VOLAP: A Scalable Distributed Real-Time OLAP System for High-Velocity Data

Frank Dehne*‡, David Robillard*‡, Andrew Rau-Chaplin†‡, Neil Burke†‡
*School of Computer Science, Carleton University, Ottawa, Canada
†Faculty of Computer Science, Dalhousie University, Halifax, Canada
‡E-mail: frank@dehne.net, d@drobilla.net, arc@cs.dal.ca, neil.burke@dal.ca

***Abstract*—This paper presents VelocityOLAP (VOLAP), a distributed real-time OLAP system for high-velocity data. VOLAP makes use of dimension hierarchies, is highly scalable, exploits both multi-core and multi-processor parallelism, and can guarantee serializable execution of insert and query operations. In contrast to other high performance OLAP systems such as SAP HANA or IBM Netezza that rely on vertical scaling or special purpose hardware, VOLAP supports cost-efficient horizontal scaling on commodity hardware or modest cloud instances. Experiments on 20 Amazon EC2 nodes with TPC-DS data show that VOLAP is capable of bulk ingesting data at over 600 thousand items per second, and processing streams of interspersed insertions and aggregate queries at a rate of approximately 50 thousand insertions and 20 thousand aggregate queries per second with a database of 1 billion items. VOLAP is designed to support applications that perform large aggregate queries, and provides similar high performance for aggregations ranging from a few items to nearly the entire database.**

I. Introduction

On-Line Analytical Processing (OLAP) is a widespread approach to knowledge discovery in large database systems. Many essential business applications rely on OLAP for structured data analysis [1]. OLAP queries often aggregate large portions of the database, which can lead to performance issues with very large databases. Many traditional OLAP systems address this problem by taking the static *data cube* approach [2] and materializing multi-dimensional views to ensure high query performance. However, such systems can only be updated periodically, e.g. once every week, which prevents queries from including the most recent data. More modern systems avoid materialization, but still incur a delay between new data being ingested and that data being available for analysis. Stale results become increasingly problematic for applications which have a high rate of change, or *velocity*. Applications that monitor high-velocity data streams require the ability to analyze new data as it arrives, in real-time.

This paper presents VelocityOLAP (VOLAP)[1], a scalable *real-time* OLAP system that supports up-to-date querying of high-velocity data in an elastic cloud environment. As is increasingly typical for high performance OLAP systems, VOLAP is an in-memory system that supports ingestion of new data, but not deletion. Unlike some other distributed OLAP systems, such as Druid [4], VOLAP does not use a special partitioning dimension. Such systems typically require queries to "slice" along that dimension to see good performance. In VOLAP, all dimensions are treated equally, and the system scales to many dimensions thanks to the properties of its underlying data structure. VOLAP is designed to support horizontal scaling on commodity hardware, which is more cost-efficient than systems like SAP HANA [5], which rely on vertical scaling (the use of a small number of very powerful compute nodes), or special purpose hardware such as an IBM Netezza data warehouse appliance [6]. Compute nodes can be added or removed as necessary to adapt to the current workload, and no single node acts as a performance bottleneck or point of failure for the entire system.

VOLAP partitions data into *shards* stored on *worker* nodes. Shards are stored using the novel *Hilbert PDC tree* [7], which supports multi-threaded insert and aggregate query operations on many hierarchical dimensions without any need for materialization or auxiliary index structures. Compared to its predecessors, the Hilbert PDC tree can sustain a much higher rate of data ingestion.

Clients interact with VOLAP via *server* nodes, which handle incoming streams of insertions and aggregate queries, and route them to the appropriate workers. Zookeeper [8] is used for managing global state information. A *manager* background process monitors the system and coordinates global real-time load balancing operations as necessary. Automatic load balancing allows VOLAP to adapt to changes in the data distribution or network topology, such as the addition of new worker nodes to accommodate increased load.

Experiments with 1 billion TPC-DS [9] items using the dimension hierarchies shown in Figure 1 show that VOLAP is able to ingest over 600 thousand items per second, and process streams of interspersed insertions and aggregate queries at approximately 50 thousand insertions and 20 thousand aggregate queries per second using 20 Amazon EC2 `c3.4xlarge`

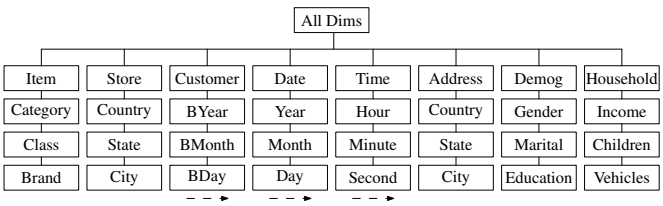


Fig. 1. Dimension hierarchies for TPC-DS data.

[1]This paper is an extended version of an earlier presentation [3] with new results on replication, data skew, and fault tolerance.

worker instances. These experiments include a wide range of queries ranging from small queries, to average size queries that need to aggregate several hundred million data items, up to queries that need to aggregate nearly the entire database.

Serializable [10] execution is particularly challenging in a highly parallel distributed system. In order to accommodate many clients, VOLAP provides different guarantees in different contexts. Each client session is attached to one of the server nodes, as illustrated in Figure 2. VOLAP can guarantee that the execution of an individual client stream is serializable, so that queries includes all relevant results from any insertion issued earlier in the stream, and none issued later. However, if an application does not require serializable execution, this can be disabled to improve performance.

Between client sessions, VOLAP provides *best-effort* freshness and aims to minimize the time required for an insertion to be included in later queries. Clients attached to the same server observe a smaller delay than those attached to different servers since inter-server synchronization is not required for one client's requests to be affected by the other. In the experiments presented here, over hundreds of billions of tests, consistency between insertions and queries on different servers was always observed in under 3 seconds, but typically in under 0.25 seconds.

In summary, VOLAP introduces novel index and worker data structures, a decentralized elastic architecture, a synchronization scheme with configurable freshness, a low-overhead implementation of serializable execution, and load balancing algorithms suitable for a fully decentralized real-time environment.

II. Related Work

Many published systems store and query large data sets in distributed environments. Hadoop [11] and its file system HDFS are popular examples, with applications typically built on MapReduce [12]. However, these systems are not designed for *real-time* operation. Instead, they are based on batch processing or "quasi real-time" operations [13], [14], [15], [16]. The situation is similar for Hive [17], HadoopDB [18], BigTable [19], BigQuery [20], and Dremel [21].

To overcome the batch processing in Hadoop based systems, Storm [22] introduced a distributed computing model that processes in-flight Twitter data. However, Storm assumes small data packets that can quickly migrate between different computing resources. This is not the case for large data warehouses. Several more recent cloud-based OLAP systems [23], [24], [25], [26] are also based on MapReduce and do not support full real-time operation.

For peer-to-peer networks, related work includes distributed methods for querying concept hierarchies [27], [28], [29], [30]. However, none of these methods provide *real-time* OLAP functionality.

Various publications on distributed B-trees for cloud platforms exist [31], however these only support 1-dimensional indices which are insufficient for OLAP. There have been efforts to build distributed multi-dimensional indices based on R-trees or related multi-dimensional tree structures [32], [33], [34]. However, these methods do not support dimension hierarchies which are essential for OLAP applications, and do not scale well to a large number of dimensions.

The systems closest to VOLAP are Druid [4], Brown Dwarf [35], SAP HANA [5], IBM Netezza data warehouse appliance [6], HyPer [36], and CR-OLAP [37].

Druid [4] is an open-source distributed OLAP store designed for real-time exploratory queries on large quantities of transactional events. Druid is specialized to operate on data items that have timestamps, such as network event logs. In particular, it partitions data based on these timestamps and queries are expected to apply to a particular range of time. This is not applicable to general OLAP where all dimensions may have equal importance. Data sets with dimension hierarchies that lack a time dimension cannot be used on Druid.

Brown Dwarf [35] is a real-time, distributed, fault tolerant OLAP store intended to be used on commodity hardware which uses a decentralized graph to quickly retrieve precomputed aggregations. However, Brown Dwarf does not support dimension hierarchies, and individual queries must specify either a single point in a dimension, or all points in a dimension, which is too restrictive for general purpose OLAP.

SAP HANA [5] is a real-time in-memory database system that also supports aggregate queries. SAP HANA relies mainly on *vertical* scaling. A basic HANA installation uses a single, special purpose, very large multi-core compute node. A limited scale-out version for multiple compute nodes is available, using a distributed file system that provides a single shared data view to all compute nodes. Horizontal scalability is restricted, however, because the system has a single master node for maintaining the shared data view, which becomes a bottleneck as the system size increases.

The IBM Netezza data warehouse appliance [6] relies on special purpose FPGA boards that provide a hardware implementation of OLAP functionality.

HyPer [36] is an in-memory database system that supports fast transactions alongside a facility for creating lightweight snapshots for OLAP sessions. HyPer makes use of the operating system's virtual memory facilities to quickly create snapshots for analysis without copying data unnecessarily. Conceptually, HyPer provides a lightweight on-demand data warehouse, which supports read-only OLAP access to a consistent snapshot of the database at a particular point in time. This is ideal for some applications, but less well-suited to those that process a high-velocity stream of mixed insertions and aggregate queries. HyPer is a single-server system, though the snapshot technique it uses may be applicable to distributed systems.

