Fault-Tolerant Query Execution over Distributed Bitmap Indices

Sam Burdick, *Amazon*
Jahrme Risner, *University of Puget Sound*
David Chiu, *University of Puget Sound*
Jason Sawin, *University of St. Thomas*

Good morning. I'm Sam Burdick, a software engineer at Amazon. Today I will be presenting a paper written by myself, Jahrme Risner and David Chiu at the University of Puget Sound, and Jason Sawin at the University of St. Thomas, entitled Fault-Tolerant Query Execution over Distributed Bitmap Indices.

Outline

- Motivation
- Background
  - Bitmap Index
- System Details
- Experimental Results
- Conclusion and Future Work

For this talk, I'll start with a motivating example problem.

Distributed Data Stores

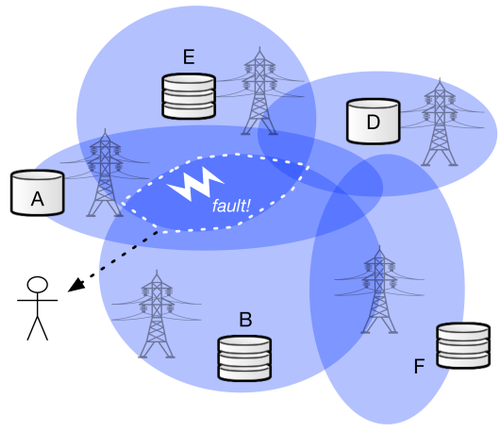
- Organizations are increasingly dependent on distributed cloud storage and filesystems

- Characteristics:
  - Fast data collection rates preclude a DBMS solution
    - Partial data is generally stored in raw (but relational) format
  - Geo-distributed nodes store partial data
  - Data may be replicated for availability and performance

Modern organizations are increasingly relying on storage and analytics of large data sets. Distributed cloud storage and distributed filesystems have been increasingly put to use to satisfy these needs. The result is that organizations' data is distributed across multiple physical nodes. These types of distributed systems have several important characteristics. First, fast data collection rates preclude using a parallel DBMS due to the amount of setup required to handle the data. The data will be stored in a raw but relational format, meaning that it is also queryable. In addition, the data tends to be produced by geographically distributed nodes, meaning that each node holds a subset of an organization's data. The data may also be replicated in multiple nodes to improve availability of the data to geographically distributed clients, as well as speeding up queries on the data made by said clients.

Distributed Data Stores (Cont.)

- Real-world example: Smart Grid project (Bonneville Power Administration)
  - 1000s of sensors deployed across NW power grid
  - 120 Hz measurement (write) rate
- Problems
  - Data may be unreachable in the event of node failure
  - Query execution is extremely costly and difficult to manage and orchestrate



An example of such a data store is the Smart Grid project (by the Bonneville Power Administration -- or BPA), which collects data from thousands of sensors on the Pacific Northwest power grid. Each sensor collects high-resolution power transmission data at a rate of 120 Hz, resulting in roughly 2 gigabytes of data stored per hour on local servers. The shaded blue areas in the picture here denote geographical regions represented by data stored in a nearby disk. However, problems abound when using large distributed systems such as these: If there were a power event and subsequent node unreachability, for example, the data contained within the nodes would also become unavailable. Even if the data was available, it is still prohibitively expensive to transfer all the data onto a centralized location for processing. Therefore, in order to satisfy queries in the presence of failures, users must in real time identify which storage nodes may contain the data that will satisfy their query, retrieve only the subset of data necessary to perform the analysis to minimize data transfer, and finally orchestrate data transfer, dependency resolution, and execution of processes to obtain a result.

What We Want

In order to solve this class of problems, we propose building a distributed system that meets several requirements. We want to support high-level SQL-like queries over our distributed data sets to satisfy, for example the BPA Smart Grid use case, using a fast data structure known as a bitmap index. Aside from storing data, we also would like to support distributed query execution. This entails generating optimal plans to speed up execution. Finally, we also want to be able to support query execution in the presence of node faults. We can do so with help from the consistent hashing algorithm and data replication, and want to allow for queries to be replanned at as little of a cost as possible.

What We Want

- We want to build a system that supports...
  - High-level (SQL like) queries over distributed data sets
    - Exploit a fast data structure, *Bitmap Index (next)*
  - Distributed query execution
    - Automatic query planning
  - Fault-tolerant query execution
    - Consistent hashing, data replication
    - Dynamic query replanning in the presence of faults

Outline

- Motivation
- Background
  - Bitmap Index
- System Details
- Experimental Results
- Conclusion and Future Work

Next I'll discuss bitmap indices, which is one of the core data structures of our system.

Background: Accelerating Queries

- Example high-level query
  - Find everyone who is under 66 years old and makes more than $100000/yr
  - Without indexing: Prohibitively slow using sequential file scan

Name	Age	Salary ($)	City	⋯
Julia	20	65,000	Tacoma	
Tim	76	25,000	Spokane	
Maria	42	130,000	Seattle	
⋮				

The system is built to support queries on standard relational data. Consider a US Census database with attributes salary age, etc, and each tuple (or row) represents a person. An arbitrary query such as “Return people with salaries of at least $100,000 and are under the age of 66” would require a scan of at least 326 million tuples **in the worst case**, which could take more than 10 minutes on a modern SSD. Clearly, this performance is unacceptable for real-time data analysis.

Bitmap Index

Name	Age	Salary ($)	City	⋯
Julia	20	65,000	Tacoma	
Tim	76	25,000	Spokane	
Maria	42	130,000	Seattle	
⋮				

We can get around our problem by indexing the data. **There are many different types of indices, but we chose bitmaps, because they can be easily partitioned, since there are no dependencies between different elements of the index. That makes them ideal candidates for data distribution.**
A bitmap index is a set of binary **sequences,** called *vectors,* **(make sure to point out vectors are vertical, not horizontal)** that represent truth values about records. For example, because Julia is under 21, and her row is the first one, in the first row of the bitmap index, her value in the "Age < 21" vector is 1. For the other folks, their value is 0, because they have ages in different ranges.

Bitmap Index

Name	Age	Salary ($)	City	⋯
Julia	20	65,000	Tacoma	
Tim	76	25,000	Spokane	
Maria	42	130,000	Seattle	
⋮				

Bitmap Index

Name	Age	Salary ($)	City	⋯
Julia	20	65,000	Tacoma	
Tim	76	25,000	Spokane	
Maria	42	130,000	Seattle	
⋮				

v0	v1	v2	v3	v4	
Age < 21	**21 ≤ Age < 66**	**66 ≤ Age**	**Salary ≤ 100,000**	**Salary > 100,000**	**⋯**
1	0	0	1	0	
0	0	1	1	0	
0	1	0	0	1	
...					

Query Revisited

- Queries on bitmap indices are performed with bitwise operators, exploiting fast hardware support

- Back to example:
  - Find everyone who is under 66 years old and makes more than $100,000/yr
  - Query is answered through performing *(v0 | v1) & v4 (= 001)*
    - *(Then chase down only the candidate records on disk, pruning rest)*

Let's revisit the problem from before using a bitmap index. We can use bitwise operators on the vectors to satisfy the query. In this example, v_0 OR v_1 represents persons who are less than 21, or at least 21 and under 66, which is everyone under 66. Also, v_4 represents people who make more than $100k per year.
So the complete query is $(v_0 \mid v_1) \& v_4$. The answer ends up being 001, which tells us to look on the third disk block to satisfy the query.
In general, we are given a superset of the tuples that should be returned from the query, but a significant number of the possible tuples are pruned away, **saving ourselves from excessive disk accesses**.
In order to execute the query, we just had to use bitwise operators on values in memory, which is much faster than having to sequentially scan the entire file on disk.

Query Revisited

- Queries on bitmap indices are performed with bitwise operators, exploiting fast hardware support

- Back to example:
  - Find everyone who is under 66 years old and makes more than $100,000/yr
  - Query is answered through performing *(v0 | v1) & v4 (= 001)*
    - *(Then chase down only the candidate records on disk, pruning rest)*

v0	v1	v2	v3	v4	
Age < 21	**21 ≤ Age < 66**	**66 ≤ Age**	**Salary ≤ 100,000**	**Salary > 100,000**	**⋯**
1	0	0	1	0	
0	0	1	1	0	
0	1	0	0	1	
...					

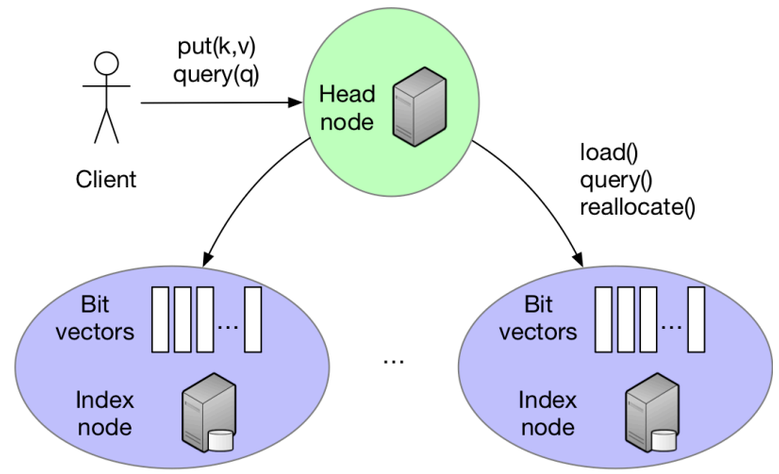
Outline

- Motivation
- Background
  - Bitmap Index
- System Details
- Experimental Results
- Conclusion and Future Work

Next I will discuss some of the design choices and algorithms used in our system.

System Details

- Master-slave architecture
  - Each index node (slave) stores bit-vectors for carrying out queries
  - Head node (master) provides client interface:
    - PUT(k,v)
    - QUERY(q)

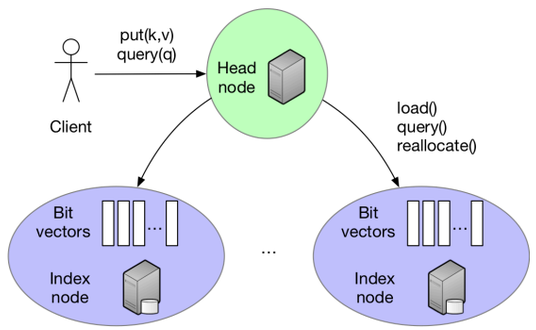


Our distributed indexing system was built using the master-slave paradigm. The system comprises one “master” head node and a collection of “slave” index nodes. Each index node stores a set of bit vectors and performs logical operations on them to satisfy queries.

The head node exposes an interface to the client, which we expect to have a set of bitmap vectors to add to our system. We have
- PUT(k, v) which adds a bit-vector with identifier k and value v into the system (or replaces vector k with value v if there is already a vector k in the system), and
- QUERY(q) which returns the result of a given query string q on the distributed index, the details of which I’ll explain later.

Head Node Responsibilities

- Data placement to $r \geq 2$ index nodes
- Adding new nodes to the system, handling redistribution
- Checking that all index nodes are still alive
- Constructing query plans and initializing queries
- We still need a mechanism for determining where vectors should be placed.



The head node has several important responsibilities.

- Data placement: upon receiving a PUT request, the head node distributes the vector to r, which is short for the “replication factor” distinct index nodes, where r is greater than or equal to 2. This way, when an index node fails, there is at least 1 backup copy to use.

- When a new index node is added into the network, it is communicated to the head node, which determines the existing and future vectors that should be moved to this node for fault tolerance, and manages the data transfer
- It also pings all index nodes periodically. if one node does not respond within a certain timeout interval, it has the index nodes redistribute vectors such that each vector is on r distinct nodes
- When given a query, it constructs a query plan that specifies which index nodes will help satisfy the query, and the order in which the nodes are to be accessed to resolve dependencies; the algorithms by which queries are carried out will be given later.

In the first, third, and fourth steps here, we still need an algorithm for determining where the vectors should be placed (or where they have been placed).

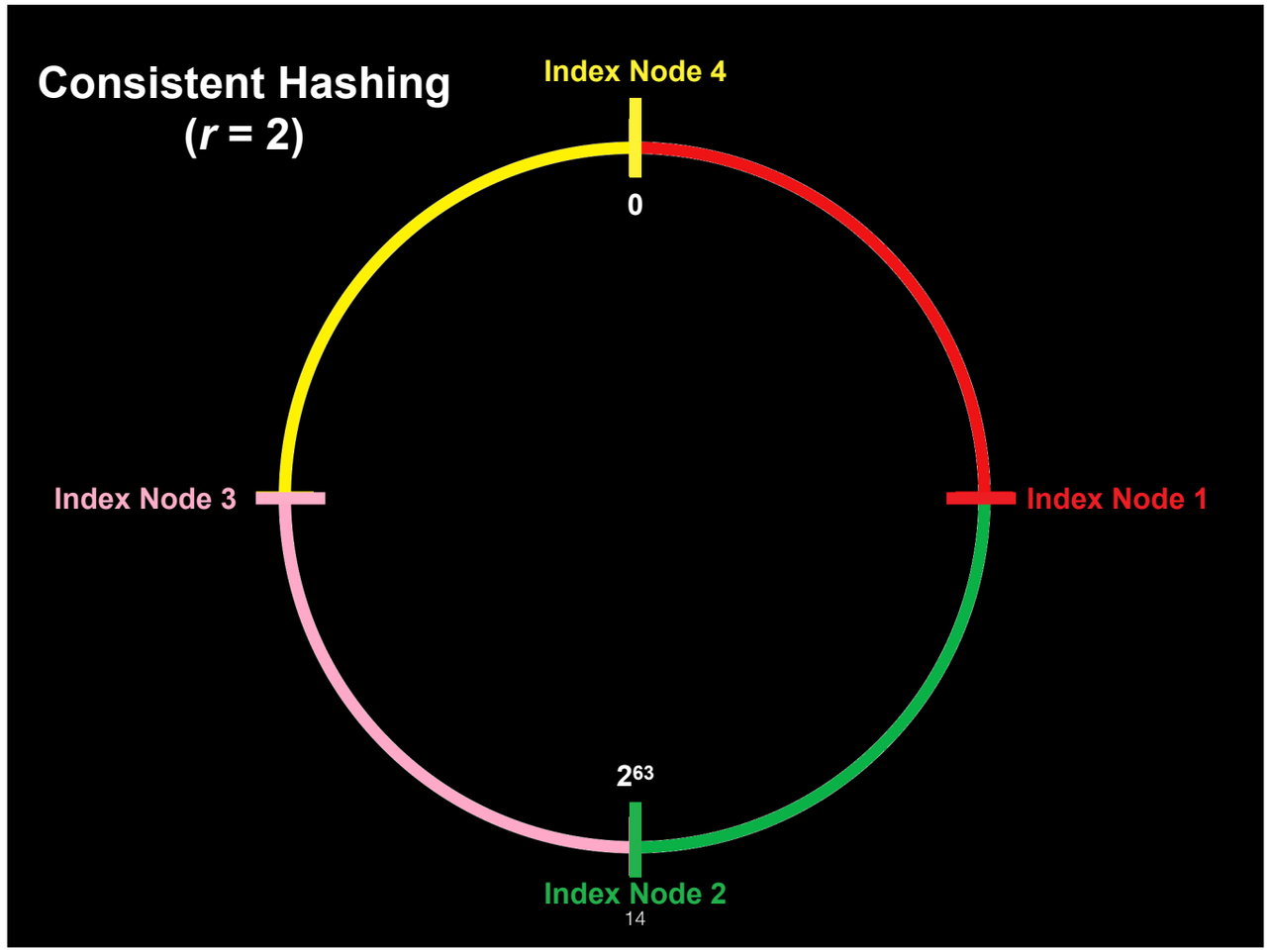
Consistent Hashing (Karger, *et al.*, 1997)

- Define the hash function:

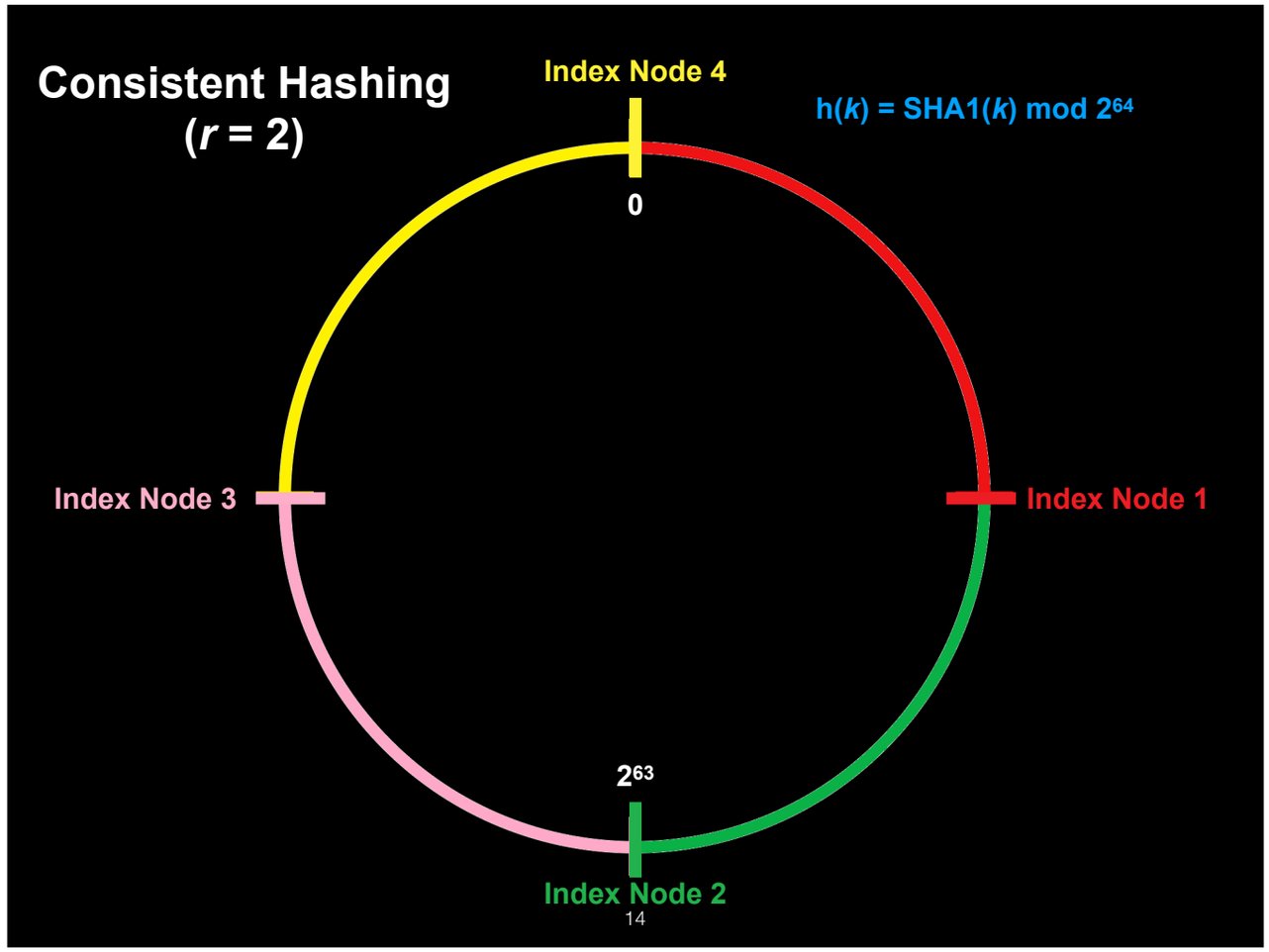
$$h(k) = \text{SHA1}(k) \bmod 2^{64}$$

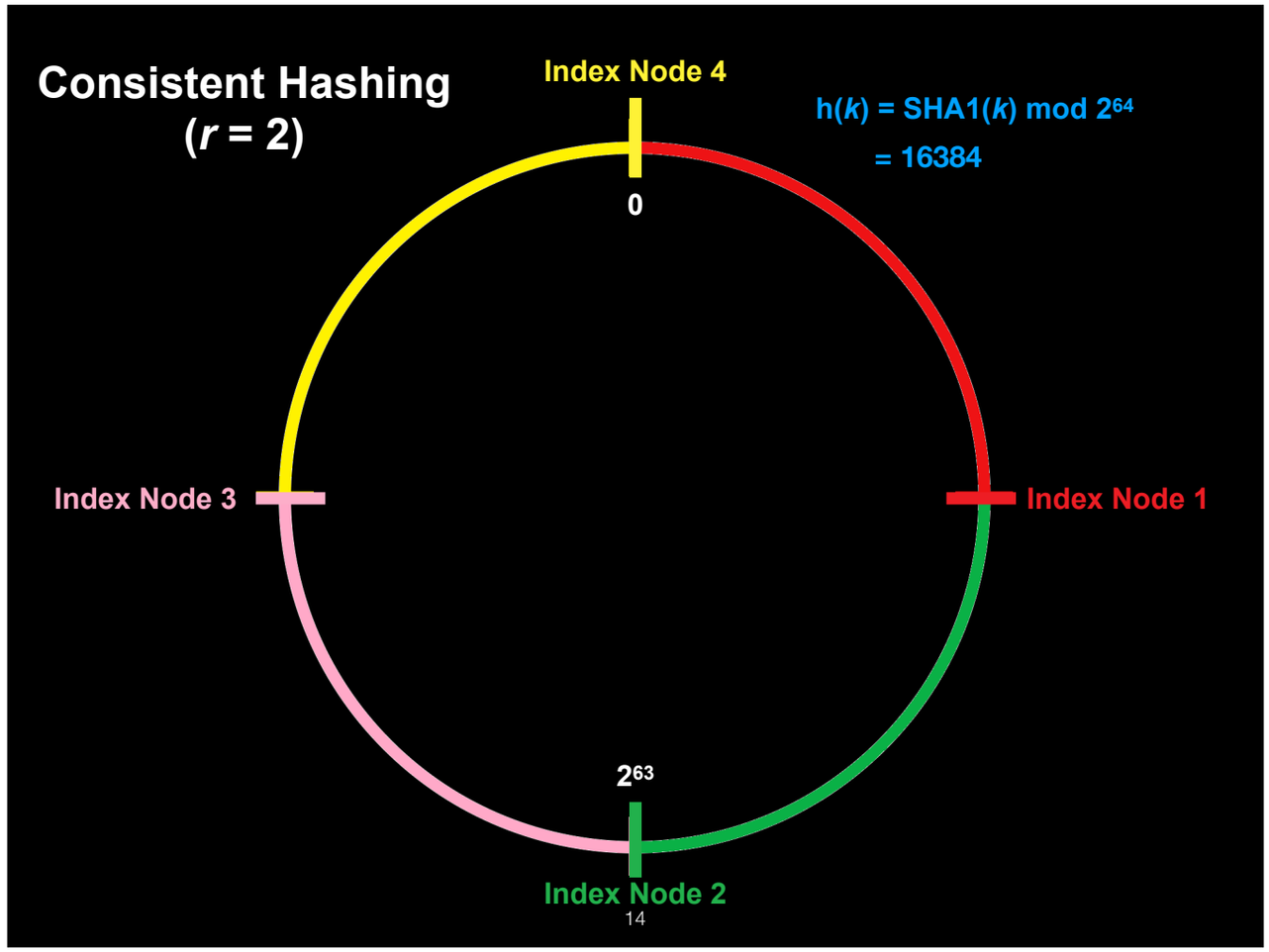
- Using $h(k)$
  - Assign each index node to a point on a circle.
  - To locate a bit-vector, compute $h(k)$, then walk clockwise on the circle until you get to an inode point. Keep walking until you've collected r nodes, and return them

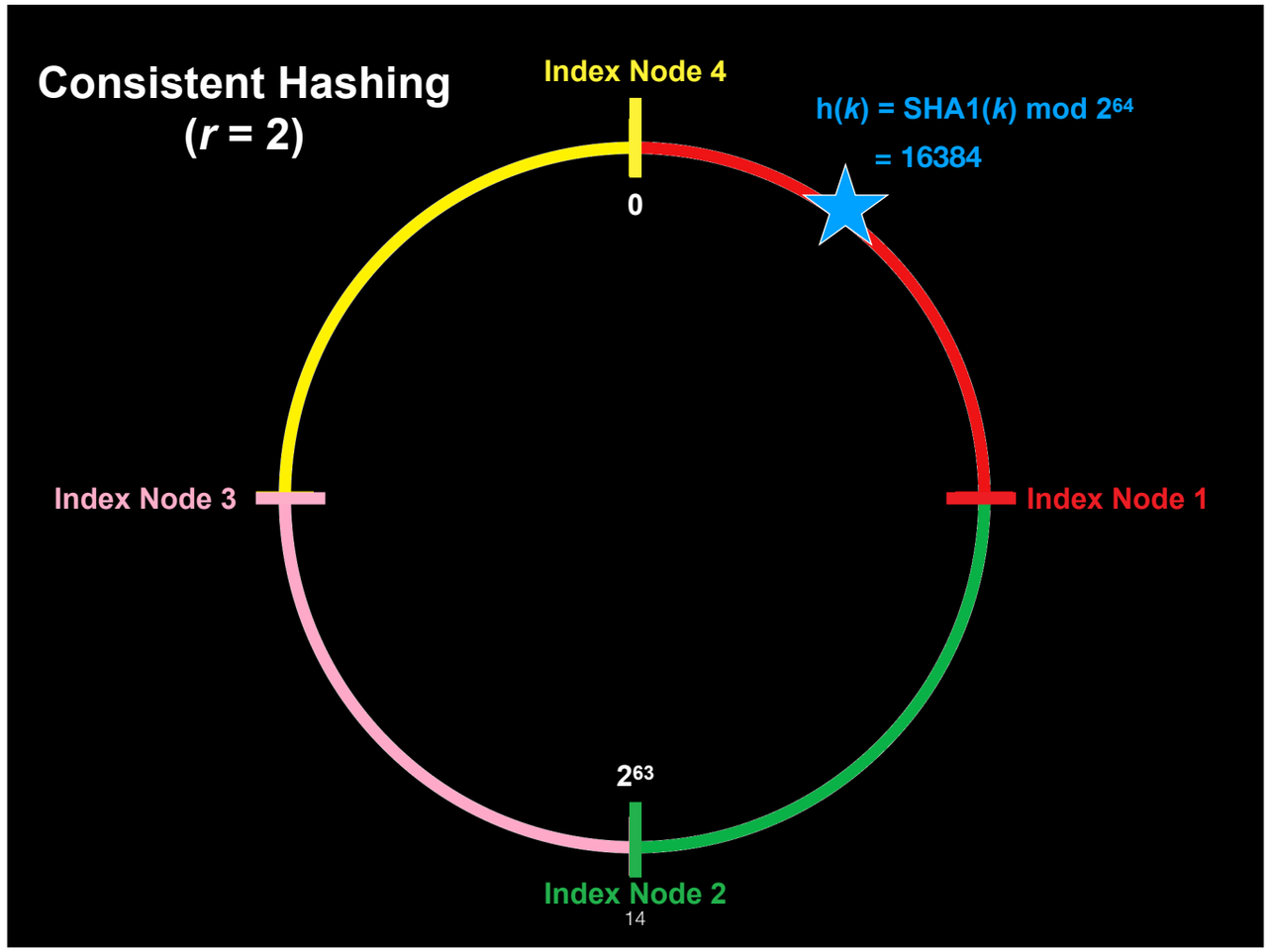
The method for locating vectors is consistent hashing, a technique originally developed by David Karger et al in the 1990s for web cacheing, and we define it as follows: First define the hash function h(k) = SHA1(k) mod 2^64. When index nodes are added to the system, the head node assigns it to a point on a ring, where each point on the ring corresponds to a value between 0 and 2^64 - 1. To determine which nodes a vector k should appear on, we compute h(k), find the corresponding point on the circle, and walk clockwise until we find an index node, and continue until we have found r nodes.

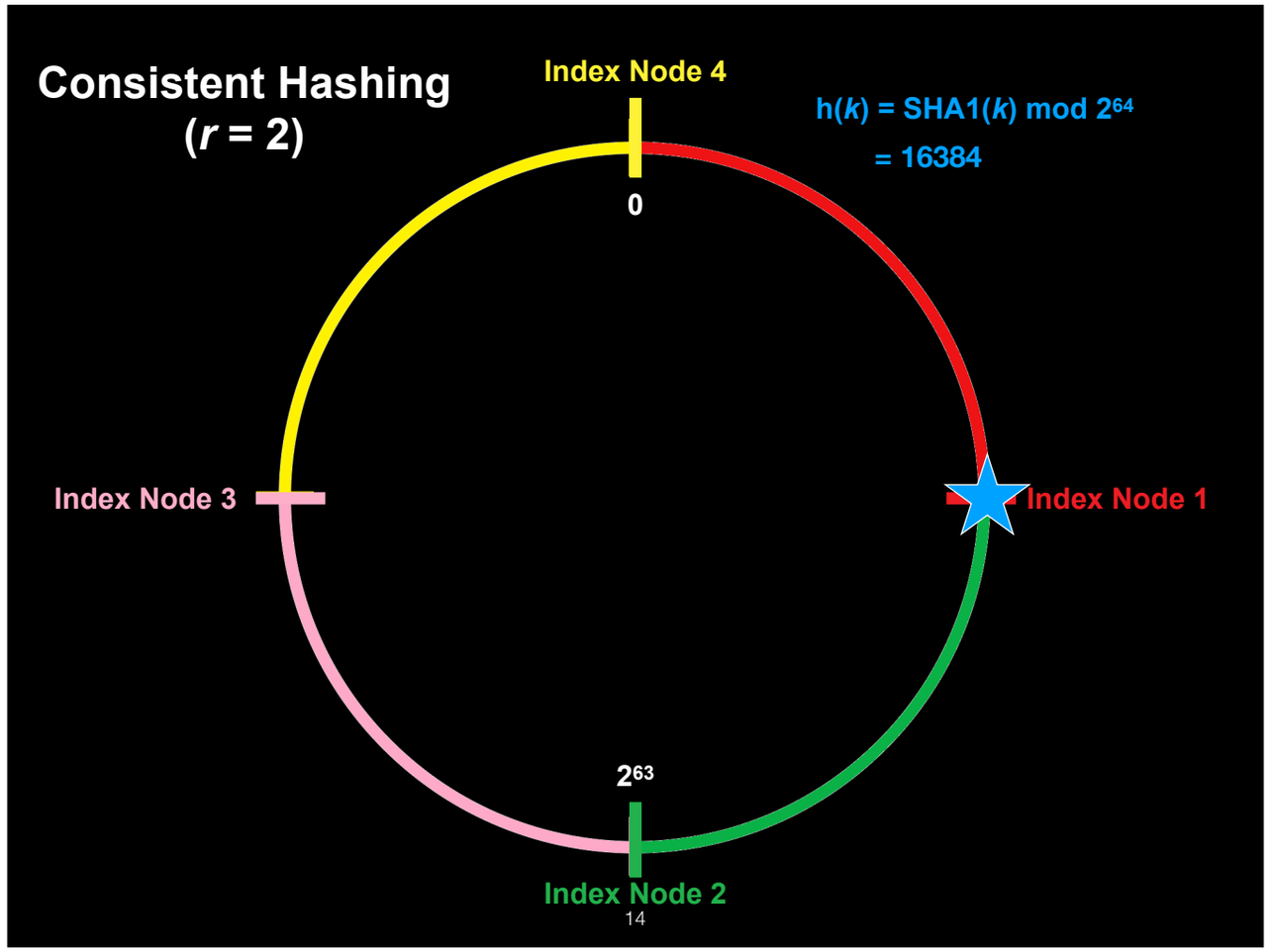


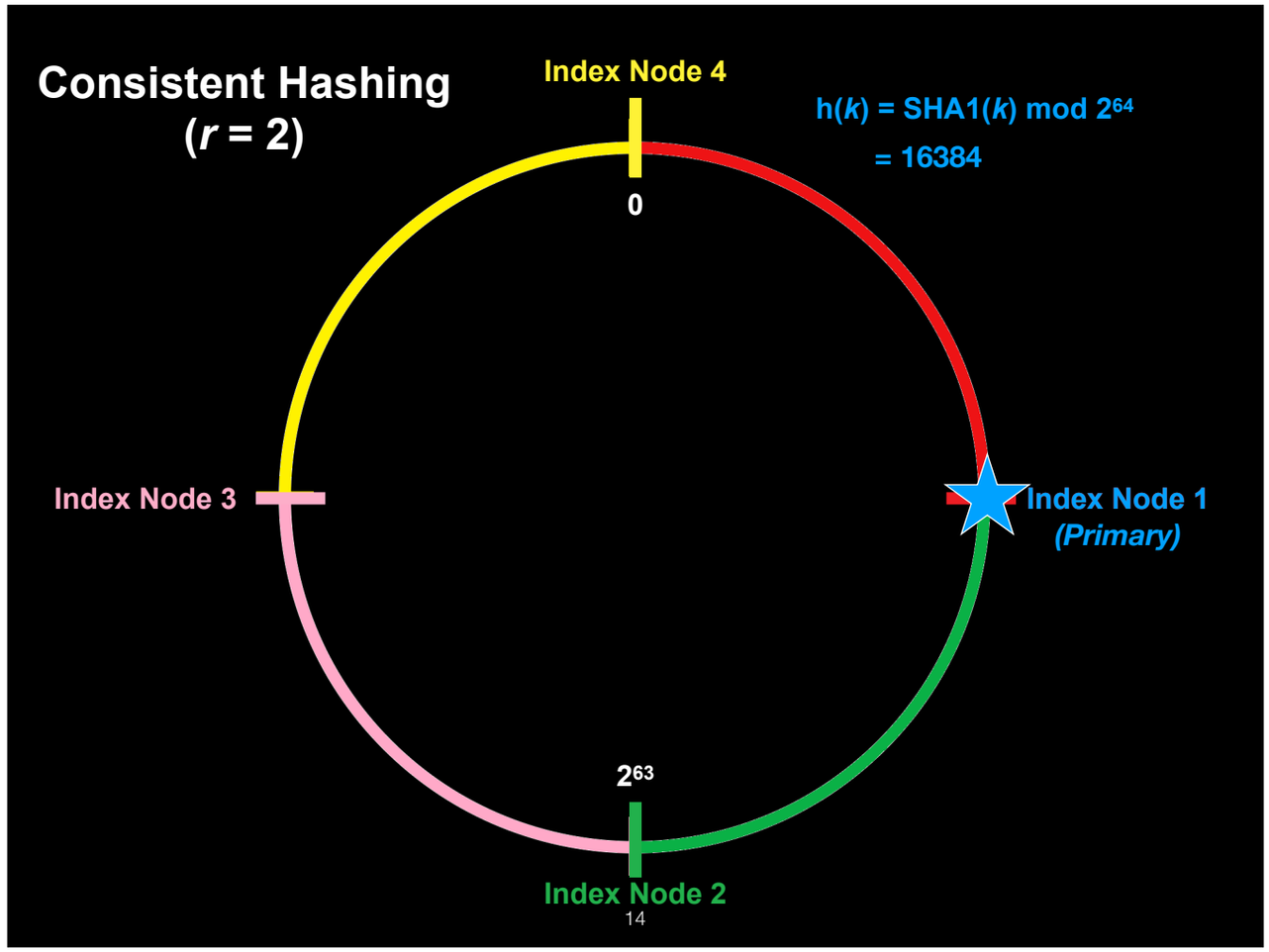
Here is an example of consistent hashing. We begin by hashing the key onto the ring, which places it at position 16384. We move clockwise to index node 1, which we designate as the “Primary” node, than to index 2, which is the first “Backup” node. Since r = 2 in this example, we have collected all the nodes we need for this vector, so we return index nodes 1 and 2.

