Fault-Tolerant Query Execution over Distributed Bitmap Indices

Sam Burdick †
Amazon.com
Seattle, WA USA

Jahrme Risner David Chiu
Mathematics and Computer Science
University of Puget Sound
Tacoma, WA USA

Jason Sawin
Computer and Information Sciences
University of St. Thomas
St. Paul, MN USA

***Abstract*—Advances in storage software and filesystems have proliferated a vast array of easy-to-use distributed storage services, removing the barrier for a growing number of organizations to geo-distribute large data sets. While leaving data in their distributed environments is convenient for data collection, various types of processing (that might use multiple data sources) are precluded due to the prohibitive costs of data movement. Users are therefore burdened with finding creative ways of performing data analysis, often requiring expert knowledge in multiple domains.**

This paper reports on the design and implementation of a query engine that enables high-level queries over distributed data sets. Our system generates bitmap indices at multiple geo-distributed data sources in order to approximate large amounts of raw data values. The bitmaps are replicated for fault-tolerance and performance. Upon accepting a high-level (SQL-like) query, our system generates a query plan, resolves dependencies, and schedules for its execution over the distributed system. The system has been tested rigorously, and experimental results show that most overheads (*i.e.*, query planning, node spawning, etc.) are negligible. Our testing also shows that our system is capable of delivering query results in the face of node failures, with no observable impact on query execution for up to 20% of the system failing. The system also provides a framework that is easily extendible for future research on the interplay between distributed systems and bitmap indices.

I. Introduction

Distributed data storage systems were initially designed to incorporate geographically distributed users while improving availability and performance of access to data [1]. The same set of motivations are even more pronounced in today's data-driven applications. The value of analytics has incentivized organizations to store data sets of high resolution and size. Also, the success of cloud computing has significantly reduced the cost barrier to access high-capacity distributed storage. Finally, the rate of data generation is so rapid among emerging applications that organizations lack the time to prepare and load the data into a relational database for query processing [2]. Therefore, a growing number of organizations now store their data, possibly piecemeal, in raw formats across high-capacity distributed storage systems. That an organization regularly uses distributed cloud storage [3]–[5] and freely available distributed filesystems [6], [7] has become commonplace.

While cloud storage providers and distributed filesystems have effectively abstracted away various low-level complexities to enforce data consistency and availability, users are still laden with the challenges of orchestrating the processing of their distributed data sets. Programmers need an intuitive and efficient mechanism to retrieve the desired data to participate in query execution.

Consider a motivating real-world example in which an organization generates, stores, and processes data sets in a distributed system: The Bonneville Power Administration (BPA.gov), a U.S. federal energy marketing authority, is completing its implementation of a Smart Grid project, in which hundreds of remote sensors are geographically deployed across the northwest power grid [8]. Each sensor collects high resolution power-transmission data at a rate of up to 120 measurements per second, resulting in ~ 2 GB per hour stored in nearby servers.

Figure 1 illustrates an important application within BPA's Smart Grid. The shaded blue areas denote the geographical regions represented by data stored in a nearby disk. During a power event (*e.g.*, line failure), it is imperative for grid operators to obtain the prior state of the grid within the faulting region [9]–[11]. The state of the grid in the moments leading up to the fault must be reconstructed using data pertaining to several geographically-distributed disks. However, problems abound: First, due to the power failure, the nodes storing the data are also presumably unreachable. Second, even if the data was obtainable (*e.g.*, power backup was available), it is still prohibitively expensive to transfer all the data onto a centralized location for processing.

Therefore, users must in real-time (1) identify which (remaining) storage nodes might contain the data pertaining to the region of interest, *i.e.*, A, B, E, and D in the figure, (2) retrieve only the subset of data necessary to perform the analysis to minimize data transfer, and (3) orchestrate data transfer, dependency resolution, execution of processes, and interpretation of the information that is extracted.

To solve this class of problems, we propose a comprehensive system that supports efficient high-level (*e.g.*, SQL style) queries and fault-tolerance over distributed data sets. At the heart of our system are *bitmap indices*, built to summarize and approximate the raw data with which they are co-located. As we will show in Section II, bitmaps are fast and effective in summarizing large amounts of data. We employ data-replication strategies to ensure that queries can be answered

† Work completed at the University of Puget Sound.

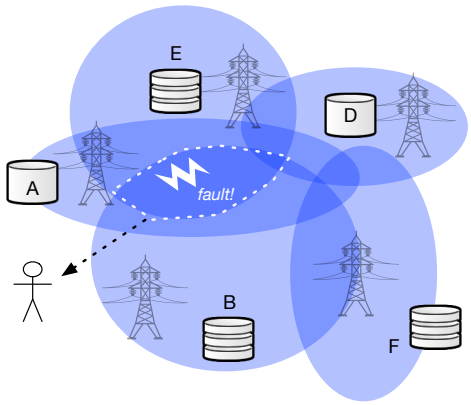


Fig. 1: Distributed Data Generation and Querying

even in the presence of node failures. Our system accepts and parses a high-level input query, then decomposes it into a set of simple bitwise operations, each known as *subqueries*, that are performed across the distributed bitmaps. As we will later explore, because subqueries are independent units, they can be executed in parallel, leading to higher performance. The subqueries' partial results are merged and reduced, before the final result is returned to the user.

The remainder of this paper is organized as follows. Section II presents a brief background on bitmap indices. Then Section III gives an overview of our distributed indexing system, and the algorithms involved in ensuring the index is both consistent and fault-tolerant. Section IV outlines our query planner and execution engine. We show that queries can be processed efficiently in terms of minimizing the time-complexity as well as the number of messages required. Experimental setup and results are explained in Section V. Section VI describes, to the best of our knowledge, the related work. Finally, Section VII describes the future work and conclusion.

II. Background

In this section we present a brief overview of bitmap indices, and how they enable fast query execution in database systems.