VOLAP's predecessor, CR-OLAP [37], is similar to HANA in that it is also a centralized system with a single master server node. As in HANA, this becomes a bottleneck in larger systems and restricts horizontal scalability. CR-OLAP uses the PDC tree [38] as a building block, but as one large conceptual tree, where the top few levels are stored on the master node and subtrees are stored in memory on worker nodes. This design scales well to a point, but has high insertion overhead and does not allow for a distributed index.

III. VOLAP Architecture

VOLAP represents a d-dimensional database with N_i data items and d dimension hierarchies. Clients send an ordered stream of insert and aggregate query operations, and receive an acknowledgement or result when the operation is complete.

Each query specifies, for each dimension, a set of values at any level of the respective dimension hierarchy, or a wildcard indicating that the entire range of the dimension should be included. The query result is the aggregate of the specified items. The *coverage* of a query is the percentage of items that are included in the result. For example, on a database with the Store, Item, Date, and Time dimensions as in Figure 1, the query `({Canada.Ontario},{Books,Music},*,*)` would aggregate all sales of books or music in Ontario, Canada.

A. Architecture Overview

The VOLAP architecture, shown in Figure 2, consists of:

- m *servers* $S_{1...m}$ for handling client requests.
- p *workers* $W_{1...p}$ for storing data.
- A *Zookeeper* [39] cluster for global system state.
- A *manager* background process for analyzing global state and initiating load balancing operations.

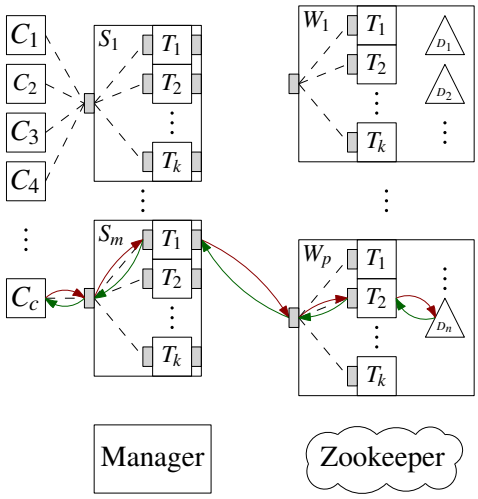


Fig. 2. System architecture. Arrows illustrate a possible path for an insertion or query through the system.

Workers and servers are multi-core machines which execute up to k parallel threads and store all data in main memory. VOLAP is *elastic* in that more workers and servers can be added if necessary. With increasing database size and/or changing network topology, data is reorganized to make the best use of the currently available resources.

Workers are used for storing data and processing OLAP operations. The global data set is partitioned into data *shards* $D_1, \ldots, D_n$. Each shard D_i has a *bounding box* B_i which is a spatial region containing D_i, represented by either a Minimum Bounding Rectangle (MBR, one box) or Minimum Describing Set (MDS, multiple boxes) [40]. Bounding boxes may overlap, though an individual data item is stored in only one shard. Each worker typically stores several shards.

Servers receive OLAP operations from clients, determine the shards relevant to each operation, and forward the operations to the worker(s) responsible for those shards. Once the workers respond, the server reports the result to the originating client.

All nodes communicate using ZeroMQ [41], a high-performance asynchronous messaging library designed for scalable distributed applications.

B. System Image

The *system image* represents the global system state, and is stored in Zookeeper [39], a fault tolerant distributed coordination service. The image contains the global information required by servers and the manager, including lists of all nodes, configuration parameters, and each shard's size, bounding box, and worker addresses.

Each server maintains a *local image* which serves as an in-memory cache to prevent Zookeeper from becoming a bottleneck. Given an insertion or query, the server uses the local image to find the relevant shards as well as the address of their corresponding worker(s).

The server updates the global image in Zookeeper at a configurable rate if the local image has changed due to insertion, for example every 3 seconds as in the experiments below. Servers make use of Zookeeper's *watch* facility to be notified of changes, and update their local image as necessary. Workers update shard statistics in Zookeeper periodically as well, to allow the manager to plan load balancing operations.

C. Index Data Structure

Since the local image is responsible for finding the shards relevant to each insertion or query, a fast index structure is crucial for high performance. Two key aspects of the index affect performance: search speed, and the global structure that results from choosing a given shard for an insertion. In particular, overlapping shards increase the likelihood that queries must be sent to many workers.

VOLAP uses a modified PDC tree [38] to serve this purpose. The basic structure of the tree is conventional: nodes have a bounding box which encompasses those of all its children. The index tree has exactly n leaves which correspond to the data shards in the system. Each leaf has the bounding box B_i of the corresponding shard D_i, and contains the ID D_i which is used by the server to locate the shard on a remote worker.

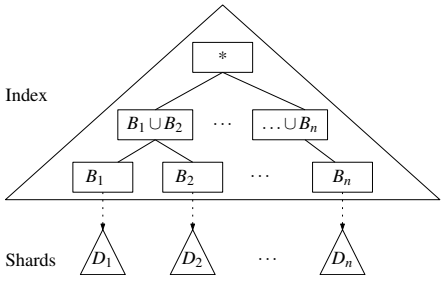


Fig. 3. Server index data structure

The server tree exploits the PDC tree cached aggregate values by storing a set of shard IDs in each node, and using set union as the aggregation function. Directory nodes higher in the tree thus contain the set of all shard IDs in the subtree

rooted at that node, as illustrated in Figure 3. For example, if the system receives a query for a large box that contains the entire database, only the root node is accessed and no tree traversal is necessary, since the root node contains the set of all shard IDs in the system.

Insertions, however, differ from those in a conventional tree, since the leaves are fixed. When a leaf is reached, its bounding box is expanded but children are not added. Consequently, an insertion never results in a node split. There are many algorithms for choosing the best subtree for insertions in an R-tree-like structure, with various trade-offs. The server index chooses the child which results in the least overlap, since the high global cost of overlap dominates the cost of performing overlap calculations in the index.

Synchronization with the global image in Zookeeper may require structural modifications to the index tree. Adding a new shard to the system inserts a new leaf, expanding and possibly splitting internal nodes in the process. When a bounding box in Zookeeper expands, the corresponding leaf's bounding box is expanded accordingly, as well as those of all nodes on the path from the leaf to the root as necessary. If a shard is split, the corresponding leaf is removed and replaced with two leaves which correspond to each side of the split.

As a PDC tree, the index is thread-safe and uses minimal locking to allow for parallelism. Servers use many threads, all using the same index in parallel, to be able to maintain a high throughput to fully utilize workers.

D. Initialization and Skew Tolerance

VOLAP is designed to ingest data exclusively from clients in real-time. This introduces challenges for a new empty system since it is not possible to initially analyze the data set to determine an optimal data distribution. However, the range of the dimension hierarchies is known at system startup, which provides coarse bounds for the keys which may be inserted.

The system is initialized using the dimension hierarchy ranges as well as the initial number of workers and shards. To prepare for data ingestion, the manager retrieves this information from Zookeeper and derives the minimal box which encompasses all possible keys. This box is then recursively split along each dimension in decreasing breadth order until a box has been produced for each initial shard. The resulting boxes are ordered such that boxes which were produced by splitting a larger box are adjacent. For example, if B is split into B_l and B_r, then the resulting array contains $[B_l, B_r]$. If B_l is then further split into B_{ll} and B_{lr}, then the resulting array contains $[B_{ll}, B_{lr}, B_r]$. A shard is created for each box, and these are distributed among the initial workers in a round-robin fashion with replication.

The initial number of shards thus provides a degree of control over load balancing agility and skew tolerance. If the initial number of shards is much greater than the initial number of workers, then a given spatial region is initially distributed among many workers. For example, if it is likely that the majority of inserts will be within one quarter of the key space, choosing at least four times as many initial shards as workers will balance this load evenly. Note, however, that this initial configuration is not fixed, skew will be adusted for over time by the real-time load balancing scheme described in Section IV. An appropriately fine-grained initial distribution allows VOLAP to handle the initial loading of the system efficiently, and increases load balancing agility since smaller shards allow the load to be more easily balanced among workers without splitting.

E. Shard Data Structure: Hilbert PDC Tree

Each shard is stored in an in-memory multi-threaded data structure that handles a stream of insertion and aggregate query operations. VOLAP includes four data structures for shards: the PDC tree [38], the Hilbert PDC tree [7], and an R-tree and Hilbert R-tree [42] variant based on the same underlying tree implementations but using MBR rather than MDS keys.

High-velocity OLAP applications are generally best served by the Hilbert PDC tree, which is designed to suit the needs of VOLAP. The Hilbert PDC tree is, like its predecessors, a multi-dimensional index where each node has a bounding box which encompasses those of all its children. The key improvements are a much higher rate of data ingestion than the PDC tree, and support for many more dimensions than an R-tree.

This is achieved by ordering nodes based on their Hilbert index, rather than performing geometric calculations at every level of the tree to determine an insert position. The *Hilbert curve* is a fractal space-filling curve with locality-preserving properties, and a *Hilbert index* is the distance along a discrete approximation of the Hilbert curve. For example, Figure 4 shows the third approximation of a 2D Hilbert curve with labelled indices. Here, $(3, 1) \Rightarrow 12$ and $(2, 0) \Rightarrow 14$, with the two nearby points mapping to nearby Hilbert indices.

