A Distributed Bitmap Index Engine

Jahrme Risner and **Sam Burdick**
Department of Mathematics and Computer Science
University of Puget Sound

Abstract

We have implemented a distributed system that reliably stores a bitmap index and supports distributed queries on the index. The system uses WAH bitmap compression, Two-Phase Commit to ensure vectors arrive at their destination slaves, and consistent hashing for determining the locations of bitmap vectors. The system also provides a framework that is easily extendable for further research on the interplay between distributed systems and bitmap indexes. Experimental results confirm that query execution time in the distributed system is slower than on a centralized system, however the benefits of fault tolerance and increased storage capacity in aggregate provide capabilities not available in a centralized system.

Introduction

As datasets grow to be terabytes—or even petabytes—in size, performing queries on them can become prohibitively time-consuming if not done efficiently. An example of such a situation could be performing a query on US census data (see Table 1 for a small set of example data). If we were to search for all Americans with salaries of at least $\$100\,000$ who are also under the age of 50 (equivalent to the SQL query `SELECT * FROM CENSUS WHERE SALARY > $100000 AND AGE < 50`), we would have to scan each of the $326\,000\,000$ lines (*tuples*) of the database sequentially, since each line corresponds to a unique person. Each line read requires a separate disk access which can take upwards of $15\,\mathrm{ms}$, even on fast computers. As a result, this query could take $326\,000\,000 \times 15\,\mathrm{ms} = 1400\,\mathrm{h}$ which is just under two months; obviously queries cannot take this long.

Table 1: Example of a Relational Database, `CENSUS`

Tuple	Salary ($)	Age	City	Name
t_0	65 000	20	Tacoma	Julia
t_1	25 000	76	Spokane	Tim
t_2	130 000	42	Seattle	Maria

Using a *bitmap index* is one method for improving query execution time. A bitmap index is a collection of binary strings that represent truth values pertaining to a relational database (see Tables 2 and 3 for a possible index of the age and salary data in Table 1). In this paper we refer to each string as a *bitmap vector*. Queries on a bitmap index can satisfy common database queries—such as the SQL query given above—and are very efficient as they consist primarily of machine-level bitwise operators. To satisfy the query, we first find all Americans who make over $\$100\,000$ by `OR`ing vectors v_2 (Americans with salaries between $\$100\,000$ and $\$300\,000$) and v_3 (Americans with salaries over $\$300\,000$). Second, we find all Americans under the age of 50 by `OR`ing bitmap vectors v_4, v_5, and v_6 which will include all Americans under the age of 66 (which is a superset of the tuples relevant to the query). After `AND`ing together the two ranges, the resulting bitmap vector will have a value of `1` in rows corresponding to Americans who *may* be included in the result of the SQL query. To find the exact query return value, the original tuples corresponding to the rows with `1`s must be read. While this may result in some tuples that do not satisfy the query being scanned, the total number of tuples scanned is *significantly* fewer than in a naive sequential scan.

Table 2: Bitmap Index for Salary (S, in thousands of dollars) in Table 1

	$S \le 60$	$60 < S < 100$	$100 < S \le 300$	$300 \le S$
	v_0	v_1	v_2	v_3
t_0	0	1	0	0
t_1	1	0	0	0
t_2	0	0	1	0

Table 3: Bitmap Index for Age (A) in Table 1

	$A < 18$	$18 \le A < 21$	$21 \le A < 66$	$66 \le A$
	v_4	v_5	v_6	v_7
t_0	0	1	0	0
t_1	0	0	0	1
t_2	0	0	1	0

A system comprising multiple computers (*nodes*) is known as a *distributed system*. In comparison, a system containing only one node is called a *centralized system*. There

are two principal advantages of using a distributed system over a centralized system: first, there is no single point of failure, and second, the storage capacity of the system in aggregate can reach a size infeasible for a centralized system. Should an individual node in a distributed system fail, the data can be redistributed such that there is again a backup of all data in the system. Such resilience to hardware (or software) failure cannot be be obtained in a centralized system, in which node failure would necessitate regenerating the entire index.