As an example, let us consider indexing the US census data (see Table I for a small subset of example data). If stored in its raw un-indexed format, then answering even a simple query, "Return people with salaries of at least $100 000 who are under the age of 50," would require a scan each of the 326 000 000 tuples (or rows) of the file, as each tuple corresponds to a unique person in the US. Assuming that each tuple requires only 1 KB of storage, the query processing time would exceed 10 minutes on a modern solid-state drive with 550 MB/s read-bandwidth. Clearly, the performance would be unacceptable for real-time data analysis.

To accelerate query execution, modern database systems typically employ some form of an *index*. Generally, an index is a data structure that stores a key by which to identify a tuple on disk. The query engine would then first consult the index to prune away true-negatives, and reserve expensive

TABLE I: Example of a Relation, `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	...
t_3	90 000	38	Tacoma	David	...

disk accesses to check if candidate tuples meet the selection criteria. Due to the emergence of modern bitmap-compression techniques [12]–[15], the decades-old *bitmap index* [16] has reemerged and gained favor with the broader big-data community [17]–[21]. A bitmap index is a set of *bit-vectors* that represent truth values pertaining to underlying data. An attribute, or column, in the database can be indexed by first enumerating all possible values, or more commonly, ranges of values (known as bins). Then each tuple's value for that attribute is examined, and if it matches the particular value or bin, it is assigned 1, and otherwise, 0. Tables II and III show four possible bins, and thus bit-vectors (read vertically), for the corresponding Age (A) and Salary (S) attributes seen in Table I.

TABLE II: Bitmap for Salary (S, in thousands) in Table I

	$S \leq 60$ v_0	$60 < S \leq 100$ v_1	$100 < S \leq 300$ v_2	$300 < S$ v_3
t_0	0	1	0	0
t_1	1	0	0	0
t_2	0	0	1	0
t_3	0	1	0	0

TABLE III: Bitmap for Age (A) in Table I

	$A < 18$ v_4	$18 \leq A < 21$ v_5	$21 \leq A < 66$ v_6	$66 \leq A$ v_7
t_0	0	1	0	0
t_1	0	0	0	1
t_2	0	0	1	0
t_3	0	0	1	0

Queries over a bitmap index are efficient, as they principally comprise bitwise operations. To satisfy the previous query, we first find all people making over $100 000 by `OR`ing vectors v_2 and v_3. Next, we find everyone under the age of 50 by `OR`ing bitmap vectors v_4, v_5, and v_6, which will include all who are under the age of 66 (a superset of the tuples relevant to the query). After `AND`ing together the two intermediate results, the final bit-vector will have a value of 1 in rows corresponding to those who *may* satisfy the selection. To identify the exact tuples that satisfy the query, the query engine traces the 1-bits to their corresponding tuple on disk and retrieves the value to run the final check. The total number of tuples scanned is *significantly* fewer than in a naïve sequential disk scan.

III. System Overview

The following section details various design choices made during the implementation of the distributed bitmap engine.

A. System Architecture

Our distributed indexing system implements the master-slave paradigm, comprising a *head node* and a collection of *index nodes*. Each index node stores a set of bit-vectors and, when requested, can perform logical operators across these bit-vectors to satisfy queries.

The head node serves as the interface to the client, exposing two important functions: $PUT(k, v)$, which adds a bit-vector with id k and value v into the system (or replaces the value of vector k with v if k exists). It also supports the $QUERY(q)$ function, which returns the results of a given query string q on the distributed index. Details of the query string and query processing will be given in Section IV.

In addition to the client interface, the head node also manages the index nodes. When an index node joins the network, it is communicated to the head node, which then records its existence, determines the existing and future bit-vectors that should be moved to this node for fault tolerance, and manages the data transfer. The head node periodically pings all index nodes to ensure their availability, and upon receiving a timeout, it commences the failure protocol, discussed in Section III-C.

Another task for which the head node is responsible is data placement. Upon receiving a $PUT(k, v)$ request from the client, the head node sends the given bit-vector to $r \geq 2$ unique index nodes, where r is the replication factor. The bit-vector is saved on r distinct index nodes, so that if one node became inoperative, then the bit-vectors it held are not lost. In other words, our system can absorb $r - 1$ simultaneous failures.

After determining *where* to store the replicas, we use the two-phase commit (2PC) protocol [22] to ensure that the data actually arrived at the appropriate index nodes. Before committing a vector to an index node, the head node checks that the r index nodes are available: if so, the vector is sent to both, if either is unavailable, the inaccessible index node is removed from the system and the commit of the vector is restarted.

Finally, the head node is responsible for both planning and carrying out query execution over the index nodes. To satisfy a given query, the head node constructs a *query plan*, which specifies which index nodes will help satisfy the query, and the order in which the nodes are to be accessed to resolve dependencies. The index nodes work together to satisfy queries using the bit-vectors they contain, and return partial query results to the head node. The head node then reduces the results from each index node and returns it to the client. The algorithms by which queries are satisfied are given in later sections.

B. Consistent Hashing

The method we locate and store bit-vectors is through *consistent hashing* [23]. When index nodes are added to the system, the head node assigns it to a point on a ring, 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 index nodes (should) contain bit-vector k, we "walk" clockwise from point $\mathrm{h}(k)$ until reaching the first index node (known as the *primary* node), and continue walking until reaching the succeeding node (known as the *backup* node). This process continues until all remaining backup nodes are identified. The primary and the $r-1$ backup nodes contain a replica of bit-vector k.

Figure 2 shows an example when $r = 2$. To locate bit-vector v_n, it is first hashed onto the ring. A clockwise walk determines that node C is the primary index node storing v_n. A further walk from C determines that D is the lone backup node containing a replica.

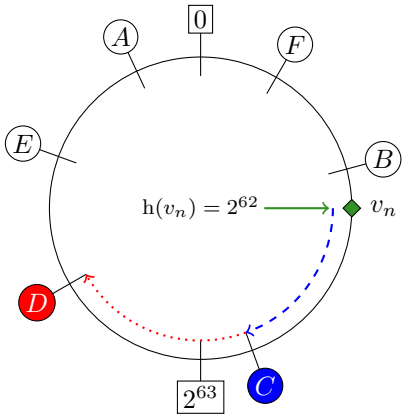


Fig. 2: Visualization of Ring Consistent Hashing (when $r = 2$)

In our system the consistent-hashing structure is implemented using a red-black tree. Each tree node corresponds with an index node in the distributed system and contains pointers to its left child, right child, and parent. The figure above that shows the placement of v_n is revisited as a red-black tree in Figure 3.