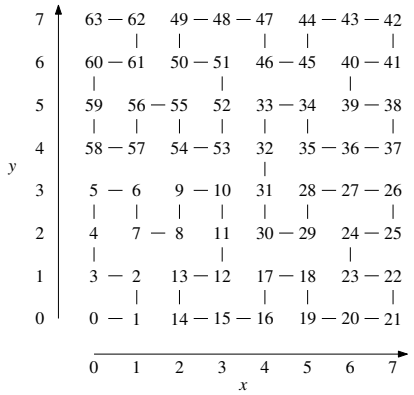


Fig. 4. The third approximation of a 2D Hilbert curve with discrete coordinates and indices

The locality-preserving nature of the Hilbert curve is not perfect, for example, $(4, 0) \Rightarrow 16$ above, which is further from $(3, 1) \Rightarrow 12$ despite having the same geometric distance as $(2, 0) \Rightarrow 14$. However, in general, geometrically nearby points map to nearby Hilbert indices.

Insertion into The Hilbert PDC tree works by first mapping multi-dimensional points to their Hilbert index, then exploiting this linear ordering to use essentially the same insertion algorithm as a B+-tree. Since linear comparison is much faster than R-tree-like geometric calculations, particularly with MDS keys, ingestion throughput increases dramatically. Querying

works much like any R-tree-like structure, that is, querying does not directly make use of Hilbert indices. The locality-preserving nature of the Hilbert curve indirectly ensures that the tree is well-structured for efficient geometric queries.

The hierarchical nature of OLAP data introduces issues with Hilbert ordering that are not addressed by structures designed for flat dimensions like the Hilbert R-tree. In particular, keys in the tree are expressed at various levels, where nodes higher in the tree tend to have coarser keys. Keys are therefore often compared at different levels than the leaf level for which the Hilbert ordering was calculated. Since the breadth of levels may vary across dimensions, the Hilbert order for leaves may not provide ideal locality for keys higher in the tree which are expressed at higher levels in the dimension hierarchy.

To improve this situation, IDs are first *expanded* such that a given level in any dimension spans the same numeric range. This is achieved by shifting the associated bits left to match the maximum possible value of an ID in that level for any dimension. As a result, the Hilbert index for leaf-level keys will still have a good distribution at higher levels in the tree. The dimension number at the start of each ID is removed entirely, since this is implicit in the Hilbert mapping.

Figure 5 shows a simple example for an ID with two dimensions. At level 4, dimension 2 uses only two bits, but dimension 1 uses four. To compensate, level 4 in dimension 2 is shifted left two bits, causing values to span roughly the same numerical range as those in dimension 1. This transformation is only performed in order to calculate the corresponding Hilbert index, the keys in the tree used for querying are unmodified. To minimize space overhead, compact Hilbert indices [43] are stored, which use the minimum number of bits necessary given the span of each dimension.

	Dim	L1	L2	L3	L4
ID	01	xx11	xx11	x111	1111
	10	1111	xxx1	xxx1	xx11
Expansion	xx	11xx	11xx	111x	1111
	xx	1111	1xxx	1xxx	11xx

Fig. 5. Transforming hierarchical IDs for Hilbert mapping

Since the ordering of child nodes is fixed, the node splitting algorithms of the PDC tree or other R-tree-like structures are not applicable to the Hilbert PDC tree. Instead, the overlap that would result from splitting a node at each index is calculated in linear time, and the node is split at the index that causes the least overlap between the resulting children.

F. Fault Tolerance

VOLAP uses redundant replication of shards to ensure that the system is able to operate without interruption or loss of data in the event of worker node failures. When replication is enabled, each shard is stored on N_r different workers. If a worker fails or stops responding, $N_r - 1$ workers remain available to handle queries on that shard. Shard replicas are kept up to date by sending each insertion from the server to all N_r workers replicating the relevant shard. Since insertions will invariably arrive at different workers at different times, shard replicas will rarely be identical at any one point in time. To mitigate this, quorum consensus is used for insertions and queries. As illustrated in Figure 6, the write quorum parameter W dictates the number of worker replies the server must receive before notifying the client that the insert is considered complete. This is similar to how write quorums work in many fault tolerant key-value stores [44], [45]. Similarly, queries are sent from the server to all workers storing replicas of each relevant shard, and R read quorum responses must be received from each relevant shard before the server performs the intermediary aggregation.

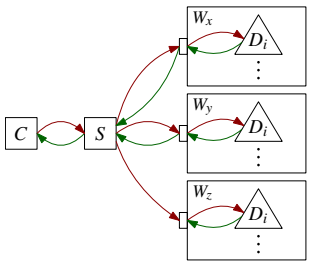


Fig. 6. An insertion with $N_r = 3$, $W = 2$

Once a server receives R query responses from a set of shard replicas, it must decide which shard aggregation is the best or "most correct" representation of the true result. Since each result is the aggregation of any number of points, timestamps of individual insertions or points can not be used to solve this problem, unlike in key-value stores. VOLAP uses different methods based on the aggregation function of the query to determine which shard replica's result is closest to the true result. For example, assuming points are never deleted from the system as is typical in OLAP, for monotonically increasing aggregation functions like `max` or `count`, the best query replica result is always the greatest. Likewise, for monotonically decreasing functions like `min`, the best result is the smallest. For non-monotonic functions like `sum` and `mean`, the result with the largest `count` value is likely the most accurate.

When worker nodes die or become unresponsive in VOLAP, the manager removes the worker from the system image and alerts servers not to send messages to the unresponsive worker. Periodically, the manager checks the system image to ensure that each shard is replicated on N_r workers. If a shard has fewer than N_r replicas, the manager coordinates a copy to a new worker in order to create another replica of the shard. Since VOLAP currently does not have any anti-entropy mechanisms to recover insertions lost during worker downtime, if a worker "comes back from the dead", its memory is wiped and it is treated as a new worker which has been added to the system.

Server threads always monitor a local command socket which broadcasts system changes like worker death. If a worker dies while a server is waiting for a reply from that worker, the server adjusts accordingly to ensure execution proceeds. If the worker is a redundant replica, the server removes the worker from its pool of expected responses, and completes the operation and responds to the client if all other workers have responded. If the operation can no longer be completed successfully because too many workers have died, the server aborts the operation and responds with an error.

Higher values of N_r increases the number of workers the

system can lose while still maintaining correct operation. Increasing N_r also consumes more resources, as each insertion and query must be executed on N_r different workers. The quorum parameters W and R thus give the user a level of control over the consistency of their queries. Lower values of W and R decrease latency at the cost of decreased consistency, while high values of W and R tune the system for high data consistency at the cost of insert and query latency.

G. Multi-Threaded Message Handling

Careful handling of messages is necessary to maintain high performance while ensuring correctness in a highly parallel system like VOLAP. Servers and workers have a similar design for handling incoming requests: a single receiver thread reads messages from the network socket, and several processing threads handle requests. The receiver thread distributes requests to processing threads via local sockets. All sockets are ZeroMQ sockets, using the appropriate transport backend (`tcp` or `inproc`). Using sockets for inter-thread communication avoids many synchronization issues and allows for a high degree of parallelism.

The receiver thread balances requests among processing threads by maintaining a list of currently idle threads. When a request is received, an idle thread is popped off the list, and the request is forwarded to the corresponding local socket. When the thread has finished processing the request, it sends the reply back to the receiver (and ultimately the client) via the same socket, followed by a special message to indicate that the thread is once again idle. The processing thread is then placed back on the list of free threads.

In the server, requests typically require sending further requests to workers. One of two possible configurations can be chosen by the user:

- *Thread sockets*: Each processing thread has its own network socket for sending messages directly to workers.
- *Single socket*: Each processing thread has a local socket that forwards to a sender thread, which uses a single network socket to send all messages to workers.

Thread sockets eliminate the hop through the sender thread, and thus can achieve slightly higher throughput. However, the lack of a single ordered stream of messages to workers precludes serializable execution, which requires a single network socket.

H. Serializability

A serializable execution guarantees the same results as a serial execution of the operation stream [10]. In the context of VOLAP, this requires that any queries include all relevant results from insertions issued prior in the stream, and none issued later. Note that this guarantee does not impose an execution order for a series of successive insertions or queries: insertions only affect the outcome of queries, so successive insertions can safely be executed in parallel until a query arrives. Similarly, queries do not affect the database state, so successive queries can be executed in parallel until an insertion arrives. Synchronization is therefore required at any *transition* from query to insertion or vice-versa, as illustrated in Figure 7.