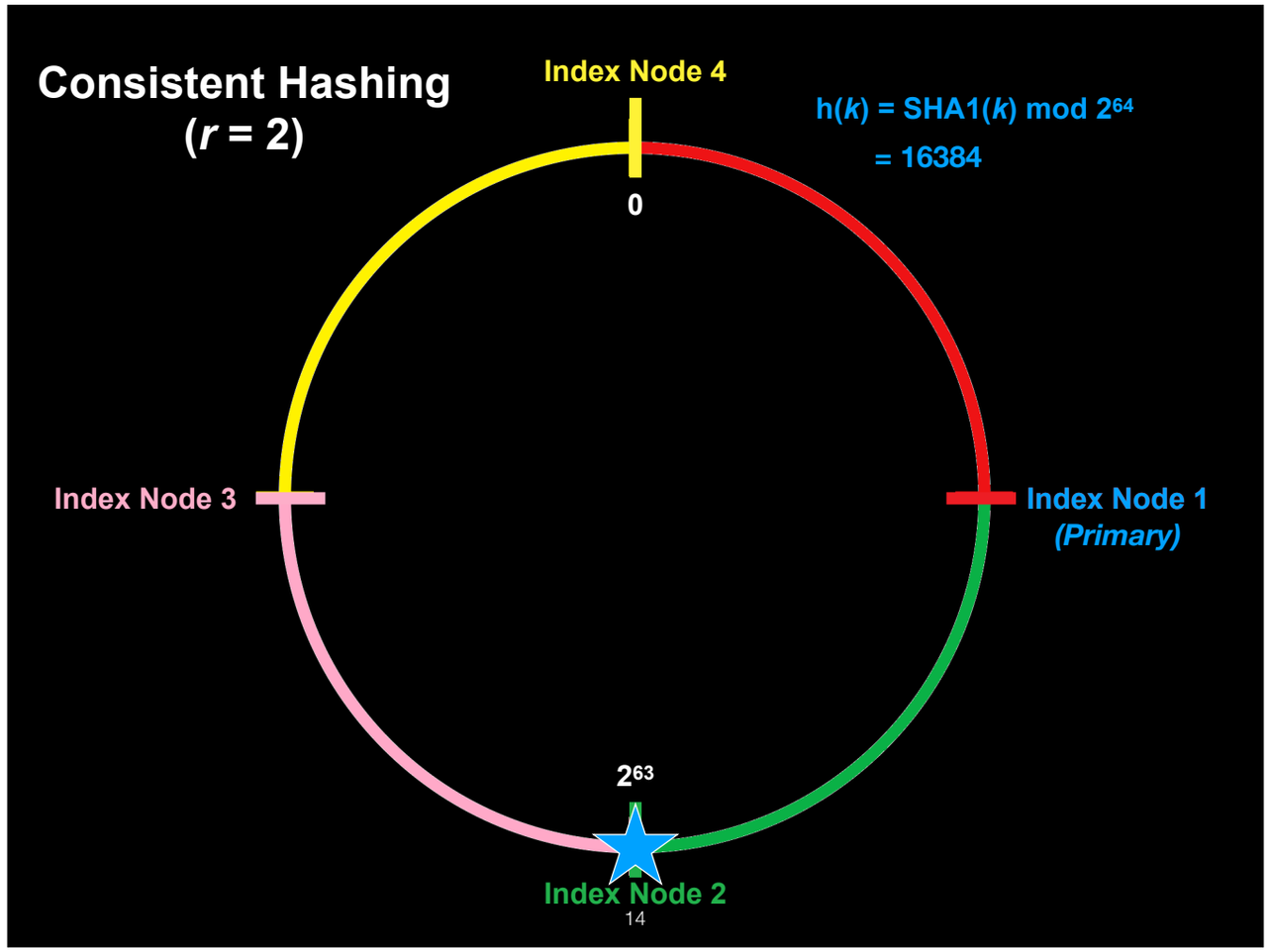

Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
0
Index Node 3
Index Node 1
2^63
Index Node 2

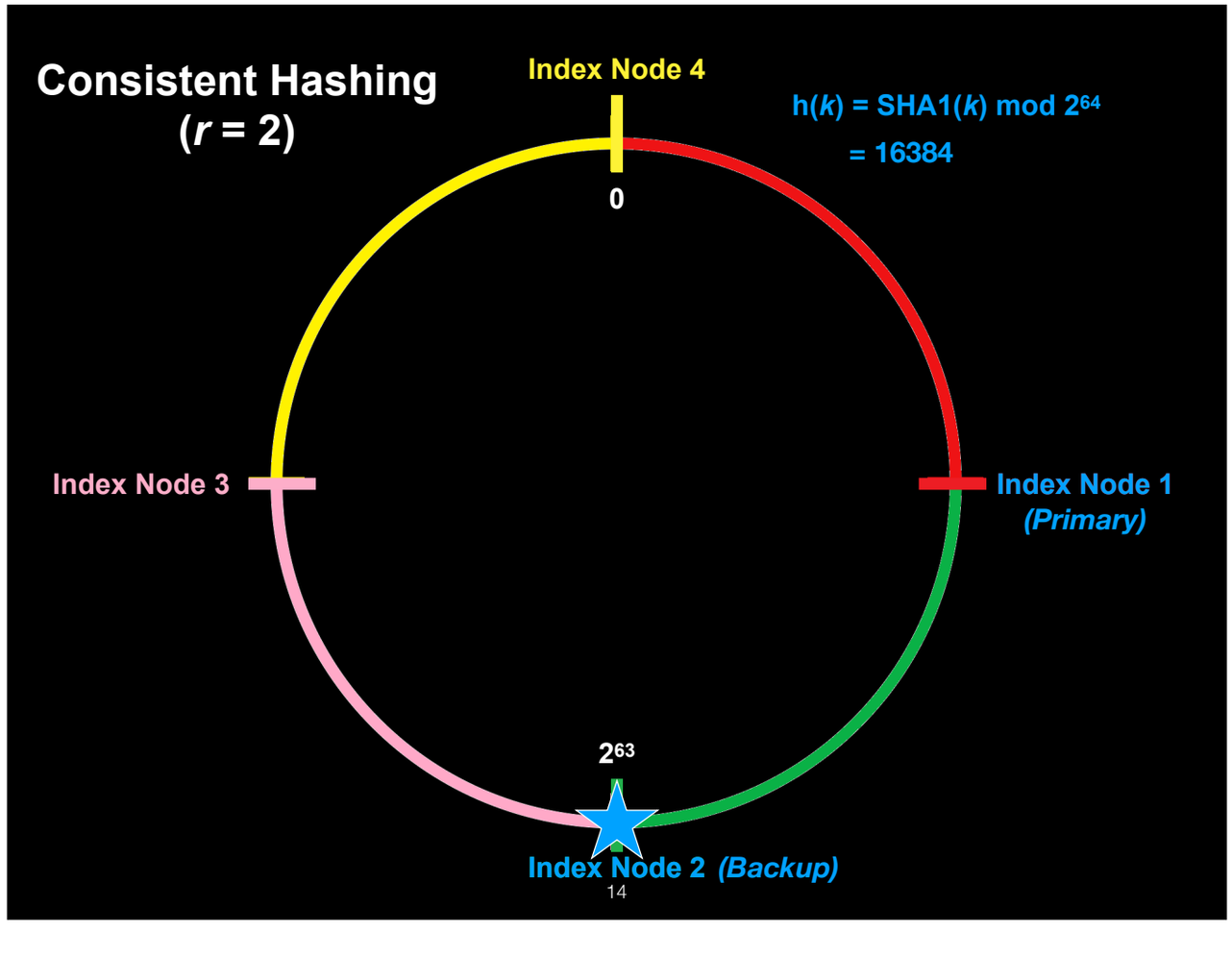

Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
2^63
Index Node 2

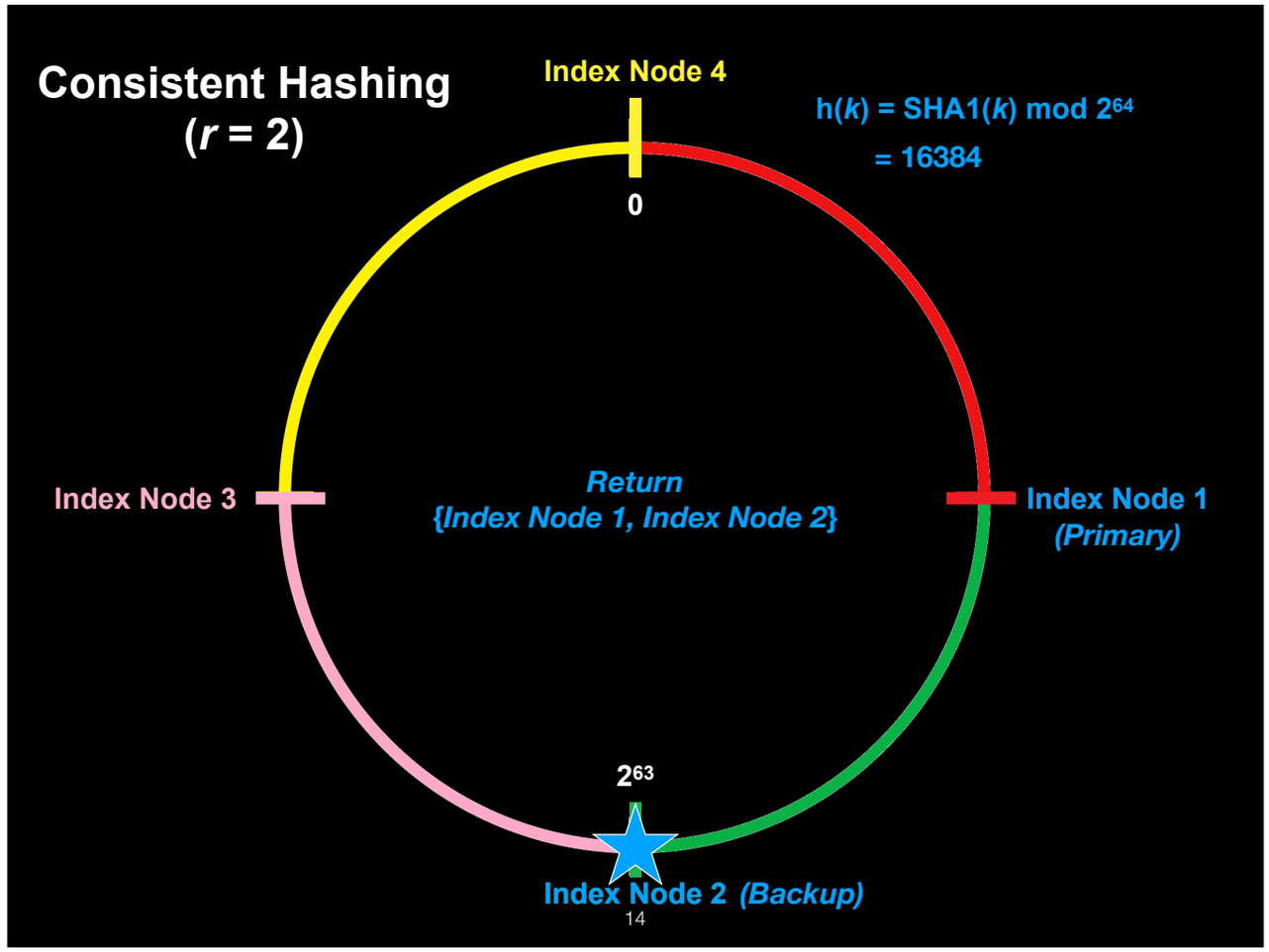

Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
2^63
Index Node 2


Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
2^63
Index Node 2


Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
(Primary)
2^63
Index Node 2


Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
(Primary)
2^63
Index Node 2


Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Index Node 1
(Primary)
2^63
Index Node 2 (Backup)


Consistent Hashing
(r = 2)
Index Node 4
h(k) = SHA1(k) mod 2^64
= 16384
0
Index Node 3
Return
{Index Node 1, Index Node 2}
Index Node 1
(Primary)
2^63
Index Node 2 (Backup)

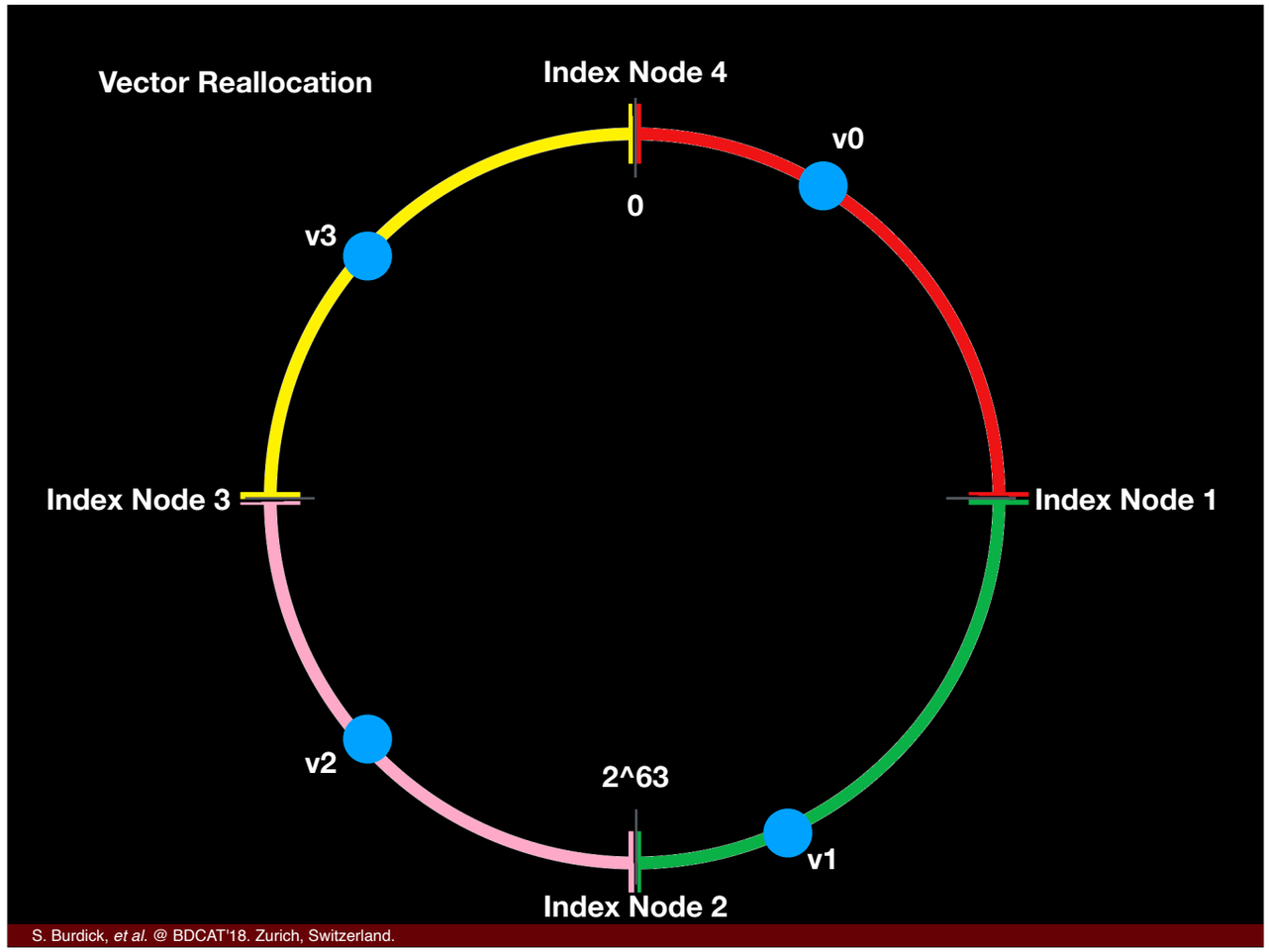
Consistent Hashing

- Implemented as a red-black tree for O(r log n) performance.

- When a node dies, some bit-vectors must be redistributed to ensure the replication factor of every bit-vector.
  - The hash ring manages the redistribution of the bit-vectors

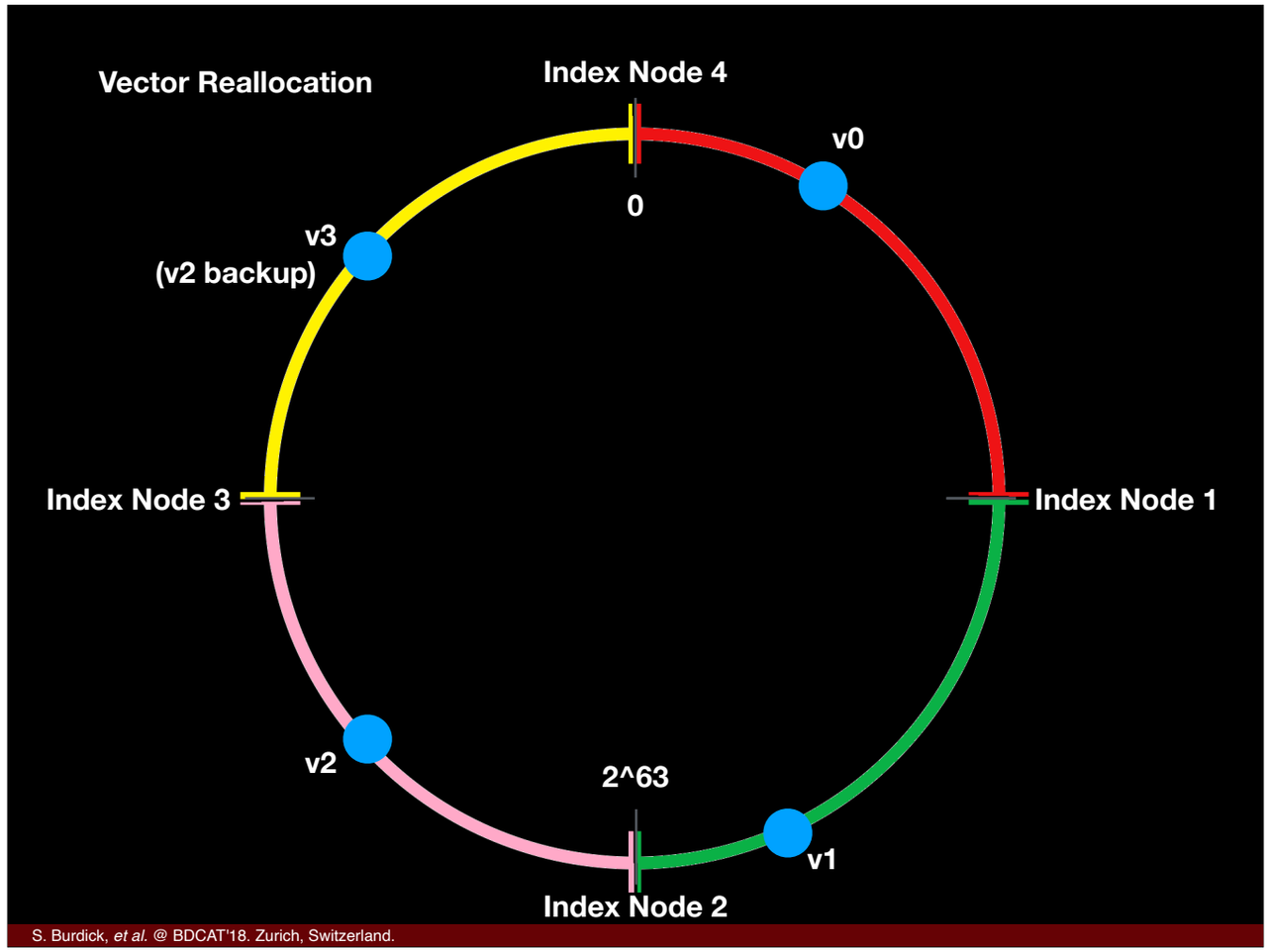
Consistent hashing is implemented via a red-black tree, where each node of the tree is an index node. Going around the circle requires finding the next successor node, since the tree represents a symbol table ordered on the hash of the index nodes' identifiers. Calling it requires r traversals of the tree, therefore the algorithm has O(r log n) runtime complexity.

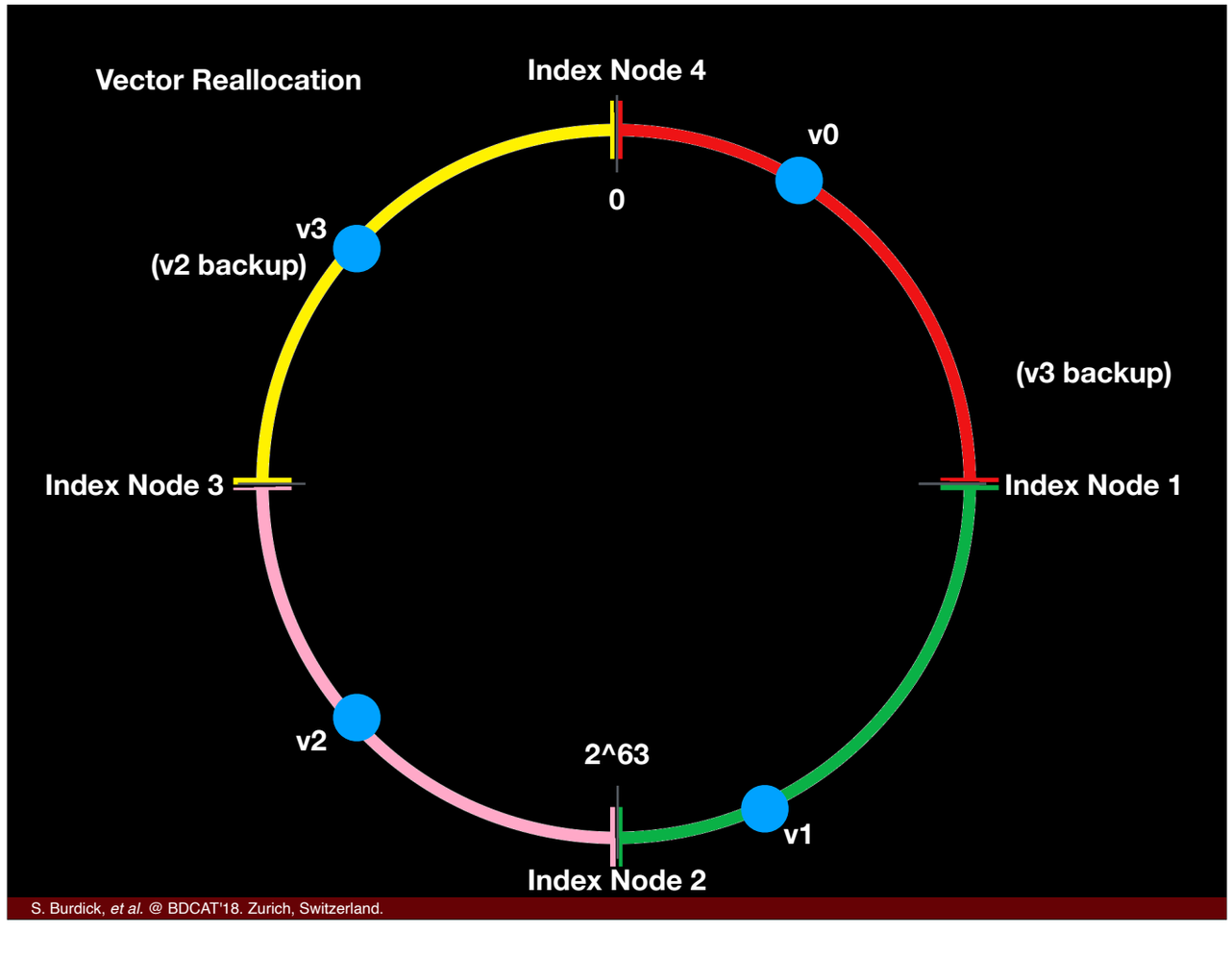
The main reason for employing consistent hashing is to support fault tolerance. When one of the nodes in the system fails, we must move some vectors around in order to guarantee the replication factor of each vector. The hash-ring structure helps us significantly in managing redistribution of bit vectors as well — the algorithm for reallocation also relies heavily on finding successor nodes. In the next slide, we'll illustrate how reallocation is done.

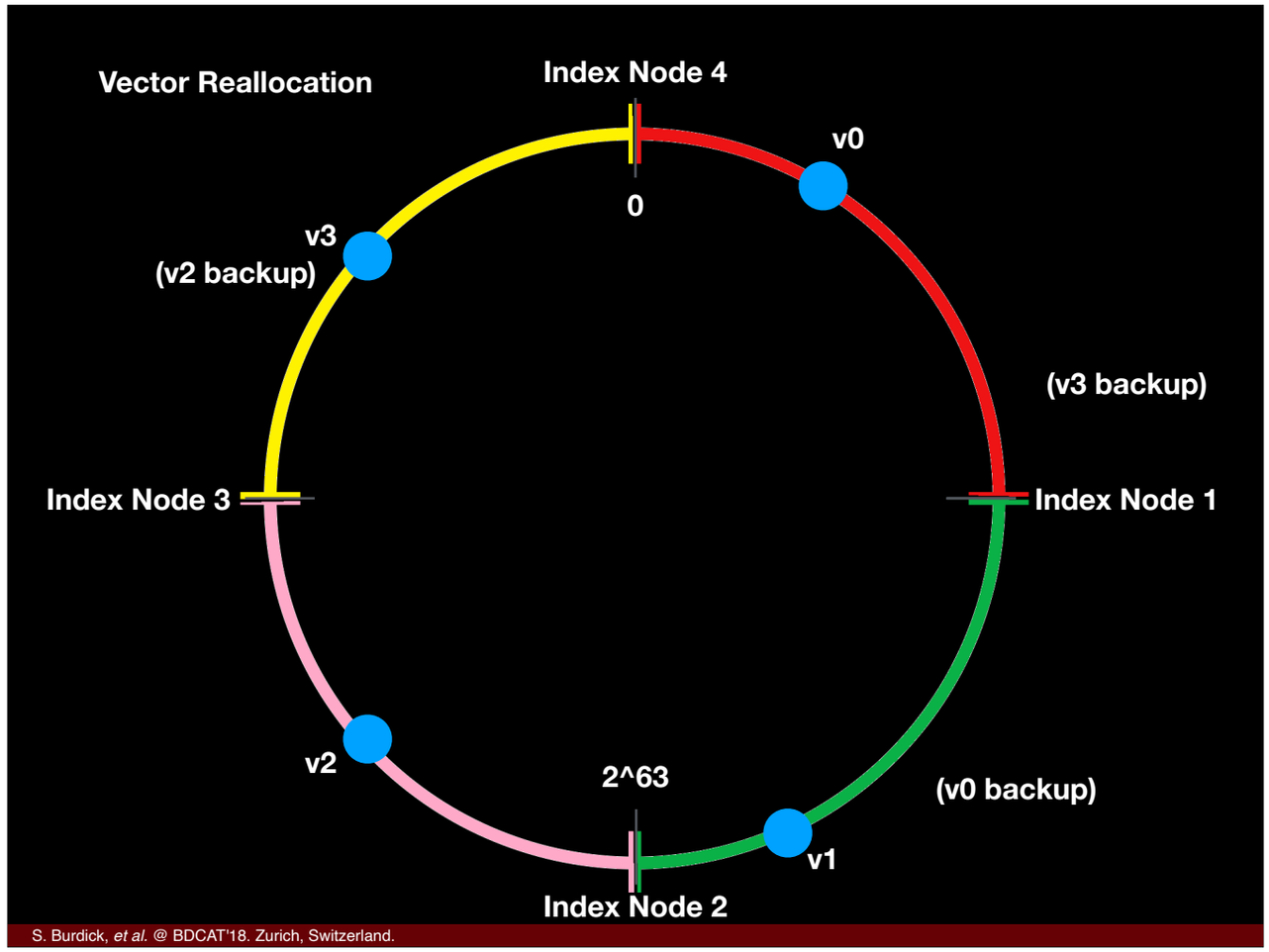


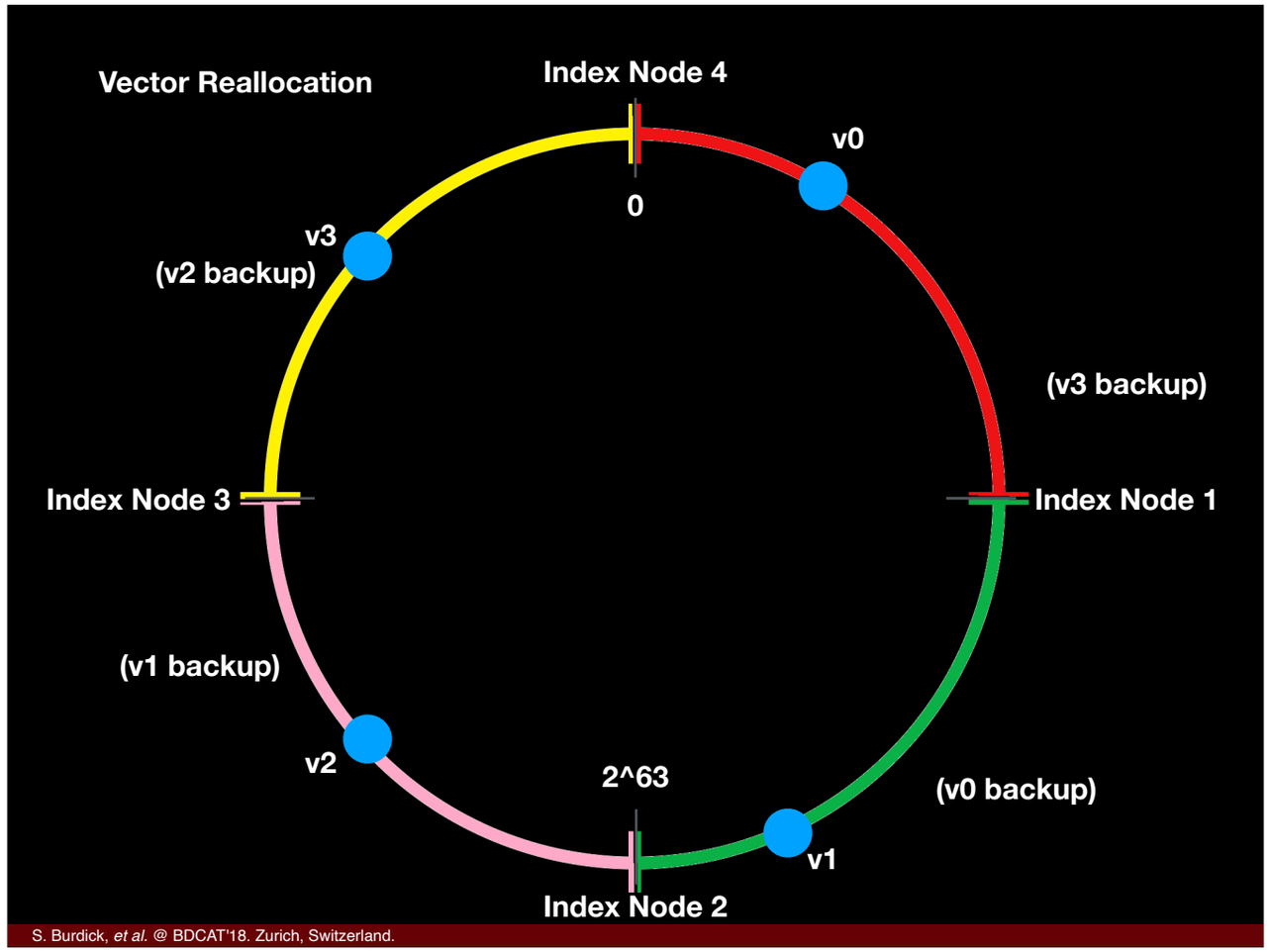
Suppose you had 4 vectors assigned to 4 different inodes as shown. Each blue dot represents a inode's primary vector as shown previously. Also note that each inode has the primary vectors of the inode placed before it as backup. Inode 4 has inode 3's primary vector, v2, and so on. Now suppose that inode 1 crashed. To recover from this fault, we reassign v0 to inode 1 as a primary vector—it was formerly a backup vector. Since Inode 1's previous machine is now Inode 4, it has to keep a backup of Inode 4's primary vectors. Therefore, inode 4 sends v3 to inode 2 so it can keep it as a backup. Likewise, inode 2 sends v0 to inode 3 so it can keep a backup of its vectors. Now, every vector is stored on at least 2 index nodes.

