VOLAP: A Scalable Distributed System for Real-Time OLAP with 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, and exploits both multi-core and multi-processor parallelism. 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 400 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 powerful technology for 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. Accordingly, many traditional OLAP systems take the static *data cube* approach [2] and materialize multi-dimensional views to ensure high query performance. However, such systems can only be updated periodically in batches, e.g. once every week, which prevents queries from including the most recent data. More modern systems avoid materialization, but still often incur a delay between new data being ingested and that data being available for analysis. The well-known "5 V's of Big Data" (volume, velocity, variety, veracity, value) highlight the importance of processing large data sets that have a high rate of change, or *velocity*. As data velocity increases, stale results become increasingly problematic. Applications that monitor high velocity data streams require the ability to analyze new data as it arrives, in real-time.

This paper introduces VelocityOLAP (VOLAP), 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 [3], VOLAP does not use a special partitioning dimension that needs to be manually configured. All dimensions are treated equally, and VOLAP scales well to a high number of 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 [4], 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 [5]. 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 is a fully distributed system that partitions data into shards stored on *worker* nodes of the cloud environment. Shards are stored using the novel *Hilbert PDC tree*, which supports multi-threaded insert and aggregate query operations on many dimensions. Dimension hierarchies are exploited at a low level to allow aggregating large portions of the database quickly without the need to materialize multi-dimensional views. 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 [6] is used for managing global state information. A *manager* background process monitors the status of the system and

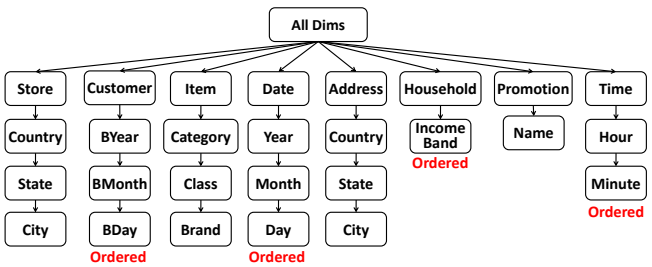


Figure 1. Dimension Hierarchies for TPC-DS Data.

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 additional load.

VOLAP has been tested on data from the TPC-DS [7] test suite which uses the dimension hierarchies shown in Figure 1. Using 20 `c3.4xlarge` worker instances on Amazon EC2 with a database of 1 billion items, VOLAP is able to ingest new data at a rate of over 400 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. 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.

Each user session is attached to one of the server nodes. To maximize throughput, VOLAP processes all requests in parallel, but aims to minimize the time required for an insert to be included in later queries. Analysis of VOLAP in terms of Probabilistically Bounded Staleness (PBS) [8] shows that the probability of an inconsistency between an insert operation issued at time t_1 on one server and a subsequent query issued at time t_2 on a different server drops to close to zero after *elapsed time* $t_2 - t_1 = 0.25$ seconds. In our experiments, over hundreds of billions of tests, consistency between insert and query operations executed on different servers was always observed in under 3 seconds.

Compared to previous work, VOLAP introduces novel index and worker data structures, a fully decentralized elastic design with support for multiple servers, a new synchronization scheme with configurable freshness, and improved load-balancing algorithms for a fully decentralized environment. The new Hilbert PDC tree supports a high rate of data ingestion while providing good performance for queries that aggregate both a large or small portion of the database with many dimensions.

The remainder of this paper is organized as follows: Section II discusses recent related work. Section III describes VOLAP's design, including network architecture, synchronization, data representation, and load balancing. Section IV shows experimental results that demonstrate the performance of the prototype implementation, and Section V presents the conclusion.

II. Related Work

Many published systems store and query large data sets in distributed environments. Hadoop [9] and its file system HDFS are popular examples, with applications typically built on MapReduce [10]. However, these systems are not designed for *real-time* operation. Instead, they are based on batch processing or "quasi real-time" operations [11], [12], [13], [14]. The situation is similar for Hive [15], HadoopDB [16], BigTable [17], BigQuery [18], and Dremel [19]. To overcome the batch processing in Hadoop based systems, Storm [20] introduced a distributed computing model that processes in-flight Twitter data. However, Storm assumes small, compact Twitter style 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 [21], [22], [23], [24] 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 [25], [26], [27], [28]. However, none of these methods provide *real-time* OLAP functionality.

Various publications on distributed B-trees for cloud platforms exist [29], 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 [30], [31], [32]. However, these methods do not support dimension hierarchies which are essential for aggregate queries, and do not scale well to a large number of dimensions.

The systems closest to VOLAP are Druid [3], SAP HANA [4], IBM Netezza data warehouse appliance [5], HyPer [33], and CR-OLAP [34]. The remainder of this section will discuss these in more detail.

Druid [3] is an open-source, distributed, scalable, 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 have equal importance.

SAP HANA [4] 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, and this single master node becomes a bottleneck as the system size increases.

The IBM Netezza data warehouse appliance [5] relies on special purpose FPGA boards that provide a hardware implementation of OLAP functionality.