The purpose of our system (hereafter called DBIE) is to expand upon the functionality of Chiu, et al.'s bitmap engine by distributing a set of bitmap vectors, and the work of executing bitmap queries, among multiple nodes. Our system is designed to be capable of recovering from single-node failure at any point during execution. Our approach was to handle vector input and query requests from a database management system (DBMS) via a *master process*. A process is a program in execution. The master and DBMS processes run on the same node. When asked to store a vector, the master sends replicas of the vector to two distinct nodes (*slaves*). To satisfy queries, the master delegates work to the slaves that have the requisite vectors.

Related Works

Dynamo (DeCandia et al. 2007) is a key-value store designed by Amazon. DBIE drew inspiration from Dynamo in the choice of hashing algorithm used to locate vectors, but it is different from Dynamo (and open source distributed data stores such as Voldemort (Project Voldemort 2017) and Redis (Sanfilippo 2018)) because it was designed to process queries using multiple nodes, instead of just storing and retrieving data.

Pilosa (Pilosa 2017) is a distributed bitmap engine built by the eponymous company in 2017. It uses the Roaring Bitmaps (RoaringBitmap 2018) compression algorithm, was written in the Go programming language, and runs each node in a cluster in lieu of using the master-slave model. Pilosa also allows data replication on multiple nodes. Due to architectural differences, Pilosa was not a large influence upon the design of DBIE.

System Description

DBIE is designed to run on Linux and its operability has been confirmed on Ubuntu 16.04.4 LTS. The following sections detail various design choices made during the implementation of DBIE.

Client Interface

Our system's "client" is a database management system (DBMS). The DBMS runs as a separate process from the distributed system and interfaces with our system through two functions: `PUT(k,v)` which adds a single vector numbered `k` with value `v` to the system (or replaces the value of vector `k` with `v`), and `QUERY(q)` which returns the results of a given query string `q` on the index. In production, the DBMS could be a full-featured product like SQLite, but we created a faux DBMS that provided fine-grained control for testing.

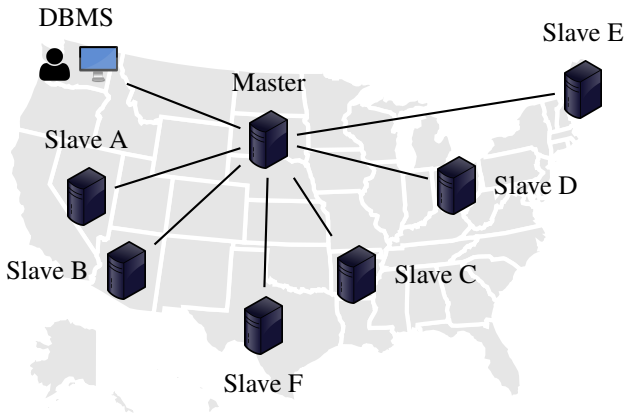

Figure 1: Visualization of the System Architecture

System Architecture

Our system is built using the *master-slave model*. There is one master node to which the DBMS makes `PUT` and `GET` requests. Upon receiving a `PUT` request, the master sends the vector to $r \geq 2$ unique slave nodess where r stands for the *replication factor* of the system (i.e., the number of replicas of each piece of data). The vector is saved on r slaves so that if one node became inoperative the vectors it held are not lost. In our implementation we assume that $r = 2$.

To satisfy a given query, the master node constructs a *query plan*, which specifies which slaves will help satisfy the query, and in what order. The slaves work together to satisfy queries using vectors they contain, and return partial query results to the master. The master collates the results from each slave and returns it to the DBMS. The algorithms by which queries are satisfied are given in later sections.

Interprocess Communication

Distributed systems require additional methods of communication beyond those used in single-process systems. Communication between nodes was accomplished using remote procedure calls (RPCs). (Oracle 2014) An RPC is simply a function call from one process to another. (Tanenbaum 1994) RPCs are a method of simplifying communication between nodes by hiding the networking details and instead allowing developers to make function calls between nodes. This helps to make the source code more readable and avoid mistakes in manually opening and closing network connections. In the current implementation, the master node resides on the same node as the DBMS. The master and DBMS processes communicate via message queues. (Oracle 2010) Requests are passed as messages, and the results could be fed into a different queue but are currently printed to the standard out device.