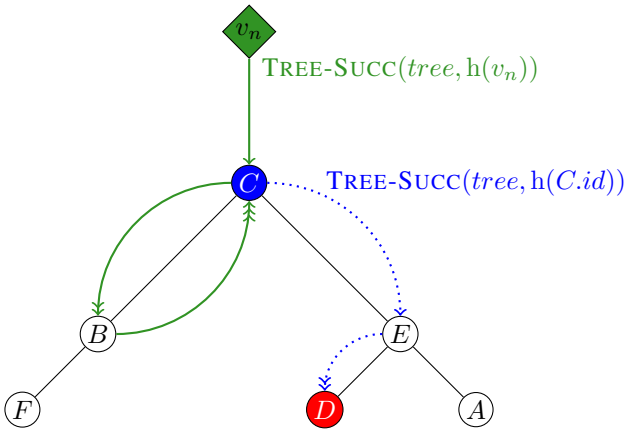


Fig. 3: Red-Black Tree Traversal

This hashing scheme is formalized in Algorithms 1, 2, and 3, respectively. Calling Algorithm 1 will return a set of r index nodes that store the given bit-vector.

Lemma 1. TREE-SUCC $\in \mathrm{O}(\log n)$, *where* n *is the number of nodes in the system.*

Algorithm 1 Consistent Hashing

```
procedure CONSISTENT-HASH(tree, key, r)
    inodes ← {TREE-SUCC(tree, h(key))}
    for i ← 1 to r − 1 do
        inodes ← inodes ∪
                {TREE-SUCC(tree, inodes[i − 1].id)}
    return inodes
```

Algorithm 2 Successor Node

```
procedure TREE-SUCC(tree, key)
    if h(key) ≥ h(TREE-MAX(tree).id) then
        return TREE-MIN(tree)
    else
        root ← ROOT(tree)
        return RECUR-SUCC(tree, root, root, key)
```

Proof. If $\mathrm{h}(key) > \mathrm{h}(\textsc{Tree-Max}(tree).id)$, TREE-SUCC will be called at most h times until one of the first two branches is taken, where h is the height of the tree. In the second branch, either loop will run for up to h iterations. In all cases, since $h = \mathrm{O}(\log n)$ [24], $\textsc{Tree-Succ} \in \mathrm{O}(\log n)$. □

Lemma 2. $\textsc{Consistent-Hash} \in \mathrm{O}(r \log n)$.

Proof. Follows trivially from the implementation and Lemma 1. □

C. Fault Tolerance with Bit-Vector Redistribution

The principal reason for employing consistent hashing is to support fault tolerance. Upon receiving a message from the client, the head node checks the living status of its index nodes. If an index node times out, it reallocates the index node's bit-vectors using REALLOCATE (Algorithm 4), passing the timed-out node as a parameter. In each node in the red-black tree we store the identifiers of the primary vectors on the associated index node (contained in the $vectors$ list), specifically for this purpose.

Algorithm 3 Recursively Determine Successor Node

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

Algorithm 4 Reallocation

```
procedure REALLOCATE(tree, inode, r)
    S_0 ← TREE-SUCC(tree, inode.id)
    S_1 ← TREE-SUCC(tree, S_0.id)
    for i ← 0 to r − 3 do
        S_1 ← TREE-SUCC(tree, S_1.id)
    SEND-VECTORS(S_0, inode.vectors, S_1)
    S_0.vectors ← S_0.vectors ∪ inode.vectors
    P ← TREE-PRED(tree, inode.id)
    for j ← 0 to r − 2 do
        S_1 ← TREE-PRED(tree, S_1.id)
        SEND-VECTORS(P, P.vectors, S_1)
        P ← TREE-PRED(tree, P.id)
    RB-DELETE(tree, inode)
```

To understand the REALLOCATE procedure, suppose there are $r + 1$ index nodes. Let I_{d+k} denote the index node positioned k nodes ahead of node I_d on the consistent hashing ring. Suppose that the head node calls $\textsc{Reallocate}(tree, I_d, r)$. I_d's primary vectors are backed up on nodes $\{I_{d+1}, I_{d+2}, \ldots, I_{d+r-1}\}$. To ensure that I_d's primary vectors are contained in r index nodes, we must have I_{d+1} pass said vectors to I_{d+r}, which is found by repeatedly finding successors of I_d as shown.

Consistent hash mappings of I_d's primary vectors will map to I_{d+1} upon deleting I_d from the tree, so I_{d+1} must inherit I_d's primary vectors. Next we must ensure that the vectors that I_d held as a backup are transferred to additional nodes to satisfy our r-replication requirement. We do this by taking making the $r - 1$ predecessors of I_d forward their primary vectors to the index nodes r places ahead of them. Afterward, every index node's primary vector is contained the proceeding $r - 1$ nodes, thus guaranteeing that each vector is located on r unique index nodes.

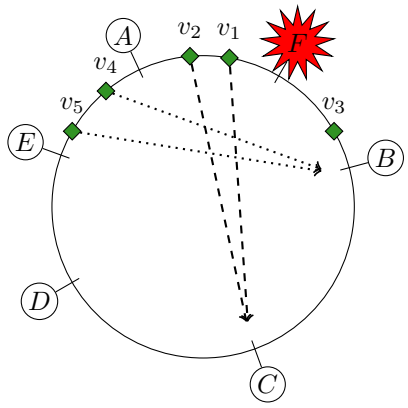


Fig. 4: Visualization of Vector Reallocation (when $r = 2$)

An example of this procedure when $r = 2$ is shown in Figure 4. Here, node F has failed. The vectors that must

be reallocated are those for which node F was the primary location *and* those for which node F was the backup location. In the figure, 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 copies of v_1 and v_2 to node C, which will serve as the new backup location, as it succeeds B on the ring. Second, vectors v_4 and v_5 will be sent from A (their primary location) to B, which will now serve as the backup node for F. After this, the system will once again contain two copies of each vector.