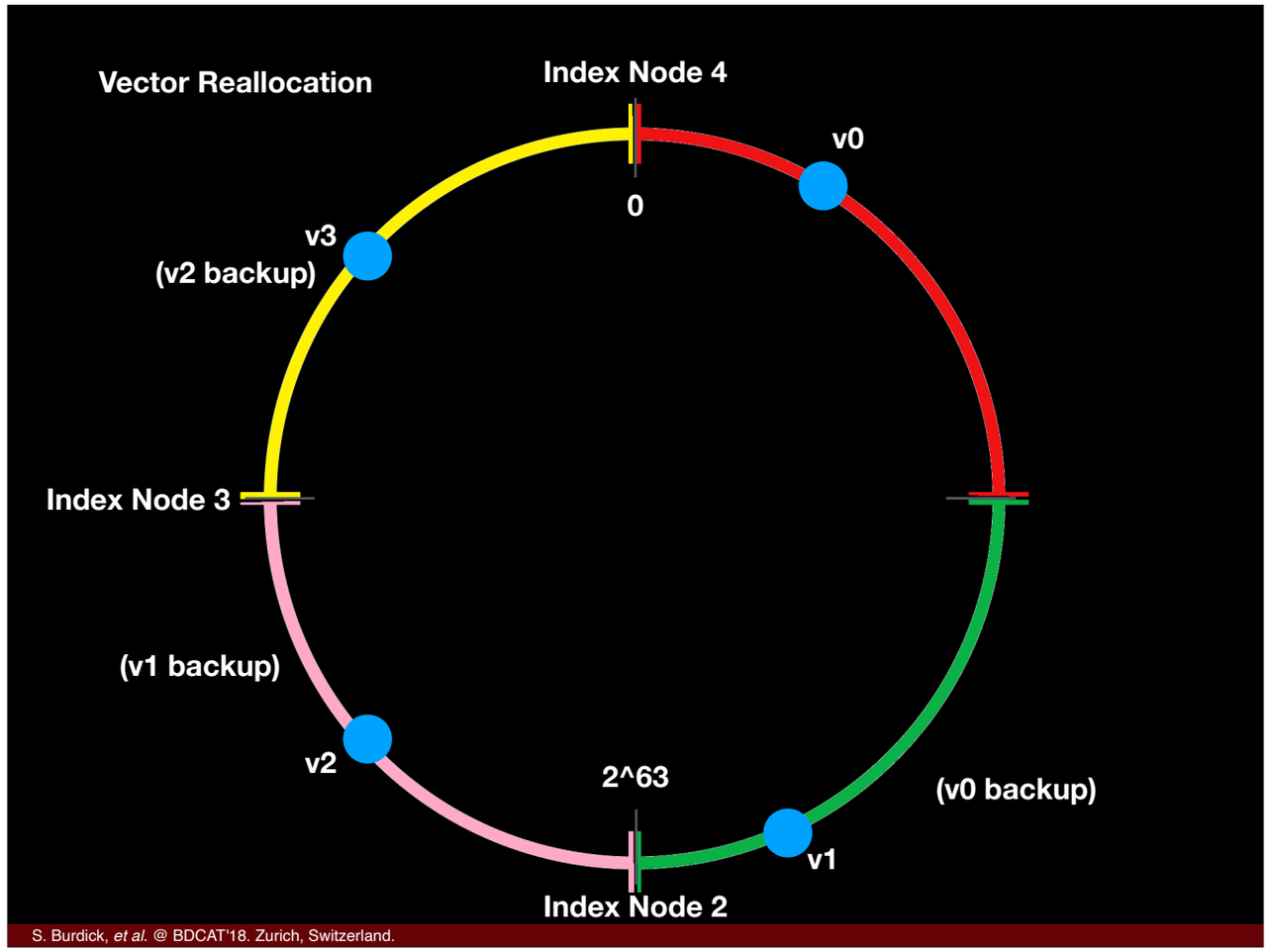
Note that we needed only 2 messages to redistribute the vectors. It is possible under naive hashing techniques that every bit-vector would have to be rehashed, which would require significantly more message-passing for data reorganization. If we use consistent hashing, however, we only require a total of r messages.

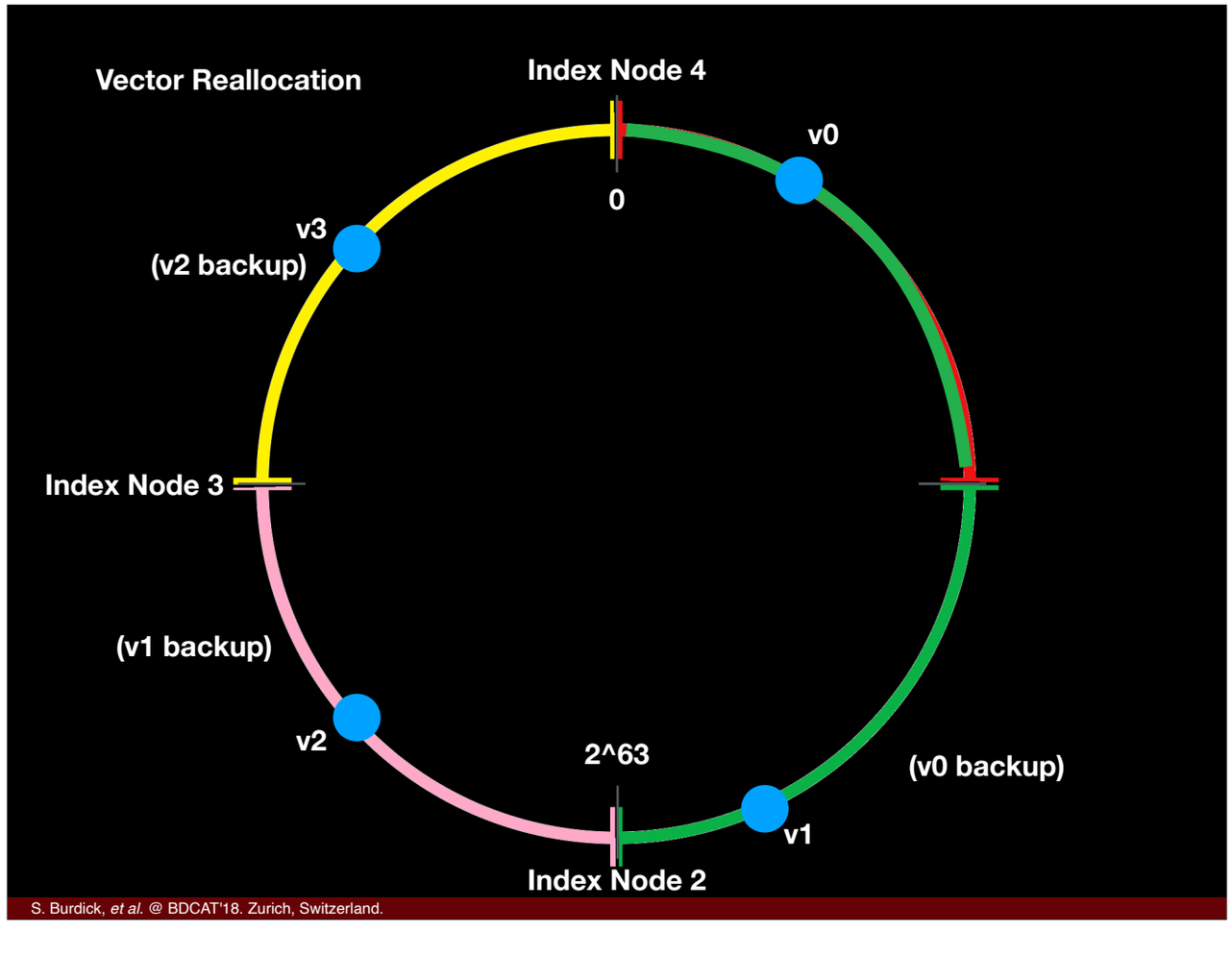

Vector Reallocation
Index Node 4
0
v0
Index Node 1
v1
2^63
Index Node 2
v2
Index Node 3
v3
(v2 backup)

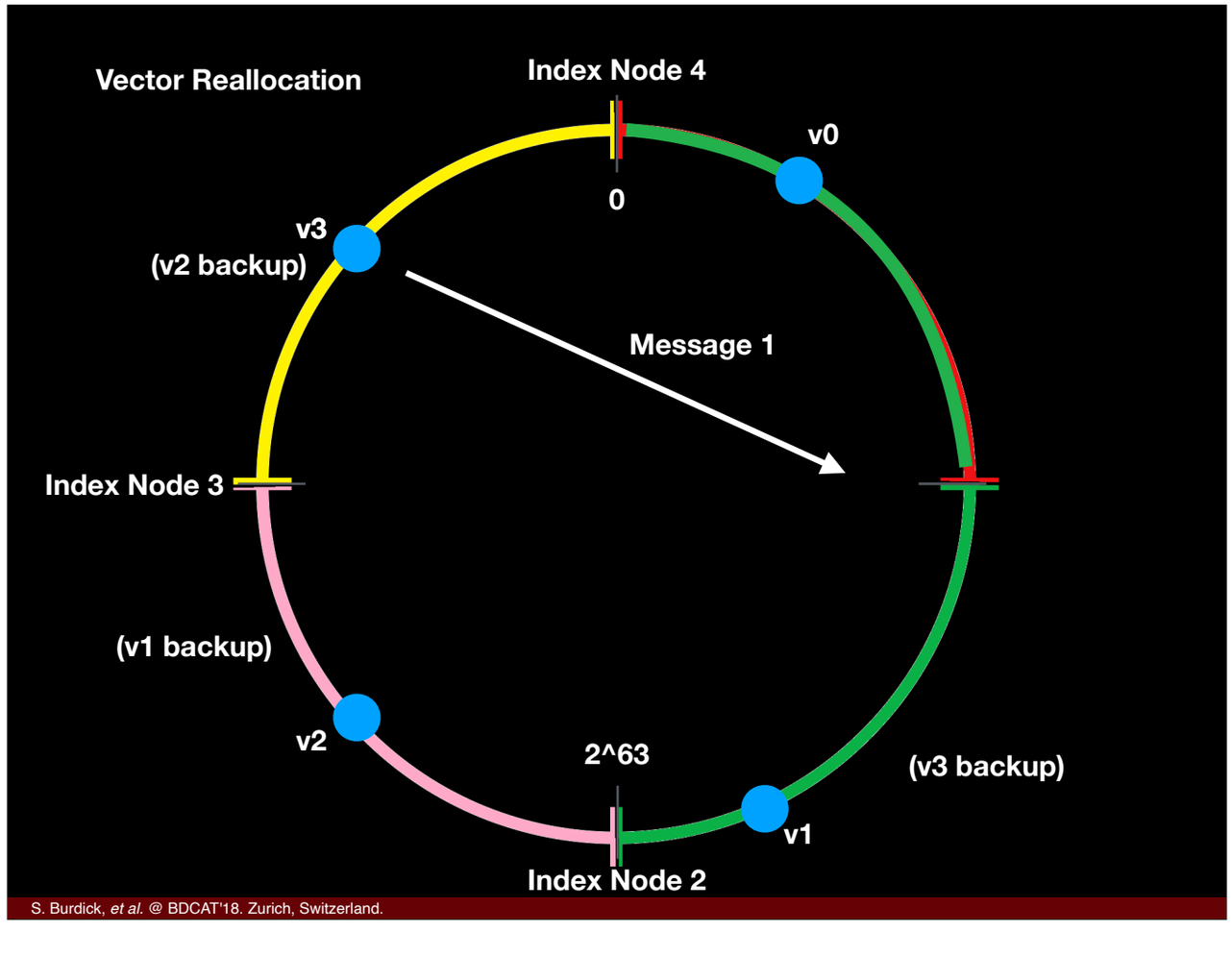

Vector Reallocation
Index Node 4
0
v0
(v3 backup)
Index Node 1
v1
Index Node 2
2^63
v2
Index Node 3
v3
(v2 backup)

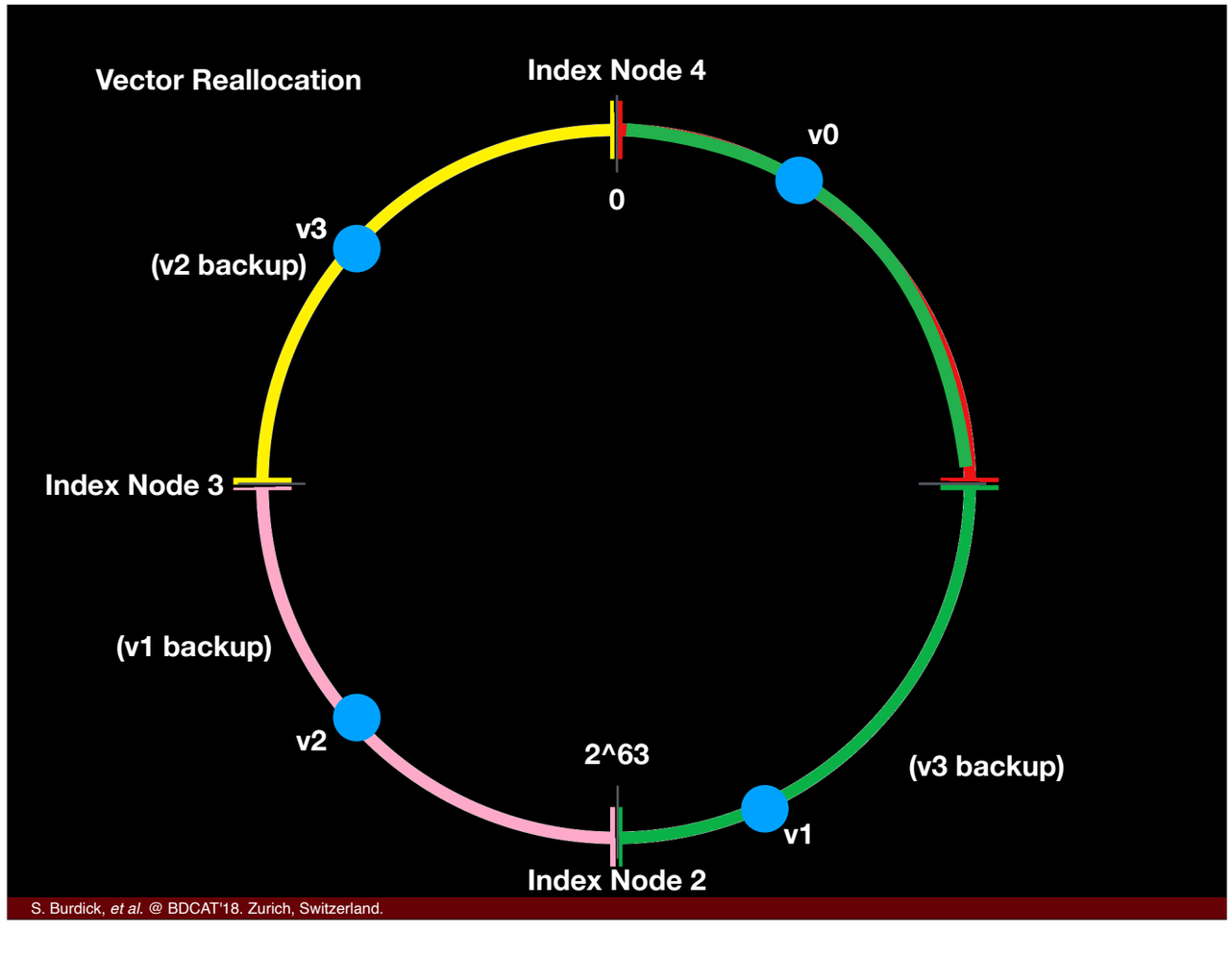

Vector Reallocation
Index Node 4
0
v0
(v3 backup)
Index Node 1
(v0 backup)
v1
Index Node 2
2^63
v2
Index Node 3
v3
(v2 backup)

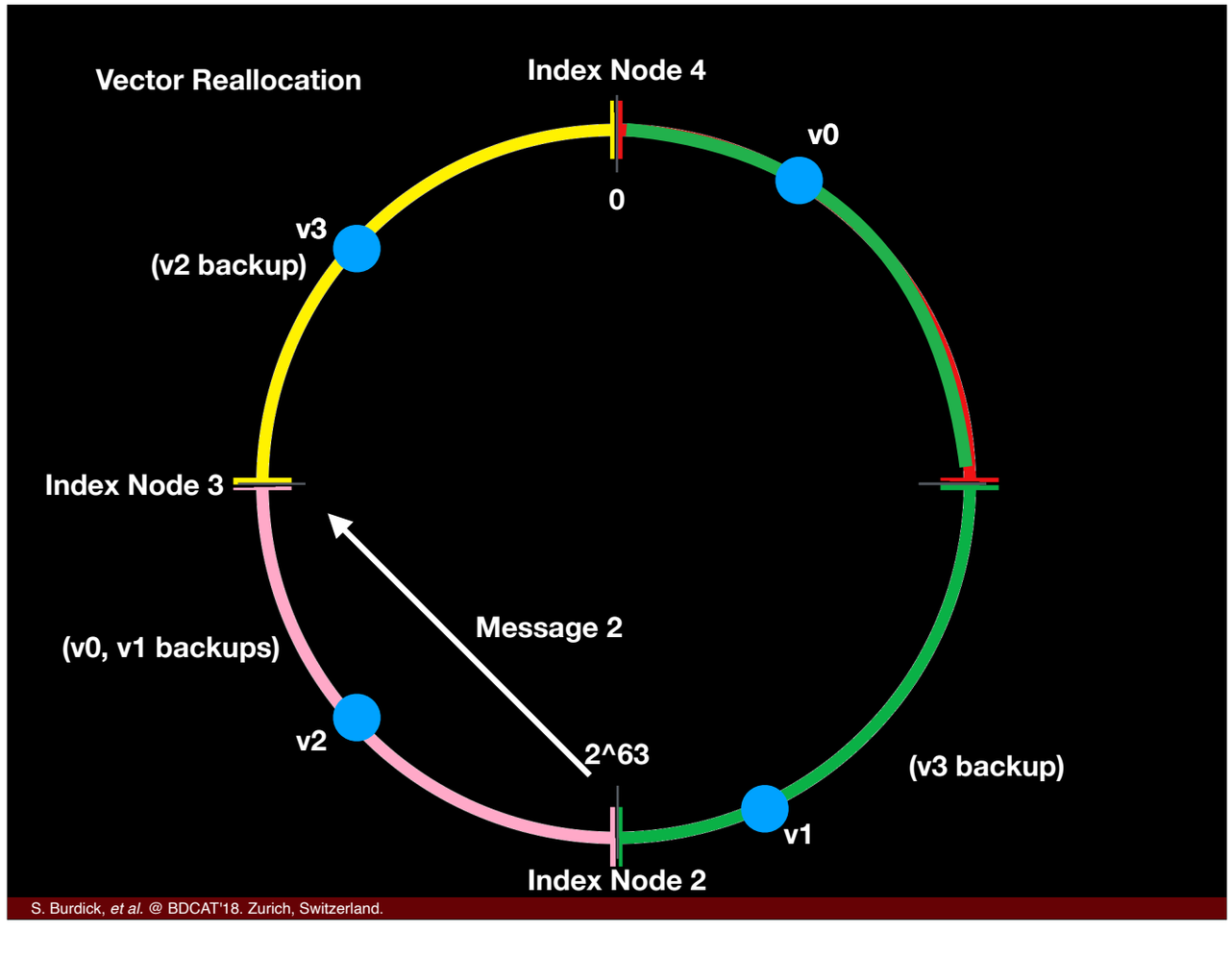

Vector Reallocation
Index Node 4
0
v0
(v3 backup)
Index Node 1
(v0 backup)
v1
Index Node 2
2^63
v2
(v1 backup)
Index Node 3
v3
(v2 backup)

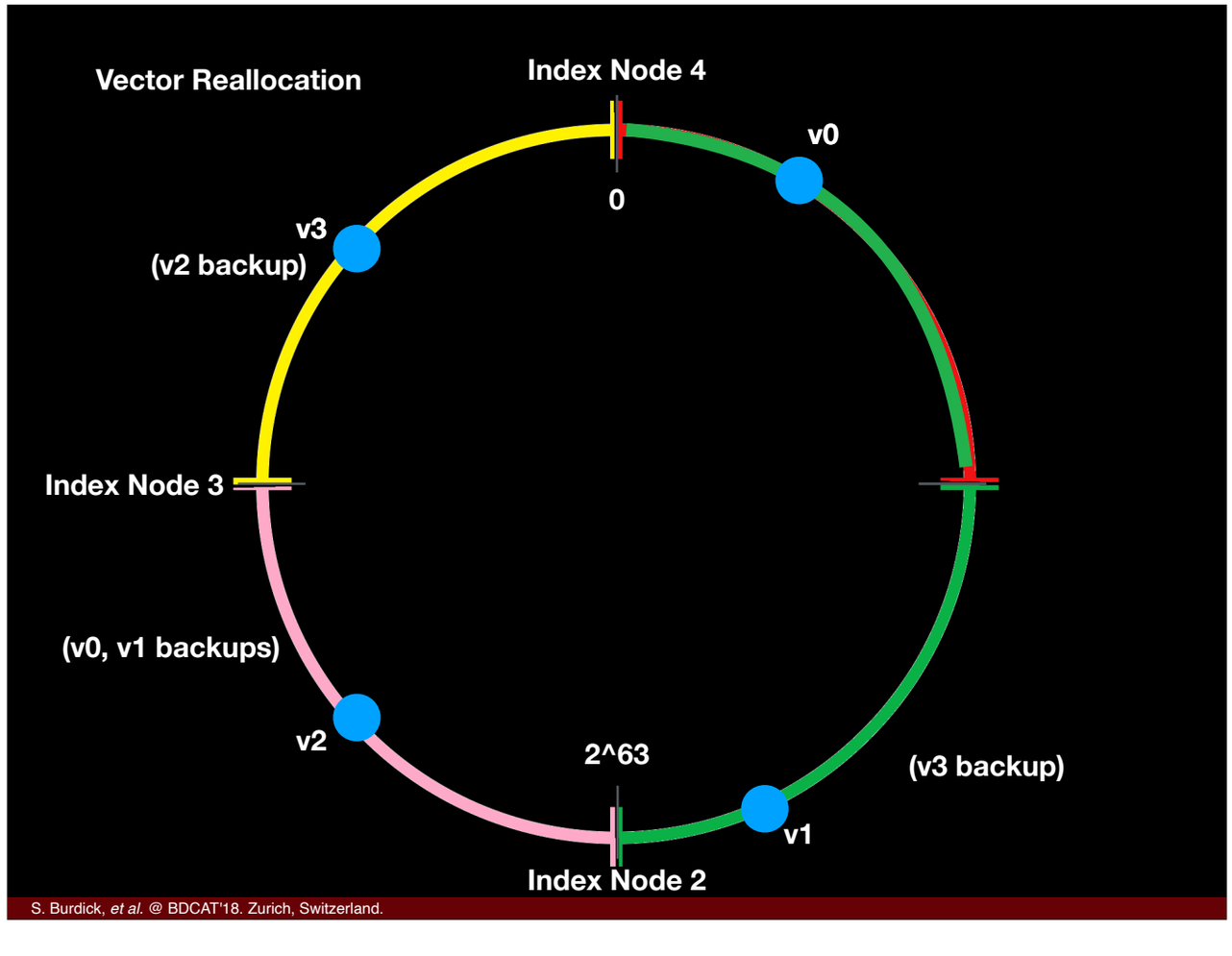

Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Index Node 3
(v1 backup)
v2
2^63
(v0 backup)
v1
Index Node 2


Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Index Node 3
(v1 backup)
v2
2^63
(v0 backup)
v1
Index Node 2


Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Message 1
Index Node 3
(v1 backup)
v2
2^63
(v3 backup)
v1
Index Node 2


Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Index Node 3
(v1 backup)
v2
2^63
(v3 backup)
v1
Index Node 2


Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Index Node 3
Message 2
(v0, v1 backups)
v2
2^63
(v3 backup)
v1
Index Node 2

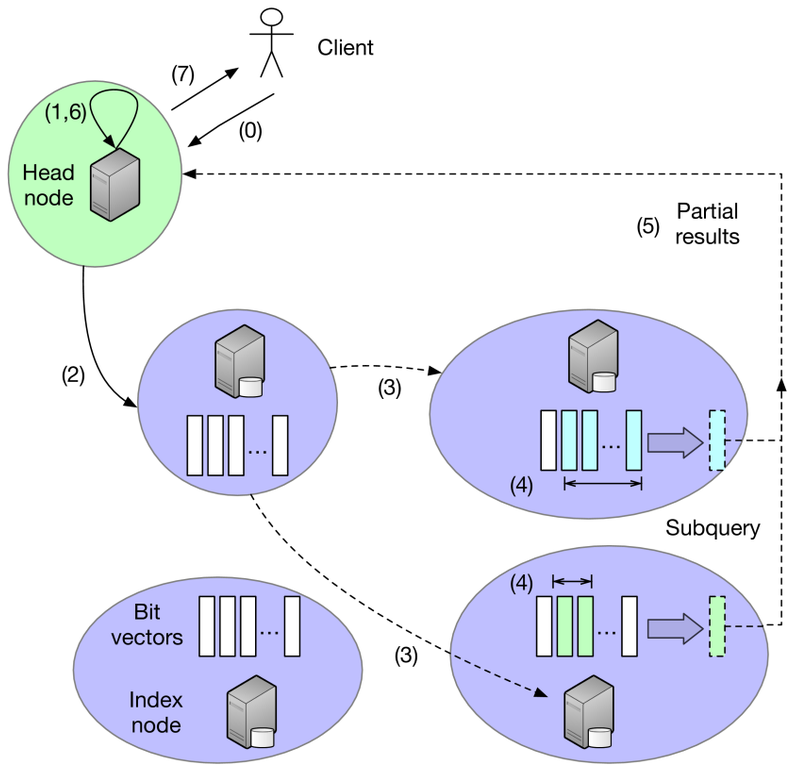

Vector Reallocation
Index Node 4
0
v0
v3
(v2 backup)
Index Node 3
(v0, v1 backups)
v2
2^63
(v3 backup)
v1
Index Node 2

Query Input and Planning

- Example point queries:
  - *(v0)*
  - *(v2&v3)*
- Example range query:
  - *(v2|v3) & (v4|v5|v6)*
- Each parenthesized expression is treated as an individual “subquery” in the plan generated by the head node.
  - Plan is a set of lists of (index node, vector) pairs
  - Index nodes work together to yield the result, which is returned to the head node.

Our system is designed to handle two types of queries: point queries, and range queries. Point queries contain one set of vectors to be AND-ed together, such as (v0) or (v2&v3). A range query (an example of which we gave in the bitmap slide) contains a set of parenthesized sub-ranges, where each vector in the subrange is OR’d together, and each subrange is AND-ed together. Each parenthesized expression is treated as an individual subquery in the plan generated by the head node. The plan is a set of lists, where each list specifies which index nodes to visit and which vectors to obtain from them. The index nodes then work together to satisfy the query based on the plan, then return the results to the head node, which collates the results and gives the final answer to the client.

Distributed Query Execution



Query execution goes as shown here. First, the client submits a query request to the head node. The head node breaks the query string into subqueries as discussed. Then, using consistent hashing, the head node decides which index nodes will be used in completing the subquery to locate the vectors. In an effort to distribute the work to all index nodes as evenly as possible, we choose a random index node from consistent hashing for each vector in the subquery.
Next, the index node at (2) is told to execute a portion of the subquery. That node then sends partial results to two other index nodes, (4) which use their vectors to complete the subquery. The subquery partial results at (5) are then returned to the head node. This process is repeated for all subqueries that make up the original query. Once all the subqueries have been completed, the head node collates all the subquery results and returns the final answer to the client.

Outline

- Motivation
- Background
  - Bitmap Index
- System Details
- Experimental Results
- Conclusion and Future Work

We also ran a series of experiments on our system to test the efficiency of our implementation.

Experimental Results Setup

- Each head/index node runs as a distinct process, where each can communicate with one another via RPC.
- Each process ran as a Docker container on the same machine.
- For all experiments we used the TPC-C benchmark
  - 100000 bit-vectors were derived and loaded
  - Replication factor of 2 *(r = 2)*

We wanted to emulate a real, horizontally-scaled distributed system as much as we could. To do so, we wanted to ensure that each node ran as a separate process that communicated via Remote Procedure Calls. Therefore we used one Docker container for each process, so that the nodes could communicate over a Docker bridge network on one physical machine. We used the TPC-C dataset, a commonly used benchmark that models business transactions, to derive about 200,000 bit-vectors. We also used a replication factor of 2.

