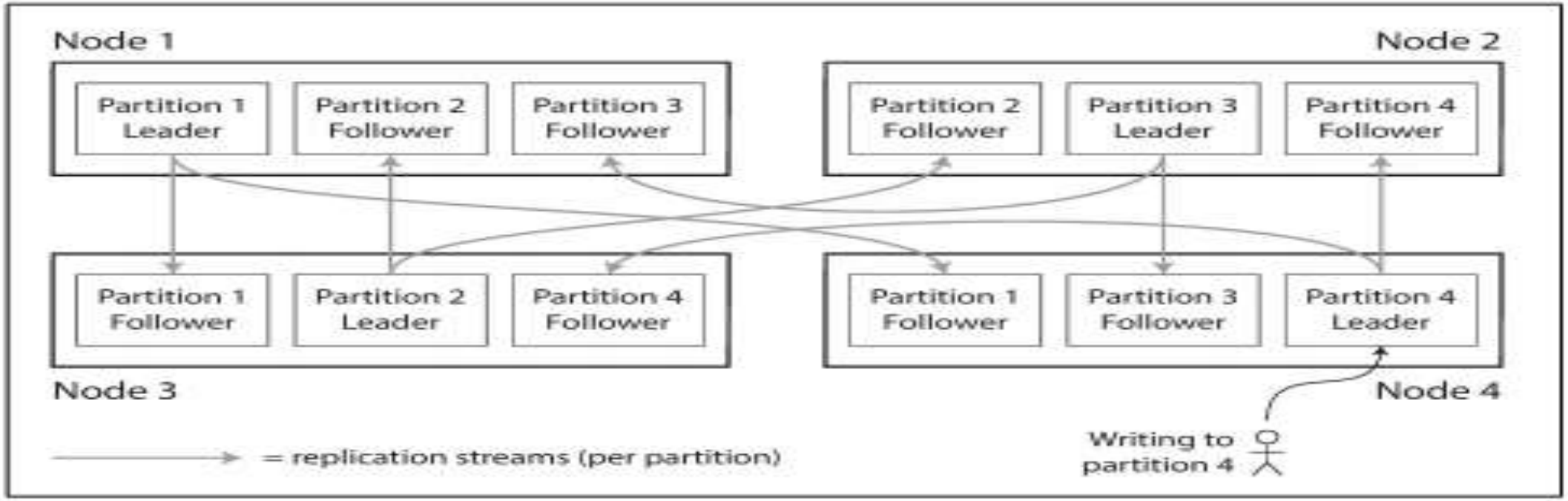
Partitioning

- Cloud Applications typically scale using elastic computing resources for addressing variations in the workload. This could lead to a bottleneck in the backend in-spite of scaling the hosts, if they continue to use the same data store.
- Partitioning is a way of intentionally breaking this large Volume of data into smaller partitions of data, where each of these partition can be placed on a single node or distributed across different nodes in a cluster, such that query or IO operations can be done across these multiple nodes supporting the performance and throughput expectations of applications.
- Partitions are defined in such a way that each piece of data (if its say a DB, then each record, row, or document) belongs to exactly one partition. There could be many operations which will touch partitions at the same time
- Each node can independently execute the queries or do IO and operate on these single (own) partition enabling scaling of throughput with the addition of more nodes.

  Eg. Partitioning a large piece of data to be written to disk into multiple partitions and distributing these partitions to multiple disks will lead to better total IO performance.

Partitioning

- Thus Large complex queries or large IO workload can potentially be parallelized across many nodes
- Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance. (We will discuss about replication in subsequent sessions)
- A node may store more than one partition



Goal of Partitioning and some terminologies

- Goal of partitioning is to spread the data and the query load evenly across nodes.
- If some partitions have more data than others, we call it **skewed**
- The presence of skew makes partitioning much less effective. In an extreme case, all the load could end up on one partition.
- A partition with disproportionately high load is called a **hot spot**

CLOUD COMPUTING

Different Approaches of Partitioning

There are 4 different approaches of partitioning

1. **Vertical Partitioning**
   - In this approach, data is partitioned vertically.
   - Its also called column partitioning where set of columns are stored on one data store and other to a different data store (could be on a different node) and data is distributed accordingly.
   - In this approach no two critical columns are stored together which improve the performance