In our implementation we utilized algorithm TREE-PRED to locate the predecessor of a given node on the ring, which is symmetrical to the TREE-SUCCESSOR algorithm given in [24]. We also use RB-DELETE to remove an index node from the tree [24]. We made use of an RPC, SEND-VECTORS$(I_1, \mathcal{K}, I_2)$, which tells index node I_1 to send each vector in $\mathcal{K}$ to index node I_2.

Theorem 1. REALLOCATE *has* $\mathrm{O}(r)$ *message complexity and* $\mathrm{O}(r(\log n + K/n))$ *time complexity, where* K *is the total number of vectors in the system.*

Proof. According to Lemma 1, TREE-SUCC and TREE-PRED run on order $\mathrm{O}(\log n)$. They are called a total of r and $2r-1$ times, respectively. Because any vector has an $\mathrm{O}(1/n)$ probability of being an arbitrary index node's primary vector [23], we would expect $|\mathcal{K}| = \mathrm{O}(K/n)$, where $\mathcal{K}$ is an index node's set of primary vectors. SEND-VECTORS$(I_1, \mathcal{K}, I_2)$ will require I_1 to send $\mathrm{O}(K/n)$ vectors, so it will need to marshal the contents of $\mathrm{O}(K/n)$ vector files into an RPC call it makes to I_2, and therefore has $\mathrm{O}(1)$ message complexity and $\mathrm{O}(K/n)$ time complexity; it is called r times. Finally, RB-DELETE has $\mathrm{O}(\log n)$ time complexity [24]. Thus total time complexity is $\mathrm{O}(3r \log n + r(K/n))) = \mathrm{O}(r(\log n + K/n))$ and total message complexity is $\mathrm{O}(r)$. □

The rationale for using consistent hashing is to reduce data movement during reallocation. For instance, when the system changes (*e.g.*, an index node is added or removed) it is possible under naïve hashing techniques that every bit-vector may need to be rehashed [25]. This hash disruption imposes a significant amount of message passing for data re-organization than our REALLOCATE function. While naïve methods, such as static hashing, requires $\mathrm{O}(K)$ remappings [23], consistent hashing only requires $\mathrm{O}(K/n)$ remappings (Theorem 1).

IV. QUERY PLANNING AND DISPATCH

Using bitmap indices, our system can handle *range* and *point (exact-match)* queries. Other queries, including joins and aggregation are also possible, but were not yet implemented at the time of writing.

A range query is given as a sequence of pairs of bit-vector IDs, where each pair specifies the first and final vectors in the range. Within these ranges, the vectors are ORed together. An example of a range query is coded `[2,3]&[4,6]` which would, in the context of Tables II and III, correspond to the SQL query, `select * from CENSUS where salary>100000 and age<50`. Using a bitmap index, the query can be satisfied by evaluating the expression:

$$(v_1 \vee v_2) \wedge (v_4 \vee v_5 \vee v_6)$$

In this paper we refer to each parenthetical quantity as a *subquery*. Note that subqueries, which involves sequence of ORs, can be performed independently. The results of multiple subqueries are then reduced to complete the process.

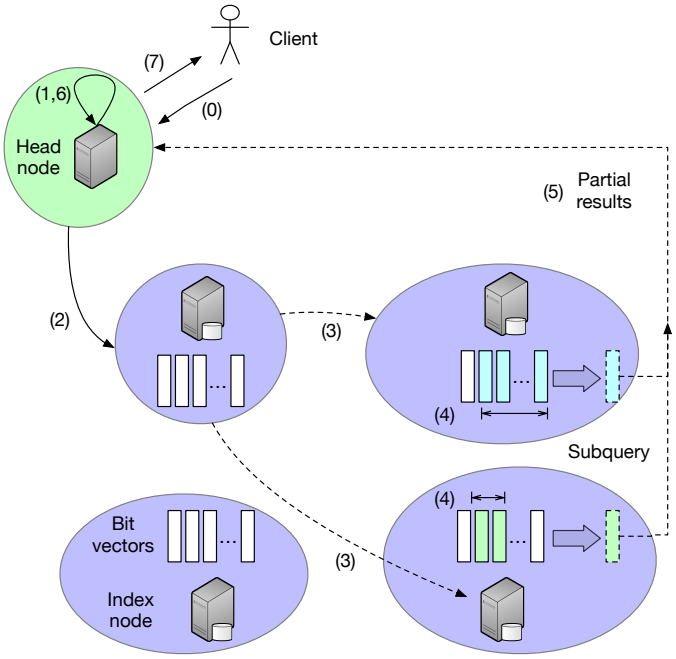


Fig. 5: Query Execution

A point (exact-match) query is the same as a range query, except that each pair is of the form `[n,n]`. For instance, `select * from CENSUS where salary=100000 and age=50` can be simply coded as the range query `[1,1]&[6,6]`. Point queries require no intermediate subquery execution, as the bit-vectors v_1 and v_6 can be returned immediately by their respective index nodes for reduction. Because bit-vectors are distributed throughout our system, there may exist multiple plans to facilitate their access. In the following subsections, we outline an optimal query-planning strategy to execute the query as efficiently as possible.

Figure 5 shows the comprehensive query-execution scheme. (0) A client initiates the process by submitting a high-level to the head node. For each query, the head node generates an optimized query plan, then dispatches the plan to at least one index node to shepherd the query. All of the necessary bit-vectors may be contained on one index node (1), in which case, the results are returned to the head node immediately. (7) This occurs more often for point queries and for range queries that contain short ranges.