Bitmap Compression

In order to perform the necessary bitmap operations (`AND`s and `OR`s), we utilized Chiu et al.'s bitmap engine which was

developed by several students under the direction of David Chiu. The bitmap engine has three compression and query methods implemented: Byte aligned Bitmap Code (BBC), Word aligned Hybrid Code (WAH), and Variable Aligned Length Code (VAL). (Guzun et al. 2014; Wu et al. 2001) Each of these compression and query methods provides substantial benefits over the use of uncompressed bitmap vectors. Of the three, we chose to use WAH as it has been shown to be faster than BBC and it has a simpler formatting than VAL. WAH allows compression of vectors such that all information the vector gives is encoded, and it allows compressed vectors to be queried. A complete description of WAH is given by (Wu et al. 2001).

Bitmap Vectors

The lengths and quantities of bitmap vectors grow proportionally to the size of the data they index. Slaves save each vector as a file. The vector is broken up into 64-bit pieces, and each piece is saved sequentially into the file.

Consistent Hashing

To determine on which two slaves a given vector is located (or should be located, if the vector identifier is new to the system) we use an algorithm known as *consistent hashing*. Consistent hashing was introduced by Karger et al. (Karger et al. 1997) and can be understood conceptually as follows.

Each slave node is assigned a point on a circle where each point corresponds to a value between 0 and $2^{64} - 1$. The point for a node with identifier i is calculated as

$$\mathrm{h}(i) \coloneqq \mathrm{SHA1}(i) \bmod 2^{64}.$$

To determine which nodes (should) contain the vector k, walk clockwise from the point $\mathrm{h}(k)$ until reaching a slave node (the *primary* node), and continue walking until reaching the next node (the *backup* node). Once the primary and backup nodes have been determined, return the set comprised of those two nodes. This procedure is represented visually in Figure 2 and formalized in Algorithm 1.

In our system, the consistent hashing structure is maintained as a red-black tree, and determining the next node is given by the algorithms TREE-SUCC (Algorithm 2) and RECUR-SUCC (Algorithm 3 (Geeks for Geeks)). When the system is initialized, each node is assigned a nonnegative integer as an identifier. Each node of the tree is associated with a slave in the system, and is identified by the corresponding slave's id. After the identifiers are assigned, they are inserted into the tree. Each node has pointers to its left child, right child, and parent (accessible via the functions LEFT, RIGHT and PAR, respectively). If a node does not have a parent or a child, that pointer will hold the value $null$. The algorithms for TREE-MIN and TREE-MAX are given in (Cormen et al. 2009).

Two-Phase Commit

After we had decided on a method for determining *where* to store data, we needed a method of ensuring that the data actually arrived at the appropriate slaves. To do this, we used the Two-Phase Commit protocol (2PC). Before committing