System Initialization time

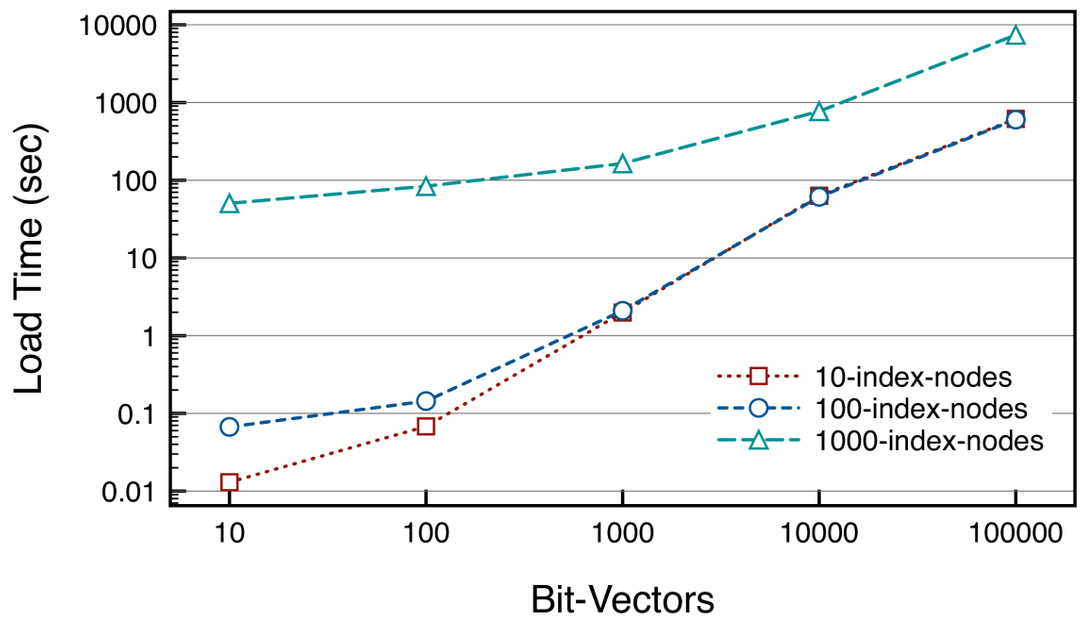
- We measured the time required for the head node to handshake, and insert a new index node into the consistent-hash ring, each index node.

- Time: consistently ~0.5s per node

10 Nodes	100 Nodes	1000 Nodes
4.501 sec	48.253 sec	667.21 sec

For our first experiment, we measured the time required for the head node to handshake, and insert into the consistent-hash ring, each index node, for 10, 100, and 1000 new index nodes.
The initialization time is about 0.5 seconds per node, which shows the expected efficiency of consistent hashing.

Performance of PUT operator

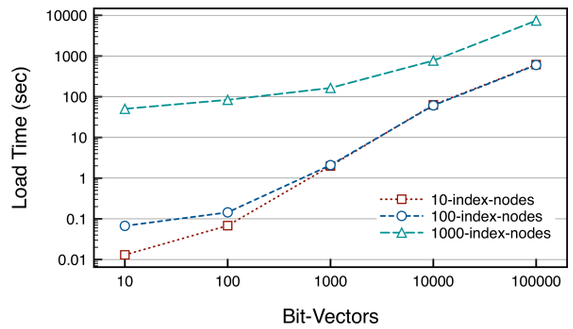


For our second experiment, we measured the performance of the PUT operator by taking 10 index nodes and loaded varying numbers of bit-vectors into the system, then repeated that for 100 and 1000 nodes. Note that the results are shown in log scale on both axes for readability.

Results

- Load time is log-linear, *O(r log n)*, meaning that our system scales to large numbers of bit-vectors

- Gaps between settings:
  - Smaller gap: due to consistent hashing
  - Large gap: due to heartbeat monitoring



The results confirm that the load time is log-linear, i. e., O(r log n), which shows that our system scales to handle large numbers of bit-vectors. Initially, there is a gap between the 10 and 100 index node settings. We believe that this is due to the log n factor of consistent hashing, which noticeably contributes more load time initially, when only a few bit-vectors are being stored. As more vectors are added, the hashing overhead is amortized due to the constant number of messages sent per vector.

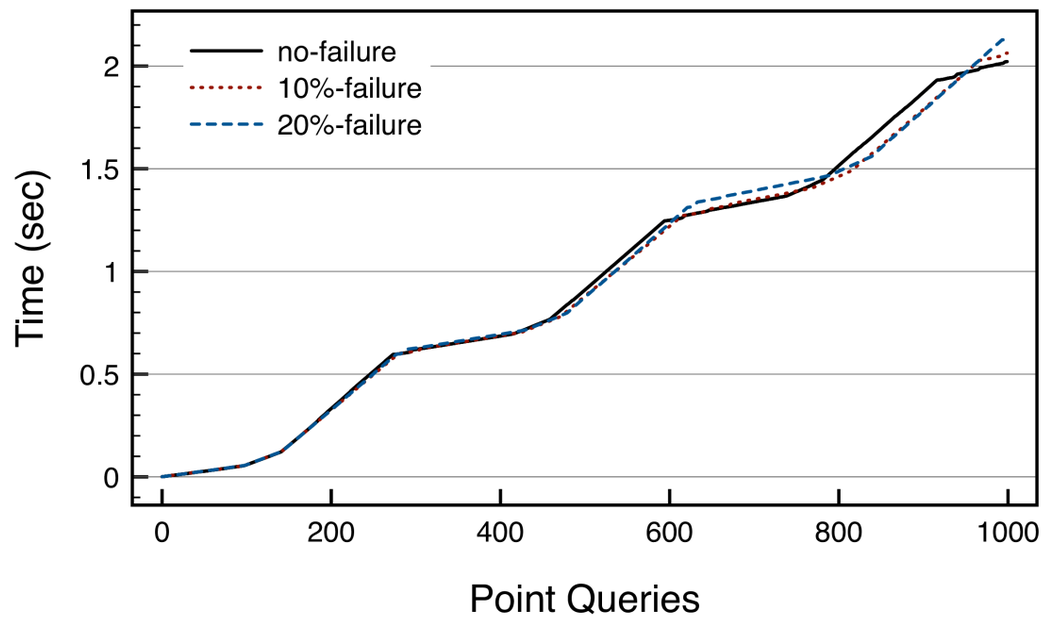
However, there is a larger gap between the 100 and 1000 node settings. We believe that this discrepancy is due to the head node's heartbeat monitoring which runs on each PUT command, which was simply not exposed between the smaller clusters. The overhead introduces an additional term for the heartbeat messages, which explains the rather constant sized gap between the two lines.

Query Execution and Fault Tolerance

- Measured cumulative execution time of queries
- Showed impact of node failures by running 3 different types of experiments with these characteristics:
  - No node failures
  - Deliberately killing a node every 100 queries (10%)
  - Deliberately killing a node every 50 queries (20%)

For our next set of experiments, we measured the cumulative amount of time required to execute a given number of queries from 1 to 1000. 100 index nodes were deployed and stored 10000 bit-vectors, running 1000 point queries and 1000 range queries, where all queries are randomly generated. In order to test fault tolerance, we want to show the impact of node failures by comparing the time required to execute the queries without any of them failing against the time required in the presence of node failures. To do so, for the second experiment we had an index node deliberately terminate itself every 100 queries, and for the third experiment we had one deliberately terminate every 50 queries.

Point Queries



Because point queries are simple, requiring only a single vector per subquery, the results are stable and predictable. The entire 1000-query workload completed in roughly 2 seconds with our without faults.

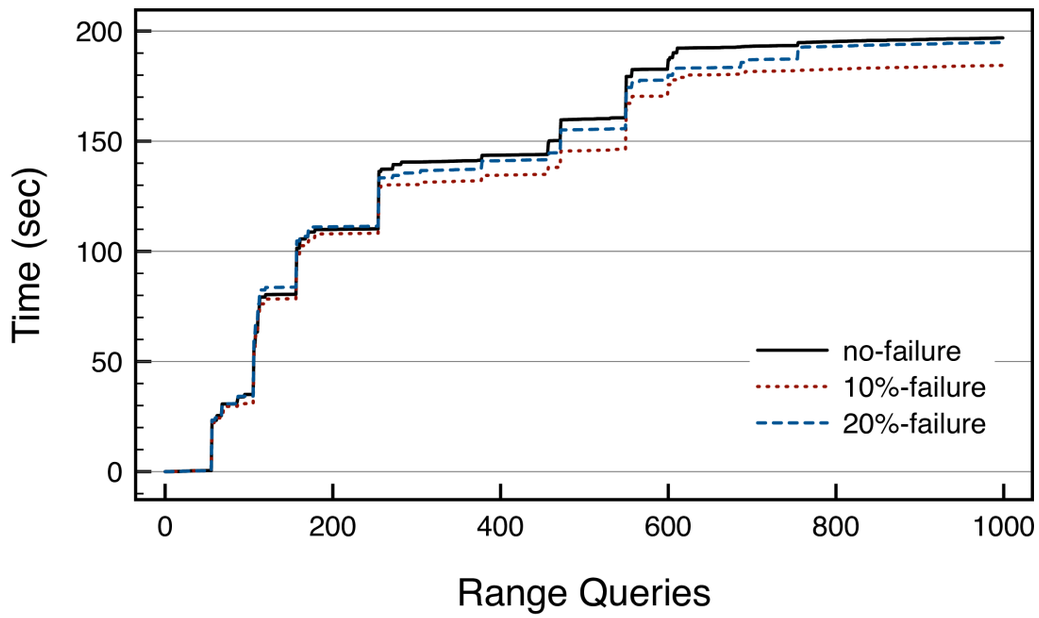
In the no-fail case, we averaged 2.02ms per query.

In the 10% fail case, we averaged 2.06ms per query, which is a 2% slowdown from no-fail case.

Finally in the 20% fail case we averaged 2.13ms per query, or 5% slowdown from no-fail case

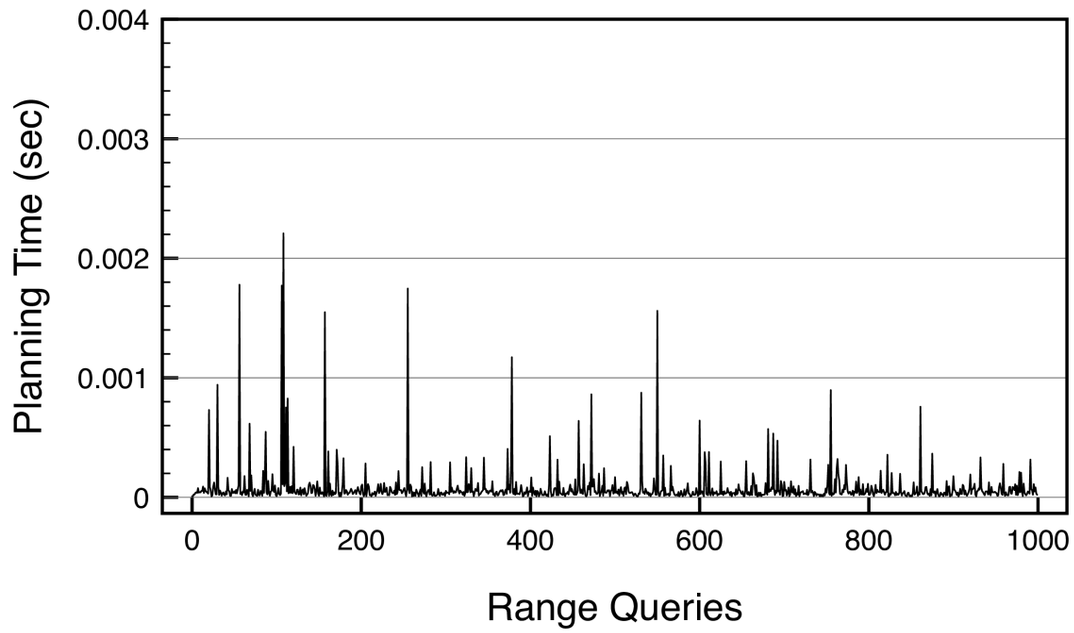
These results are due to the fact that only a handful of nodes had to be involved in each query.

Range Queries



For range queries, we have a similar set of results. The cumulative time increases in a steplike manner since some queries were significantly more I/O-intensive than most. Similarly to point queries, there is not a significant variance between workloads with and without failure. These results suggest that node failures do not significantly impact query performance, due to consistent hashing and vector replication.

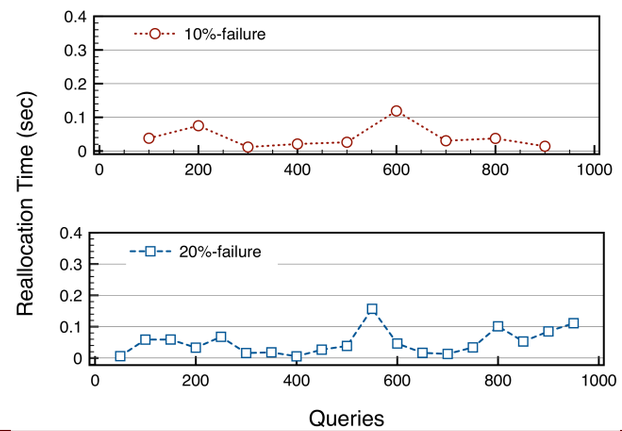
Query Planning Overhead



Our next experiment measured the amount of time required for the head node to create a distributed query plan from each of the same set of range queries. The planning overhead is generally negligible as it averages 0.07 ms, which is 0.03% of the average execution time.

Reallocation Overhead

- After each index node failure, we take the sum of the time needed for:
  - Head node to communicate to the new nodes regarding which bit vectors to transfer
  - Index nodes to transfer the vectors
  - Remove failed node from consistent hash ring



Our final experiment measures the time needed for bit-vector reallocation following a node failure. This is the sum of the time needed for the head node to communicate to the new nodes regarding which bit vectors to transfer, then for the index nodes to actually transfer the vectors, then for it to remove the failed node from the consistent hash ring.
Similarly to the last set of experiments, we ran a sequence of queries, and for the first experiment allowed a node to fail 10% of the time, while in the second experiment we let one fail 20% of the time. In both cases, the reallocation time never took more than 200 milliseconds, with an average time of 44.8 milliseconds. We believe that this speed shows the strength of consistent hashing, as only a small subset of the data has to be migrated upon an index-node failure.

Outline

- Motivation
- Background
  - Bitmap Index
- System Details
- Experimental Results
- Conclusion and Future Work

Finally, we give our conclusion and some areas in which we would like to improve the system.

Conclusion

- Organizations requiring large-scale data generation and querying on their data also need fault tolerance
- Our system is designed to automatically handle such problems by supporting
  - Distributed replication of bitmap vectors
  - Reallocation in event of node failure
  - Distributed query execution
- Experiments show that our system can efficiently satisfy queries in the presence of node failures, and quickly reallocate vectors when needed.

Going back to the original problem, recall that when you need to generate a geographically distributed data store, and also want to run queries on the data, it can easily become difficult to handle both tasks in the presence of faults. We propose a system built around distributed bitmap vectors to solve this class of problems. In order to automatically ensure fault tolerance in distributed queryable data-stores, we developed a system designed to support the distributed replication of bitmap vectors, dynamically reallocate the vectors in the event of a node failure, and execute queries on the distributed data sets. Our experimental results show that our system can efficiently satisfy queries in the presence of node failures, which was an important goal we set out to achieve. The fault-tolerant execution also had to be supported by reallocation, which we handled efficiently.

Future Work

- Improve heartbeat mechanism, possibly by reversing message order to index node to head node
- Distributed query optimization
- Alternatives to consistent hashing
- Other types of queries

We would like to improve the performance of our system in a few different ways. The first is to improve the heartbeat mechanism. We decided to have the head node check the status of each index node at the start of each put or query transaction. This could instead be done in reverse, with each index node periodically messaging the head node, allowing the head node to conduct time-critical tasks faster.

The second area improvement is in distributed query optimization. We could have the system short-circuit a subquery that has obtained a partial result vector of entirely 1’s or 0's, since the result will be the same regardless of what the remaining vectors are. In addition, we could try a different set of algorithms for planning out the queries, which would exploit more parallelization in query execution than we currently are.

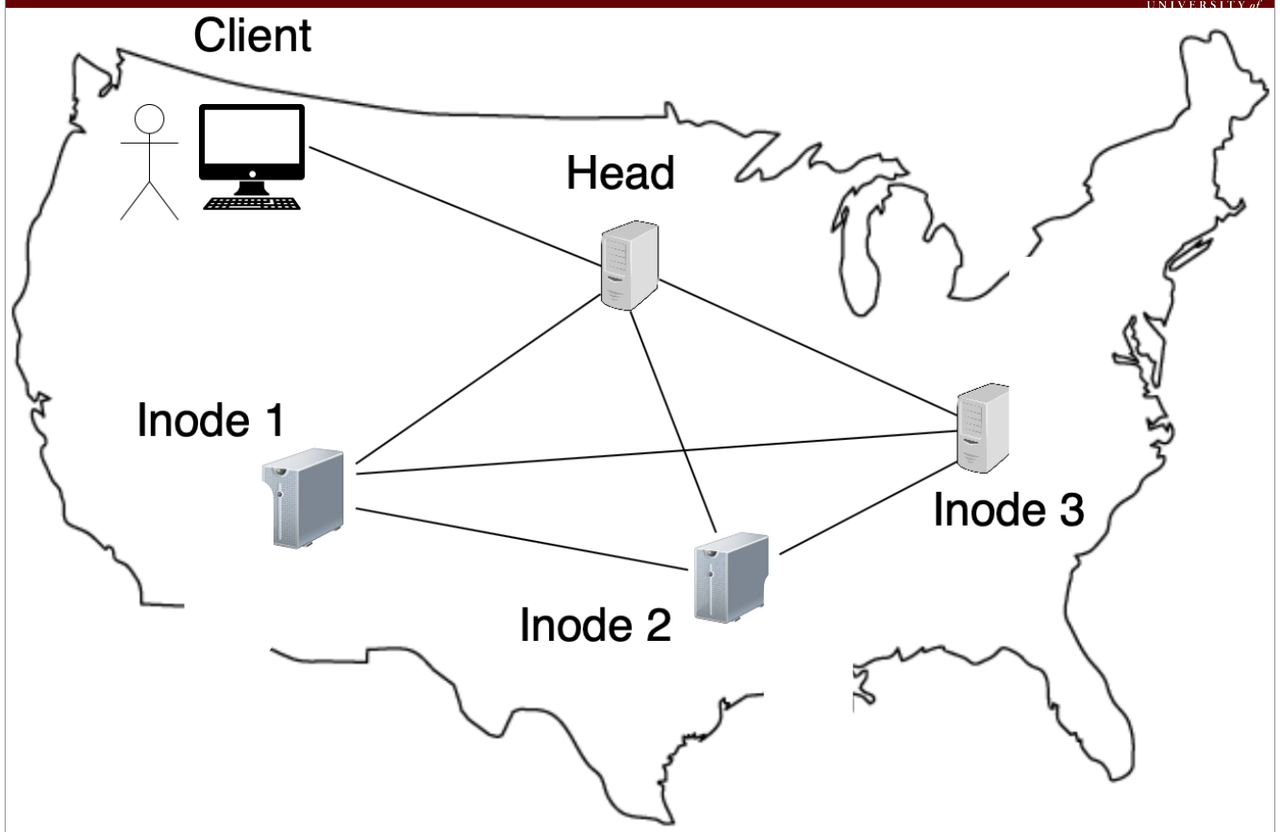
We could also try other key-assignment schemes such as dynamic partitioning, which would allow the vector assignment in the system to be more even, but would be more difficult to code. We can also use alternative data structures to the red-black tree to make consistent hashing run in O(1) time, which would further improve query performance.

Finally, we would also like to support other types of relational queries such as joins and aggregation.

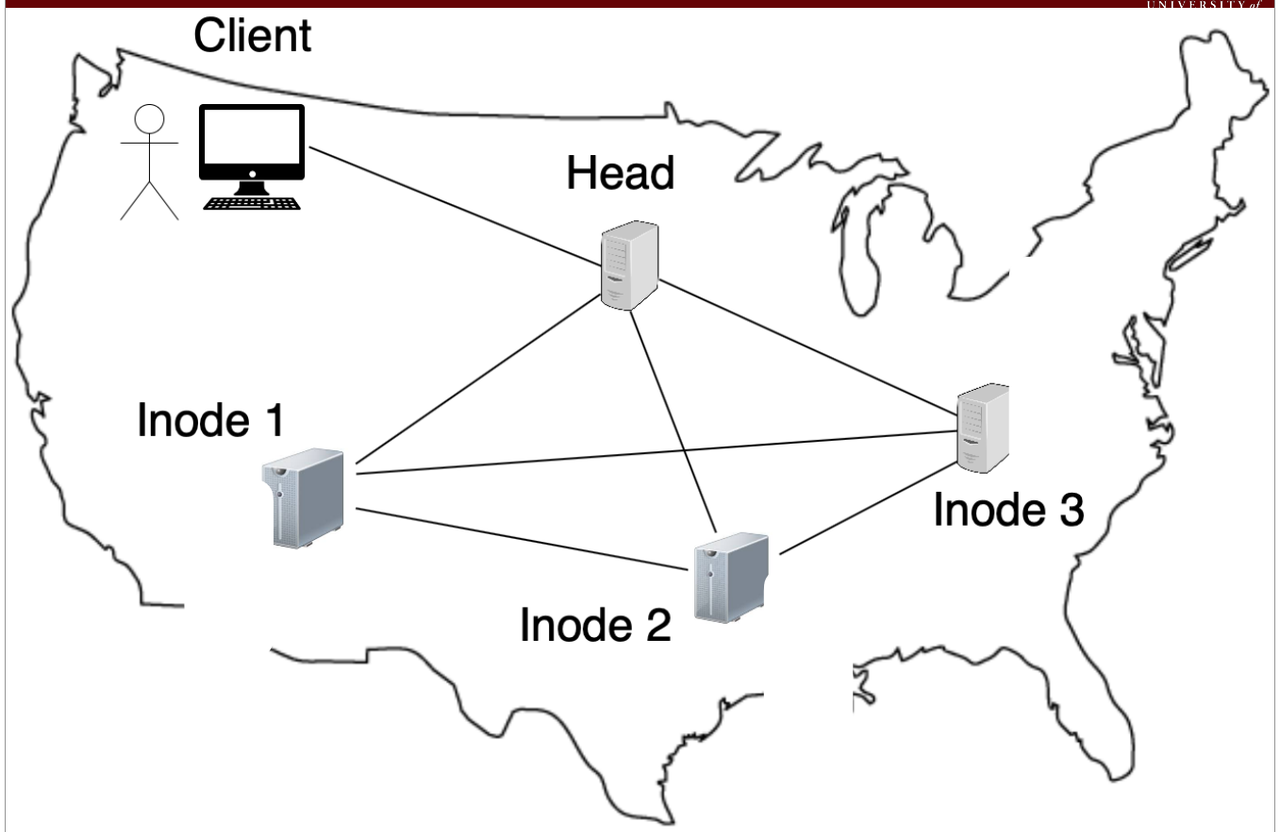
Q&A

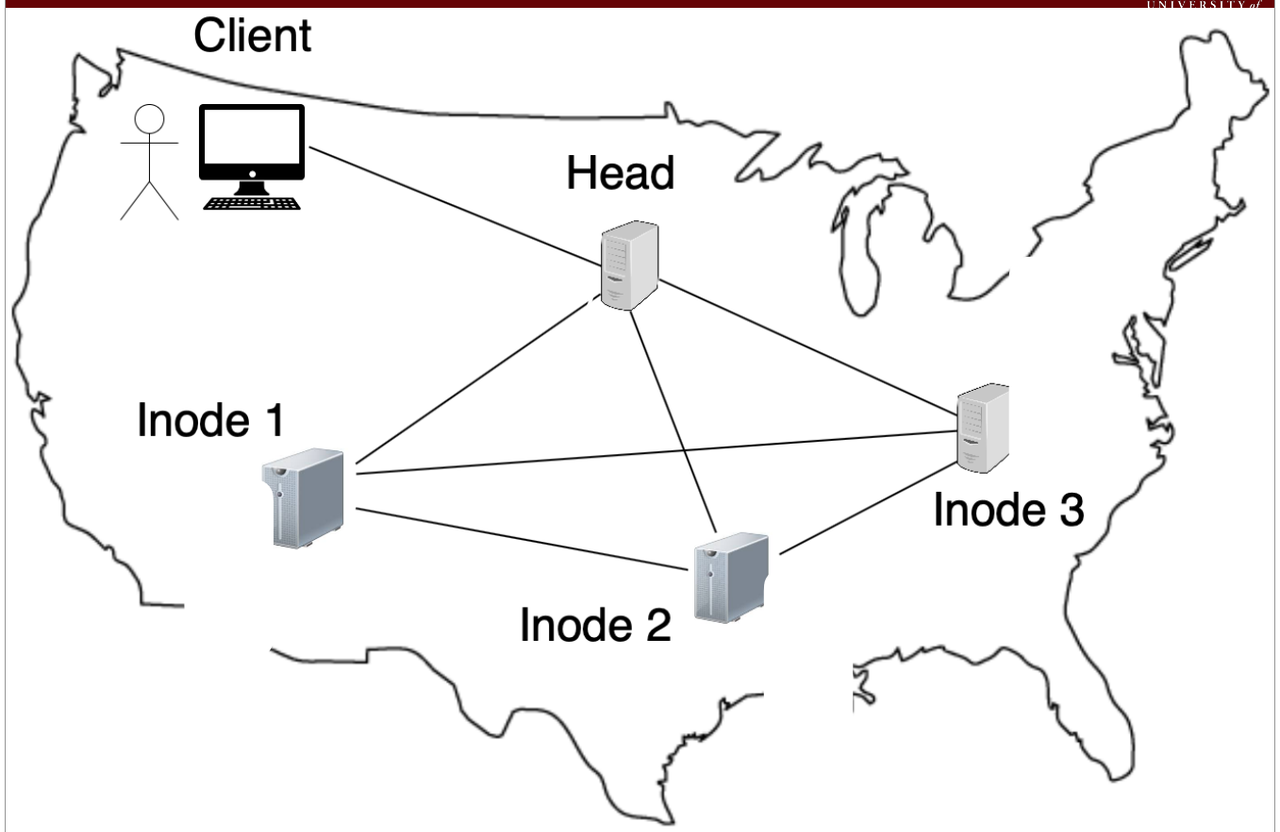
Thank you for listening… any questions?

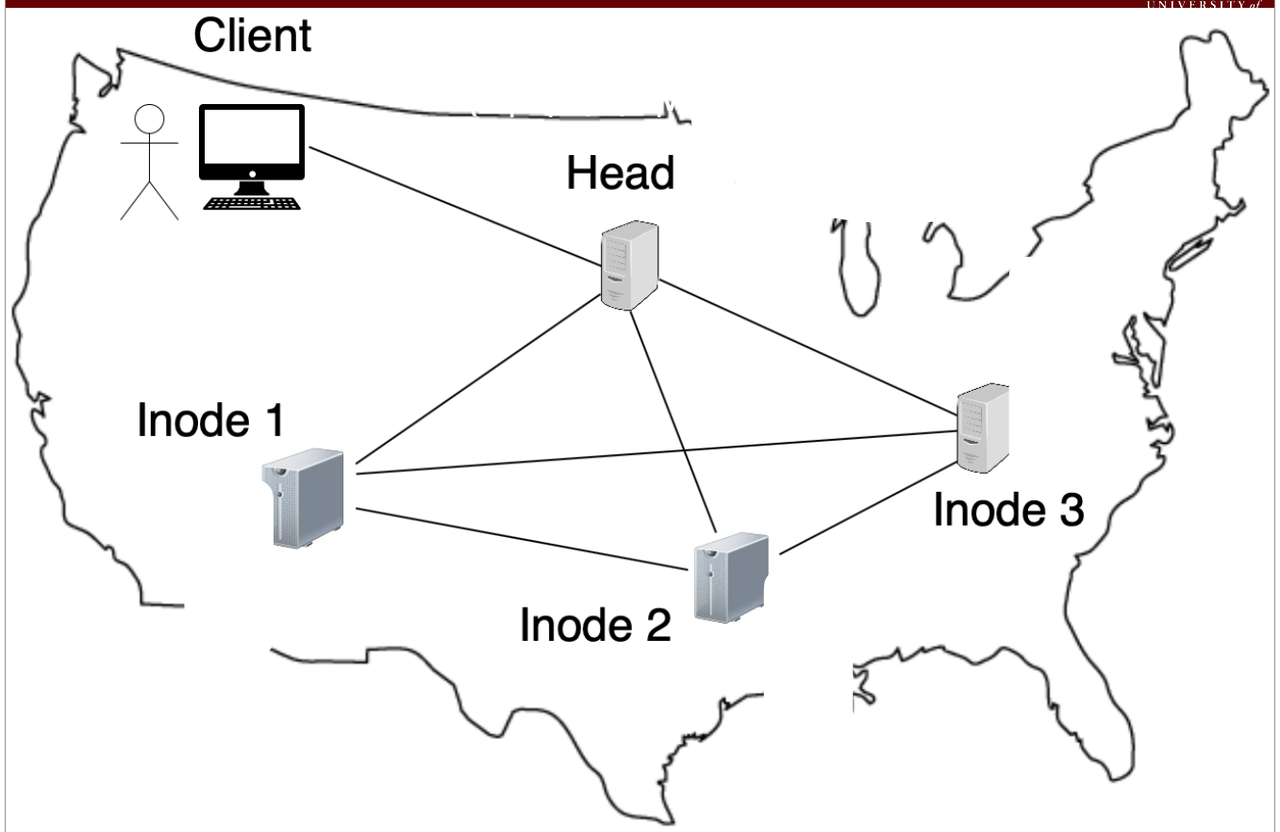
Query demo (time permitting)

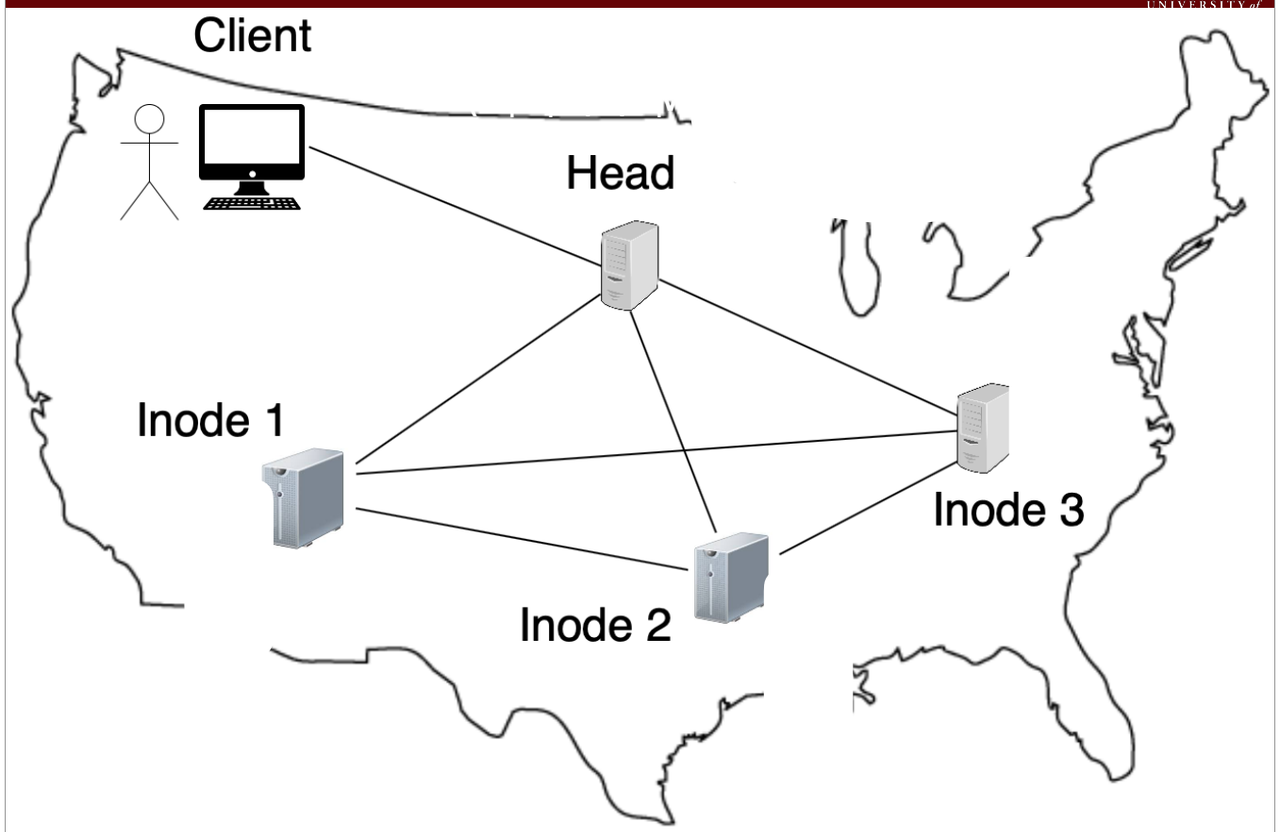


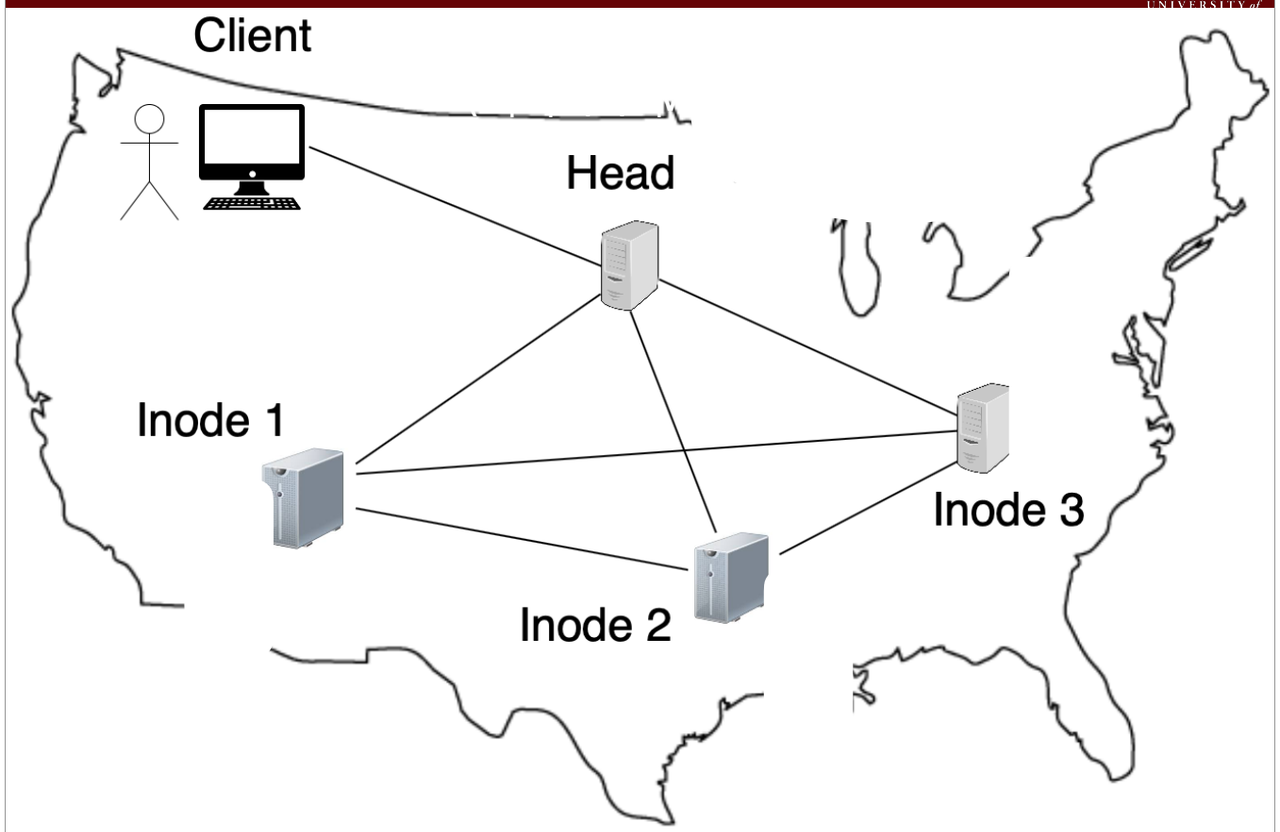
The DBMS will construct the query which it will send to Head.
Once the Head node has received the query, it will break the query apart into the parenthetical query expressions.
If all of the vectors needed for an expression are available on a single inode, that Slave will be sent that query.
For example, here the expression v_3 OR v_4 can be handled by Slave 3.
If, on the other hand, there is no single Slave containing all the vectors involved in a query, the Master node will send a request to a Slave containing some of the vectors along with information about where the remaining vectors can be found.
In this example, Slave 1 is able to handle the v_0 OR v_1 portion of the v_0 OR v_1 OR v_2 expression.
In order to complete the request, Slave 1 will request v_2 from Slave 2.
Once this is done, the Slaves will locally calculate the result of the requests they received.
The will then each return that result, which is a single vector, to the Master.
Once the Master has the result of each expression, it will calculate the final result with is returned to the DBMS.