Otherwise, the head node dispatches additional *subqueries*, which are partial ranges, to the specific index nodes that

store the appropriate bit-vectors. (4-5) Their partial results are delivered to the head node, where the partial results of the subqueries are reduced. (6) Finally, the head node transfers the results back to the client. (7)

A. Query Planning

In step (1) of the query execution scheme described previous, the query-planning algorithm (Algorithm 5) inputs $tree$, which is the red-black tree used in consistent hashing, the replication factor r, and a list of pairs, R, where each pair $(i, j) \in R$ denotes a subquery range starting at bit-vector v_i and ending at v_j (inclusive), where $i \leq j$. Sorting of $subpaths$ is performed so that, in each portion of the query, index nodes do not have to be visited more than once, making query-processing linear with respect to the number of index nodes. Because the bitwise operator $\vee$ is commutative, the order in which the vectors are ORed within the subquery is arbitrary, and therefore, the subqueries can be processed in parallel. In an effort to distribute the work to all index nodes as evenly as possible, we choose a random return value from CONSISTENT-HASH as the index node to visit for each of the given vectors.

Algorithm 5 Query Planning

```
procedure QUERY-PLAN(tree, r, R)
    paths ← ∅
    for all (first, last) ∈ R do
        subpaths ← ∅
        for v_id ∈ [first, last] do
            inodes ← CONSISTENT-HASH(tree, v_id, r)
            inode ← inodes[RANDOM(0, r)]
            subpaths ← subpaths ∪ {(inode_id, v_id)}
        Sort subpaths on inode_id
        paths ← paths ∪ {subpaths}
    return paths
```

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

Theorem 2. QUERY-PLAN $\in \mathrm{O}(|R| \cdot r \cdot K \log(K \cdot n))$.

Proof. Operations in the innermost loop are $\mathrm{O}(r \log n)$ and $\mathrm{O}(1)$ (Lemma 2). That loop runs for $(last - first) \leq K$ iterations, taking $\mathrm{O}(K \cdot r \log n)$ time. $subpaths$ is at most K in length, and sorting it requires $\mathrm{O}(K \log K)$ time; because the outer loop executes $|R|$ times, the entire procedure has $\mathrm{O}(|R| \cdot \ K(\log K + r \log n))$ time complexity. Since

$$\lim_{n \to \infty} \frac{r \log n + \log K}{r \log(K \cdot n)} = 1,$$

QUERY-PLAN $\in \mathrm{O}(|R| \cdot r \cdot K \log(K \cdot n))$. □

B. Query Execution

Execution of queries received by the head node is handled using Algorithm 7, which first plans the query using Algorithm 5 and then delegates each subquery to its index nodes using Algorithm 6.

Algorithm 6 is an RPC that inputs an index node identifier and a set of pairs representing a subquery. The index node iterates over the pairs referencing bit-vectors it contains, and ORs the vectors together. RETRIEVE-VECTOR(k) returns the value of vector v_k. Once the index node has operated on all requested vectors it holds, it makes an RPC to the index node in the subsequent pair, recursively satisfying the remainder of the subquery.

Theorem 3. *If s is the number of index nodes involved in the range, then the initial call to* RANGE-SUBQUERY *has* $\mathrm{O}(s)$ *message complexity.*

Proof. Because $subquery$ is sorted on $node$, each call will target an index node not previously accessed, and will not be called more than once by any index node, as it has finished ORing its own vectors to the result at that point. Therefore, if each index node is involved in the subquery, a total number of s messages will need to be sent. □

Algorithm 6 Index Node Subquery

```
procedure RANGE-SUBQUERY(inode_id, subquery)
    r ← 0⃗
    for all (inode, vec) ∈ subquery do
        if inode = inode_id then
            r ← r ∨ RETRIEVE-VECTOR(vec)
            subquery ← subquery \ (inode, vec)
        else
            s ← RANGE-SUBQUERY(inode, subquery)
            r ← r ∨ s
    return r
```

Algorithm 7 accepts a complete query, divides the work among the index nodes, ANDs the results of the subqueries (denoted by R) together, and returns the result to the DBMS. By Theorem 3, the procedure will require $\mathrm{O}(|Q| \cdot s)$ messages, where Q is the set of subqueries.

Algorithm 7 Head Query Root

```
procedure HEAD-QUERY-ROOT(Q)
    R ← ∅
    for all subquery ∈ Q do                ▷ Delegate subqueries.
        inode_id ← subquery[0][0]
        r ← RANGE-SUBQUERY(inode_id, subquery)
        R ← R ∪ {r}
    v ← 1⃗
    for all w ∈ R do                       ▷ AND the results.
        v ← v ∧ w
    return v
```

V. Experimental Results

Our system is implemented in C and tested on Ubuntu 16.04.4 LTS. To create our RPCs, we specified RPCs in the ONC+ RPC language [26], [27].

A. Experimental Setup

Most experiments involved stress-testing the distributed system, which require significant horizontal scale-out capabilities. Therefore, due to physical limitations, virtualization was necessary and we decided to conduct all of our experiments on a Docker test-bed running on a single Linux machine with an Intel Quad-core i5-6500 3.20 GHz CPU, 16 GB RAM, and a 120 GB SSD. Each container executed a single node (head or index) as a lightweight process. The nodes communicated via RPC over the Docker bridge network, which emulates a star network between all nodes.

In order to the conduct the experiments, we used the `TPC-C` data set, a commonly-used benchmark that models business transactions [28], to derive 199406 bit-vectors, each 5 KB to 6 KB in size.

B. System Initialization Time

The first experiment observes the node initialization time, which is the total time the head node used to handshake, and insert into the consistent-hash ring, each index node. The node initialization times are given in Table IV.

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

TABLE IV: Node Initialization Time

The initialization time is consistently ~ 0.5 seconds per node, regardless of the number of nodes already registered in the system. This result is intuitive, and shows expected the efficiency of consistent hashing, whose time-complexity is clearly dominated by node spawn time.

Next, we were interested in observing the performance of the $PUT(k, v)$ operator. We spawned 10 index nodes and loaded varying numbers of bit-vectors, then repeated on 100 and 1000 index nodes. We only chose to evaluate `TPC-C`, because it contains a large number of bit-vectors. Figure 6 depicts the results for a replication factor of $r = 2$. Note that both axes are presented in log-scale for readability.