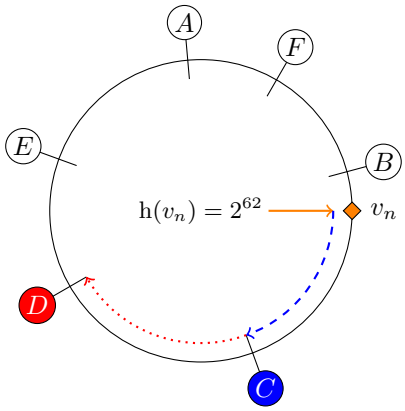

Figure 2: Visualization of Ring Consistent Hashing

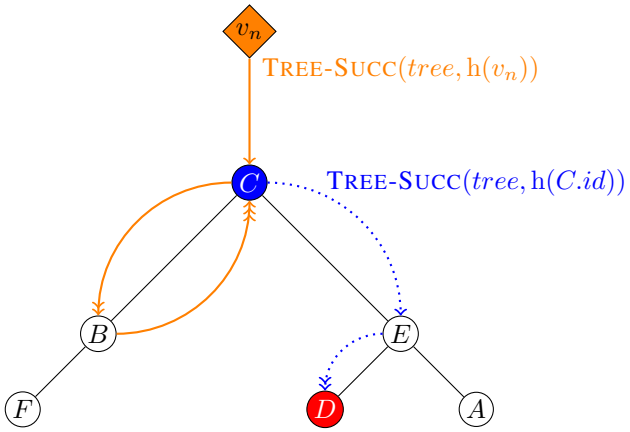

Figure 3: Red-Black Tree Traversal

Algorithm 1 Consistent Hashing

procedure CONSISTENT-HASH($tree, key$)
  $p \leftarrow$ TREE-SUCC$(tree, \mathrm{h}(key))$
  $b \leftarrow$ TREE-SUCC$(tree, \mathrm{h}(p.id))$
  return (p, b)

Algorithm 2 Successor Node

procedure TREE-SUCC($tree, key$)
  if $\mathrm{h}(key) \geq \mathrm{h}($TREE-MAX$(tree).id)$ **then**
    return TREE-MIN$(tree)$
  else
    $root \leftarrow$ ROOT$(tree)$
    return RECUR-SUCC$(tree, root, root, key)$

Algorithm 3 Recursively Determined Successor Node

```
procedure RECUR-SUCC(tree, root, succ, key)
    if root = null then
        return succ
    else if key = h(root.id) then
        succ ← RIGHT(root)
        if succ = null then
            succ ← root
            while PAR(succ) ≠ null ∧ h(succ) < key do
                succ ← PAR(succ)
        else
            while LEFT(succ) ≠ null do
                succ ← LEFT(succ)
        return succ
    else if h(root.id) > key then
        left ← LEFT(root)
        return RECUR-SUCC(tree, left, root, key)
    else
        right ← RIGHT(root)
        return RECUR-SUCC(tree, right, succ, key)
```

a vector to a slave, the master checks that both slaves are available: if so, the vector is sent to both, if either is unavailable, the inaccessible slave is removed from the system and the commit of the vector is restarted. (van Steen and Tanenbaum 2017)

Fault Tolerance

The primary reason for using consistent hashing is for fault tolerance. While waiting for messages from the DBMS, the master checks the status of its slaves by pinging each. If it does not hear back from a slave after a reasonable length of time (one second in our system), it assumes that the slave is out of commission, and reallocates the slave's vectors using REALLOCATE (Algorithm 4). In each node in the tree we store the identifiers of the vectors on the associated slave, specifically for this purpose.

An example of this procedure can be seen in Figure 4. There, node F has failed, and the master had determined that vectors must be reallocated. The vectors that must be reallocated are those for which F was the primary location and those for which F was the backup location. First, vectors v_1 and v_2 had F as their primary location and B as their backup location (which will now become their primary location). So, B will send (in one message) copies of v_1 and v_2 to node C which will serve as the new backup location. Second, vectors v_4 and v_5 will be sent (in one message) from A (their primary location) to B which will take over as backup for F. After this, the system will once again contain two copies of each vector.

In our implementation we utilized algorithms provided by (Cormen et al. 2009) to find the predecessor of a slave (TREE-PRED), to find the successor of a slave (TREE-SUCC), and to delete a slave from the tree (RB-DELETE). We also made use of an RPC, SEND-VECTOR(s_1, k, s_2), that we wrote which makes slave s_1 send the vector k to slave s_2.

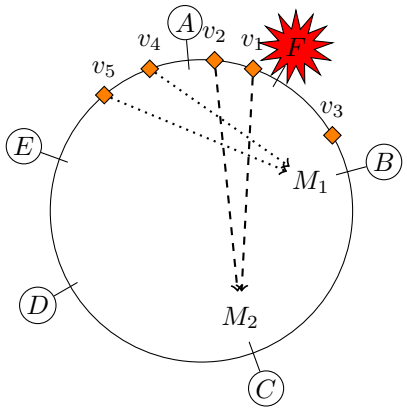

Figure 4: Visualization of Vector Reallocation

Using consistent hashing, we know that the failed slave's backup vectors are the primary vectors of its predecessor. The successor must now maintain copies of the predecessor's vectors, so the master notifies the predecessor to communicate its primary vectors to the successor. Likewise, since the successor node now holds the slave's primary vectors alongside its own, it must send the slave's primary vectors to its successor as a backup.