A naïve implementation could simply block processing of the stream entirely at each transition, but this would severely restrict throughput for streams with interspersed insertions and queries. Instead, VOLAP uses a parallel pipeline model where synchronization occurs at various stages throughout the processing of a request, allowing insertions and queries to be executed in parallel at different stages in the pipeline. There are several stages in the execution of a request where the correct order must be ensured. At each such stage, if a synchronization point in the stream is encountered, processing must be blocked until the previous operation has completed execution of that stage. This is achieved with a series of barriers which allow any number of insertions, or any number of queries, to execute in parallel, but block processing if an insertion is encountered while queries are being executed or vice-versa. The synchronization barriers encountered while processing a request, as illustrated in Figure 8, are:

1) Server Receive: When the request arrives at the server's receive socket, the first barrier is reached by the (single) receiver thread. Then, the request is forwarded to a server thread for processing via an internal socket.
2) Server Prepare: When a server thread receives the request, it deserializes the request and applies it to the index.
3) Index: After the request is applied to the index, the corresponding worker request(s) is/are enqueued for sending to the relevant worker(s).
4) Enqueue: After all messages have been enqueued to be sent to workers, a special message is also enqueued to signal the sender thread to trigger the next barrier.
5) Send: When the sender thread receives the special synchronization message, it knows all worker messages for this request have been enqueued, so serializable delivery of requests to workers has been ensured.
6) Worker Receive: Identical to step 1 in the server.
7) Worker Prepare: Identical to step 2 in the server.
8) Commit: After the request has been applied to all relevant shards, serializable execution has been completed.

In order to maximize throughput, more barriers are used than are strictly necessary to ensure correctness. For example, barrier 2 could be omitted, but its inclusion allows following events to be distributed to server threads and prepared in parallel while the current operation is being committed to the index. This design allows many requests to execute in parallel at different stages, so serializable execution has only a moderate impact on performance, as demonstrated in Section V-B.

IV. Load Balancing

Effective load balancing is crucial for scalable distributed systems. When the workload of the system is unevenly partitioned among its resources, some portion goes underutilized while the remainder struggles to pick up the slack. This has a negative impact on throughput, response time, and stability which tends to get further compounded as the system scales up in size. However, the load balancing operations themselves can also incur significant costs due to the overhead of moving potentially large amounts of data over the network. Maintaining consistently high performance requires a load balancing scheme

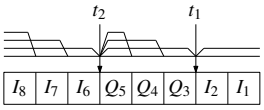


Fig. 7. The *transitions* t_i within a stream of insertions I_j and queries Q_j where synchronization is required

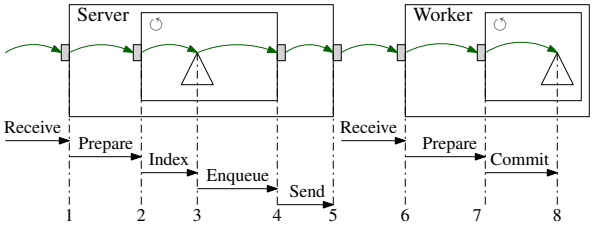


Fig. 8. Path of a request through the system with serialization barriers

which offers a good trade-off between load balancing overhead and effectiveness.

VOLAP uses a real-time load balancing scheme which allows workers to be added, removed, or replaced dynamically in order to maintain performance in the face of changing system load. The operations performed during load balancing are carefully designed to not interrupt processing, so insertions and aggregate queries can continue to execute while load balancing is performed.

A separate background process called the *manager* initiates load balancing operations. The manager periodically analyzes the system state stored in Zookeeper and decides on suitable load balancing operations. It then initiates these operations, coordinating the necessary actions between workers and servers. For example, the manager may identify a worker that is overloaded and about to run out of memory, then send messages to workers instructing them to perform the appropriate splits and/or migrations. The manager is not a bottleneck for insertion or query performance, and can reside anywhere in the system.

A. Shard Operations

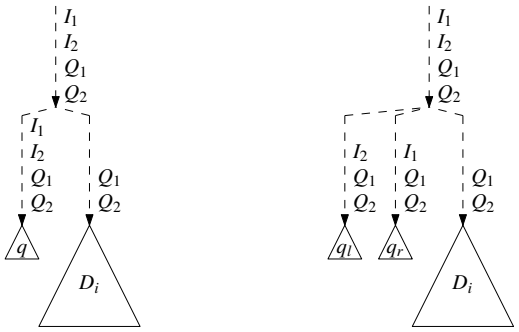
A shard D_i stored on a *source worker* W_s can be migrated to a *destination worker* W_d if, for example, W_s is running out of memory or W_d is a new worker allocated for spreading the load. A shard can also be split if load balancing requires smaller shards to migrate. The shard data structures provide four operations in order to support these scenarios:

- $\mathrm{SplitQuery}(D_i, B_i)$ which returns a hyperplane h that partitions D_i into D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are of approximately equal size.
- $\mathrm{Split}(D_i, B_i, h)$ which returns $(D_i^1, B_i^1, D_i^2, B_i^2)$ where D_i is partitioned into D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are spatially separated by hyperplane h.
- $\mathrm{SerializeShard}()$ which returns a flat binary blob b containing the data in D_i (suitable for network transmission).
- $\mathrm{DeserializeShard}(b)$ which builds the data structure from such a blob.

B. Processing Requests During Load Balancing

Correctly performing a split or migration requires a consistent snapshot of the relevant shard. However, real-time operation requires that operations can be processed at any time, including when shards are being split or migrated. For example, if a shard is being serialized for migration, but data items are inserted during this process, it is unknown which insertions are included in the serialized shard and which have been missed.

To avoid such problems, workers create *insertion queues* for shards during load balancing operations. During the operation, insertions for the shard are inserted into a queue rather than the shard itself, as illustrated in Figure 9. A single queue is used for split queries and migrations, and two are used during a split with inserts directed according to the split hyperplane. Queries are directed to both the shard and the queue(s) to ensure results are up to date. The insertion queue uses the same data structure as shards, allowing insertions and queries to be processed with the same performance as shards themselves.



(a) Split query and migration (b) Split

Fig. 9. Distribution of inserts and queries to a shard D_i and its insertion queue(s) during load balancing.

When a migration is finished, the shard on the source worker is no longer necessary and is destroyed along with the corresponding queue. When a split query is finished, the queue is drained into the shard. When a split is finished, for each replica, the two queues are drained into their corresponding shard.

C. Migration

The basic strategy for maintaining responsiveness during migration is to have the source worker continue to serve requests while the shard is migrating. Once the destination worker receives the shard from source worker, it is activated there and removed from the source worker. The process of migrating a shard from a *source worker* to a *destination worker* is initiated and coordinated by the manager.

The migration process consists of several stages, each associated with a message delivery as illustrated in Figure 10:

1) `MigratePrepDest`(W_s, s):
   To begin a migration, the manager notifies the destination worker that it will be receiving a new shard s. The

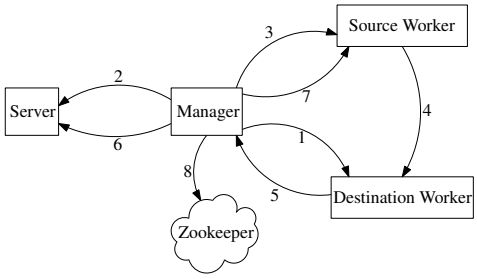


Fig. 10. Migration process

destination worker begins queuing up any insertions for s that may arrive during the migration.

2) `MigratePrepServ`(W_s, W_d, s):
Once the destination worker is ready, the manager notifies all servers that shard s is about to migrate from W_s to W_d. The servers begin sending any insertions for s to both the source and destination workers.
3) `MigrateBegin`(W_d, s):
Once all servers are aware of the migration, the manager instructs the source worker to send shard s to W_d.
4) `MigrateData`(s, W_d, d):
The source worker serializes shard s into a blob b and sends it to W_d.
5) `MigrateDone`(s):
When the destination worker receives s, it adds it to its set of data structures, applies all pending queued inserts, and notifies the manager that it has received the shard.
6) `MigrateDone`(s):
When the migration is complete, the manager notifies all servers, which cease sending messages for s to the source worker.
7) `DeleteTree`(s):
Now that s is no longer considered to reside at the source worker by servers, the manager instructs the source worker to delete it.
8) Finally, the manager updates Zookeeper to reflect the new location of s, and the migration is complete.

Note that all servers are informed of the migration of a shard before starting the migration process. Thus, when shards are migrating, correct query results can be guaranteed by forwarding queries to all relevant workers.

D. Split

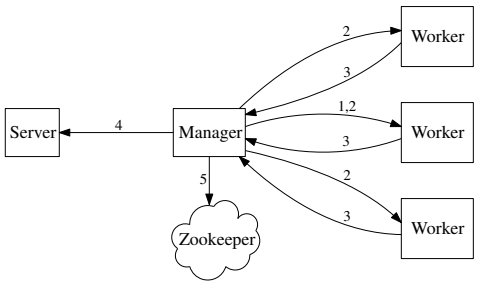


Fig. 11. Split process

Like migration, the process of splitting a shard is initiated and coordinated by the manager, and transaction processing can proceed normally while shards are being split. Unlike migration, splitting requires coordination from all workers replicating the shard. The split process consist of 5 stages, each associated with a message delivery as illustrated in Figure 11:

1) `SplitQuery`(s):
To begin the split, the manager arbitrarily selects one of the workers replicating the shard and requests that it return a hyperplane with which to split the shard.
2) `SplitRequest`(s):
Once the manager receives the split hyperplane, it sends a split request with the hyperplane to all workers replicating the shard. The workers then begin the split process.
3) `SplitComplete`(s, s', s''):
Once a replica split completes, the worker notifies the manager that s is split into s' and s''.
4) `SplitUpdate`(s):
When the manager receives the `SplitComplete` message from each worker, it notifies all servers that s has been split into s' and s''. The servers begin sending insertions and queries to s' and s''.
5) Finally, the manager updates Zookeeper to remove the old shard s and add the two new "sides" s' and s''.