The results confirm that the load time is log-linear, *i.e.*, $\mathrm{O}(r \log n)$, which shows that our system scales to large numbers of bit-vectors. One can observe a sizable gap between the settings. We believe this is due to the $\log n$ factor for consistent hashing, which noticeably contributes more to the load time initially, when the system is only storing a small number of bit-vectors. As more bit-vectors are added, the hashing overhead is amortized due to the constant r (the messages sent per bit-vector).

However, there is still another gap between the 100-node to 1000-node setting that cannot be explained by the consistent-hashing overhead. We believe this overhead is due to the head node's heartbeat monitoring for fault-tolerance, which

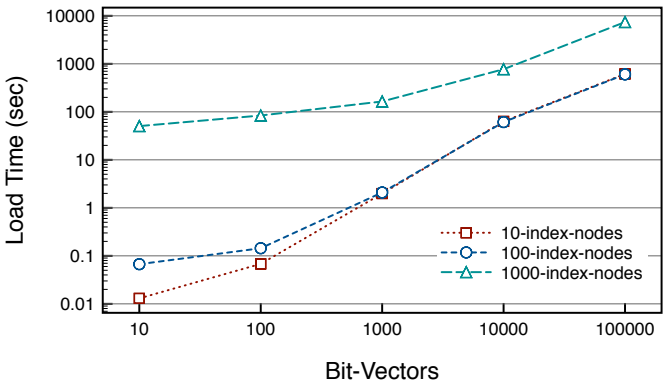


Fig. 6: Load Time

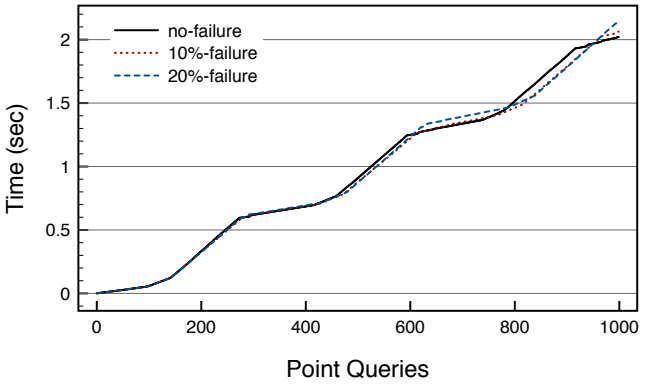
runs on each `PUT` command. This overhead introduces an additional term for the n heartbeat messages, which causes the rather constant-sized gap between the two lines. The heartbeat overhead was simply not exposed between the smaller clusters. This result indicates that we could optimize `PUT` by separating it from the heartbeat logic, a topic for future work.

C. Query Execution and Fault Tolerance

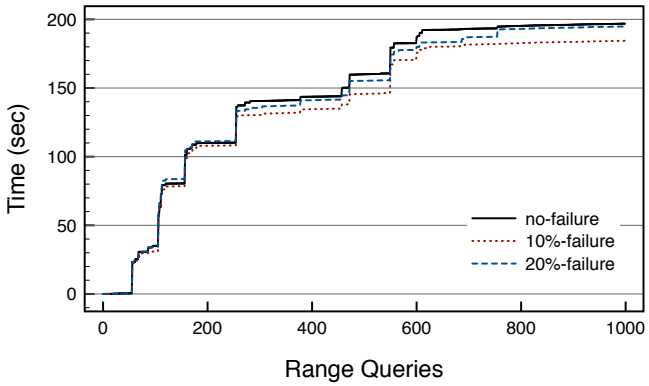
The query execution experiments presented in this section were performed under the following environment. 100 index nodes were deployed and stored 10000 total bit-vectors. We generated 1000 point queries and 1000 range queries, and ran them as separate experiments in order to understand the behavior of our system under each condition. The query workloads are skewed (*i.e.*, a subset of bit-vectors are selected more often than others). This approach is generally well-understood, as search-key and primary-key attributes are the most frequently queried in database and analytical workloads.

In addition to capturing overall query execution time, we were also interested in showing the impact of node failures on performance. In the following figures, the `no-failure` label pertains to the results in which we did not deliberately kill an index node. The `10%-failure` denotes the results in which we killed off an index-node after every 100 queries, amounting to 10% of the original node size. Finally, `20%-failure` shows the case in which we killed an index node after every 50 queries, leaving the system with only 80% of the index nodes by the time the workload completes.

Point Queries: The first experiment observes the performance of point queries both with and without index node failures. The results are shown in Figure 7(a). Because point queries are simple (requiring only a single bit-vector per subquery), their results are quite stable and quite predictable. The 1000-query workload completes in roughly 2 seconds for all three cases. The reason that performance is not affected by faults for point queries is two-fold. When an index node fails, data reallocation is handled in the background, separately from the query processor. Because the bit-vectors are replicated, point queries still can be answered, even as replication is taking place, through one of the backup nodes.



(a) Point Query



(b) Range Query

Fig. 7: Query Performance amid Failures

Range Queries: The range query results are shown in Figure 7(b). The average range query time is 0.197 seconds, which is orders of magnitude longer than point queries. This result is expected, because this workload contains queries that may involve a large number of subqueries (and therefore, more contact with index nodes). As one can observe, the cumulative time increases in a "step-like" fashion, indicating the exact queries in the workload that were I/O-heavy. However, similarly to point queries, there is not a significant variance between the workloads with and without failure.

Together, these results suggest that node failures do not significantly impact query performance, thanks to consistent hashing and bit-vector replication.

D. Overhead Evaluation

Next, we are interested in understanding the overhead of query planning, which occurs on the head node. Recall that, upon receiving a point or range query from the client, the head node first generates a distributed query plan that is to be sent for distributed query execution. We focus on range queries here, because they their plans are strictly more complex compared to point queries.