Consistent hashing is preferable to using a hash-mod-n algorithm, whose hash function modulus is the number of nodes. When the system changes while using the hash-mod-n algorithm, it is possible that every single vector would have to be remapped. This remapping requires significantly more message passing than the REALLOCATE function. This is because when a slave node is removed while using the hash-mod-n technique, the primary location of every vector has the potential to change, thus requiring the system to remap every single vector in the worst case. (Kleppmann 2017) While hash-mod-n requires $\mathrm{O}(K)$ remappings, where K is the number of vector identifiers, consistent hashing only requires $\mathrm{O}(K/n)$ remappings where n is the number of slaves. (Karger et al. 1997)

Algorithm 4 Reallocation

```
procedure REALLOCATE(tree, slave)
    pred ← TREE-PRED(tree, slave)
    succ ← TREE-SUCC(tree, slave)
    ssucc ← TREE-SUCC(tree, succ)
    for all v ∈ (slave.vectors ∩ succ.vectors) do
        SEND-VECTOR(succ, v, ssucc)
    for all w ∈ (slave.vectors ∩ pred.vectors) do
        SEND-VECTOR(pred, w, succ)
    RB-DELETE(tree, slave)
```

Queries

Our system is designed to handle *range queries*. In this context, a range query is a query written as a number of ranges which are ANDed together. Each range is given as a pair of vector identifiers specifying the first and last vector in

the range. Within these ranges, the vectors are ORed together. An example of a range query is `R:[2,3]&[4,6]` which would, in the context of Tables 2 and 3, correspond to the SQL query `SELECT * FROM CENSUS WHERE SALARY > $100000 AND AGE < 50` given in the introduction. Using the bitmap index, the query can be satisfied by evaluating the expression $(v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$. To carry out a query, we first create a plan that tells us which slave nodes contain the requisite vectors, and then perform that query.

Query Planning

Our query planning algorithm (Algorithm 5) takes three inputs. The first input, $tree$, is the red-black tree used in consistent hashing to determine which two nodes contain a given vector identifier, and the second input, r, is the replication factor of the system. The third input, R, is a list of pairs where each pair $(i, j) \in R$ represents a range starting at v_i and ending at v_j, where $i \leq j$. Sorting of $subpaths$ is performed so that, in each portion of the query, slaves do not have to be visited more than once, making query processing linear with respect to the number of slaves. Because $\vee$ is commutative, the order in which the vectors are ORed together within the subquery does not matter. Each range of the query may be run concurrently. In an effort to distribute the work to all slaves as evenly as possible, we choose a random return value from CONSISTENT-HASH as the slave to visit for each of the given vectors. This selection is determined using the RANDOM function which takes two integer arguments, a and b, and returns a random integer in the range $[a, b)$.

The return value of RANGE-QUERY-PLAN is a set of *subqueries*, Q, where each subquery comprises one or more pairs of the form $(slave\_id, vector\_id)$. These pairs are used in the query execution algorithms to determine which slaves to visit and which vectors to obtain. For example, the pair $(3, 2)$ indicates that v_2 should be retrieved from slave 3.

Algorithm 5 Query Planning

procedure RANGE-QUERY-PLAN($tree, r, R$)
  $paths \leftarrow \emptyset$
  for all $(first, last) \in R$ **do**
    $subpaths \leftarrow \emptyset$
    for $k \in [first, last]$ **do**
      $nodes \leftarrow$ CONSISTENT-HASH$(tree, k)$
      $node \leftarrow nodes[$RANDOM$(0, r)]$
      $subpaths \leftarrow subpaths \cup \{(node, k)\}$
    Sort $subpaths$ on $node$
    $paths \leftarrow paths \cup \{subpaths\}$
  return $paths$

Query Execution

Execution of queries received by the master is handled using Algorithm 6 which first plans out the query using Algorithm 5 and then delegates each subquery to its slaves using Algorithm 7.