In order to handle queries after shards have been split, each worker W_k stores a *mapping table* M_j. If a shard D_i is split into D_i^1 and D_i^2, then M_j stores an entry with key D_i and value pointing to the two data structures for D_i^1 and D_i^2. Thus, any queries in the queue which were initiated before the split can be applied to the two new shards that contain the data originally stored in the query shard.

E. Replication

When a server notices that messages being sent to a worker are timing out, the server records in the system image that the worker node has died and its shards have been lost. This triggers the manager, who periodically checks the system image for any shards that have less than N_r copies, to initiate a replication operation of the shard to another worker. The replication process is essentially a migration without deletion. The source worker begins queuing insertions for the relevant shard, and instructs servers to send insertions to both the source and destination worker. The source worker then sends a copy of the shard to the destination worker, followed by a copy of the queued inserts. Finally, the manager updates Zookeeper to reflect that the destination worker now replicates the shard.

F. Load Balancing Algorithm

VOLAP has a modular design that allows for various *optimizers* which implement load balancing algorithms. The manager provides access to the system state as stored in Zookeeper, and periodically requests load balancing operations from the optimizer. If the optimizer returns a set of operations, the manager initiates them, and, once all operations are complete, resumes periodically requesting load balancing operations from the optimizer.

The experiments below use an optimizer that balances the memory load of workers, so if a single worker becomes too

full, data will be migrated to a worker with more free memory. The *improvement* I for a migration of a shard s from worker W_s to W_d is the difference between the current imbalance of the workers and the imbalance if the migration were performed. If we denote the size of a worker or shard X as $\overline{X}$, then:

$$I = |\overline{W_s} - \overline{W_d}| - |(\overline{W_s} - \overline{s}) - (\overline{W_d} + \overline{s})| \quad (1)$$

For example, if $\overline{W_s} = 1000$, $\overline{W_d} = 500$, and $\overline{s} = 250$, then

$$\begin{aligned} I &= |\overline{W_s} - \overline{W_d}| - |(\overline{W_s} - \overline{s}) - (\overline{W_d} + \overline{s})| \\ &= |1000 - 500| - |(1000 - 250) - (500 + 250)| \\ &= |500| - |750 - 750| \\ &= 500 \end{aligned}$$

In this case, migrating s results in the two workers being completely balanced, so the maximum possible improvement (their original difference in size) is achieved.

The algorithm proceeds by evaluating every possible migration and ranking them by improvement. In order to avoid migrations that do not result in a significant enough improvement to justify their expense, the user can specify a *migration threshold* parameter T_m. For a migration to be considered, the improvement must exceed the shard size scaled by the migration threshold, that is:

$$I \geq T_m * \overline{s} \quad (2)$$

Then, potential splits are considered by evaluating the migrations they would enable. For each shard, the optimizer considers the migrations that would be possible if the shard were split in half, and adds the split to the list of potential operations using the same criteria as migrations.

The optimizer then chooses operations in order of decreasing improvement. The user can control the aggressiveness of load balancing by configuring the *minimum improvement* which an operation must exceed in order to be selected. To prevent a worker from being too heavily loaded by load balancing operations, at most one migration to or from a given worker, and at most one split on a given worker, is chosen in a single optimization round. When an operation is chosen, the relevant workers are flagged, and subsequent potential operations that involve those workers are skipped. Any such skipped operations will likely be chosen in the next optimization round if they remain worthwhile.

V. Experimental Evaluation

VOLAP performance is evaluated with respect to the system size, *workload mix* (percentage of insertions in the operation stream), and *query coverage* (percentage of the database that needs to be aggregated for a query). Data sets are either from TPC-DS with $d = 8$ hierarchical dimensions as shown in Figure 1, or a synthetic Zipf distribution with skew $s = 1$ except where otherwise noted. Experiments were performed on Amazon EC2, using `c3.8xlarge`, `c3.4xlarge`, and `c3.2xlarge` instances for servers, workers, and all other nodes, respectively. At the time of writing, these instances are based on Intel Xeon E5-2680 processors, running Amazon Linux with Linux 3.14.35, ZeroMQ 4.0.5 and Zookeeper 3.4.6.

Queries are randomly generated to span a wide range of coverages, and specify values at various levels in all dimensions. Generated queries are tested against the database and binned according to their true coverage. During benchmarking, queries are chosen uniformly at random from the appropriate bin. Except where otherwise noted, $N_r = 1$ replicas are used.

A. Data Structure Performance

The Hilbert PDC tree is designed for high-velocity environments, and supports a dramatically higher rate of insertion than its predecessor the PDC tree, as shown in Figure 12a. In this experiment, using TPC-DS data with 8 threads on a single quad-core machine, the Hilbert PDC tree ingests data over 10 times faster than the PDC tree.

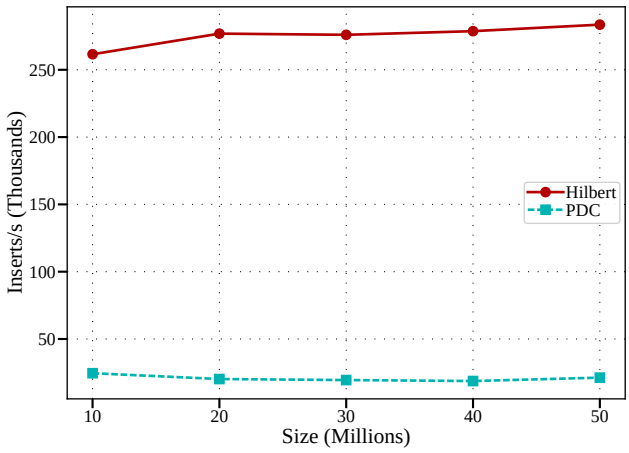
Figure 12b compares the performance of the Hilbert PDC tree and the PDC tree for queries with varying coverage. Both trees perform relatively well with high coverage, since these queries tend to completely cover high-level tree nodes. This allows the cached aggregate values in the tree to be used, avoiding the need to traverse more deeply. The performance gain of the Hilbert PDC tree for low and medium coverage queries highlights the improved tree structure obtained by using Hilbert ordering. For both low (below 33%) and medium (33% to 66%) coverage queries, the Hilbert PDC tree performs significantly better than the PDC tree. The fractal nature of Hilbert ordering combined with careful mapping of MDSs (as described in Section III-E) produces less overlap at lower levels in the tree than the R-tree-like PDC tree algorithm. This increases the likelihood that cached aggregate values are used for lower coverage queries.

The benefits of the PDC tree are most apparent with a high number of dimensions. In particular, PDC trees handle many dimensions much more efficient than R trees, as can be clearly seen in Figure 13b. Above 16 dimensions, the R tree variants become effectively unusable, but the PDC tree variants scale gracefully to many more dimensions.

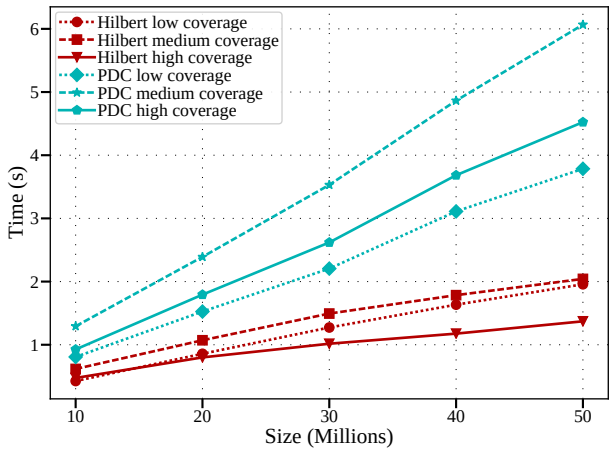
The Hilbert PDC tree preserves this scalability in query performance, and significantly improves it for insertion, as shown in Figure 13a. With over 16 dimensions, the query performance of the R tree variants degrades dramatically, but both PDC trees retain their speed. Since insertion in the Hilbert PDC tree is based on a simple linear ordering rather than geometric calculations as in the PDC tree, the cost of additional dimensions is significantly lower. As a result, insert latency is nearly flat compared to the PDC tree where insertion gets significantly more expensive as the number of dimensions increases. Accordingly, all subsequent experiments in this section use the Hilbert PDC tree as the underlying data structure.

B. Serializable Execution Impact

The performance impact of serializable execution is shown in Figure 14. The socket configurations refer to the different architectures described in Section III-G. All three experiments use the same TPC-DS data and queries. The effects of serializability can be seen by comparing “serializable” with “single socket”, which both have the same socket configuration.