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.com	3kb	3MB

Relational DB in a Block/File Storage

Object Storage

2. **Workload Driven Partitioning**
   - Data access patterns generated from the application is analysed and partitions are formed according to that. This improves the scalability of transactions in terms of throughput and response time

Different Approaches of Partitioning

3. Partitioning by Random Assignment

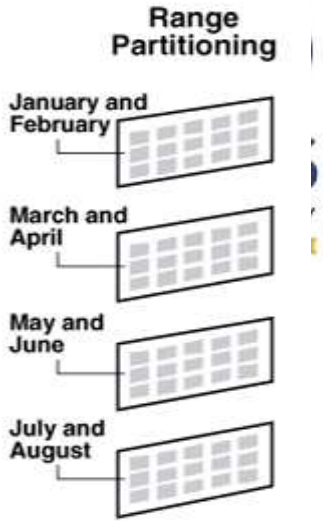
- Random assignment of of records to nodes would be the simplest approach for avoiding hot spots.
- This would distribute the data quite evenly across the nodes,
- The disadvantage of this would be, when trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.

4. Horizontal Partitioning

- This is a static approach of horizontally partitioning data to store it on different nodes
- Once its partitioned, this would not change.
- There are different techniques which are used for horizontal partitioning like
  1. ***Partitioning by Key Range***
  2. ***Partitioning using the Schema***
  3. ***Partitioning using Graph Partitioning***
  4. ***Partitioning using* Hashing**

CLOUD COMPUTING

4.1 Partitioning by Key -Range



- Range partitioning, is partitioning the cloud data by using range of keys by assigning a continuous range of keys to each partition (from some minimum to some maximum).
- The values of the keys should be adjacent but not overlapped
- Range of keys is decided on the basis of some conditions or operators.
- If we know the boundaries between the ranges, we can easily determine which partition contains a given key
- If we also know which partition is assigned to which node, then we can make the request directly to the appropriate node
- The ranges of keys are not necessarily evenly spaced, because data may not be evenly distributed Within each partition, we can keep keys in sorted order.
- **Example:**
  - Consider an application that stores data from a network of sensors, where the key is the timestamp of the measurement (year-month-day-hour-minute-second).
  - Range scans are very useful in the case where all the readings for a particular month need to be fetched.
- The disadvantage of key range partitioning is that certain access patterns can lead to hot spots
- If the key is a timestamp, all writes end up going to the same partition (the one for today), so that partition can be overloaded with writes while others are idle
- To avoid this problem, we can use some other attribute other than the timestamp as the first element of the key

4.2-4.3 : Schema Based and Graph Partitioning :

4.2 Schema Based Partitioning

- Schema Partitioning is basically designed to minimize the distributed transactions.
- In this database schema is partitioned in such a way that related rows are kept in the same partition instead of separating them in different partitions.

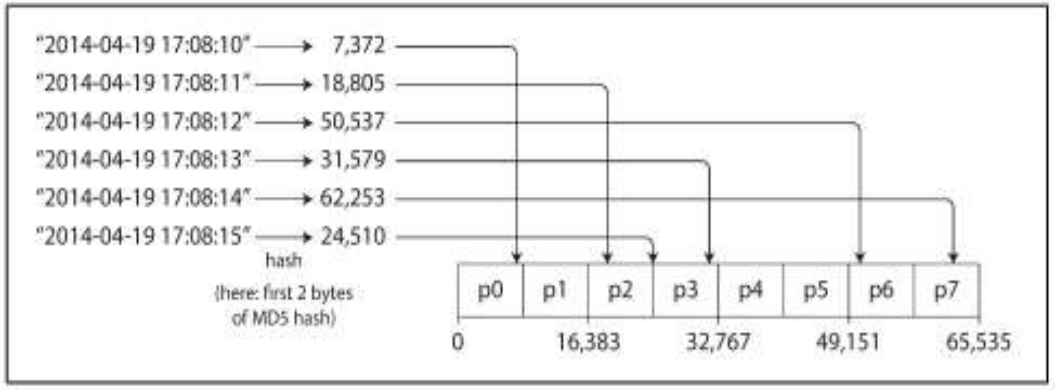
4.3 Graph Partitioning

- Graph partitioning is a workload- based static partition in which partitions are made by analysing the pattern of data access.
- Once partitioning is done, the workload is not observed for changes and there is no-repartitioning
- We could have a workload based dynamic partitioning strategy too .. Which we will discuss as part of Partition rebalancing