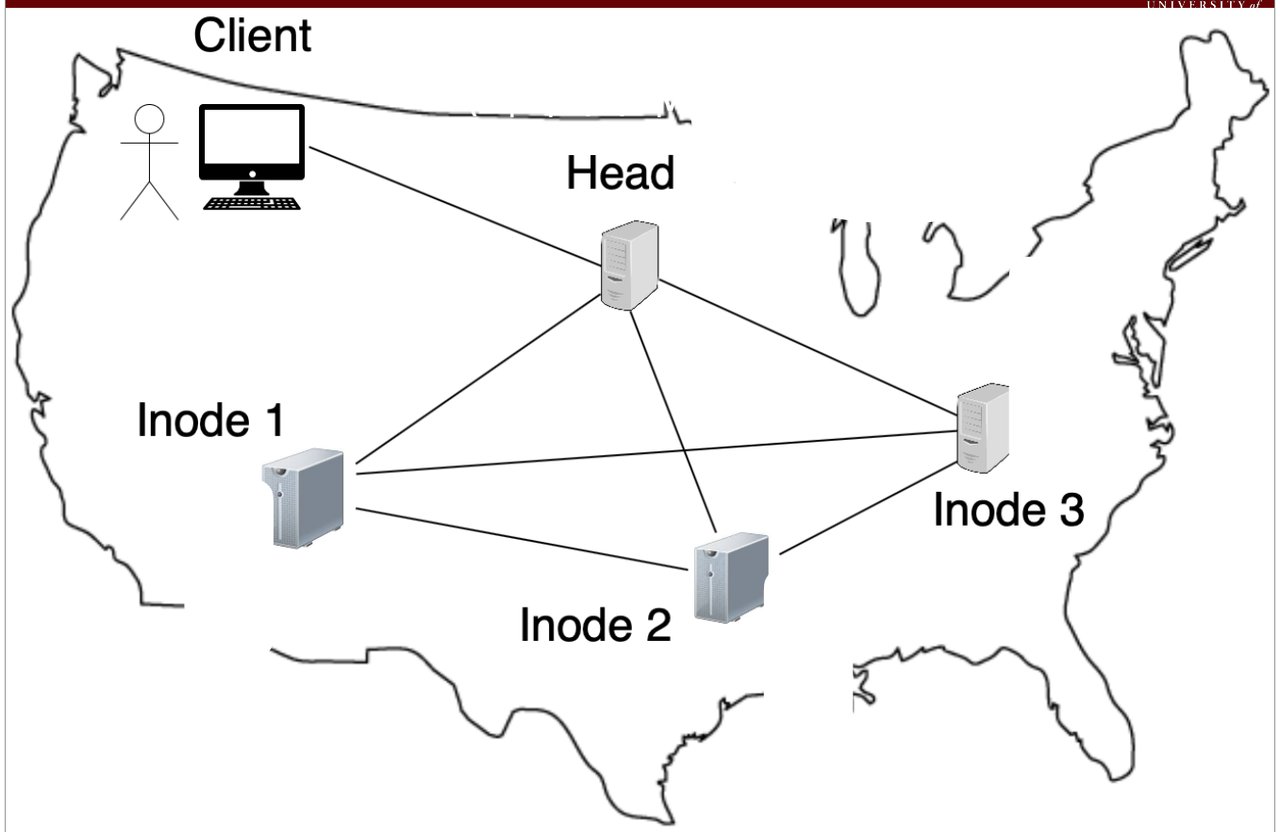

Client
Head
Inode 1
Inode 3
Inode 2

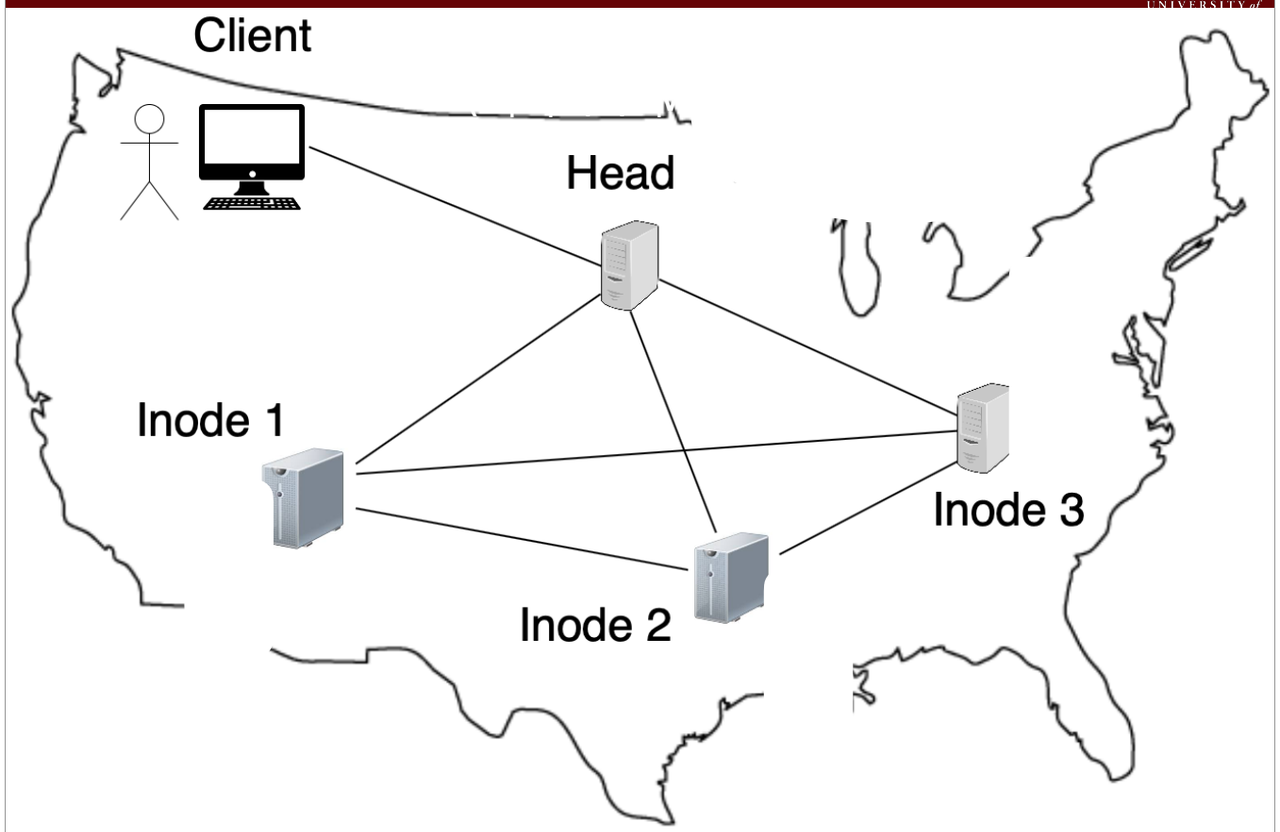

Client
Head
Inode 1
Inode 3
Inode 2

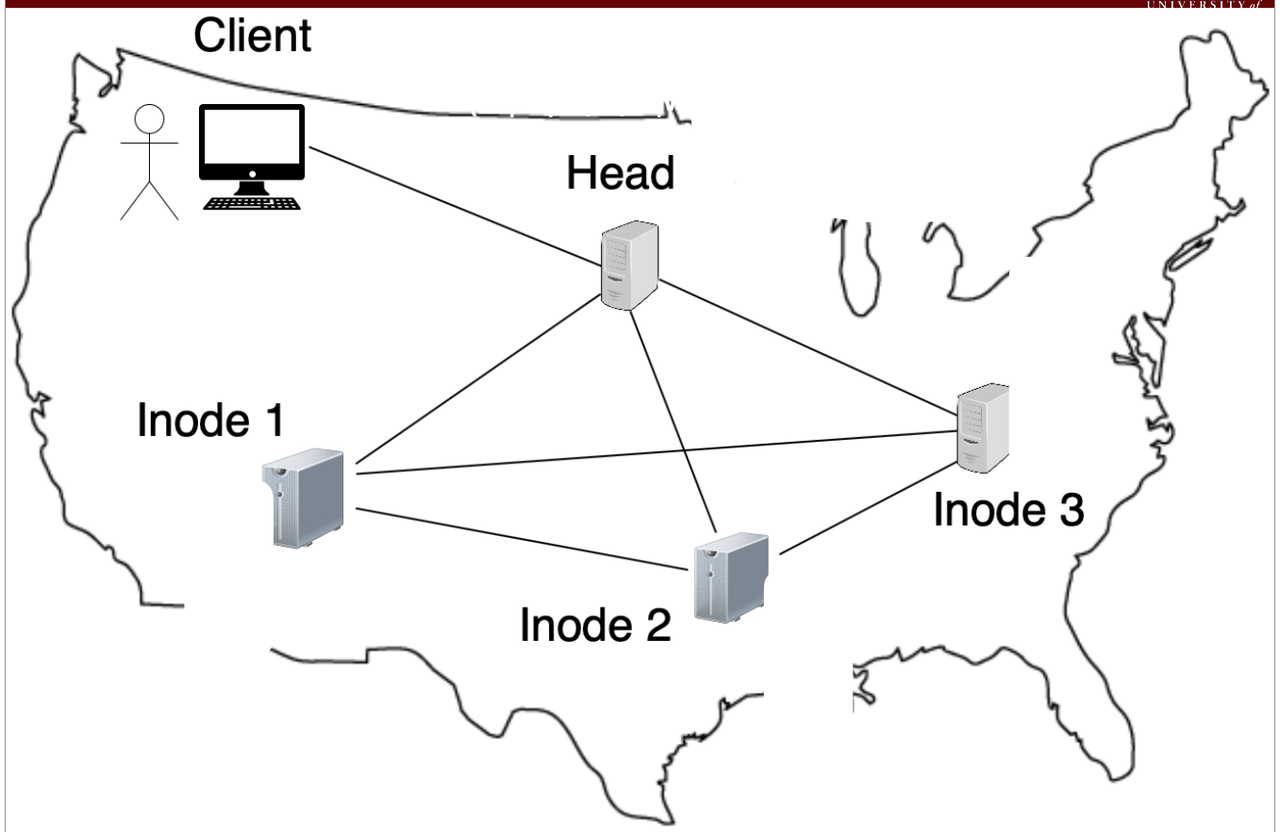

Client
Head
Inode 1
Inode 3
Inode 2

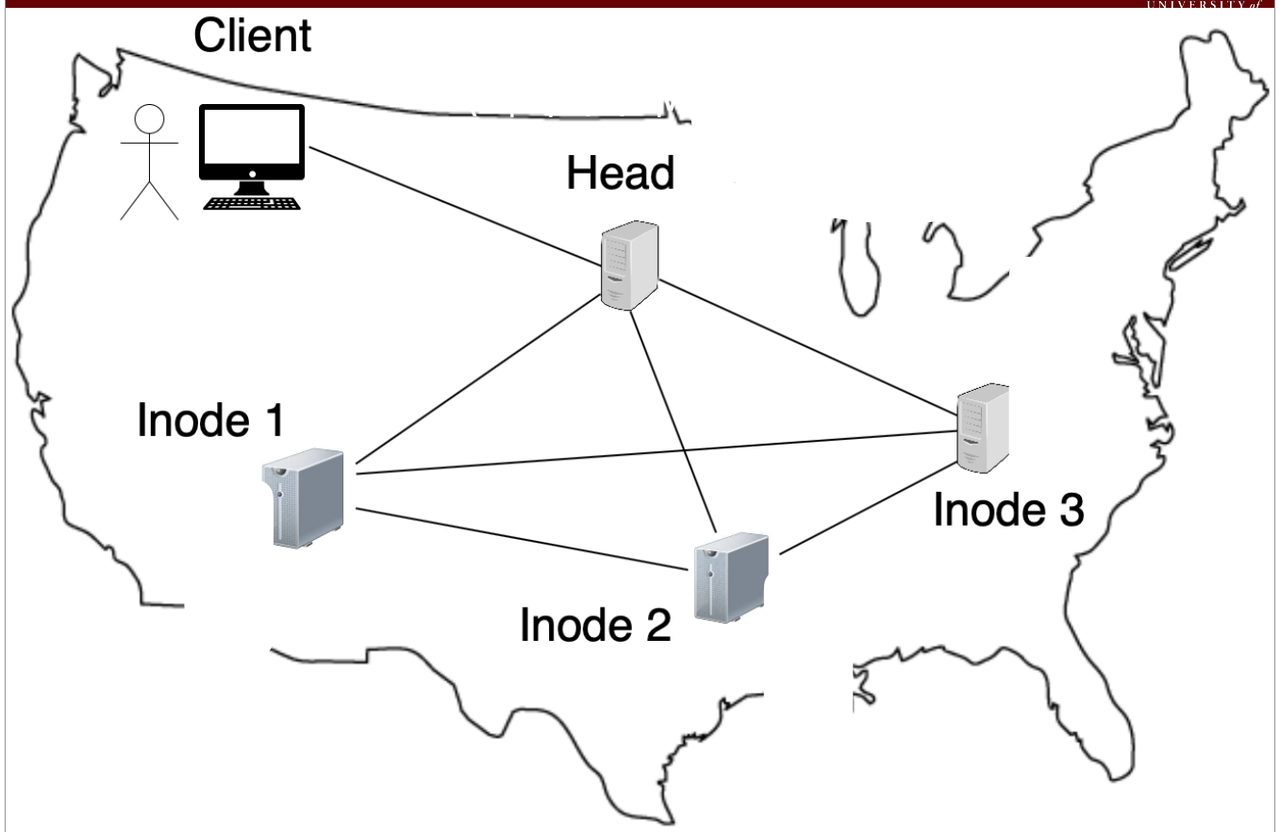

Client
Head
Inode 1
Inode 3
Inode 2

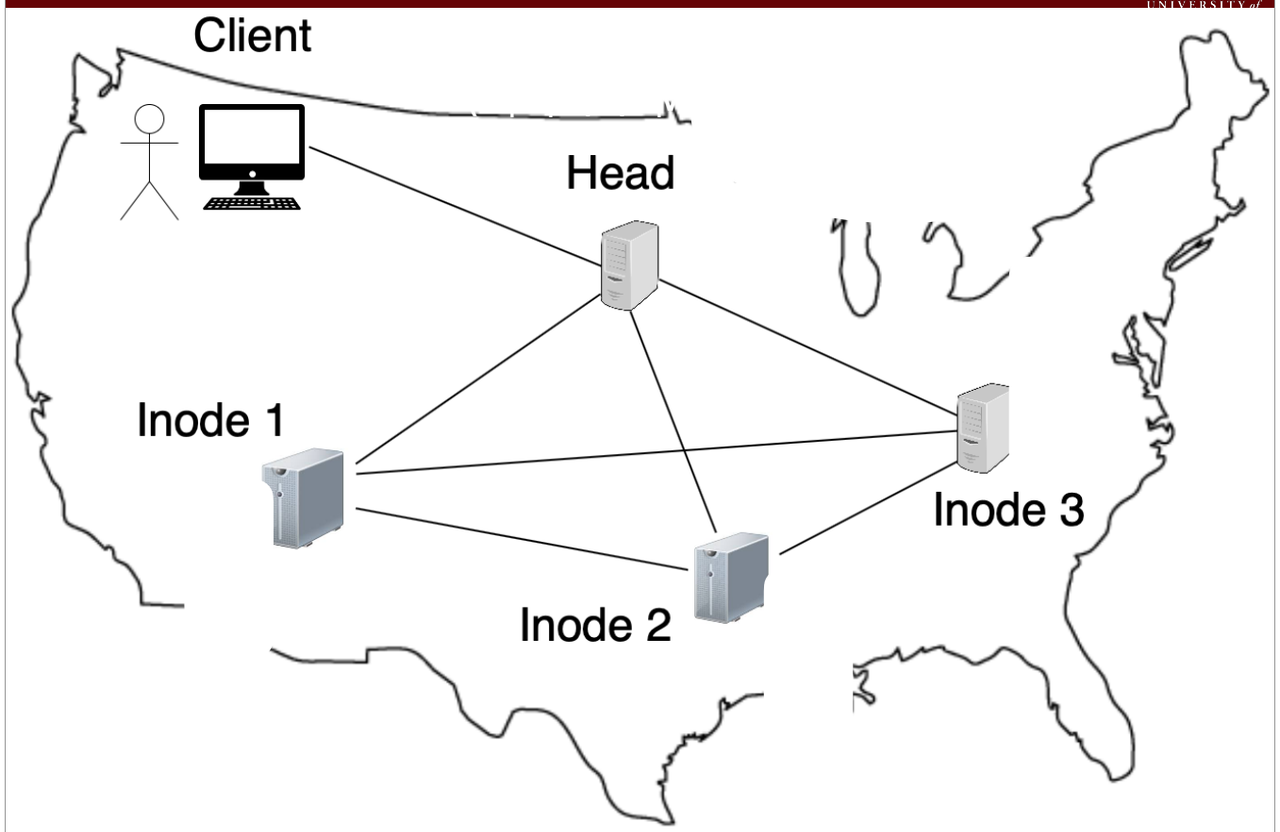

Client
Head
Inode 1
Inode 3
Inode 2

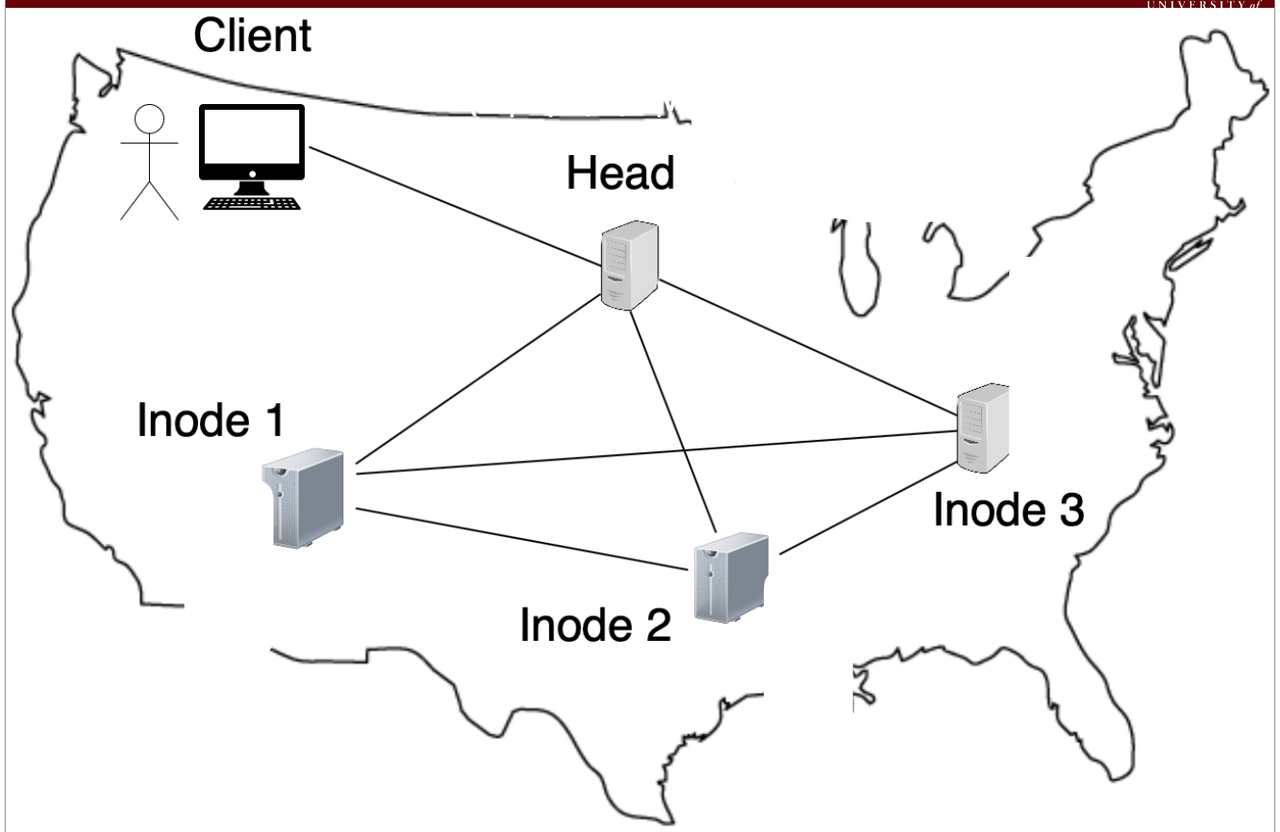

Client
Head
Inode 1
Inode 3
Inode 2

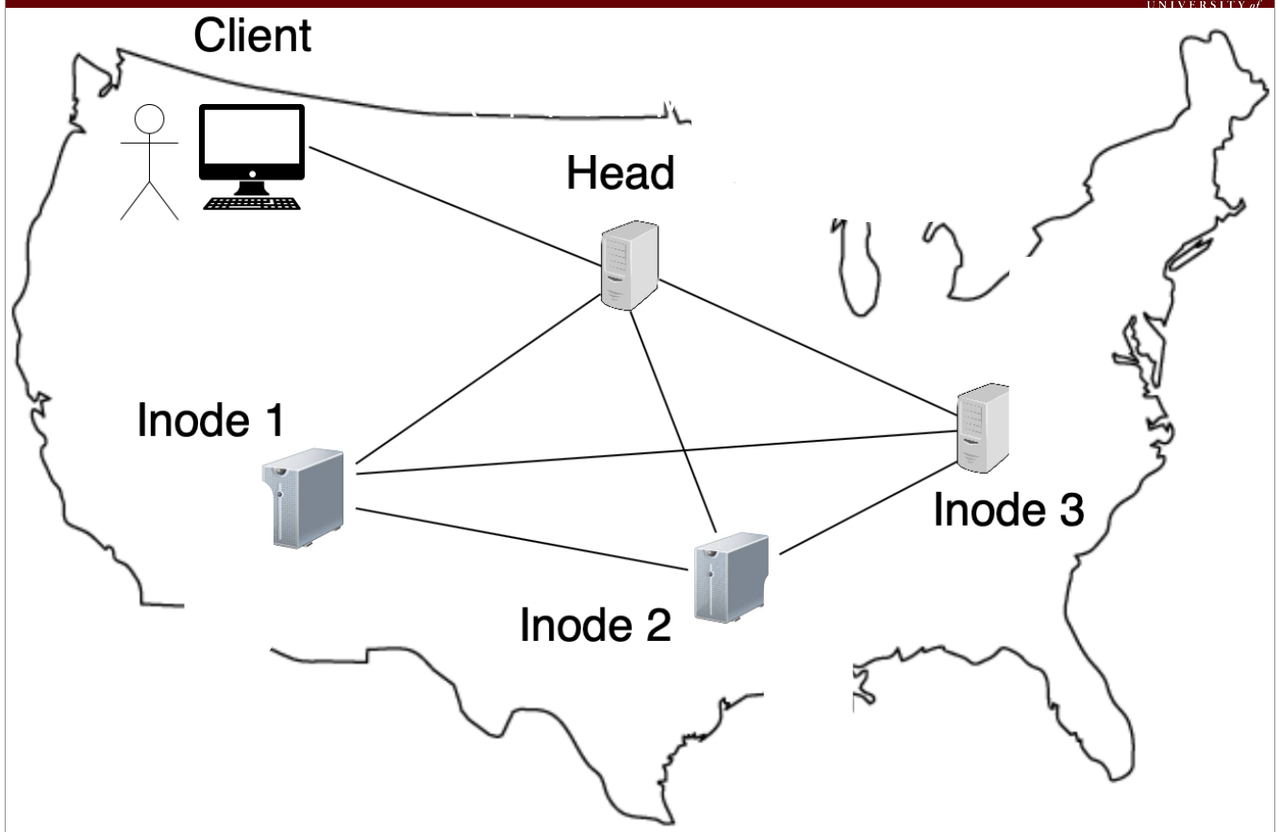

Client
Head
Inode 1
Inode 3
Inode 2

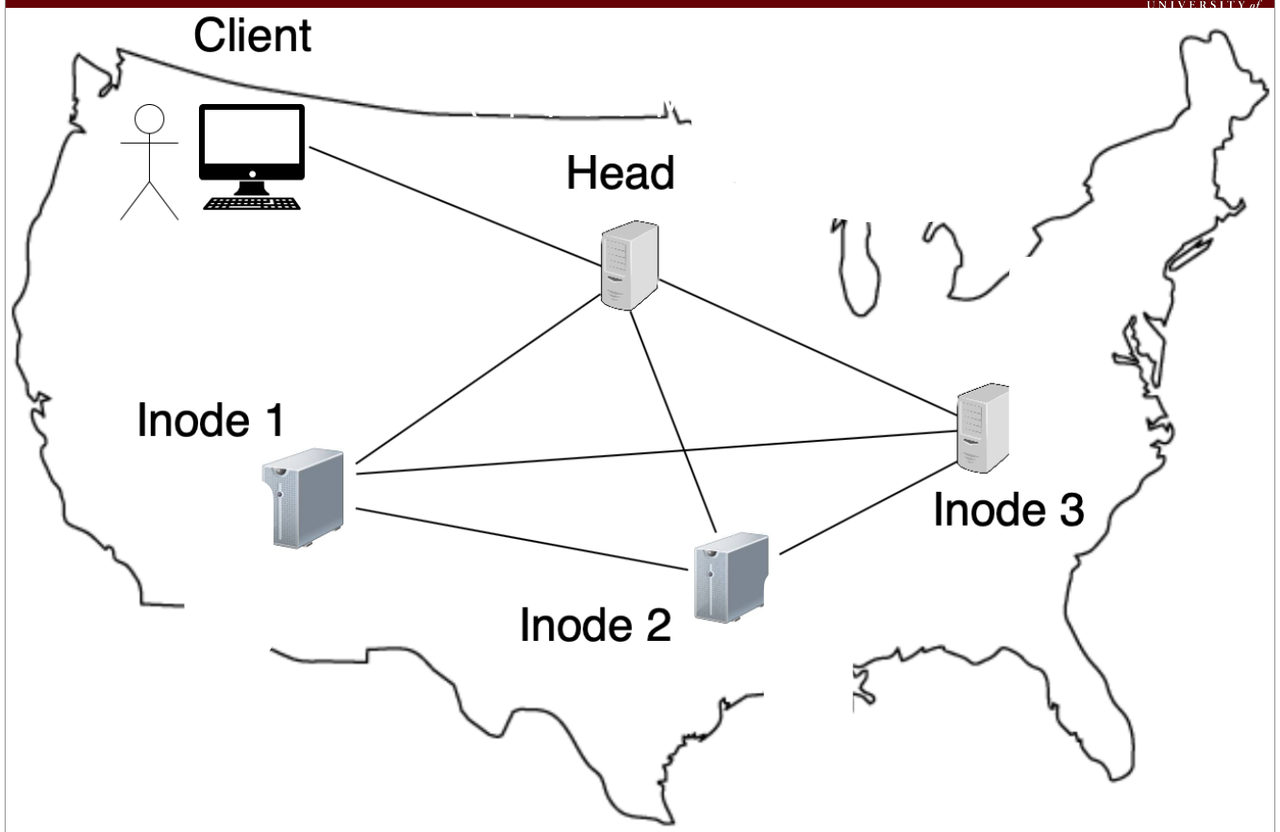

Client
Head
Inode 1
Inode 3
Inode 2

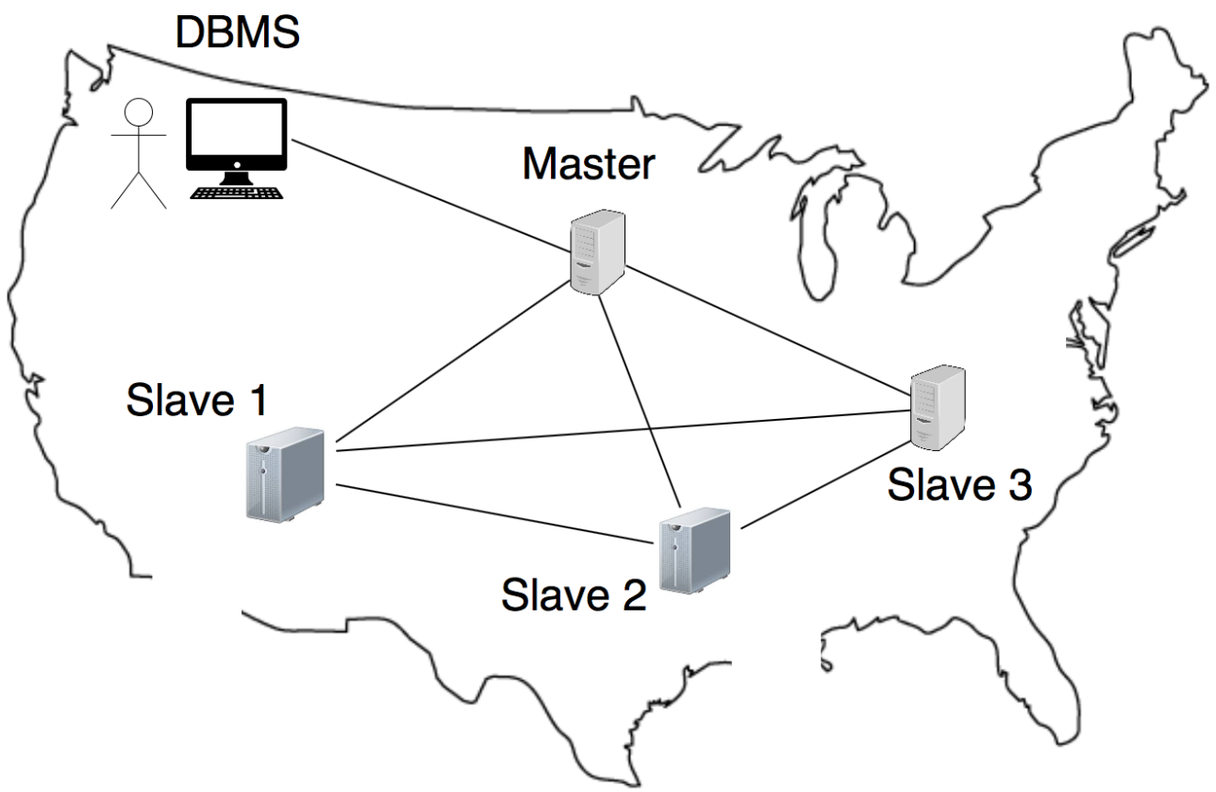

Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2

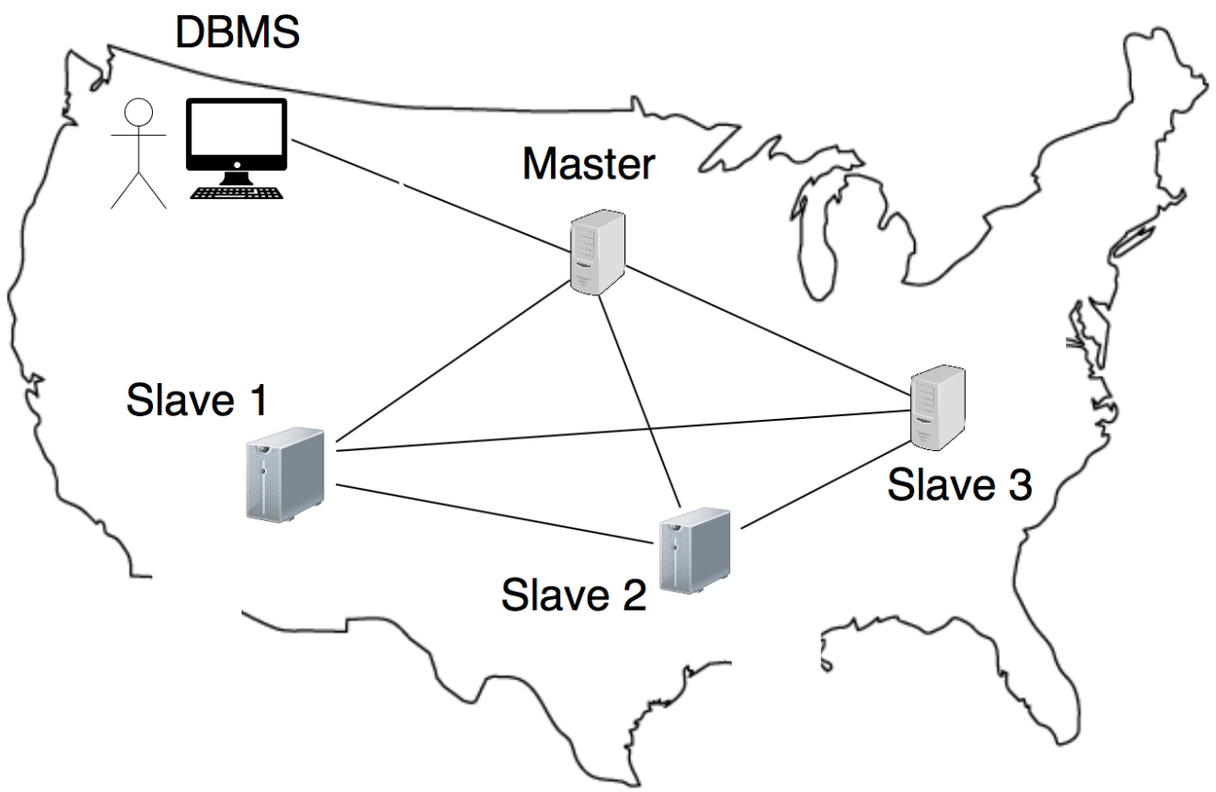

Client
Head
Inode 1
Inode 3
Inode 2

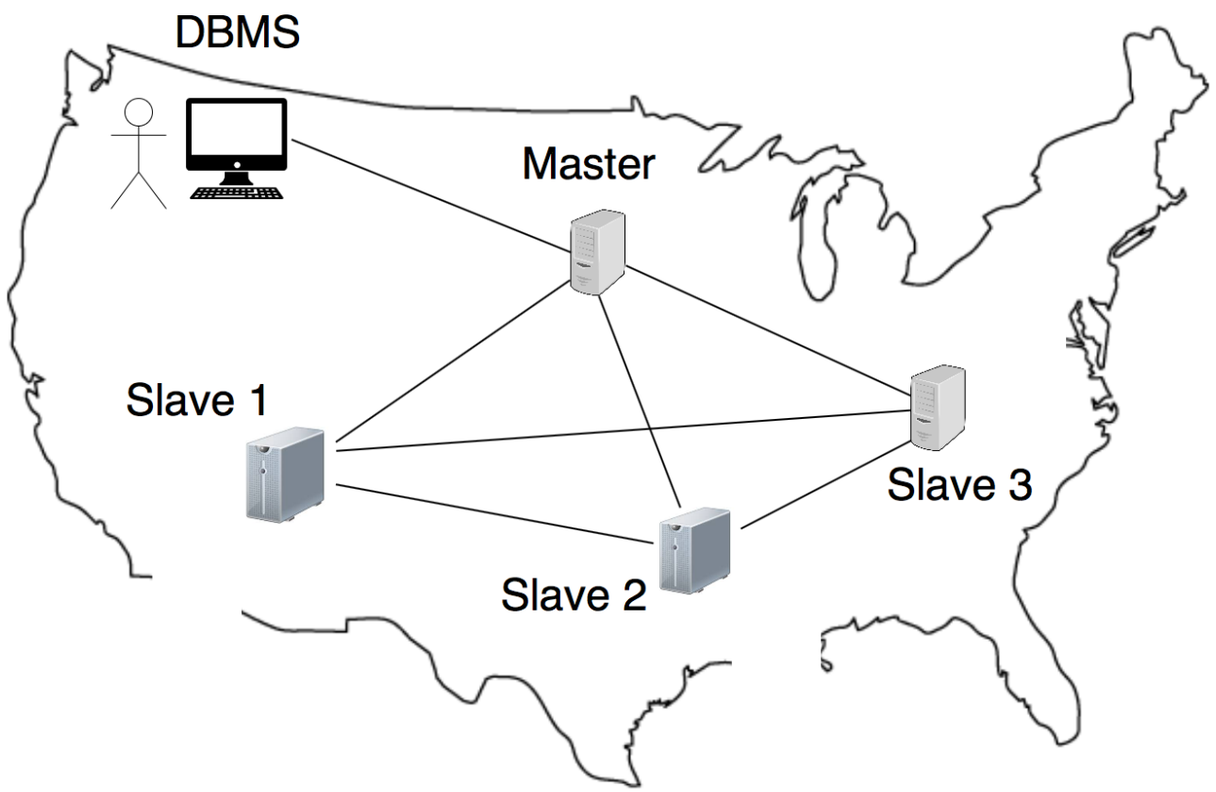