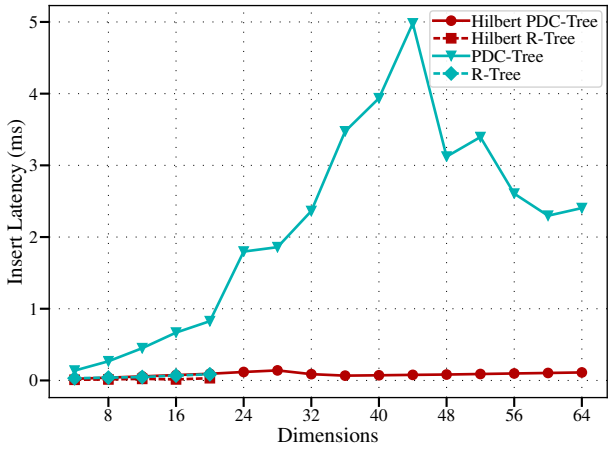


(a) Insert performance of Hilbert PDC tree vs. PDC tree

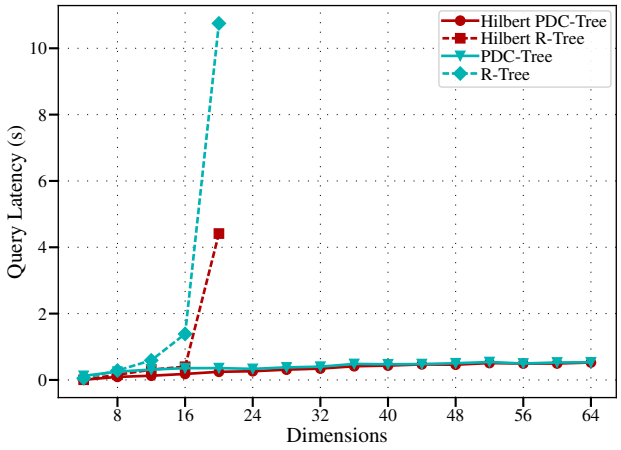


(b) Query performance of Hilbert PDC tree vs. PDC tree

Fig. 12. Performance of tree variants with increasing number of elements, $d = 8$



(a) Insert latency



(b) Query latency

Fig. 13. Performance of tree variants with increasing number of dimensions, $N_i = 50,000,000$

The impact is greater when there is a mix of insertions and queries, as expected, since synchronization is required only when an insertion is followed by a query, or vice-versa. In the worst case, at a 50% mix, serializability comes at a cost of approximately 20 thousand operations per second. A significant cost, but one that only affects the client with serializable execution enabled. Note that a 50% mix here results in a stream that is often an insertion followed by a query, then another insertion, and so on; the worst case scenario for serializable execution overhead.

C. Real-Time Load Balancing

The real-time load balancer coordinates the elasticity of the system. As workers are added, the load balancer automatically moves data items to the new workers to balance the workload. Figure 15 shows the impact of real-time load balancing during a horizontal scale-up experiment. In this experiment, load phases are interleaved with insert and query benchmarking phases. At the start of each load phase, two additional workers are added to account for the increase in database size. The red region shows the minimum and maximum number of data elements stored on a worker. When new workers are introduced they are empty, causing the minimum to go to zero. The effects of the load balancer are clearly visible as the gap between minimum and maximum worker size is reduced by moving data to the newly introduced workers. The number of migration operations for this process are shown as a dotted purple line associated with the right y-axis. Once balance is achieved, loading proceeds, increasing the minimum and maximum size per worker as new elements are inserted. Note that this experiment uses discrete phases to ensure a stable benchmarking environment, but in general, load balancing is performed concurrently with insertions and aggregate queries whenever the manager decides an adjustment is necessary.

D. Horizontal Scale-Up Performance

Figure 16 shows the insert and aggregate query performance for various workloads as the system size increases. This data is from the same experiment as shown in Figure 15, where two new empty workers are added at each scale-up step. For each system size with p workers and $N_i \approx p \times 50$ million data elements, benchmarks are performed for insertions as well as

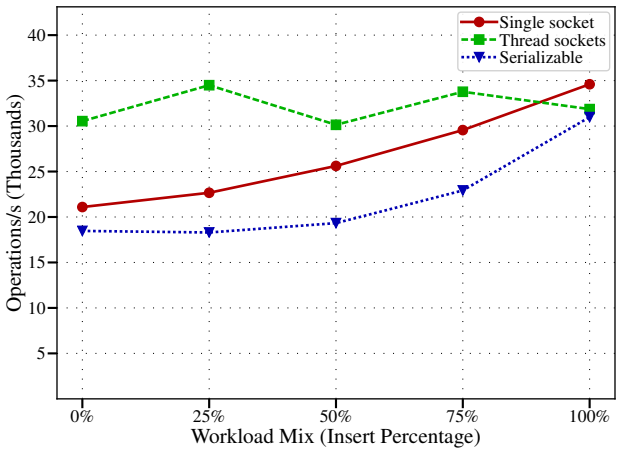


Fig. 14. Performance impact of serializable execution. "Thread sockets" has server threads communicate with workers directly using a thread-local socket. "Single socket" has all threads send to workers via a single socket, and "Serializable" uses the same architecture but with added synchronization to guarantee serializable execution.

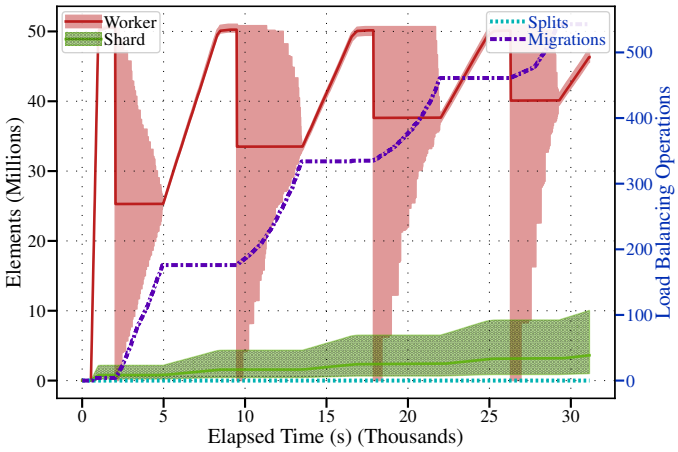


Fig. 15. Load balancing data size per worker as database size N_i and number of workers p increases. $N_i \approx p \cdot 500$ million; $p = 4 \ldots 20$; $m = 2$.

queries with low (below 33%), medium (between 33% and 66%), and high (above 66%) coverage. The throughput and corresponding latency are shown in Figure 16.

Figure 16 shows that VOLAP scales well in an elastic environment. As the database size increases and workers are added to compensate, VOLAP maintains its performance over the entire range of database sizes. The insertion curve is nearly flat at approximately 50 thousand inserts per second. Query performance is unsurprisingly more affected by increasing database size, but the gentle slope averaging approximately 20 thousand per second shows that VOLAP can sustain high throughput and sub-second aggregate queries for very large databases.

VOLAP also supports bulk ingestion which allows data to be loaded at a much higher rate than point insertion. When many records are available to be bulk inserted at once, experiments on the same system show VOLAP to be capable of ingesting data at over 600 thousand items per second for TPC-DS data loaded from disk, and over 1 million items per second for synthetic data generated by the client on the fly.

E. Insert and Query Performance

Figure 17 shows the throughput and latency for insertions and queries with a database of 1 billion items. Performance is measured for various workload mixes and query coverages. Workload mix has a significant impact on throughput because the time spent for insertions and queries may vary considerably.

Figure 17 shows that the "coverage resilience" of the Hilbert PDC tree carries through to VOLAP as a whole: query performance is nearly identical regardless of coverage. The cached aggregate values discussed in Sections III-E and V-A speed up large aggregate queries on the trees, and the parallelism of workers mitigates the impact of sending a large query to many shards.

In these experiments, insertion was approximately three times faster than querying, with a predictable linear relationship between workload mix and overall performance. This also demonstrates that insertions do not significantly impact concurrent query performance.

F. Coverage Impact

A more detailed analysis of the impact of query coverage on performance is shown in Figure 18. Both the impact on individual query time and the number of shards searched are shown as a heat map.

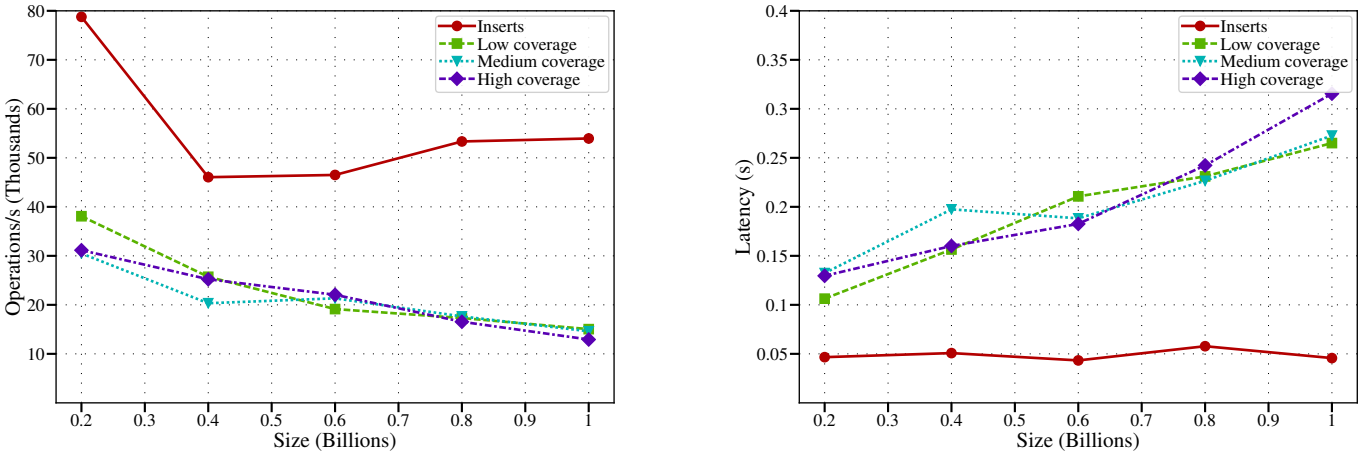
As shown in Figure 18a, the majority of queries are executed very quickly, with a few outliers at low coverage. This reflects the behaviour of the Hilbert PDC tree: with high coverage it is likely that aggregates will be found at higher levels in the tree, making deeper traversal unnecessary. However, with low coverage it may be necessary to walk to the leaf level several times to find individual values, if none of the higher level directory nodes completely cover the query region.