CLOUD COMPUTING

4.4 Partitioning by Hash of Key

- Hash partitioning maps data to partitions based on a hashing algorithm that is applied to the partitioning key identified.
- The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size and thus avoids the risk of skew and hot spots
- Hash partitioning is also an easy-to-use alternative to range-partitioning, especially when the data to be partitioned is not historical or has no obvious partitioning key
- Using a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition.



https://notes.shichao.io/dda/ch6/#:~:text=Document%2Dpartitioned%20indexes%20(local%20indexes,scatter%2Fgather%20across%20all%20partitions.

CLOUD COMPUTING

Different Approaches of Partitioning : Partitioning by Hash of Key (Cont.)

- By using the hash of the key for partitioning we lose the ability to do efficient range queries
- Some approaches to circumvent this would be to concatenated index (or a composite key) approach which can enable a one-to-many relationships.

  Example: On a social media site, one user may post many updates. If the primary key for updates is chosen to be (user_id, update_timestamp), then you can efficiently retrieve all updates made by a particular user within some time interval, sorted by timestamp.
- Different users may be stored on different partitions, but within each user, the updates are stored ordered by timestamp on a single partition.

Example :

```
CREATE PARTITION FUNCTION myRangePF1 (int)
   AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS1
   AS PARTITION myRangePF1
   TO (test1fg, test2fg, test3fg, test4fg);
GO
CREATE TABLE PartitionTable (col1 int, col2 char(10))
   ON myRangePS1 (col1);
GO
```

- Partition test1fg will contain tuples with col1 values <= 1
- Partition test2fg will contain tuples with col1 values > 1 and <= 100
- Partition test3fg will contain tuples with col1 values > 100 and <= 1000
- Partition test4fg will contain tuples with col1 values > 1000

4.4 Distributed Hashing

- The hash table discussed earlier may need to be split into several parts and stored in different servers for working around the memory limitations of a single system to create and keep these large hash tables
- In these environments the data partitions and their keys (the hash table) are distributed among many servers (thus the name of distributed hashing)
- The mechanism for distribution of the keys onto different servers could be a simple hash modulo of the number of servers in the environment. Detailing, for the partition key under consideration, compute the hash and then by using modulo of the number of servers, determine which server the key and the partition will need to be stored in or read from.
- These setups also typically consist of a pool of caching servers that host many key/value pairs and are used to provide fast access to data. If the query for a key for an partition information is not available on the cache, then the data is retrieved from the distributed hashing table as a cache miss.
  So if a client needs to retrieve the partition with key say “Name”, then it looks it up in the cache and gets location of the partition. If the cache entry for “Name” does not exist, then it does a hash and the mod of the number of servers, for getting the information on which server the partition exists and then retrieves the partition information, and also populates it back into the cache for future.

4.4 Distributed Hashing – Rehashing Problem

- This Distributed hashing scheme is simple, intuitive, and works fine .. but may have issues in say the following two scenarios
  - What if we add a few more servers into the pool for scaling
  - What if one of the servers fail
- Keys need to be redistributed to account for the missing server or to account for new servers in the pool
- This is true for any distribution scheme, but the problem with our simple modulo distribution is that when the number of servers changes, most hashes modulo N will change, so most keys will need to be moved to a different server. So, even if a single server is removed or added, all keys will likely need to be rehashed into a different server. Eg. Lets say Server 2 (of 0,1 and 2) failed. The new keys are below. Note that all key locations changed, not only the ones from server C representing 2 in the HASH mod 3.