Client
Head
Inode 1
Inode 3
Inode 2

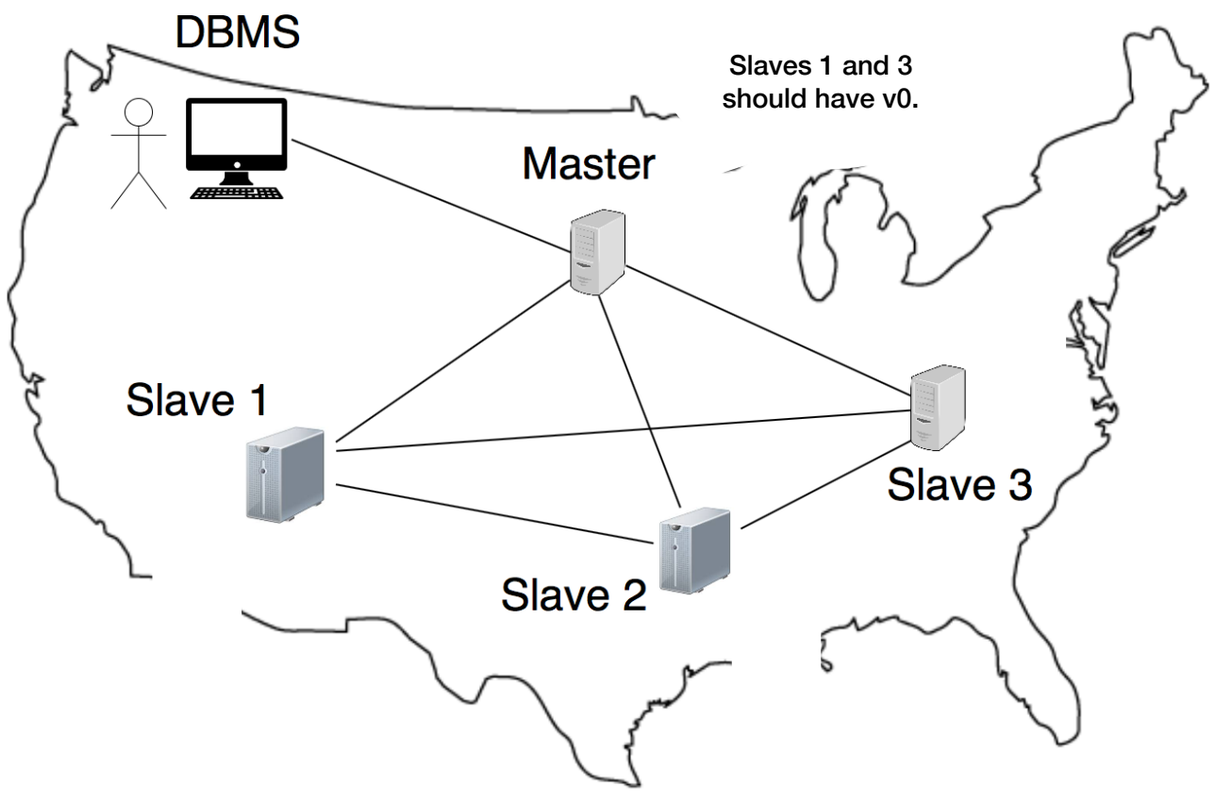

Client
Head
Inode 1
Inode 3
Inode 2

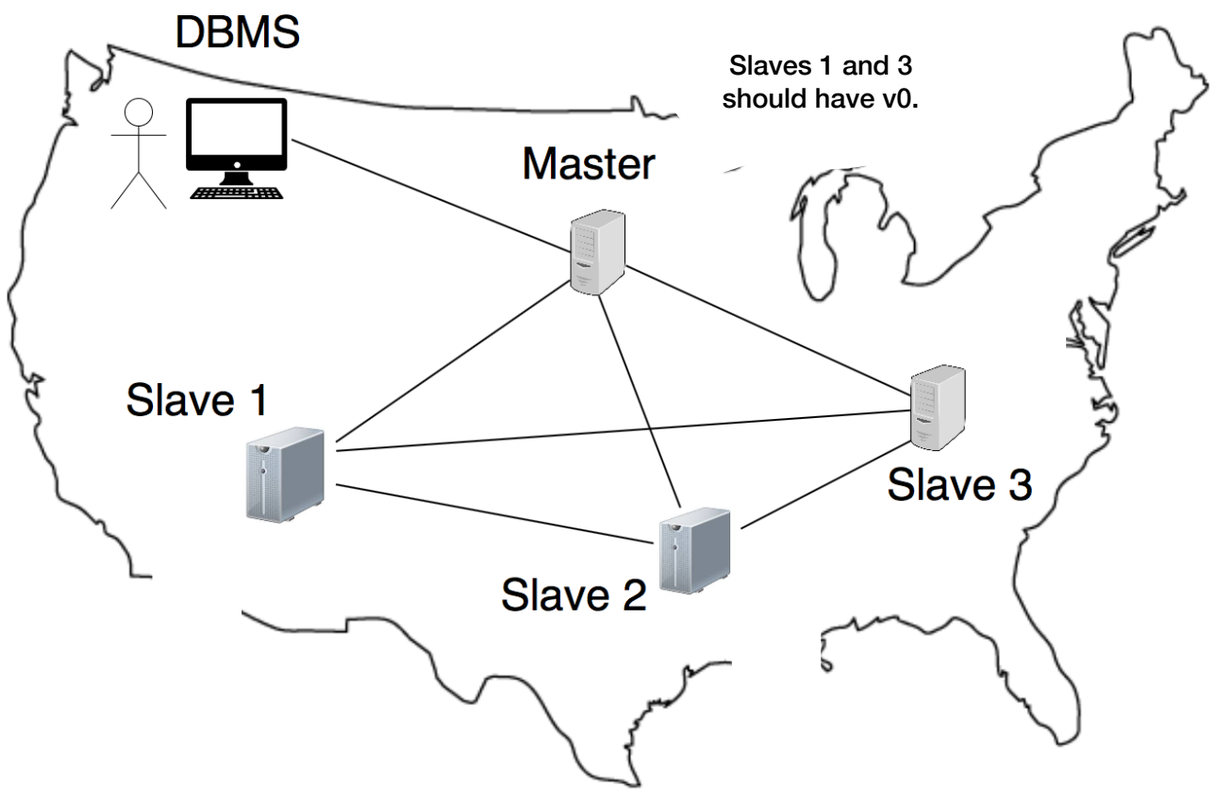


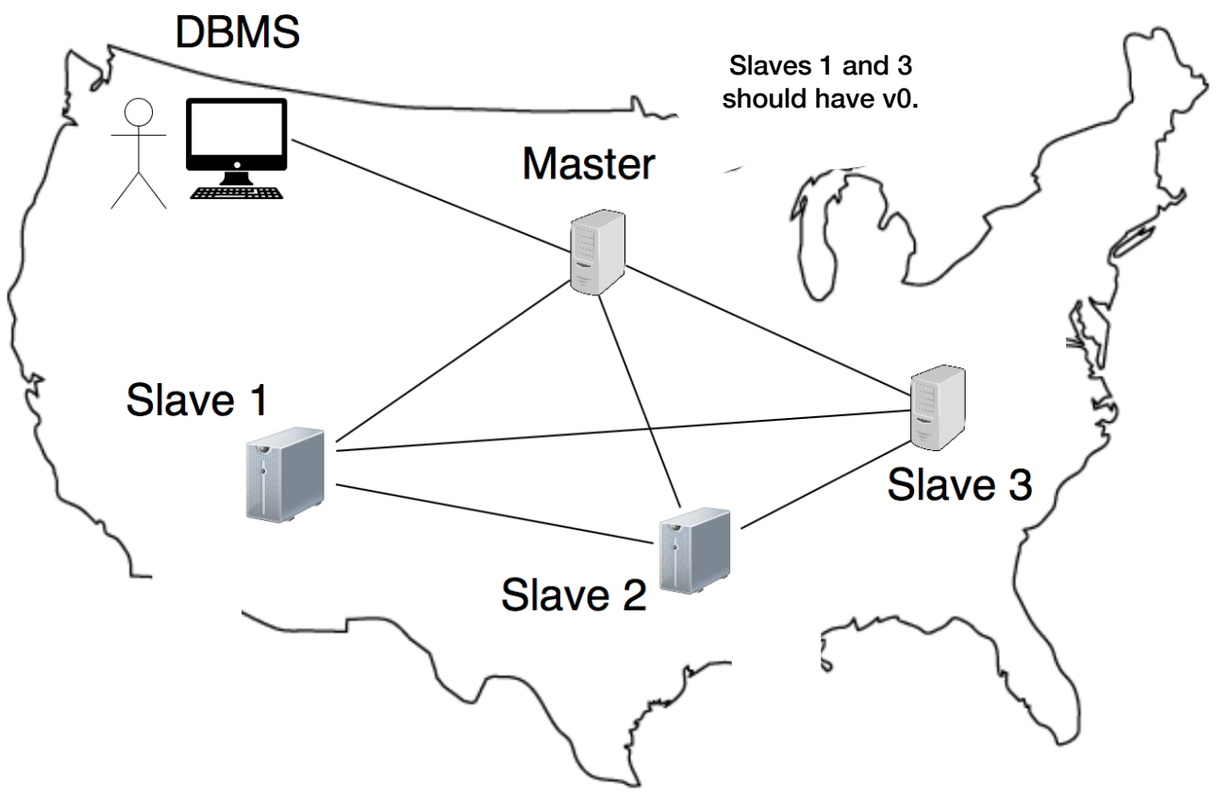
Jahrme:
Here we have a fairly simple system: we have a DBMS, a Master, and three Slaves.
The DBMS interfaces directly with the Master.
The Master and three Slaves are all able to communicate with each other.
When the DBMS sends a bitmap vector, the Master will decide which Slaves should have a copy of that vector.
For example, here v_0 is being sent to Slaves 1 and 3.
Once that has completed, the DBMS may send a second vector which the Master will distribute to the appropriate Slaves.
This vector distribution will occur any time the DBMS sends a new vector, or if it updates a vector that is already present.