Figure 8 shows the time-elapsed to generate a plan for each of the 1000 range queries in the workload. As we can see, the

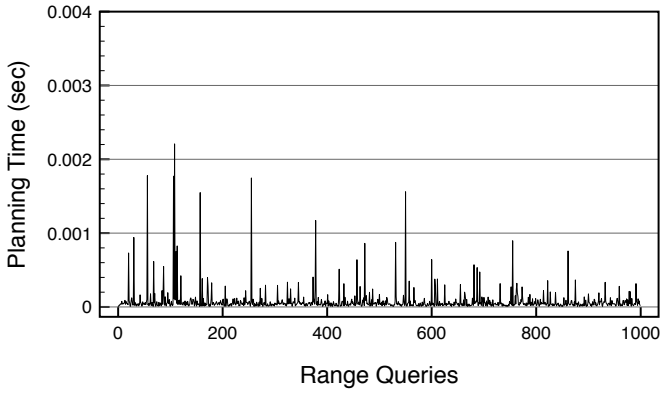


Fig. 8: Query Planning Overhead

planning overhead is generally negligible and averages 0.07 ms (or only 0.03% of the average query execution time).

Finally, we show the time taken for bit-vector reallocation following a node failure. After each index node failure, we capture the sum of (1) the time taken for the head node to communicate to the new nodes regarding which bit-vectors to transfer, (2) the time elapsed to perform the transfer, and (3) the overhead to remove the failed node from the consistent-hash ring. Figure 9 shows the time taken to reallocate bit-vectors onto remaining nodes for the 10% node-failure run (top) and the 20% node-failure run (bottom).

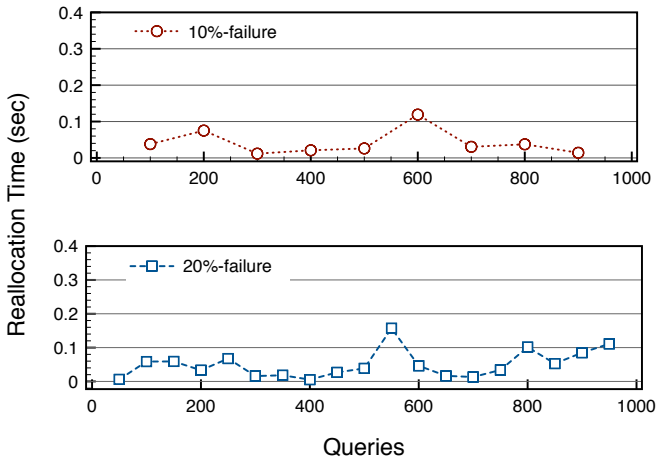


Fig. 9: Failure and Reallocation Overhead

As can be seen, bit-vector reallocation time is generally negligible (recalling that it does not impact query execution), with an average reallocation time of 44.8 ms. The overhead would expectedly grow if bit-vector sizes are large (not typical for bitmaps due to compression), or if the number of bit-vectors is much larger than our test data. However, we believe that $\sim$ 200000 bit-vectors is already on the high-end, as most bitmaps we have worked with in the past contain a few tens or hundreds of bit-vectors. Comparing the two plots, we also conclude that failure frequency does not negatively impact

reallocation overhead, as long as a significant number of nodes remain. We would, of course, expect reallocation overhead to grow when the number of remaining nodes becomes low.

VI. Related Work

Some of the most persistent challenges with creating and maintaining a distributed database system is reducing the serialization and communicating overheads that can reduce transaction and query throughput [29]. To address these challenges Alagiannis, *et al.* proposed the NoDB philosophy [2]. The tenet of this philosophy is to build systems with the goal of reducing the data-to-query time. They present a NoDB system, PostgresRaw which reduces the costs associated with accessing the raw data. PostgresRaw also uses an adaptive positional map as an indexing structure that is created on-the-fly during query processing and maintains metadata on the structure of the raw files. The bitmap index of our system is a coarse representation of the data and can be queried directly.

Aligned with the NoDB philosophy is the set of work involving scientific workflows. Workflow management systems [30]–[34] let users compose directed acyclic graphs (DAG) comprising distributed data sources, processes, and their data dependencies. Once a DAG is composed, a workflow management system schedules the jobs for processing. Parallelism is exploited through the simultaneous execution of independent jobs. Workflow composition, however, is nontrivial as it requires manual labor with expertise in the scientific and computational domains. And while automatic workflow composition engines exist [35]–[37], they require careful curation of the data and processes and are quite limited in the types of queries that are supported. Our system is more intuitive in that it accepts high-level (SQL style) queries and returns results without further user input.

Similar to the workflow management systems, Ebenstein and Agrawal created a framework that supports join-like operations over geographically distributed scientific data [38]. They present an algorithm for building and efficiently pruning distributed query execution plans. We believe that the incorporation of bitmap indices may be able to further increase the efficiency of distributed joins.

The design of our system draws inspiration from several prominent distributed object storage systems. For instance, Chord [39], Dynamo [5], Voldemort [40] and Redis [41] are persistent distributed key-value stores that guarantee high availability and *eventual consistency* on its objects. Like these systems, we also use consistent-hashing to minimize data movement when adding nodes. Instead of storing arbitrary data objects, ours stores bit-vectors with the specific purpose of answering queries. Also noteworthy are popular distributed files systems, such as HDFS [6] and Calvin [42]. The goal for these systems, however, focus on the access throughput of massive files stored across multiple nodes. Because their use-cases differ from ours, their user-interface supports primitive filesystem operations, among some others collate, and transfer the data onto local machines.

Other works have used bitmap indices in distributed systems. Fotiadou and Pitoura [43] used bitmap-based representations to compute skyline queries in a distributed setting. Su, *et al.* [44] showed that bitmap indices can be used to create lower-resolutions subsamples of massive datasets while still preserving both value and spatial distributions. These works did not explore the general use of bitmaps to speed up distributed query processing. The closest work to our own is Pilosa [21], a distributed bitmap engine built by the eponymous company. It uses the Roaring Bitmaps [15] compression algorithm 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 several core architectural differences, Pilosa was not a large influence upon the design of our system.