As shown in Figure 18b, the relationship between coverage and number of shards searched is approximately linear, where increasing coverage requires an increasing number of shards to be searched. There are some outlying points at around 50% coverage where many more shards must be searched, however. This is due to queries that intersect many boundaries of the shard partitions, requiring a larger number of shards to be queried.

G. Skew Tolerance

Figure 19 shows the performance of VOLAP on Zipf distributed data with increasing skew s. Hierarchical data is generated such that the overall distribution is close to a Zipf distribution within the contraints of the dimension hierarchies. The system is configured to maintain ingestion performance, with 80 initial shards, a minimum balance improvement of 2 million, and worker statistics updated every second. Query times shown are for the maximum database size of 1 billion items.

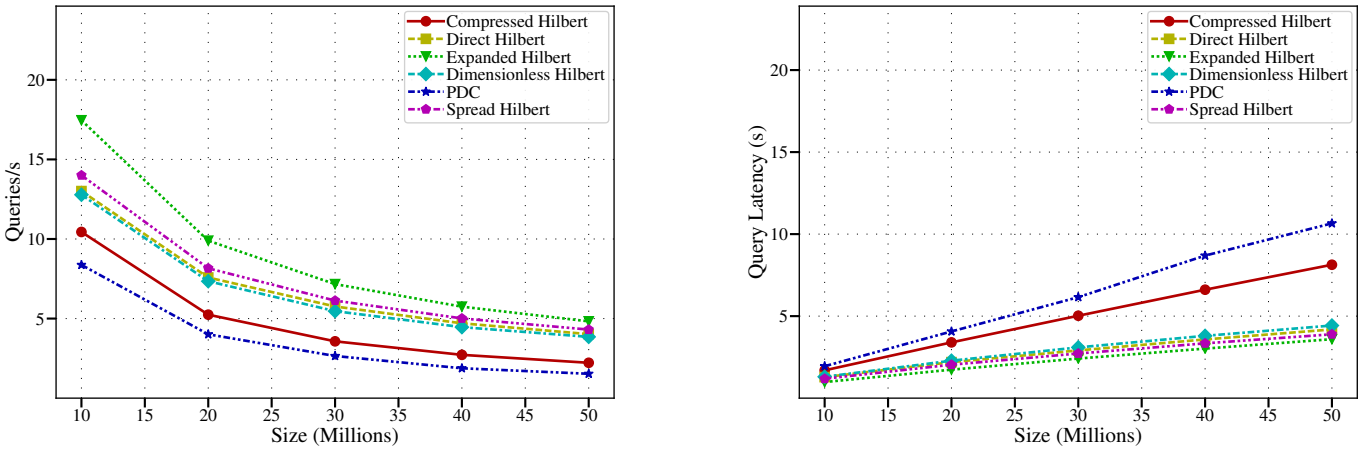
Because of the multi-dimensional nature of the data and the distribution algorithm described in Section III-D, skew and performance do not have a straightforward linear relationship, but as Figure 19 shows, more load-balancing operations are necessary to maintain performance as skew increases. Splits



(a) Throughput

(b) Latency

Fig. 16. Query and insert performance with increasing system size. Database size N_i and number of workers $p = N_i/50$ million ($4 \leq p \leq 20$) both increasing. Low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.



(a) Query Throughput

(b) Query Latency

Fig. 17. Performance for various workload mixes and query coverages. TPC-DS; $N_i = 1$ billion; $p = 20$; $m = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.

tend to impact performance most significantly, since they only occur when migration of existing shards can not address the imbalance. Worker threads are thus kept busy for longer before the migration can occur and balance be improved. This is particularly visible in this experiment with a skew of 2.0, where ingestion performance drops to just above 800 thousand inserts per second. However, in general, VOLAP maintains relatively stable performance over a wide range of skew, sustaining well over a million inserts per second on average. This is because once heavily loaded shards are split and/or migrated, their load is more evenly distributed amongst workers, so the skew of the data contributes less to skew between workers as time progresses.

The effect of load balancing over time for a run with a high skew of 8.0 can be seen in Figure 20 which shows the number of items stored on each worker over time.

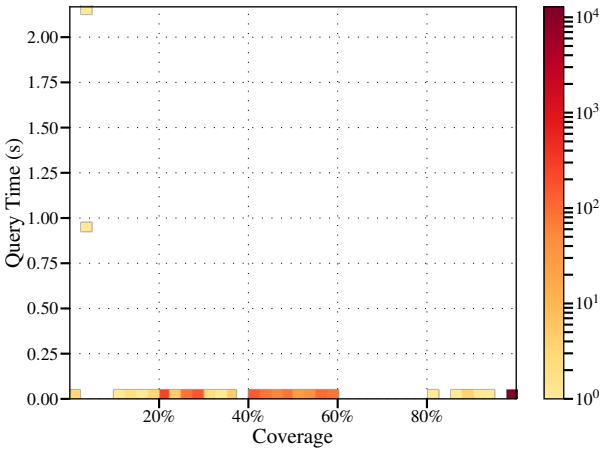
H. Replication Impact

Figure 21 shows the operation latency with varying values of N_r, W and R, with $N_i = 300{,}000{,}000$ items. The number of worker nodes is the same regardless of the value of N_r, and consequently the amount of load on the set of workers is proportional to N_r. As a result, increasing N_r with this configuration has a negative impact on performance.

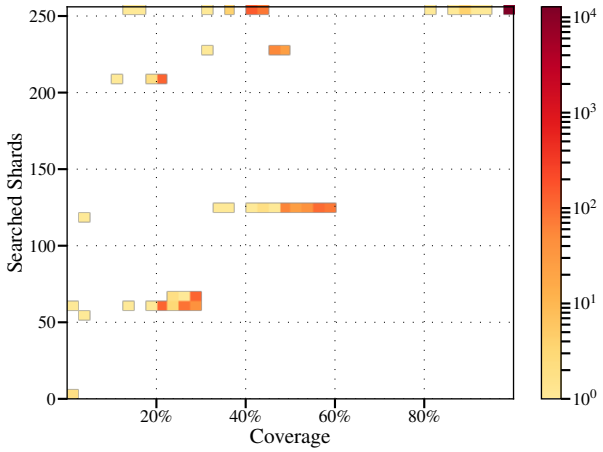
The addition of replicas increases latency by approximately 0.1 to 0.2 seconds in this configuration, as shown in Figure 21. Since all of the compute nodes in this experiment communicate over a low latency local network, there is little variance in the time it takes each worker to return a response to a server. Because of this, the value of the read and write quorum has a minimal effect on operation latency with this configuration.

I. Query Freshness

Without shard replication, user sessions attached to the same server will observe a very low time between an insert being issued and its effect being visible in subsequent queries, since no global synchronization is required. To synchronize sessions across servers, VOLAP periodically initiates a synchronization of the servers through Zookeeper at a configurable rate, set to

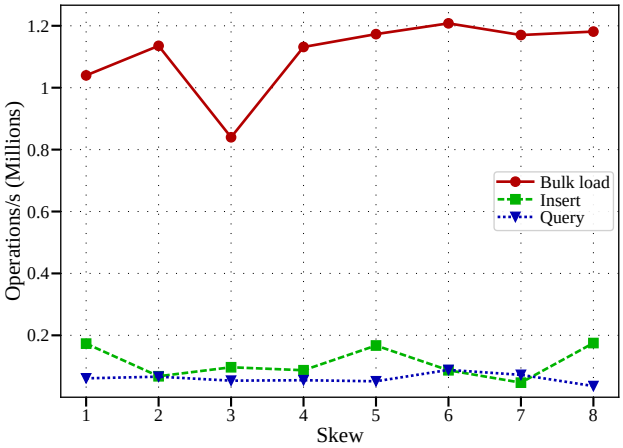


(a) Query time vs. coverage.