Algorithm 7 is an RPC that takes a node identifier, denoting the slave that the function is run on, and a set of pairs representing a subquery. The slave iterates over the pairs referencing vectors it contains, and ORs the vectors together using WAH, as denoted by $\vee$. RETRIEVE-VECTOR(k) returns the value of vector v_k. Once the slave has operated on all requested vectors it holds, it makes an RPC to the slave in the subsequent pair, recursively satisfying the remainder of the subquery.

Algorithm 6 takes a complete query, divides the work among the slaves, ANDs the results of the subqueries (denoted by R) together, and returns the result to the DBMS.

Algorithm 6 Master Query Root

procedure MASTER-QUERY-ROOT(Q)
  $R \leftarrow \emptyset$
  for all $subquery \in Q$ **do** ▷ Delegate subqueries.
    $slave\_id \leftarrow subquery[0][0]$
    $r \leftarrow$ RANGE-SUBQUERY$(slave_id, subquery)$
    $R \leftarrow R \cup \{r\}$
  $v \leftarrow \vec{1}$
  for all $w \in R$ **do** ▷ AND the results.
    $v \leftarrow v \wedge w$
  return v

Algorithm 7 Slave subquery

procedure RANGE-SUBQUERY($node\_id, subquery$)
  $r \leftarrow \vec{0}$
  for all $(node, vec) \in subquery$ **do**
    if $node = node\_id$ **then** ▷ Vector on current node.
      $r \leftarrow r \vee$ RETRIEVE-VECTOR(vec)
      $subquery \leftarrow subquery \setminus (node, vec)$
    else ▷ Make RPC to next node.
      $s \leftarrow$ RANGE-SUBQUERY$(node, subquery)$
      $r \leftarrow r \vee s$
  return r

Implementation and Testing Notes

DBIE was primarily written in the C programming language due to its speed and common usage in systems programming. Python 3 was used to script the production of testing data, which we obtained from the TPC-C data repository (Raab, Kohler, and Shah 2018). Python 3 was also used in collaboration with bash to produce startup scripts that facilitated automatic testing of the system. In order to create our RPCs, we had to specify the types of RPCs being used in the ONC+ RPC language (Stevens 1999) which is an extension of the eXternal Data Representation (XDR) language developed in the mid 1980s by Sun Microsystems (Oracle 2014).

Results

We have tested our implementation on a system comprising six slave nodes and one master against a centralized system

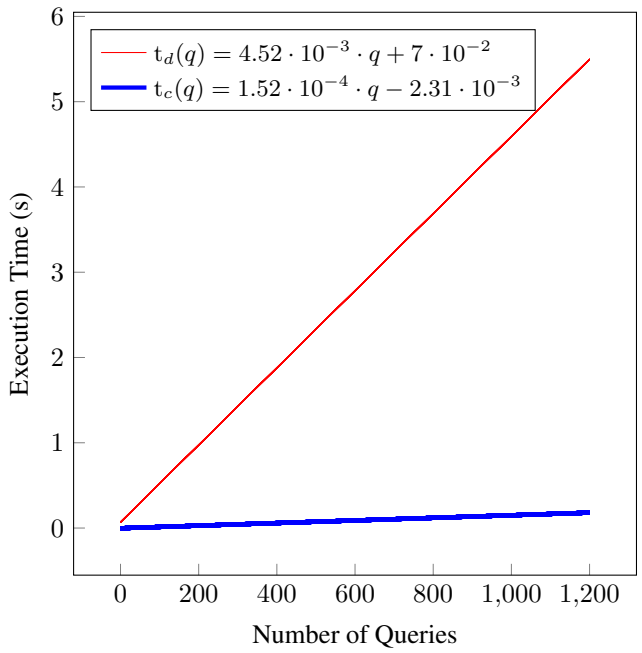

Figure 5: Query Experiment Results

that made no RPCs. All nodes used for testing were running Ubuntu 16.04.4 LTS on single-core 2.6 GHz processors. In addition, the nodes each had 2 GiB of RAM and 20 GiB of disk space. The time taken by the distibuted system to process a set of q range queries ($0 \leq q \leq 1200$) was found to be noticeably longer than the time taken by the centralized system to process the same set of queries. Figure 5 compares the execution time of the distributed system with that of the centralized system. It was found that the distributed system had an execution time of $t_d(q) = 0.00452q + 0.7$s with $R^2 = 0.9919$ while the centralized system had an execution time of $t_c(q) = 0.000152q - 0.00231$s with $R^2 = 0.9818$. Since the slope of $t_d(q)$ surpasses the slope of $t_c(q)$, we expect the distributed system to perform more slowly. This is an expected result due to the time required to perform RPCs and execute the query planner. Despite the slower measured performance, using a distributed system provides the benefits of fault tolerance and higher storage capacity (in aggregate) as discussed in the Introduction. Further, while a centralized system cannot be altered to provide fault tolerance, the execution time of the distributed system could be improved through further optimization.

