COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 10: Databases: NoSQL, Key-Value stores, Consistent Hashing

Lecturer: Emery Berger *Scribe(s): Molly McMahon, Janani Krishna*

10.1 Concurrency Control

Concurrency control is important with respect to database management, where concurrency in this context refers to a database with multiple users reading and writing in parallel.It is mostly related to semantics of the database.

- Database reads can happen in parallel with no issues
- Database writes need to be handled in a way that ensures consistency

10.2 Database Partitioning

Partitioning or "sharding" refers to distributing one database table among many machines. For example consider a giant table distributed to many machines.

	ID	Name	Qty	Uid
Machine A	23000	Brown ..	29	1
Machine B	25000	white ..	29	2
Machine C	88000	Andrew ..	70	3

The following are different strategies for partitioning:

10.2.1 Partitioning by Column

In this type of partitioning, each column of the table would be stored on a separate machine.

Cons of this approach:

- Extent of parallelization is limited to the number of columns
- A query requiring all columns requires reading from all machines

10.2.2 Partitioning by Row

In this type of partitioning, the table is partitioned horizontally; the rows are split into shards which are stored on various machines. This can be done through various hashing strategies. The possible strategies are:

10.2.2.1 Range Partitioning

To partition the rows, we can sort them by some field (e.g. ID), then define ranges of IDs and map each range to a machine. This approach only works well if the load is well balanced; for example, what happens if your IDs can range 1-1000, but most of your items have ID of 1? In this case, the majority of the rows will map to the same machine. We can attempt to solve this load imbalance by including more fields in our key.NoSQL allows the range partitioning.

10.2.2.2 Indexes

Indexes can be formed based on multiple fields in combination.For example an index can be combination of (ID,name),qty etc.This would be feasible only in case of small database. Potential issues with indexed based approach: Consider a table of n fields

No of indexes	fields involved
n	1
n^2	2
n^3	3

From the table above its clear that as the number of fields involved for creating an index increases the number of indexes increases.The problem will come in case of updates to the index.

10.2.2.3 Digest Hashing

In general hashing has one major issue ”Collision”, which is unlikely to happen in case of digest hashing.It uses the MD4 algorithm whose hash function is represented as 32 digit hexadecimal number.This particular hashing is most commonly used for integrity checks.

10.2.2.4 Hash Partitioning

To partition a table of entries, we take the entry IDs (or some other field) as keys, compute a hash function on the key, then mod the result by the number of machines to decide which machine to store the entry on.

hash(key) mod N, where N is the number of machines

The motivation of a hash function is to randomly distribute entries onto machines, so that the records will tend to be evenly distributed. This helps with load balancing.

Question from a student: Consider a query select * from db where qty=10; how does the query work??
Answer: This query is sent to all the machines.In short the entire db is read.

Potential issues with hash partitioning approach:

- Collisions - by the pigeonhole principle, some keys must hash to the same value
- Any ordering of the rows (for example, by ID) is destroyed
- Scalability - bigger hash functions are needed as the number of machines grows. In practice, this isn't really an issue, unless the range of keys is small.
- Fault tolerance - how to handle a broken machine? If we have replicas, they must all be consistent - ensuring consistency means that machines are doing less useful work. Additionally, in order to keep

new entries from hashing to the broken machine, we may need to change the hash function, which would require re-hashing of all the entries.

Locality Sensitive Hashing (LSH) Ensures that entries with similar keys will hash to similar values (not really used much in practice)

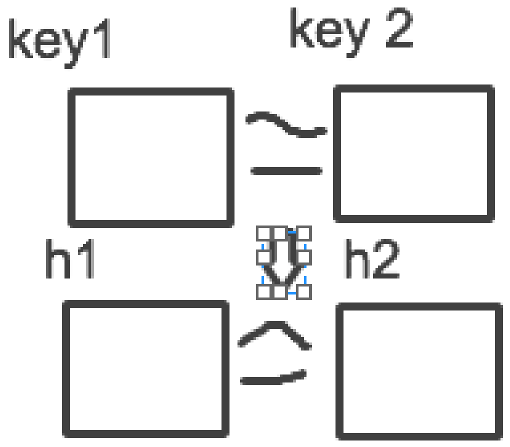

Figure 10.1: Locality hashing

10.2.2.5 Consistent Hashing

This type of hashing solves the fault tolerance problem. The hash values are arranged in a ring, and machine identifier replicas are randomly distributed around the ring. When a hash value is computed for an entry, the entry is mapped either to the machine at the value's location, or the machine nearest to the value's location. The random distribution of machine identifiers helps with load balancing, so that all entries don't hash to the same machine. When a machine fails, its identifiers are removed from the ring, and all the data that would have hashed to that machine is rehashed to the closest machine left on the ring. In this case, instead of having to compute a new hash function and rehash all the data, we only need to rehash the entries from the failed machine.

In fig 10.3 when machine B is killed ,some labels like(A,B,D) vitual nodes are added to the virtual space in order to achieve the uniform distribution of data from the machine B.

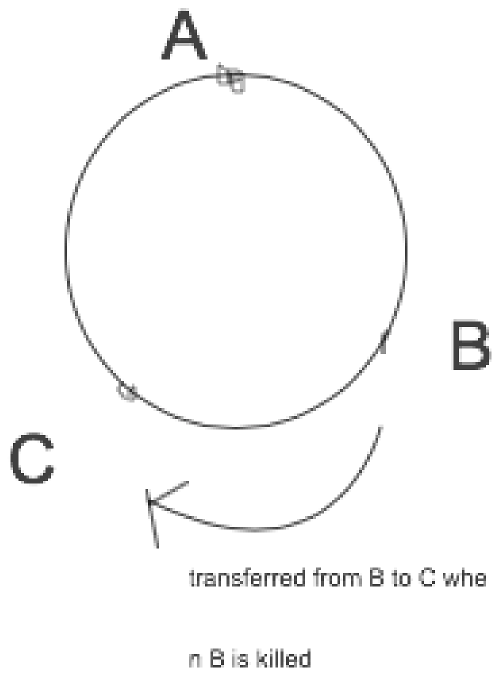

Figure 10.2: Consistent hashing

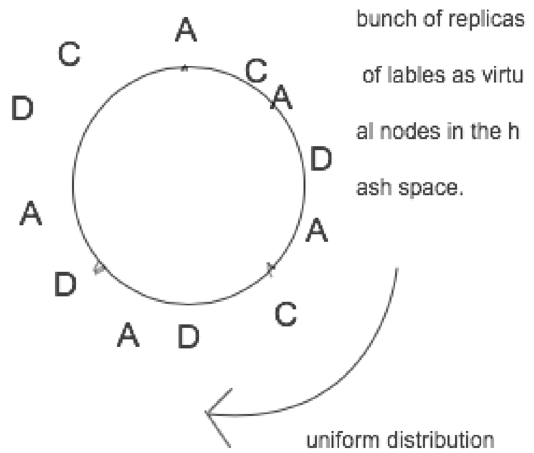

Figure 10.3: Consistent hashing with uniform distribution