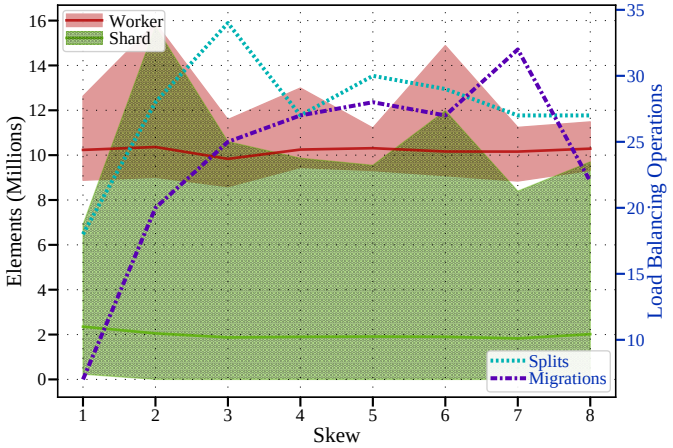


(b) Searched shards vs. coverage.

Fig. 18. Effect of coverage on query performance; $N_i = 1$ billion; $p = 20$.



(a) Performance vs. skew.



(b) Load balance vs. skew.

Fig. 19. Performance and load balancing with increasing data skew; $N_i = 1$ billion; $p = 20$.

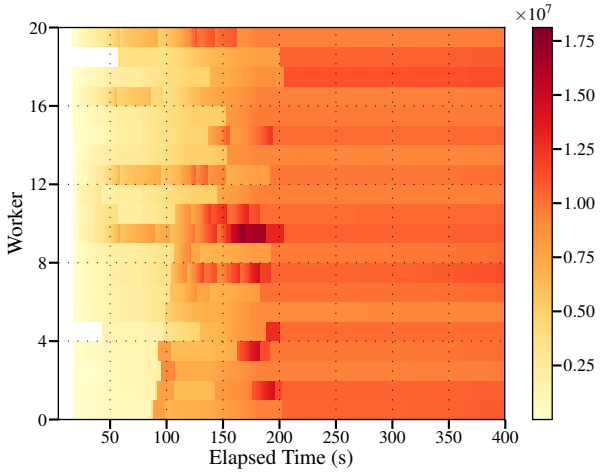


Fig. 20. Distribution of data across workers during a run with Zipf data with skew $s = 8.0$.

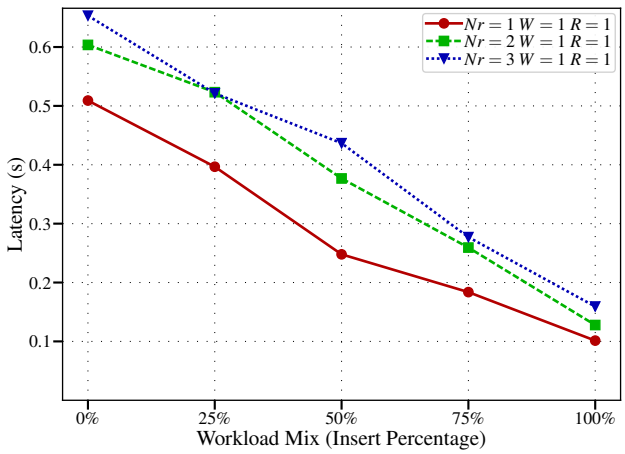


Fig. 21. Latency for a stream with 50% workload mix for various replication factors; N_i=300,000,000; $p = 20$; $m = 2$

3 seconds in these experiments. The time between an aggregate query issued on one server and a prior insert operation issued on a different server is referred to as the *elapsed time*. In these experiments the number of missed insertions drops very close to zero after only 7 ms, and, on average, to zero after 20 ms.

These figures represent typical behaviour, though it is possible for consistency to take longer in some edge cases. In the many experiments performed with VOLAP, consistency between insert and query operations executed on different servers was always observed in under 3 seconds. When shard replication is enabled, measuring consistency is more complex, and is outside the scope of this paper.

VI. Conclusion

VelocityOLAP (VOLAP) is a scalable OLAP system which allows high-velocity data to be queried in real-time. A novel underlying data structure exploits dimension hierarchies at a low level to allow aggregating large portions of the database quickly without materializing multi-dimensional views, and supports a very high rate of data ingestion. A fully decentralized architecture supports horizontal scaling, allowing VOLAP to scale up to very large sizes using only commodity hardware or modest cloud instances.

VII. Acknowledgements

Research partially supported by the IBM Center for Advanced Studies Canada and the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[2] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Know. Disc.*, vol. 1, pp. 29–53, 1997.

[3] F. Dehne, D. E. Robillard, A. Rau-Chaplin, and N. Burke, "VOLAP: A scalable distributed system for real-time OLAP with high velocity data," in *Proc. IEEE Cluster*, 2016.

[4] F. Yang, E. Tschetter, X. Leaute, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proc. ACM SIGMOD*, 2014.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA database – an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[6] P. Francisco *et al.*, "The Netezza data appliance architecture: A platform for high performance data warehousing and analytics," *IBM Redbooks*, vol. 3, 2011.

[7] D. E. Robillard, F. Dehne, A. Rau-Chaplin, and N. Burke, "The Hilbert PDC-tree: A high-velocity structure for many-dimensional data," in *Proc. IDEAS*, 2016.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX*, 2010.

[9] M. Poess, B. Smith, L. Kollar, and P. Larson, "TPC-DS, Taking decision support benchmarking to the next level," in *Proc. ACM SIGMOD*. ACM, 2002, pp. 582–587.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[11] "Hadoop," http://hadoop.apache.org/.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] R. Bruckner, B. List, and J. Schiefer, "Striving towards near real-time data integration for data warehouses," *Proc. DaWaK*, pp. 173–182, 2002.

[14] D. Jin, T. Tsuji, and K. Higuchi, "An Incremental Maintenance Scheme of Data Cubes and Its Evaluation," *Proc. DASFAA*, pp. 36–48, 2008.

[15] R. J. Santos and J. Bernardino, "Real-time data warehouse loading methodology," *Proc. IDEAS*, pp. 49–58, 2008.

[16] ——, "Optimizing data warehouse loading procedures for enabling useful-time data warehousing," *Proc. IDEAS*, pp. 292–299, 2009.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB*, pp. 1626–1629, 2009.

[18] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proc. VLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "BigTable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[20] "BigQuery," http://developers.google.com/bigquery/.

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB*, pp. 330–339, 2010.

[22] "Twitter storm," http://storm-project.net/.

[23] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "ES^2: A cloud data storage system for supporting both OLTP and OLAP," in *Proc. ICDE*, 2011, pp. 291–302.

[24] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "Providing scalable database services on the cloud," in *Proc. WISE*, 2010, pp. 1–19.

[25] H. Han, Y. C. Lee, S. Choi, H. Y. Yeom, and A. Y. Zomaya, "Cloud-aware processing of MapReduce-based OLAP applications," in *Proc. Australasian Sym. on Par. Distr. Comp.*, 2013, pp. 31–38.

[26] J. Li, F. Z. Wang, L. Meng, W. Zhang, and Y. Cai, "A map-reduce-enabled SOLAP cube for large-scale remotely sensed data aggregation," *Computers & Geosciences*, 2014.

[27] K. Doka, D. Tsoumakos, and N. Koziris, "Online querying of d-dimensional hierarchies," *J. Par. Distr. Comp.*, vol. 71, no. 3, pp. 424–437, 2011.

[28] A. Asiki, D. Tsoumakos, and N. Koziris, "Distributing and searching concept hierarchies," *Cluster Computing*, vol. 13, pp. 257–276, 2010.

[29] K. Doka, D. Tsoumakos, and N. Koziris, "Brown Dwarf: A fully-distributed, fault-tolerant data warehousing system," *J. Par. Distr. Comp.*, vol. 71, no. 11, pp. 1434–1446, 2011.

[30] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, "Dwarf: Shrinking the PetaCube," in *Proc. ACM SIGMOD*, 2002, pp. 464–475.

[31] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," in *Proc. VLDB*, 2010, pp. 1207–1218.

[32] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proc. ACM SIGMOD*, 2010, pp. 591–602.

[33] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *Proc. Int. W. Cloud Data Management*, 2009, pp. 17–24.

[34] M. C. Kurt and G. Agrawal, "A fault-tolerant environment for large-scale query processing," in *Proc. HiPC*, 2012, pp. 1–10.

[35] K. Doka, D. Tsoumakos, and N. Koziris, "Brown dwarf: A fully-distributed, fault-tolerant data warehousing system," *Journal of Parallel and Distributed Computing*, vol. 71, no. 11, pp. 1434–1446, 2011.

[36] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, apr 2011, pp. 195–206.

[37] F. Dehne, Q. Kong, A. Rau-Chaplin, H. Zaboli, and R. Zhou, "A distributed tree data structure for real-time OLAP on cloud architecture," in *Proc. IEEE Big Data*, 2013.

[38] F. Dehne and H. Zaboli, "Parallel real-time OLAP on multi-core processors," in *Proc. CCGRID*, 2012, pp. 588–594.

[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *USENIX*, vol. 8, 2010, pp. 11–11.

[40] M. Ester, J. Kohlhammer, and H.-P. Kriegel, "The DC-tree: A fully dynamic index structure for data warehouses," in *Proc. ICDE*, 2000, pp. 379–388.

[41] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[42] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *Proc. VLDB*, 1994, pp. 500–509.

[43] C. H. Hamilton and A. Rau-Chaplin, "Compact Hilbert indices: Space-filling curves for domains with unequal side lengths," *Information Processing Letters*, vol. 105, no. 5, pp. 155–163, Feb 2008.

[44] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[45] 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," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.