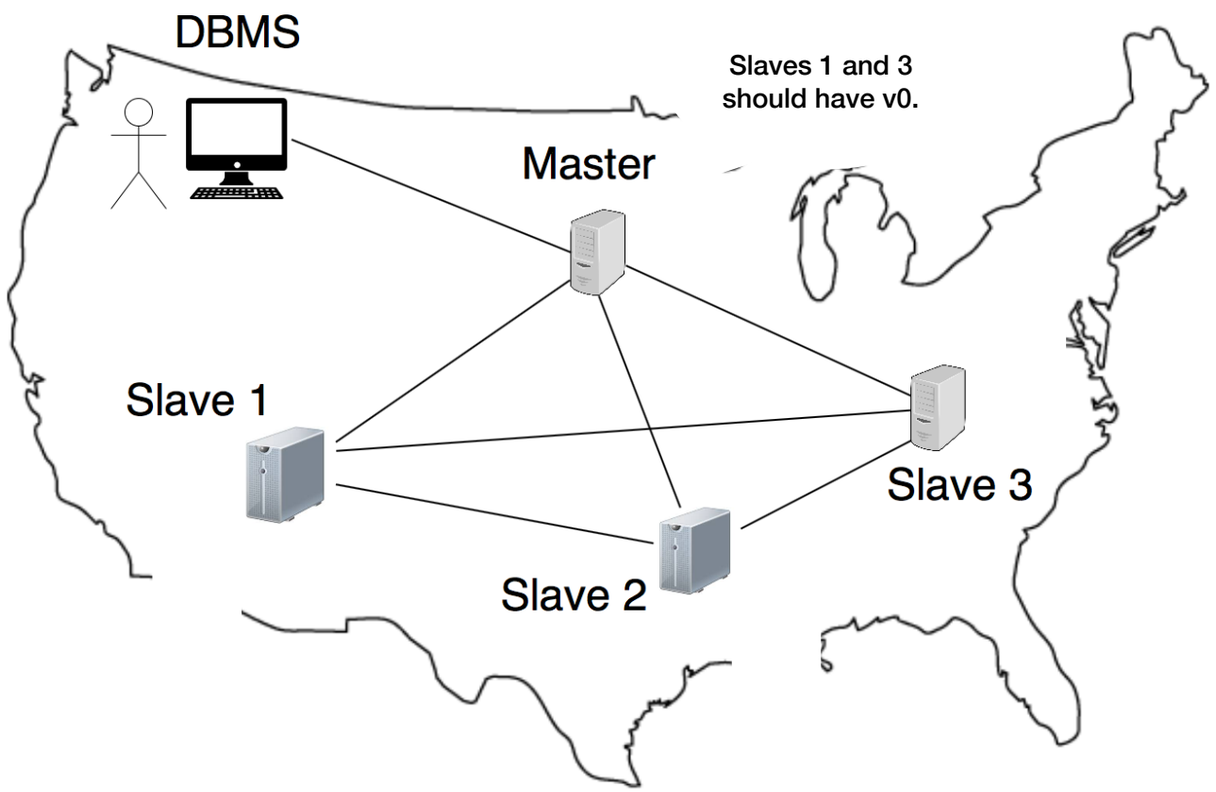

DBMS
Master
Slave 1
Slave 3
Slave 2

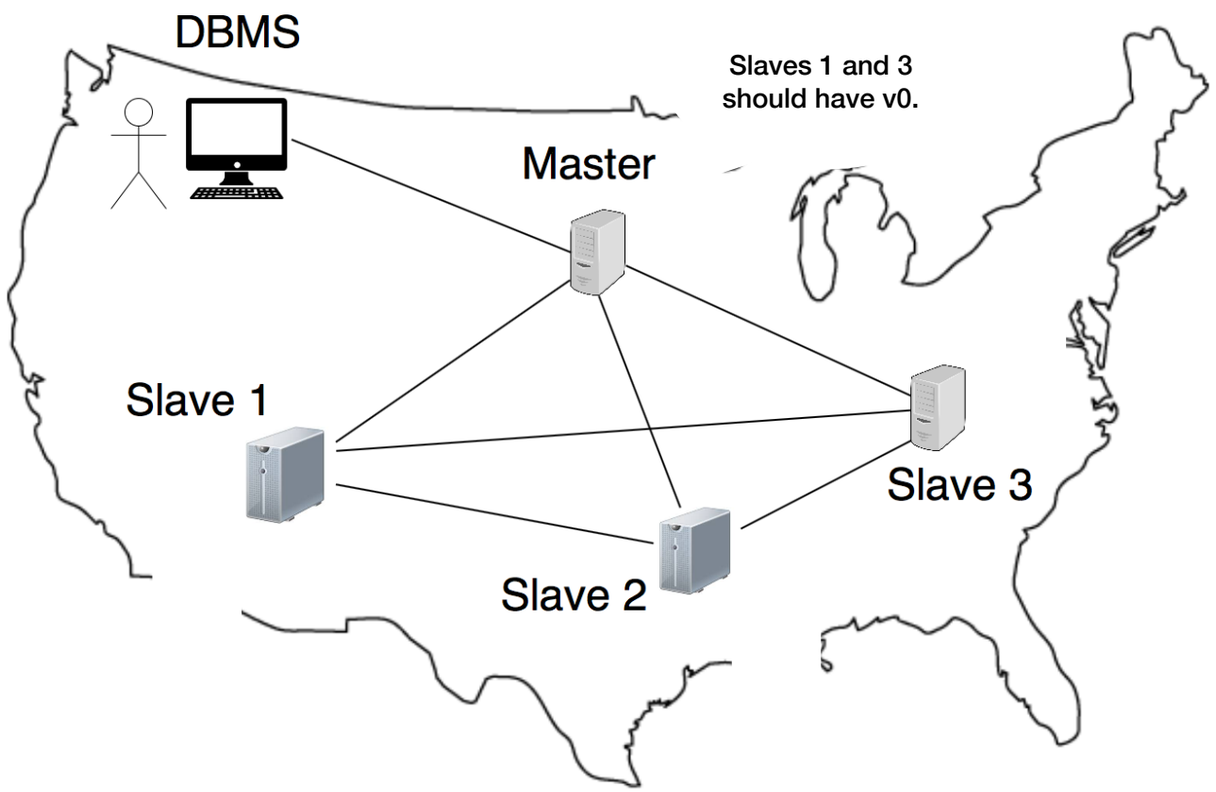

DBMS
Master
Slave 1
Slave 3
Slave 2

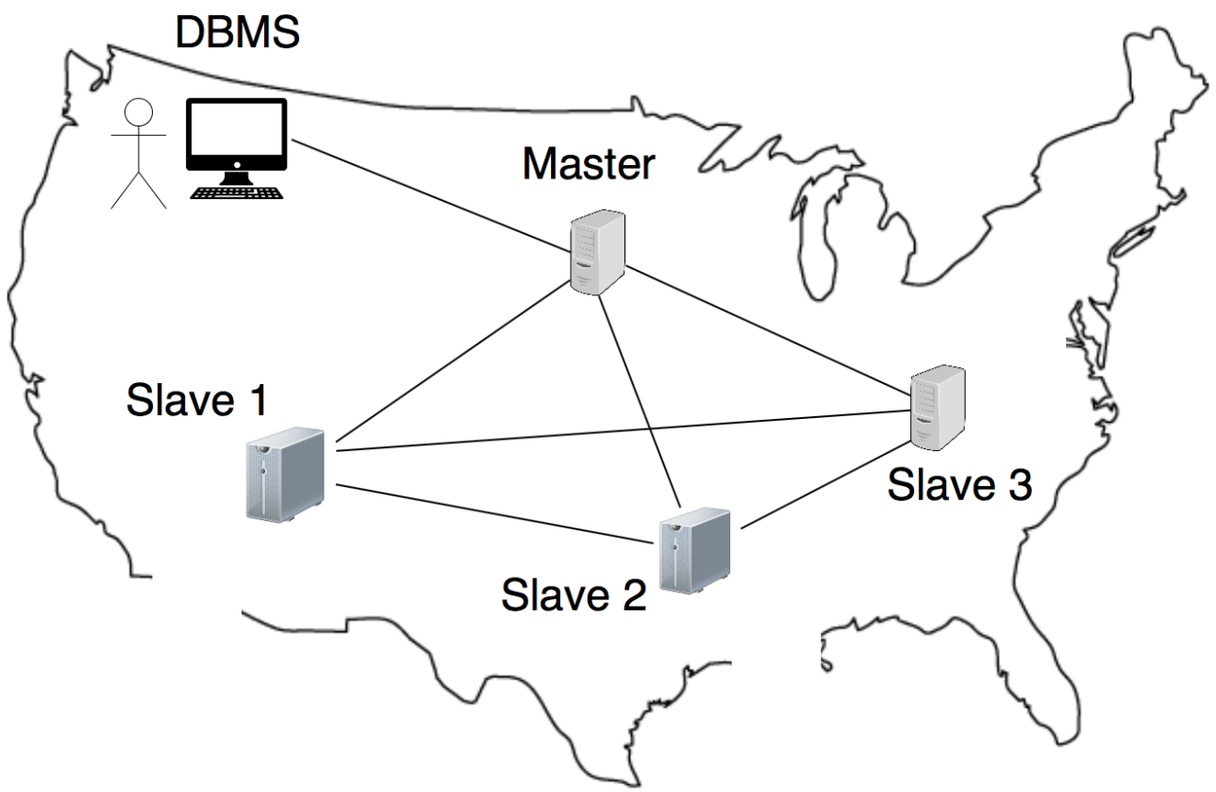

DBMS
Slaves 1 and 3
should have v0.
Master
Slave 1
Slave 3
Slave 2

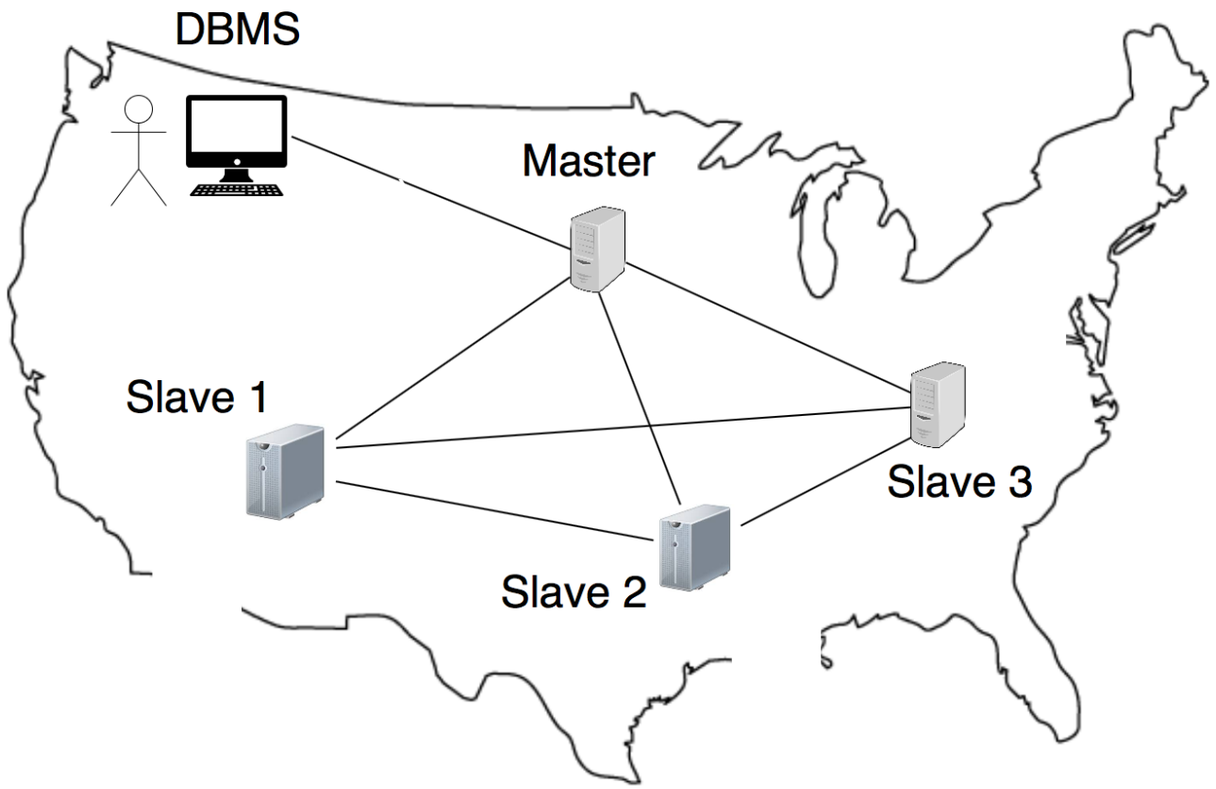

DBMS
Slaves 1 and 3
should have v0.
Master
Slave 1
Slave 3
Slave 2

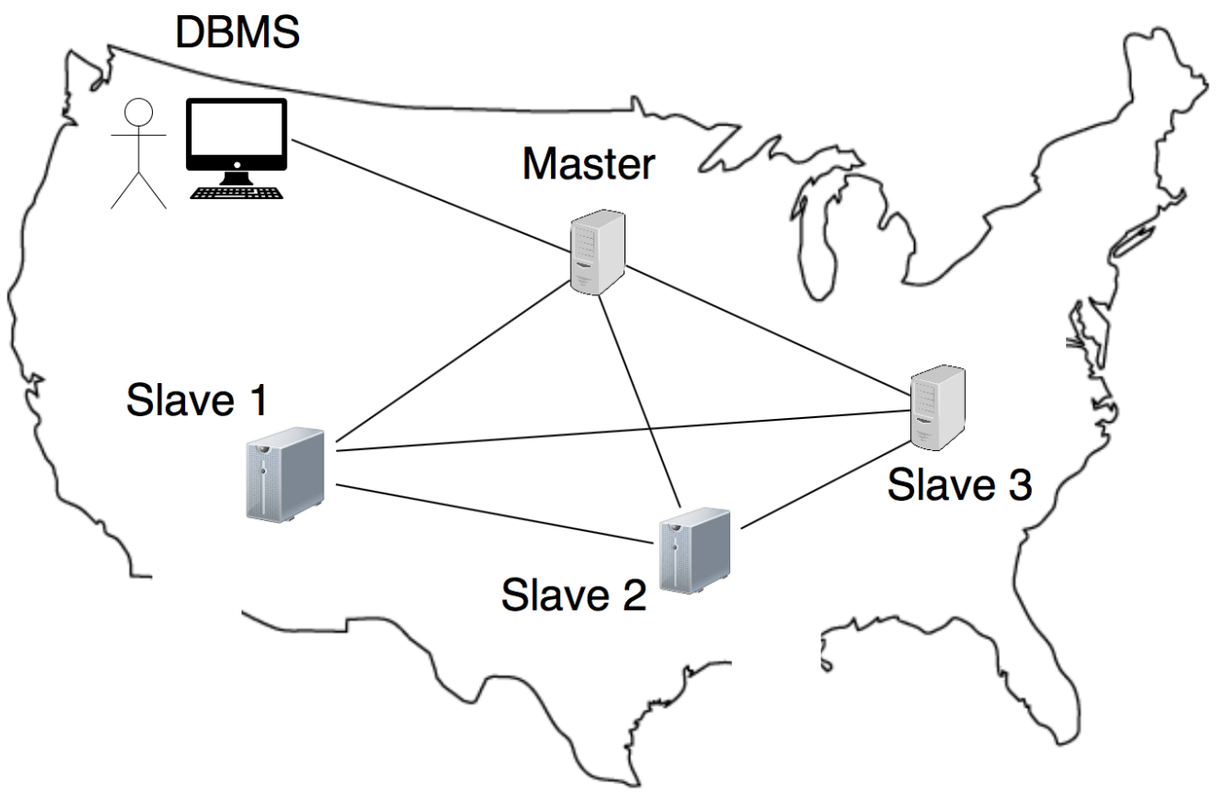

DBMS
Slaves 1 and 3
should have v0.
Master
Slave 1
Slave 3
Slave 2

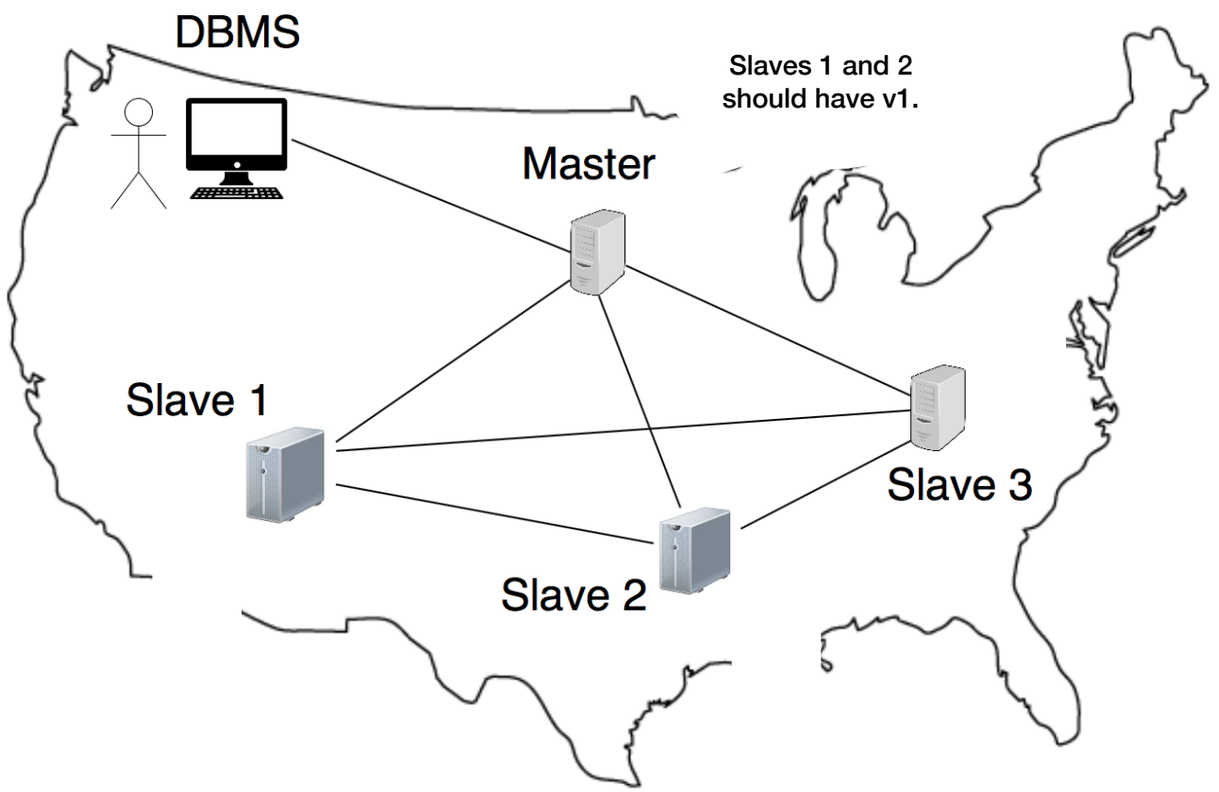

DBMS
Slaves 1 and 3
should have v0.
Master
Slave 1
Slave 3
Slave 2

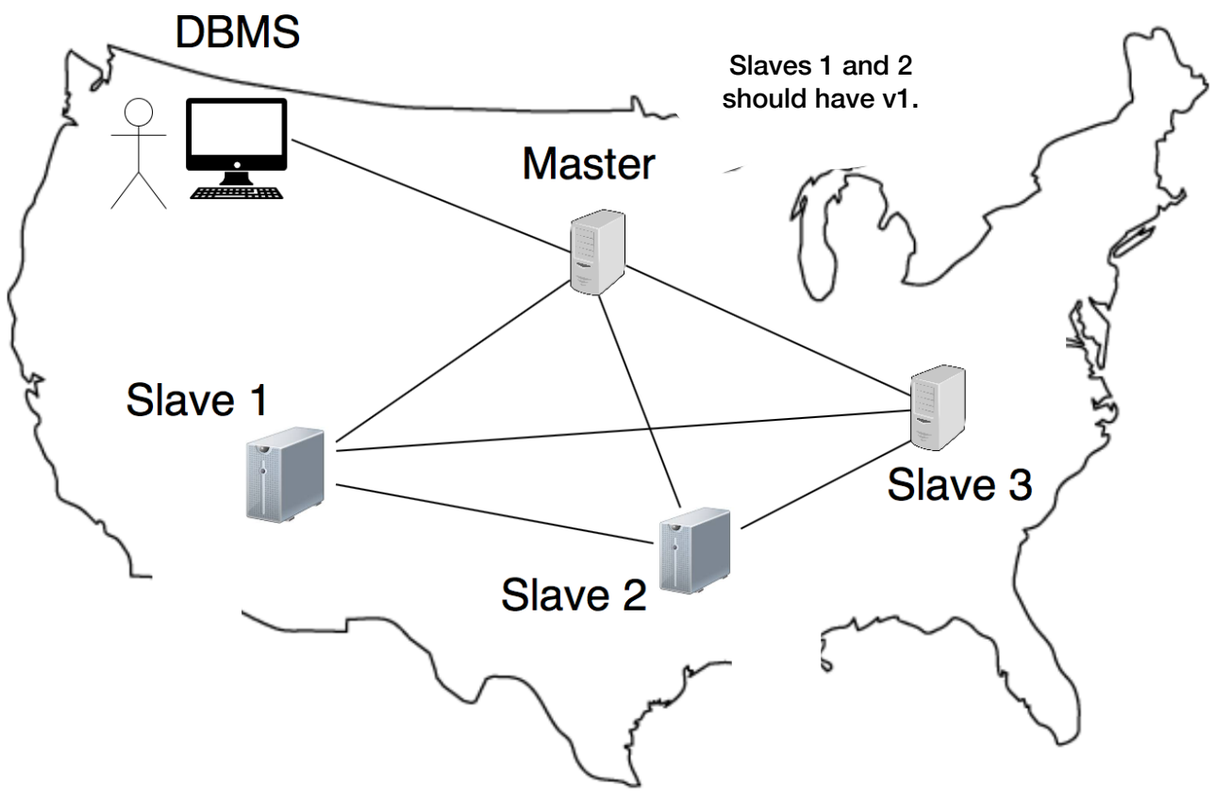

DBMS
Slaves 1 and 3
should have v0.
Master
Slave 1
Slave 3
Slave 2

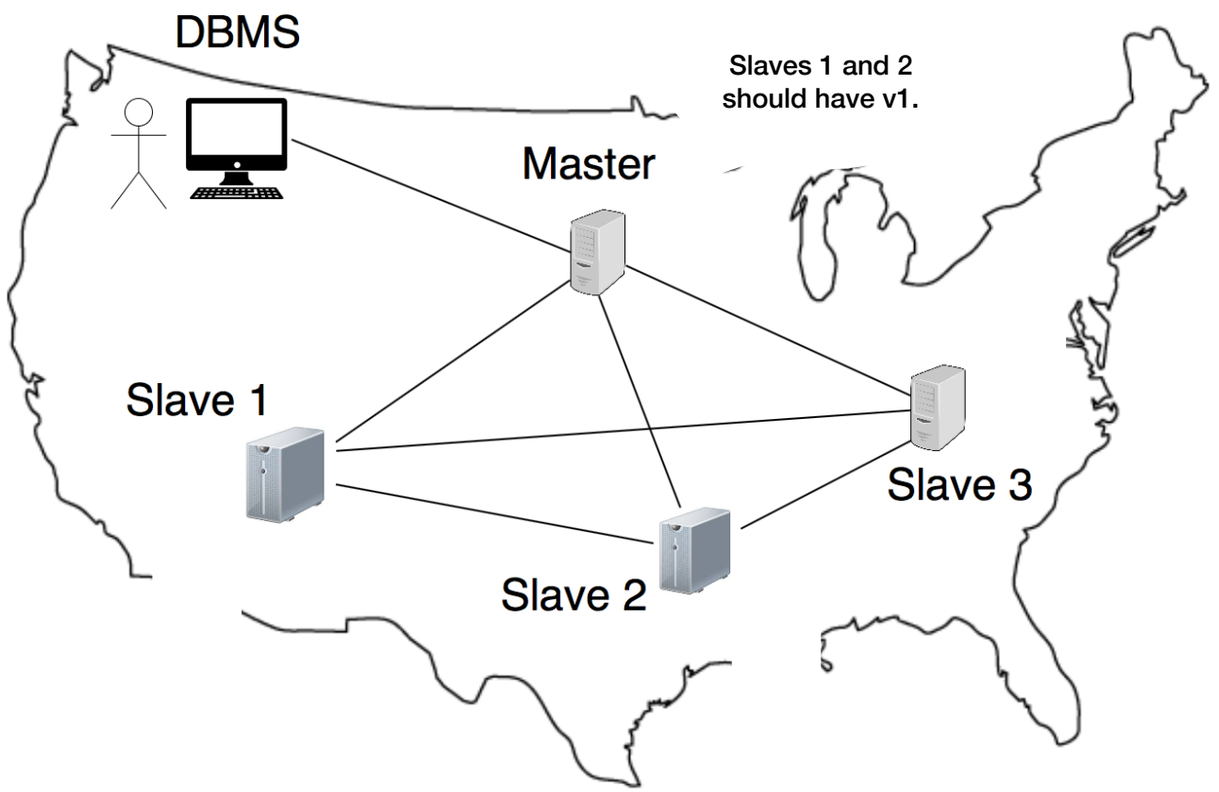

DBMS
Master
Slave 1
Slave 3
Slave 2

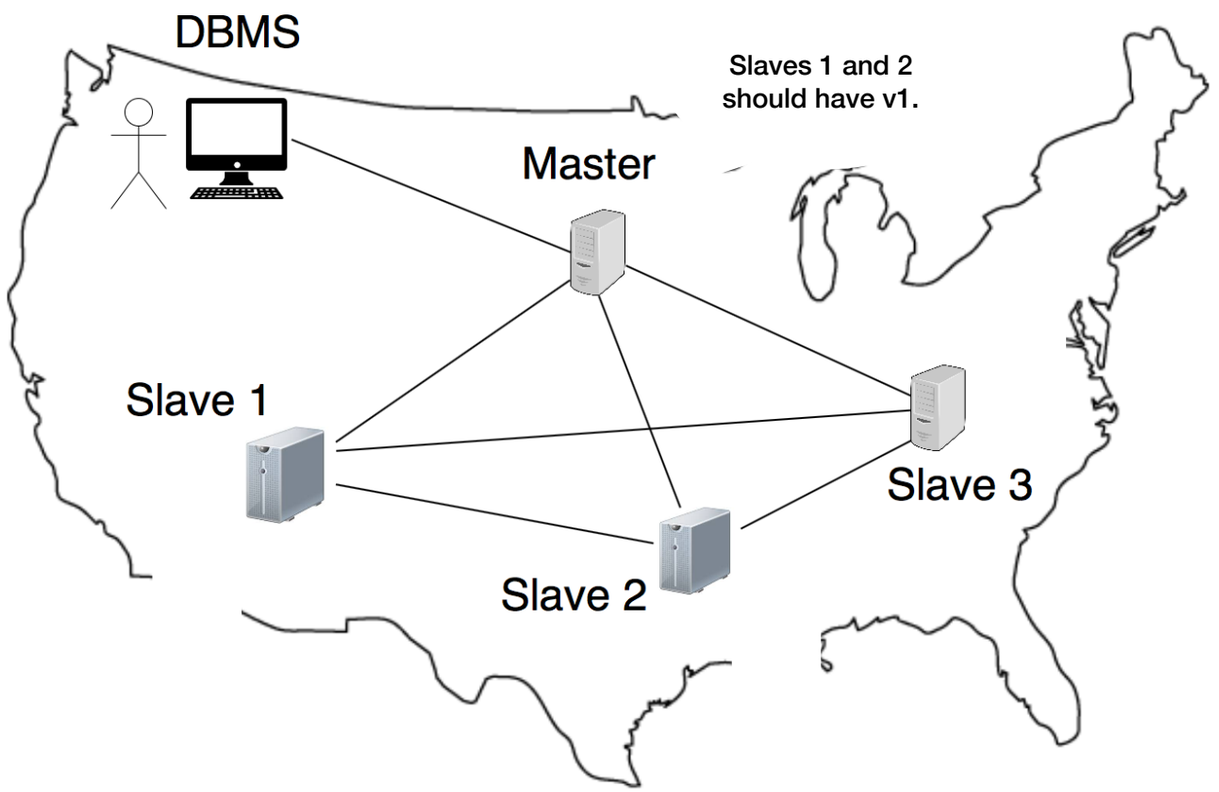

DBMS
Master
Slave 1
Slave 3
Slave 2

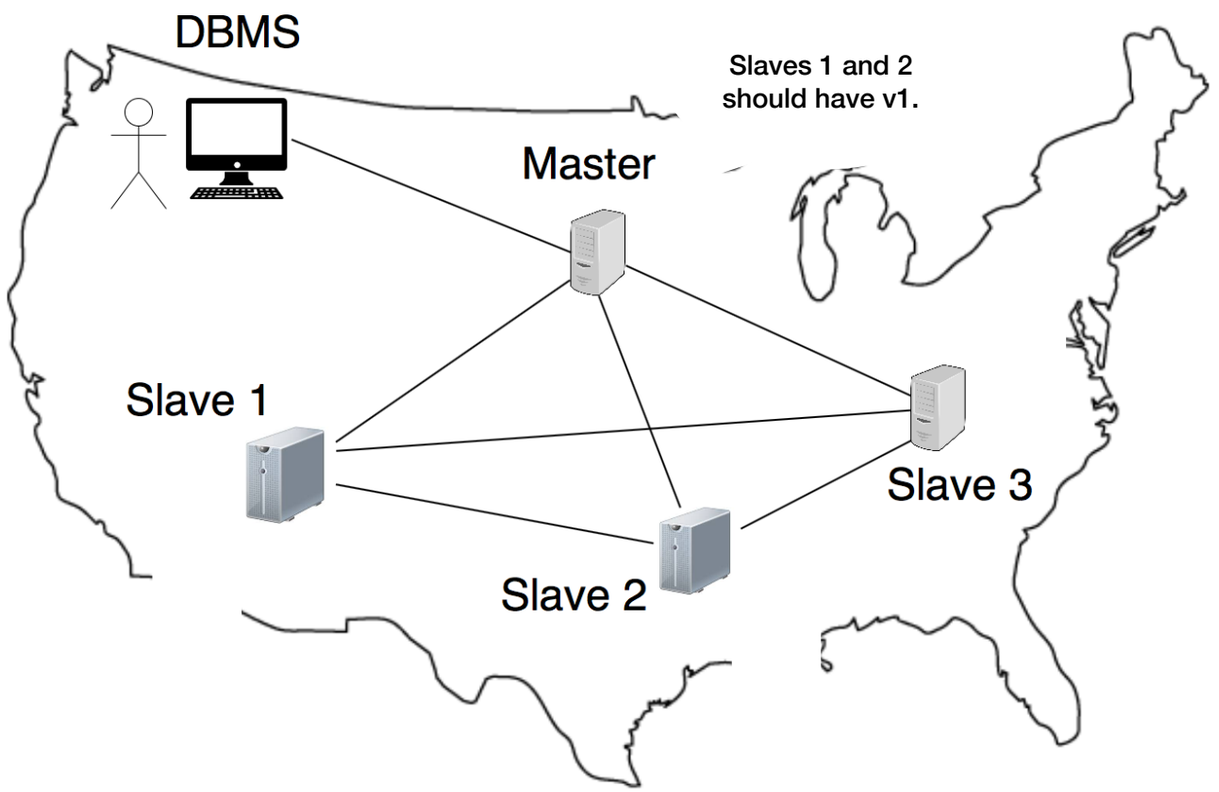

DBMS
Master
Slave 1
Slave 3
Slave 2

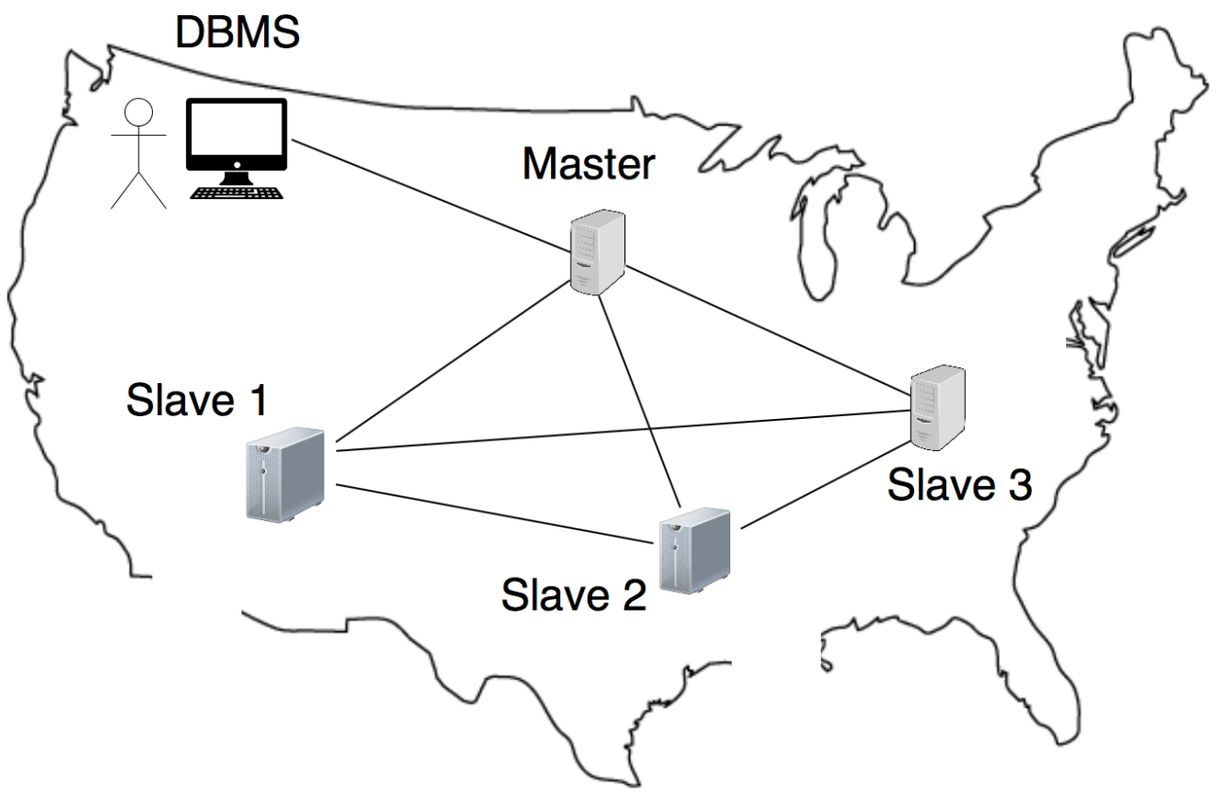

DBMS
Slaves 1 and 2
should have v1.
Master
Slave 1
Slave 3
Slave 2


DBMS
Slaves 1 and 2
should have v1.
Master
Slave 1
Slave 3
Slave 2


DBMS
Slaves 1 and 2
should have v1.
Master
Slave 1
Slave 3
Slave 2


DBMS
Slaves 1 and 2
should have v1.
Master
Slave 1
Slave 3
Slave 2


DBMS
Slaves 1 and 2
should have v1.
Master
Slave 1
Slave 3
Slave 2


DBMS
Master
Slave 1
Slave 3
Slave 2

Implementation

- The core system is written in C, targeting Linux.
- RPCs are handled by the Sun ONC+ library.
- For simplicity, we assume the DBMS is on the same node as the master.
  - DBMS–Master communication is handled by Unix System V queues.
- Query plan optimizers are written in Python and called by the master.
- Vectors are compressed and queried using Word-Aligned Hybrid (WAH).
  - Ours is the first distributed bitmap system to use WAH.

Fault Tolerance

Sam: When our system experiences a fault, such as the death of one of its slaves, the system should be re-adjusted such that each vector is still replicated on two different machines

How do we decide where a given vector belongs?

Jahrme:
So, with all of these vectors being sent to different Slaves, how does the Master actually decide where a vector should go?
Further, if that vector is needed for a query later on, how does the Master figure out where that vector is located?

Two-Phase Commit

- Whenever master put a vector on two different machines, it does not just send it to both.
- Instead, it performs two-phase-commit (TPC).
  - Phase 1: Ensure that both machines are online.
  - Phase 2:
    - If they are both online, upload the vector to both.
    - If either is unavailable, handle the failed slave.
      Then attempt TPC with the new pair of slaves.

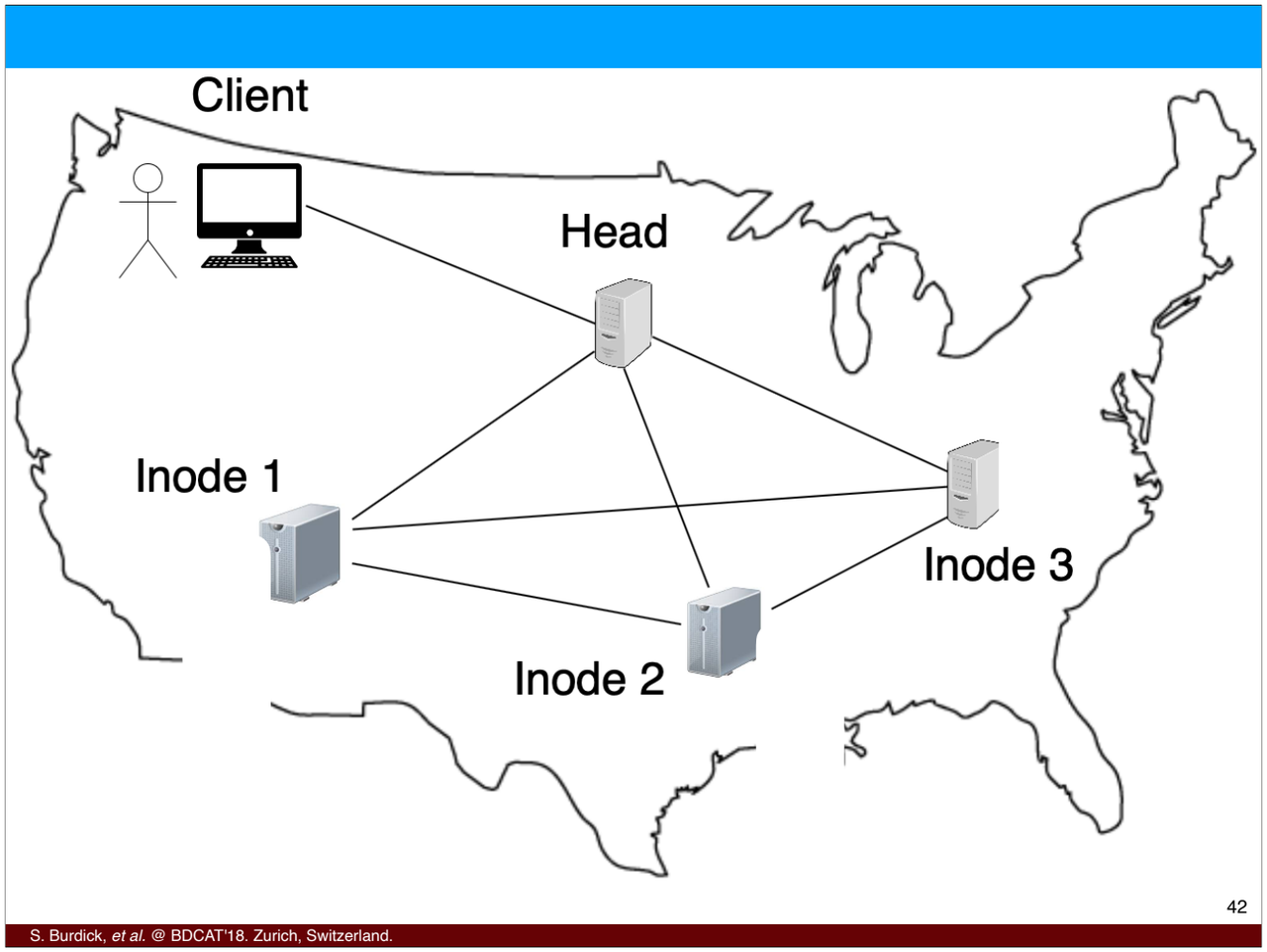
Database management system

- Databases and their indices are created and managed by a database management system (DBMS).
- Typically, the DBMS runs on a single machine.
  - What if we split the index among multiple machines?
  - What if we could use those machines to execute the bitmap queries?

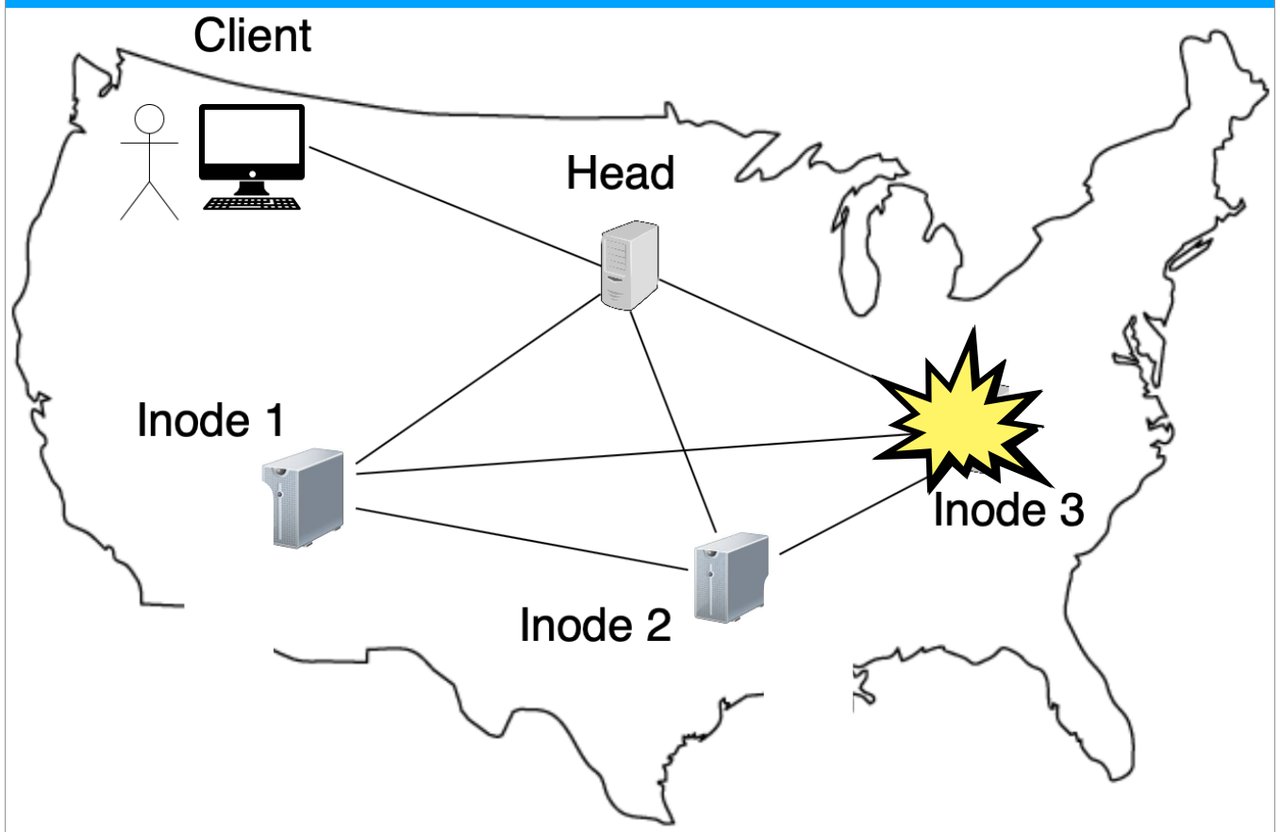
Sam:
Databases and their indices are created and managed by a program called a database management system, or DBMS. Typically, you'll run a DBMS on a single machine, and by default your index will be on the same machine.
The question we're interested in answering is: what if we partition the index vectors among multiple machines?
Furthermore, what if we could make those machines execute bitmap queries for us?