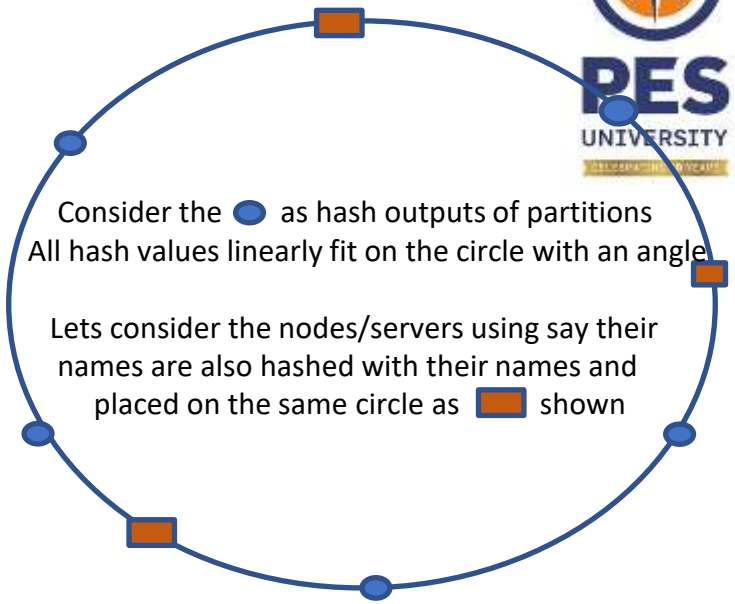
KEY	HASH	HASH mod 3
"john"	1633428562	2
"bill"	7594634739	0
"jane"	5000799124	1
"steve"	9787173343	0
"kate"	3421657995	2

KEY	HASH	HASH mod 2
"john"	1633428562	0
"bill"	7594634739	1
"jane"	5000799124	0
"steve"	9787173343	1
"kate"	3421657995	1

Typical use case discussed earlier with caching, this would mean that, all of a sudden, the keys won't be found because they won't yet be present at their new location.
So, most queries will result in misses, and the original data will likely need retrieving again from the source to be rehashed, thus placing a heavy load on the origin

CLOUD COMPUTING

Consistent Hashing



- Addressing this would need a distribution scheme that does not depend directly on the number of servers, so that, when adding or removing servers, the number of keys that need to be relocated is minimized.
- ***Consistent Hashing*** is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system.
- Imagine we mapped the hash output range onto the edge of a circle. That means that the minimum possible hash value, zero, would correspond to an angle of zero, the maximum possible value would correspond to an angle of 360 degrees, and all other hash values would linearly fit somewhere in between.
- So, we could take the keys, compute their hash, and place it on the circle's edge. If we now consider the servers too and using their names, compute a hash and place them also on the edge of the circle.
- Since we have the keys for both the objects and the servers on the same circle, we can define a simple rule to associate the former with the latter:
- Each object key will belong in the server whose key is closest, in a counterclockwise direction (or clockwise, depending on the conventions used). Thus to find out which server to ask for a given key, we need to locate the key on the circle and move in the ascending angle direction until we find a server.
- Now if a server is added and placed on the circle, then only those which are closest to it in clockwise direction will need to be moved and the rest of them will not need changes. If a server fails its similar only those behind will need to change.

Partitioning and Secondary Indexes

- We have seen till now that If records are only accessed via their primary key, we can determine the partition from that key, and use it to route read and write requests to the partition responsible for that key.
- A **secondary index** usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value