Conclusion

There are no other known distributed bitmap index systems that use WAH bitmap compression. Our system could easily be extended to be compatible with more compression schemes (as opposed to Pilosa which currently only supports Roaring), as it is not designed with a specific compression algorithm in mind. Thus, it can serve as a framework for evaluating different bitmap compression schemes in the context of distributed systems. Since fault-tolerant distributed systems are challenging to program (Kleppmann 2017) we hope that further work on our system will lead to new techniques in systems programming.

Future Work

In continuing our development of DBIE, we would like to allow the addition of new slaves (including ones that were previously down) into the system while it is running. We would like to finish the work we have started by implementing more query plan algorithms and testing their efficiency. We also intend to test fault tolerance more rigorously through a series of experiments in which slave nodes fail at random times. Another optimization might be to short circuit a query that may `OR` more vectors, but has obtained a partial vector that is entirely `1`s. Furthermore, we can improve the $O(\log N)$ performance given by consistent hashing by using dynamic partitioning of the bitmap vectors, as described in (Kleppmann 2017). The system could also be modified to allow the DBMS and master to reside on separate nodes, making `PUT` and `GET` RPCs. We would also like to make the system compatible with a real DBMS.

Acknowledgements

We would like to thank Professor David Chiu for advising us on the project and for providing invaluable design insights along the way. We would also like to thank the former developers of Chiu et al.'s bitmap engine including Alexia Ingerson, Patrick Ryan, Ian White, and Alexander Harris.

References

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*. MIT Press, 3rd edition.

DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 41(6):205–220.

Geeks for Geeks. Inorder predecessor and successor for a given key in BST. `https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/`.

Guzun, G.; Canahuate, G.; Chiu, D.; and Sawin, J. 2014. A tunable aligned compression framework for bitmap indices. In *Proceedings of the 30th IEEE International Conference on Data Engineering*, ICDE '14. Chicago, IL.: IEEE.

Karger, D.; Lehman, E.; Leighton, T.; Levine, M.; Lewin, D.; and Panigrahy, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, 654–663. New York, NY, USA: ACM.

Kleppmann, M. 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 1st edition.

Oracle. 2010. System v ipc. `https://docs.oracle.com/cd/E19455-01/806-4750/6jdqdfltg/index.html`.

Oracle. 2014. Onc+ rpc developer's guide. https://docs.oracle.com/cd/E36784_01/html/E36862/index.html.

Pilosa. 2017. White paper: Pilosa: A technical overview. Technical report, Pilosa.

Project Voldemort. 2017. Project voldemort: A distributed database. http://www.project-voldemort.com/voldemort/design.html.

Raab, F.; Kohler, W.; and Shah, A. 2018. Overview of the tpc-c benchmark, the order-entry benchmark. http://www.tpc.org/tpcc/detail.asp.

RoaringBitmap. 2018. Roaring bitmaps. https://github.com/RoaringBitmap.

Sanfilippo, S. 2018. Redis. https://github.com/antirez/redis.

Stevens, W. R. 1999. *UNIX Network Programming, Volume 2: Interprocess Communication*. Prentice-Hall Inc., second edition.

Tanenbaum, A. S. 1994. *Distributed Operating Systems*. Pearson, 1st edition.

van Steen, M., and Tanenbaum, A. S. 2017. *Distributed Systems*. CreateSpace Independent Publishing Platform, 3rd edition.

Wu, K.; Otoo, E. J.; Shoshani, A.; and Nordberg, H. 2001. Notes on design and implementation of compressed bit vectors.