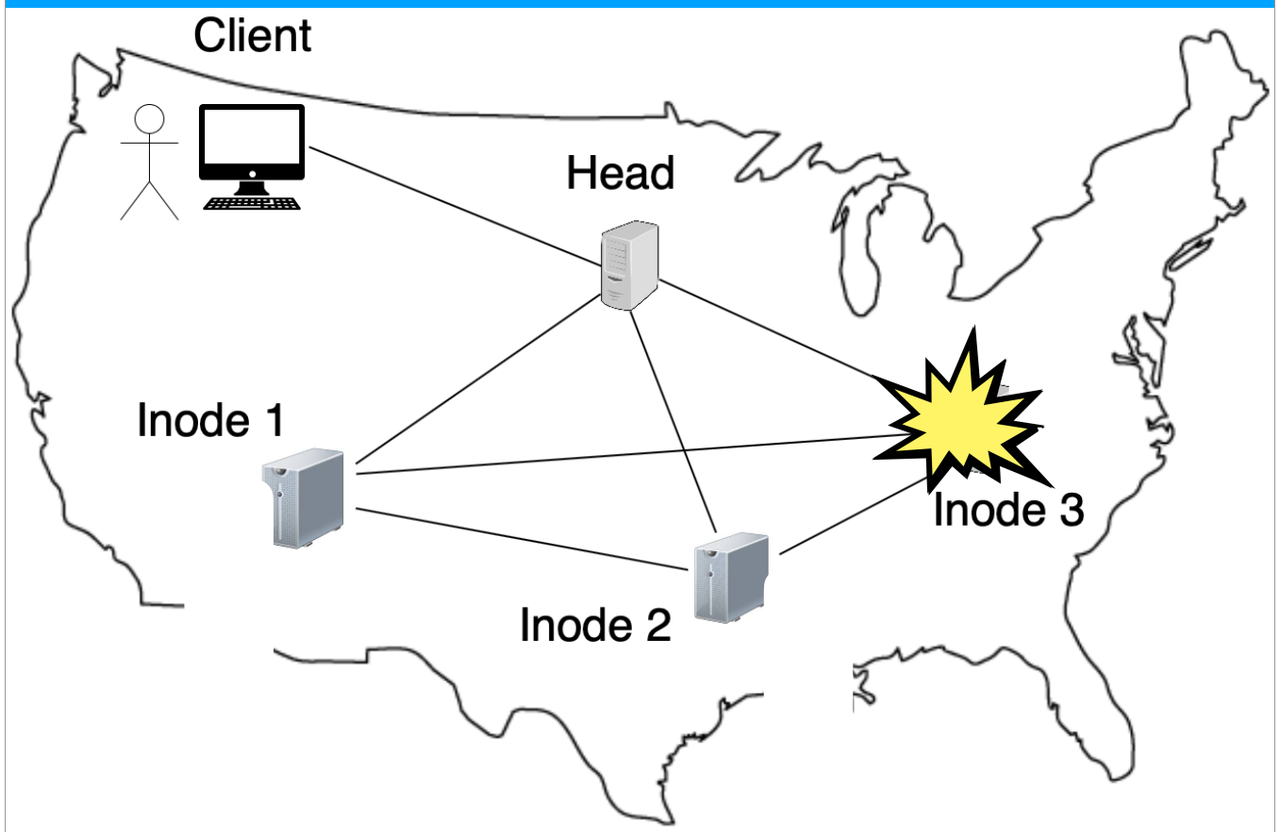
Benefit of consistent hashing

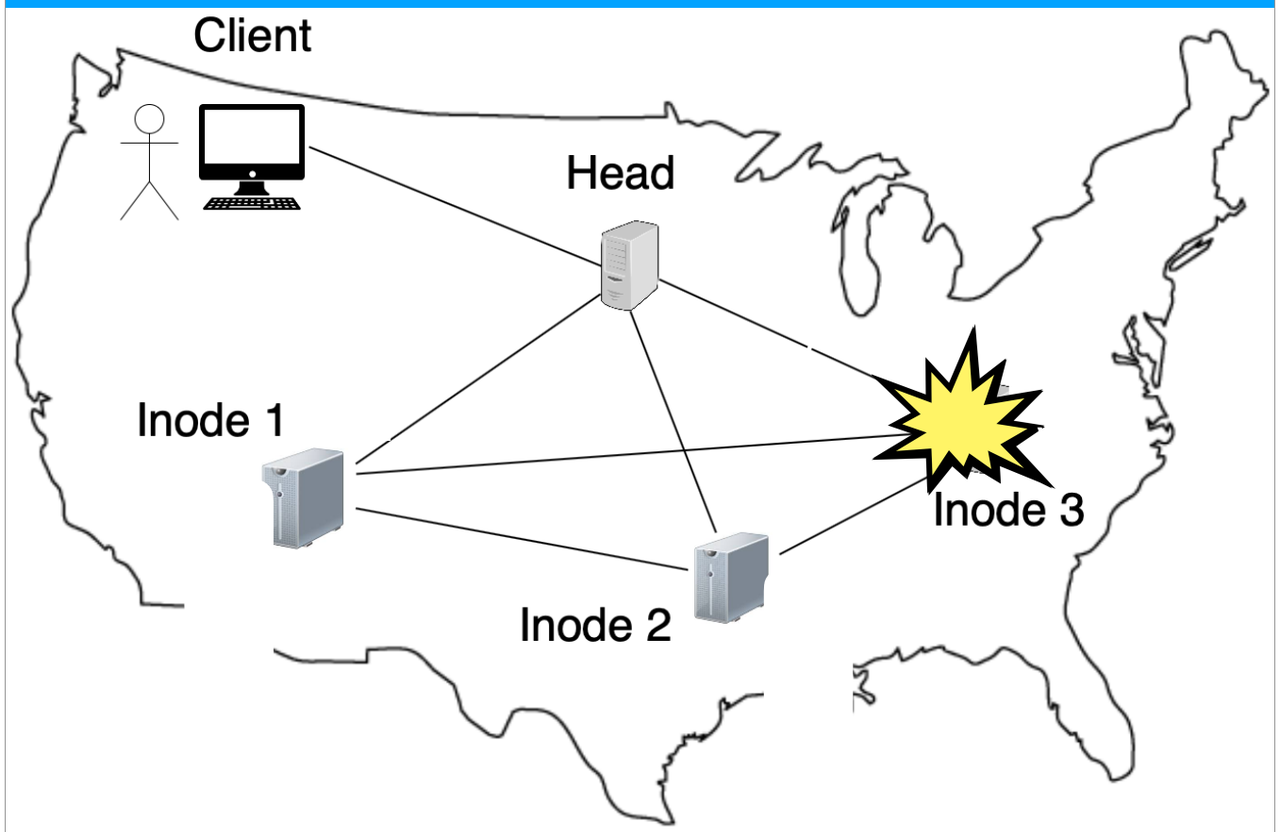
Sam:
To see the benefit of consistent hashing, let's compare the two algorithms. We saw that consistent hashing needed 2 messages to reallocate vectors, while hash-mod-n needed 4. To get the slaves to move those vectors, the master had to ask 2 slaves in consistent hashing, and 3 slaves in hash-mod-n. So in total, consistent hashing required 4 messages, and hash-mod-n required 7. It can be shown that consistent hashing *always* requires fewer messages than hash-mod-n!

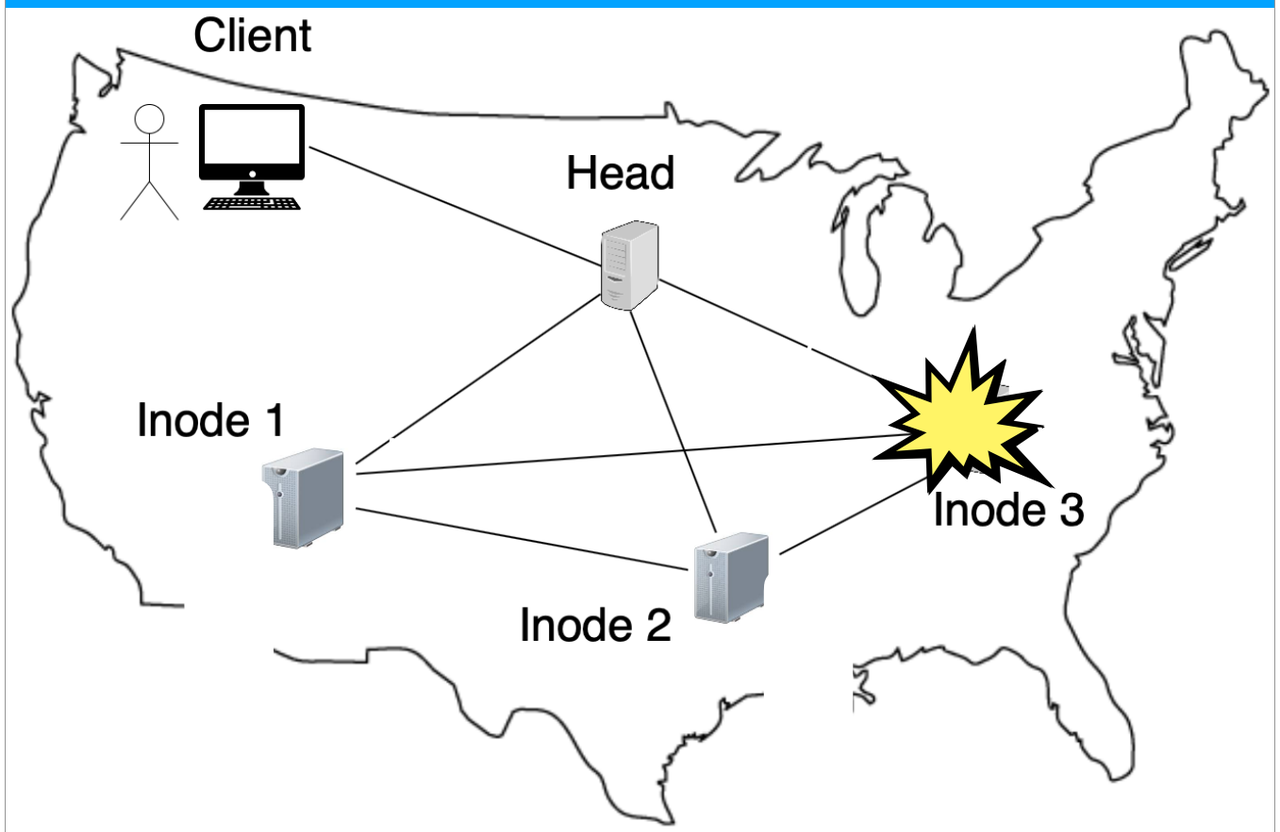


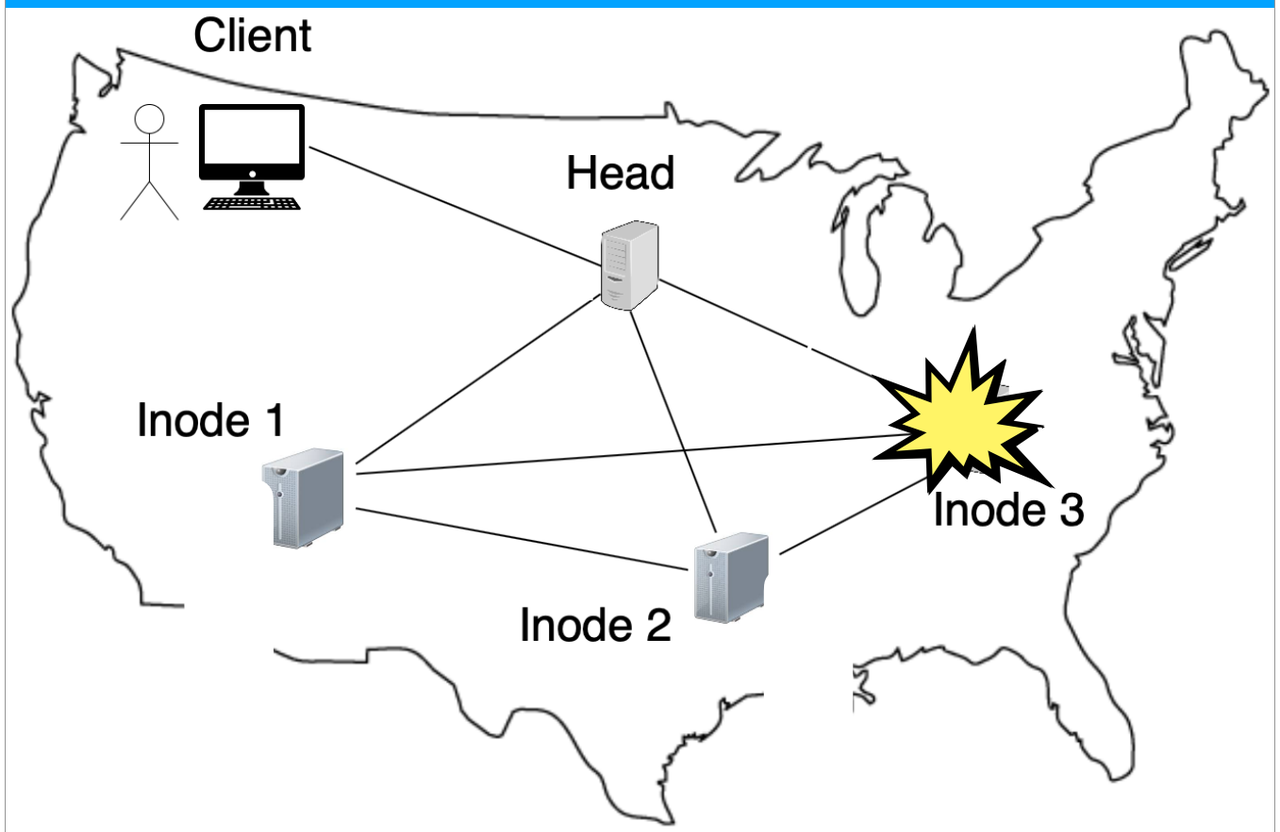
Sam: Suppose that slave 3 crashes. The master periodically checks to make sure that each slave is still alive. If it doesn't hear back from a slave it assumes that machine is dead. The master then orders the living slaves to send each other vectors such that every vector is replicated twice, in case another slave failed.

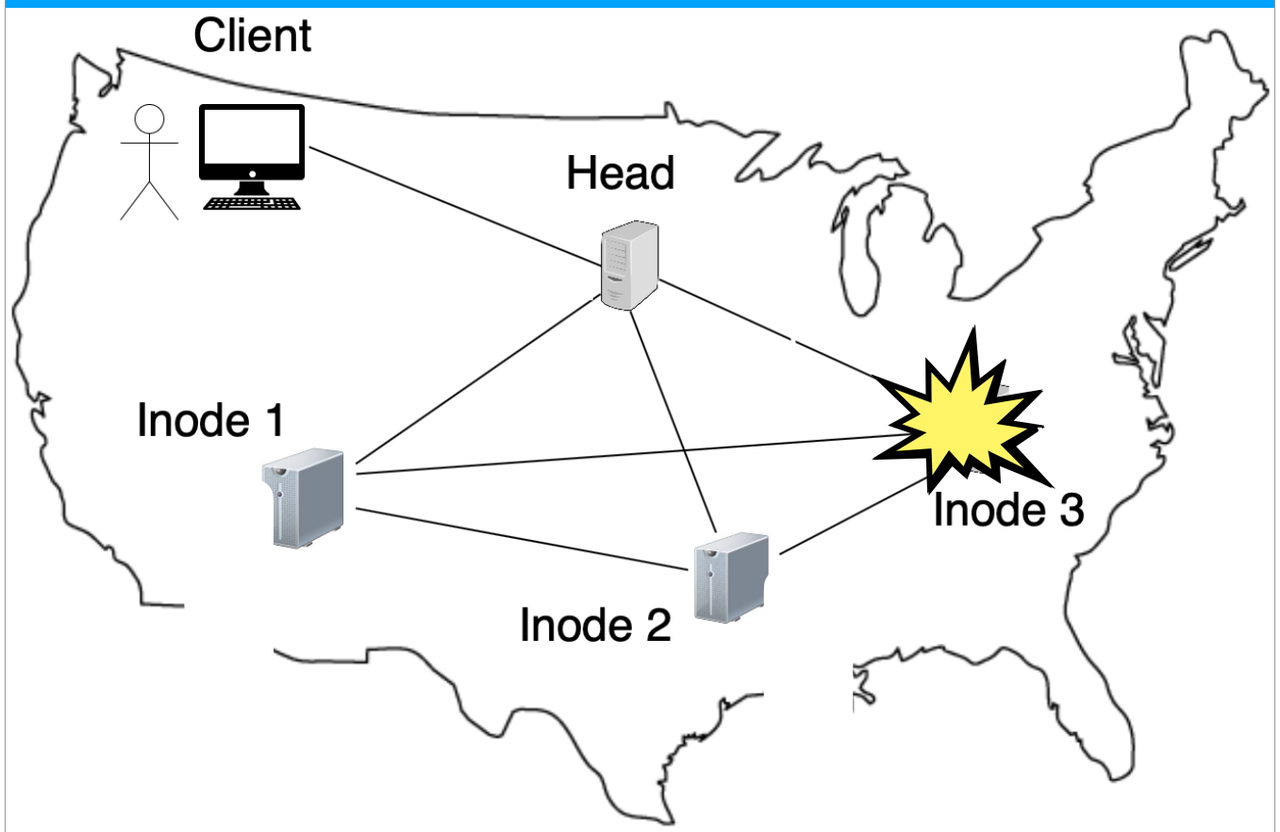

Client
Head
Inode 1
Inode 3
Inode 2

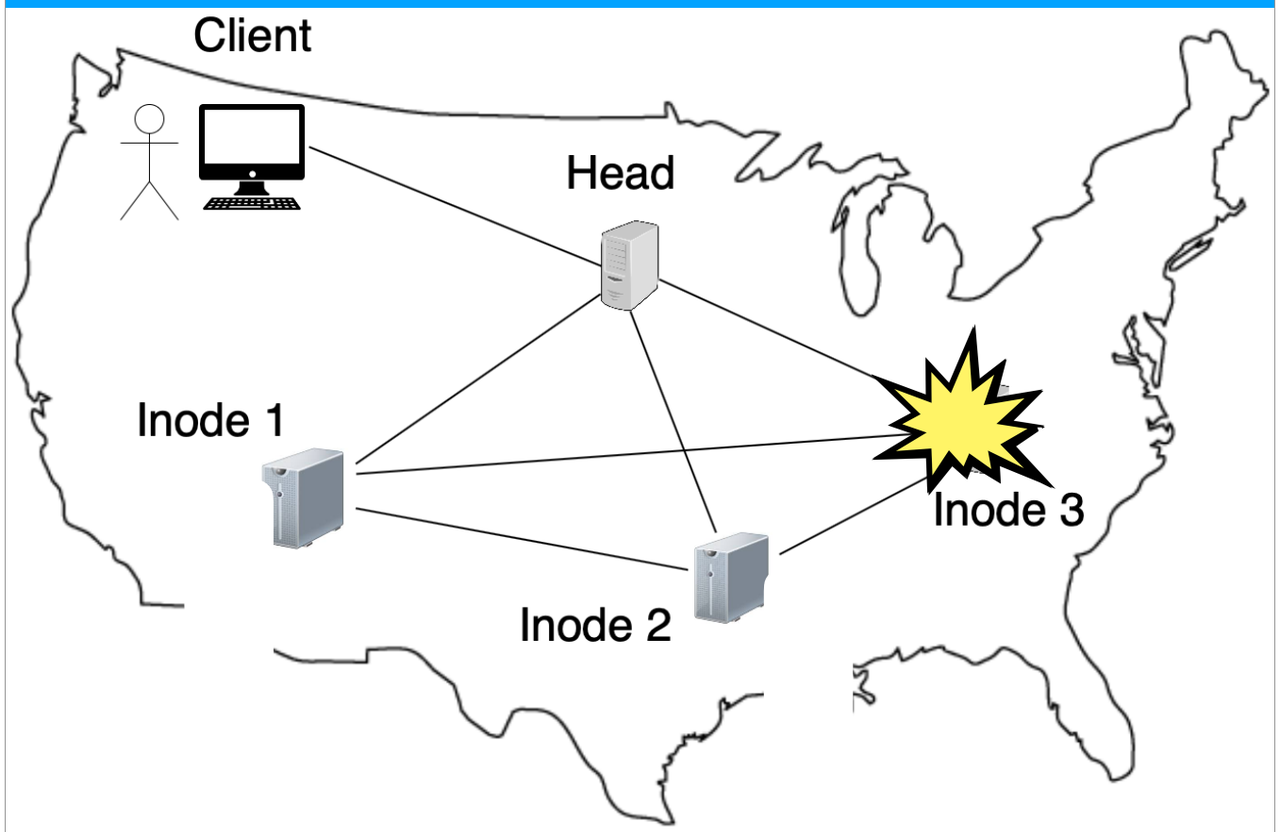

Client
Head
Inode 1
Inode 3
Inode 2

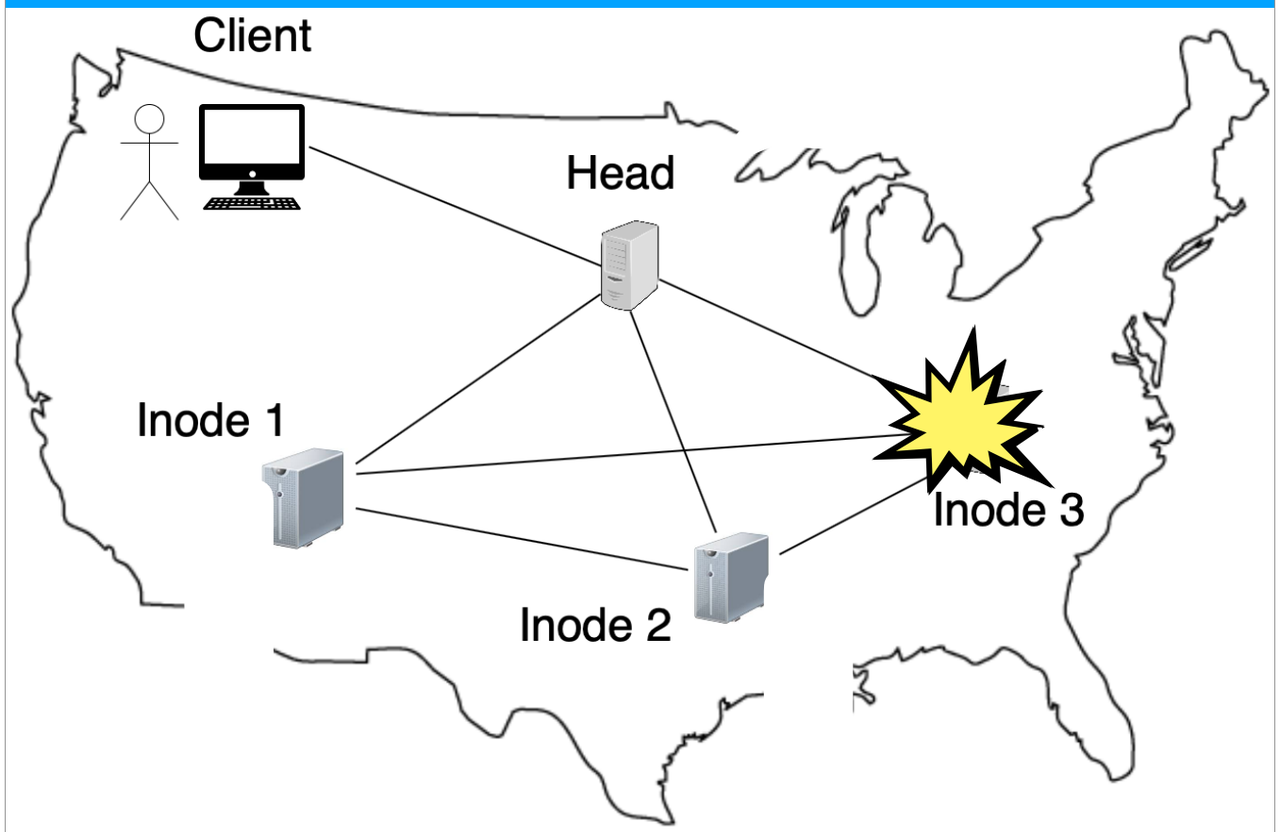

Client
Head
Inode 1
Inode 3
Inode 2

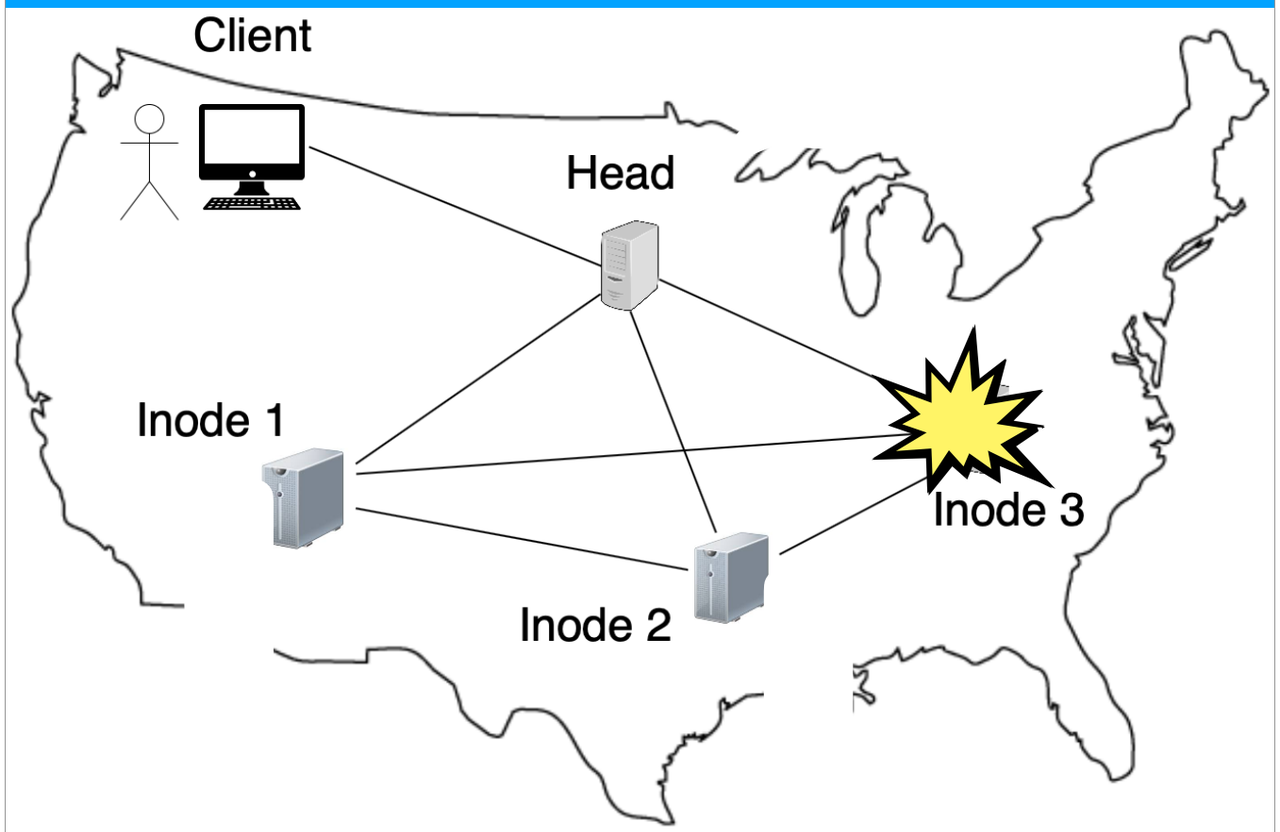

Client
Head
Inode 1
Inode 3
Inode 2

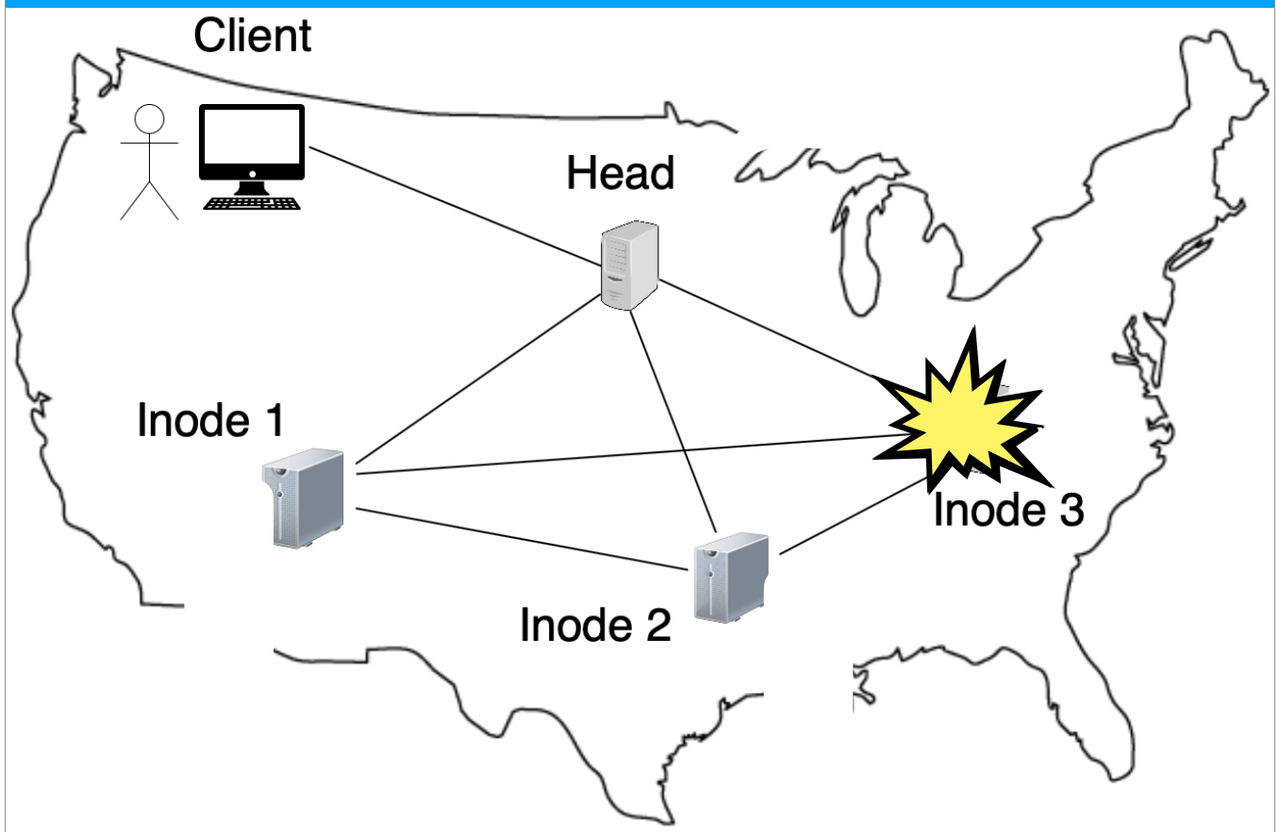

Client
Head
Inode 1
Inode 3
Inode 2

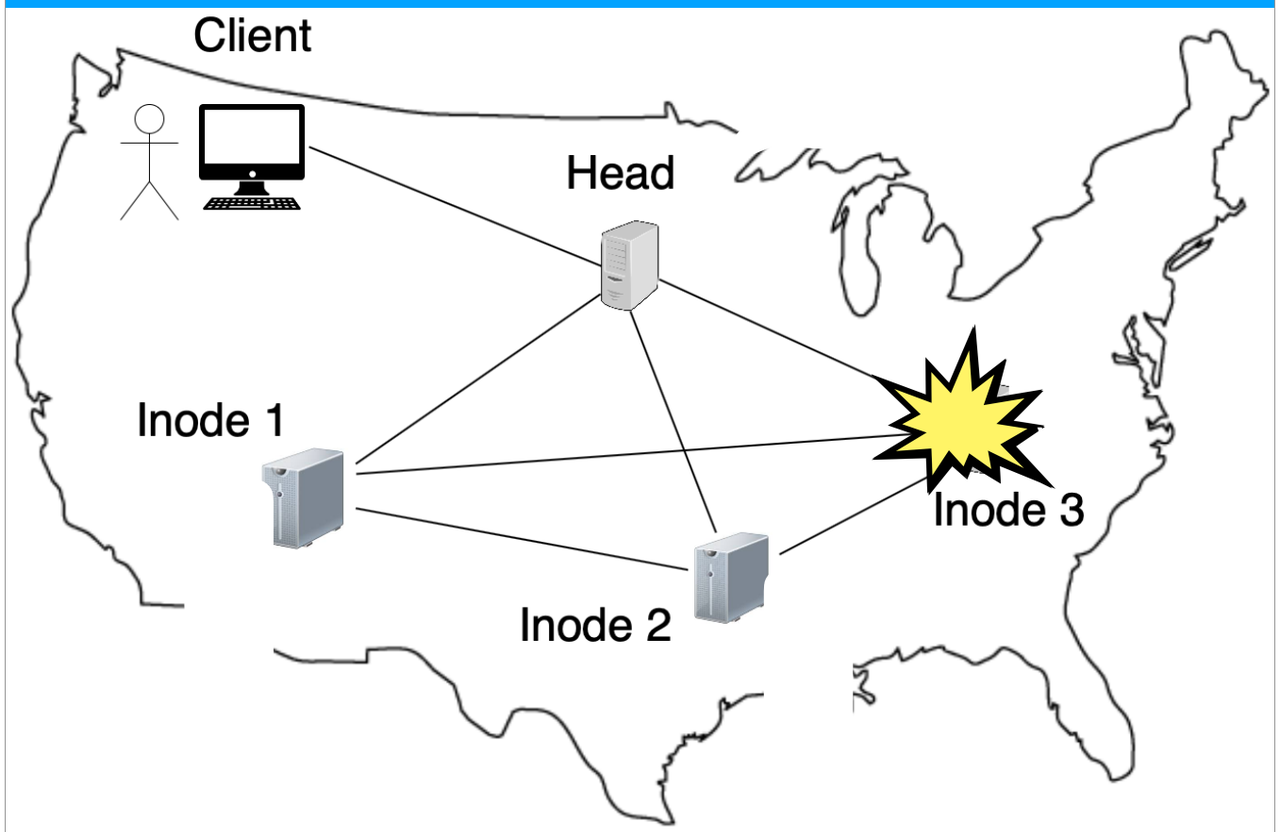

Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2


Client
Head
Inode 1
Inode 3
Inode 2

Alternative: hash-mod-n

Sam:
We may have instead used a conventional “hash-mod-n” technique like so. As with consistent hashing, we should back up each vector on the next slave as shown. But if we remove a slave and re-hash, since the modulus changed from from 4 to 3, the primary locations for every vector changes. To account for this, slave 1 has to send v0, v2, v3 to slave 3, and Slave 3 has to send v1 slaves 1 and 2, and slave 1 has to send v2 to slave 2. This requires a total of 4 messages. Using consistent hashing, we accomplished effectively the same thing with 2 messages.

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n

Alternative: hash-mod-n