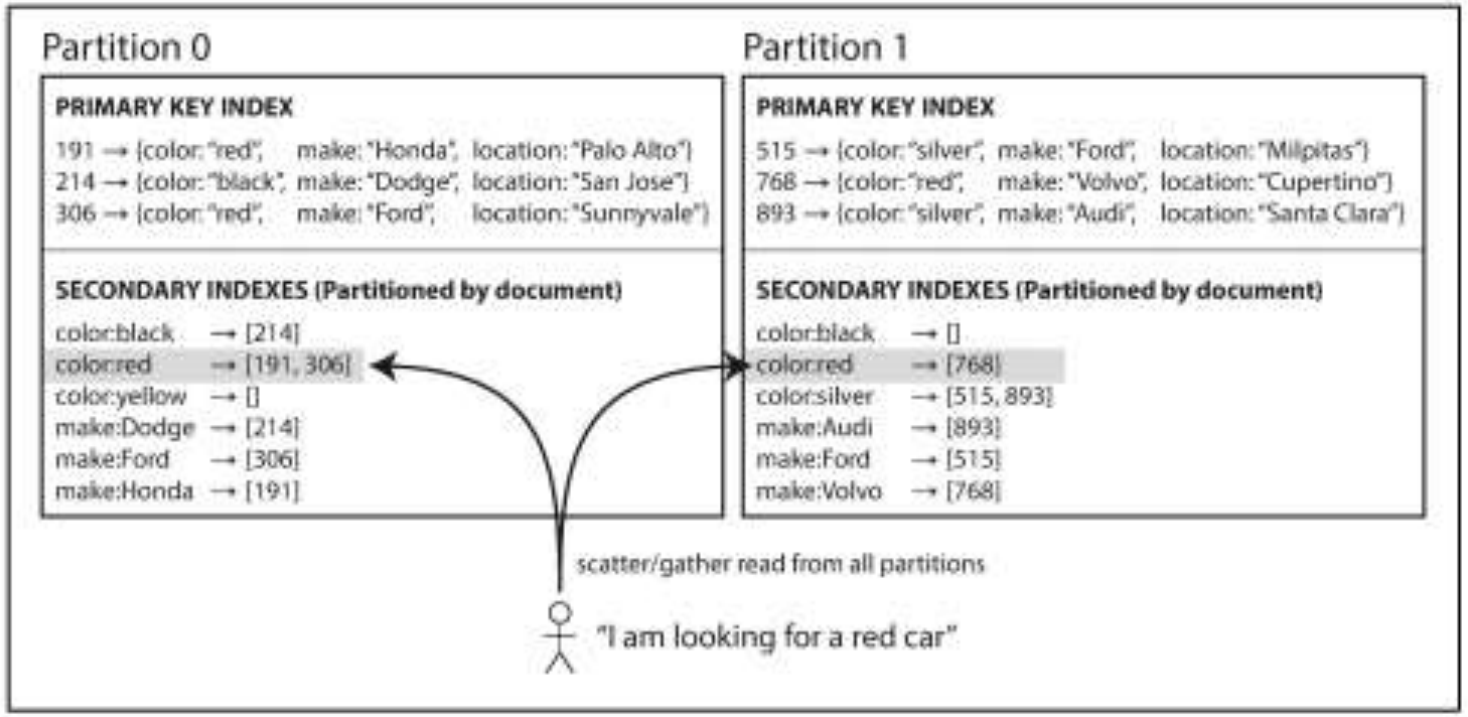
  E.g. find all actions by user 123, find all articles containing the word hogwash,
  find all cars whose color is red, and so on.
- Secondary indexes don't map neatly to partitions.
- Two main approaches to partitioning a database with secondary indexes:
  1. Document-based partitioning
  2. Term-based partitioning

Partitioning with secondary Indexes : Document based Partitioning

- In this indexing approach, each partition is completely separate
- Each partition maintains its own secondary indexes, covering only the documents in that partition. It doesn't care what data is stored in other partitions.
- Whenever you need to write to the database, you only need to deal with the partition that contains the document ID that you are writing. For that reason, a document-partitioned index is also known as a **local index**
- Reading from a document-partitioned index requires all partitions to be queried and combining all the results.
- This approach to querying a partitioned database is sometimes known as **scatter/ gather** and it can make read queries on secondary indexes quite expensive.