VII. Conclusion and Future Work

In this paper we propose a fault-tolerant distributed system that supports high-level database queries. Raw, geo-distributed data sets are index using a bitmap. Subsets of this bitmap can be replicated and distributed to enable high performance and availability. Our query planner decomposes high-level queries into subqueries, which are then dispatched for execution in the distributed system.

There are several takeaways from the system's evaluation. We showed that our system is fault tolerant, and query performance is generally predictable, even in the face of failures. Initialization and query planning are necessary overheads, but as we showed, they do not contribute significantly to the overall query execution time. We also showed that failure recovery (reallocation) overhead is independent from query performance.

The heartbeat overhead, however, was unexpectedly long, due to a design decision (and limitation of RPC) to have the head node check the statuses of the index nodes at the beginning of each PUT or $QUERY$ transaction. In hindsight, the work should have been done in reverse order, in which we task the index nodes to periodically message the head node. This decentralizes burden away from the head node, and allows it to conduct more time-critical tasks. The heartbeat mechanism will be fixed in future work.

Another future work topic involves distributed query optimization. For instance, there are opportunities to short-circuit a (sub)query that performs `OR`s over bit-vectors, but has obtained a partial bit-vector that is entirely 0s. Conversely, this idea can be extended to `AND`s over bit-vectors with partial results of 0s. Furthermore, we can improve the $\mathrm{O}(r \log n)$ performance given by consistent hashing by using dynamic partitioning of the bitmap vectors, as described in [25].

Acknowledgments

We would like to acknowledge Prof. America Chambers at the University of Puget Sound for offering valuable feedback and advice for strengthening this work.

REFERENCES

[1] J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong, "Introduction to a system for distributed databases (sdd-1)," *ACM Trans. Database Syst.*, vol. 5, pp. 1–17, Mar. 1980.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: Efficient query execution on raw data files," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 241–252, ACM, 2012.

[3] "Amazon s3 cloud object storage, https://aws.amazon.com/s3."

[4] "Dropbox," in *https://www.dropbox.com*.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, 12 2007.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, May 2010.

[7] "Apache hbase," in *http://hbase.apache.org*.

[8] "Synchrophasor technology at bpa: From wide-area monitoring to wide-area control," 2017.

[9] T.-C. Lin, P.-Y. Lin, and C.-W. Liu, "An Algorithm for Locating Faults in Three-Terminal Multisection Nonhomogeneous Transmission Lines Using Synchrophasor Measurements," 2014.

[10] M. Gol and A. Abur, "LAV Based Robust State Estimation for Systems Measured by PMUs,"

[11] V. Salehi, A. Mohamed, A. Mazloomzadeh, and O. A. Mohammed, "Laboratory-based smart power system, part II: Control, monitoring, and protection," *IEEE Transactions on Smart Grid*, vol. 3, no. 3, pp. 1405–1417, 2012.

[12] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, pp. 1–38, Mar. 2006.

[13] F. Deliege and T. Pederson, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *EDBT'10*, pp. 228–239, 2010.

[14] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin, "A tunable compression framework for bitmap indices," in *IEEE International Conference on Data Engineering (ICDE'14)*, 2014.

[15] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Softw., Pract. Exper.*, vol. 46, no. 5, pp. 709–719, 2016.

[16] H. K. T. Wong, H. fen Liu, F. Olken, D. Rotem, and L. Wong, "Bit transposed files," in *Proceedings of VLDB 85*, pp. 448–457, 1985.

[17] A. Romosan, A. Shoshani, K. Wu, V. M. Markowitz, and K. Mavrommatis, "Accelerating gene context analysis using bitmaps," in *SSDBM*, p. 26, 2013.

[18] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. P. Ahrens, "Taming massive distributed datasets: data sampling using bitmap indices," in *HPDC*, pp. 13–24, 2013.

[19] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pp. 61–72, 2015.

[20] S. Chambi, D. Lemire, R. Godin, K. Boukhalfa, C. R. Allen, and F. Yang, "Optimizing druid with roaring bitmaps," in *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, (New York, NY, USA), pp. 77–86, ACM, 2016.

[21] Pilosa, "White paper: Pilosa: A technical overview," tech. rep., Pilosa, December 2017.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[23] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "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, (New York, NY, USA), pp. 654–663, ACM, 1997.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 3rd ed., 07 2009.

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

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

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

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

[29] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Commun. ACM*, vol. 35, pp. 85–98, June 1992.

[30] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[31] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. C. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems.," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[32] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[33] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," 2004.

[34] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang, "Triana: A Graphical Web Service Composition and Execution Toolkit," in *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pp. 514–524, IEEE Computer Society, 2004.

[35] D. Chiu and G. Agrawal, "Enabling ad hoc queries over low-level scientific datasets," in *Proceedings of the 21th International Conference on Scientific and Statistical Database Management (SSDBM'09)*, 2009.

[36] D. Chiu, S. Deshpande, G. Agrawal, and R. Li, "A dynamic approach toward qos-aware service workflow composition," in *Proceedings of the 7th IEEE International Conference on Web Services (ICWS'09)*, IEEE Computer Society, 2009.

[37] D. Chiu, T. Hall, F. Kabir, and G. Agrawal, "An approach towards automatic workflow composition through information retrieval," in *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*, pp. 170–178, 2011.

[38] R. Ebenstein and G. Agrawal, "Distriplan: An optimized join execution framework for geo-distributed scientific data," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, pp. 25:1–25:6, 2017.

[39] I. Stoica, R. T. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[40] Project Voldemort, "Project voldemort: A distributed database.." http://www.project-voldemort.com/voldemort/design.html, 2017.

[41] S. Sanfilippo, "Redis." https://github.com/antirez/redis, 2018.

[42] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 1–12, 2012.

[43] K. Fotiadou and E. Pitoura, "Bitpeer: Continuous subspace skyline computation with distributed bitmap indexes," in *Proceedings of the 2008 International Workshop on Data Management in Peer-to-peer Systems*, DaMaP '08, pp. 35–42, 2008.

[44] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. Ahrens, "Taming massive distributed datasets: Data sampling using bitmap indices," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pp. 13–24, 2013.