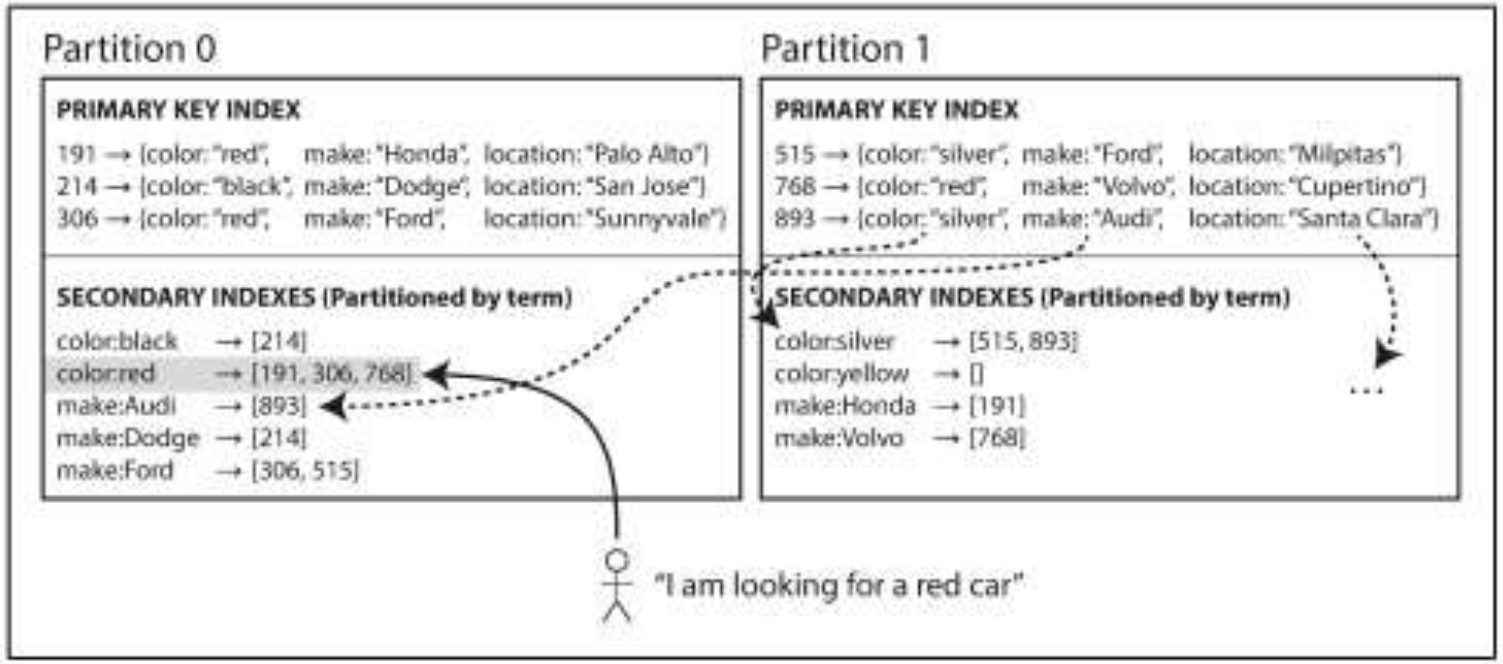
Partitioning with secondary Indexes : Document based Partitioning

- Even if the partitions are queried in parallel, scatter/gather is prone to tail latency amplification
- Most database vendors recommend that you structure your partitioning scheme so that secondary index queries can be served from a single partition, but that is not always possible, especially when we're using multiple secondary indexes in a single query



Partitioning with secondary Indexes : Term based Partitioning

- Construct a global index that covers data in all partitions
- A global index must also be partitioned, but it can be partitioned differently from the primary key index.
- This kind of index is called **term-partitioned**, because the term we're looking for determines the partition of the index.



Partitioning with secondary Indexes : Term based Partitioning

- We can partition the index by the term itself or using a hash of the term.
- Partitioning by the term itself can be useful for range scans, whereas partitioning on a hash of the term gives a more even distribution of load.
- The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants.
- The downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition on a different node).
- The global index needs to be up to date so that every document written to the database would immediately be reflected in the index. That would require a distributed transaction across all partitions affected by a write, which is not supported in all databases. Typically these updates are often asynchronous
- Therefore, if you read the index shortly after a write, the change you just made may not yet be reflected in the index

THANK YOU

Prafullata Kiran Auradkar

Department of Computer Science and Engineering

prafullatak@pes.edu