HyPer [33] 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 all the data unnecessarily. The HyPer model conceptually provides a low overhead 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 [34], 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 [35] as a building block, but as one large 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

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* [36] cluster for global system state.
- A *manager* background process for analyzing global state and initiating load balancing operations.

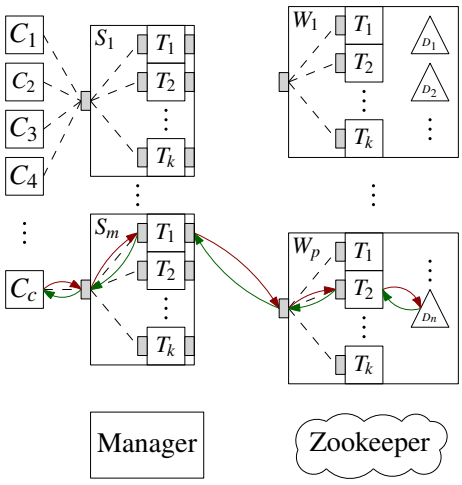


Figure 2. System architecture. Arrows illustrate the path of an insert or query operation through the system.

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 Subset (MDS, multiple boxes) [37]. 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.

Servers, workers, and the manager communicate using ZeroMQ [38], a high-performance asynchronous messaging library designed for scalable distributed applications. ZeroMQ is used both for inter-process and inter-thread communication. Each thread has local sockets for sending and receiving messages without locking, and incoming network requests are load-balanced between threads based on their availability.

B. System Image

The *system image* represents the global system state, and is stored in Zookeeper [36], a fault tolerant distributed coordination service. The image contains the global information required by servers and the manager, including lists of the current workers and servers, configuration parameters, and for each shard its size, bounding box, and the address of the worker where it is located.

Each server maintains a *local image* which is used to perform insertions and queries. The local image 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).

If the local image is changed by an insertion, the server updates the global image in Zookeeper at a configurable rate. In the experiments presented here, servers update Zookeeper every 3 seconds as necessary. Since changes take time to reach other servers, the local image of each server may become outdated. Servers make use of Zookeeper's *watch* facility to be notified of changes without wasteful polling. Workers update shard statistics in Zookeeper periodically as well, to allow the manager to plan load balancing operations.

C. Index Data Structure

The server's local image is responsible for finding the shards relevant to each insertion or query. A fast index structure for each local image is therefore crucial for high performance. Two key aspects of the index affect performance: the speed of searching for shards, and the structure that results from choosing a given shard for an insert. In particular, overlapping shards increase the likelihood that queries must be sent to many workers; hence, minimizing shard overlap is critically important.

We use a modified PDC tree [35] 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 leaves of the tree correspond to shards, i.e. there are

exactly n leaves in the index tree. The search performed for a query is straightforward: starting from the root, search every child whose bounding box touches the query box. If a leaf is reached, that shard must be queried. Insertions differ from 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 VOLAP 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.

Though insertions do not modify the structure of the tree, synchronization with the global image in Zookeeper requires operations that do. 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 must be expanded. This operation is unique, since the expansion is a bottom-up operation. Searching for a leaf via its bounding box is expensive since there may be overlap higher in the tree, forcing the search to visit several subtrees of a single node. Instead, VOLAP keeps a separate index of pointers to leaf nodes, keyed by shard ID. When a bounding box expands, the leaf is found immediately using this index, the leaf's bounding box is expanded, and the expansion is propagated up the tree towards the root as necessary. This operation temporarily violates the tree invariant that a node's bounding box encompasses all of its children. However, this is not problematic since it is only used for synchronization, and will never cause queries to miss data they would have otherwise included prior to the start of the synchronization.

Being based on the PDC tree, the index data structure is multi-threaded and uses minimal locking: operations hold only one or two node locks at a given time, and never lock entire subtrees. Thus, many operations can be performed in parallel. Servers use many threads, all using the same index in parallel, to be able to maintain a high throughput to fully utilize workers.

D. Shard Data Structure: Hilbert PDC Tree

Each shard is stored in an in-memory data structure designed for a multi-core compute node. The shard data structures handle a stream of insert and aggregate query operations. VOLAP currently includes five data structures for shards: a simple array for benchmarking purposes, two variants of the PDC tree [35], and two variants of the novel *Hilbert PDC tree*. All variants share the same multi-threaded underlying tree implementation, but use either Minimum Describing Subsets (MDS) or Minimum Bounding Rectangles (MBR) as keys.

Most applications are best served by the Hilbert PDC tree, which is designed to suit the needs of a high velocity OLAP system like VOLAP. The Hilbert PDC tree is, like the PDC tree and R-tree before it, a multi-dimensional index where each node has a bounding box which encompasses that of all its children. It supports a much higher rate of data ingestion than the PDC tree by using the maximum Hilbert number of nodes to quickly determine an insert position.

This approach is similar to that of the Hilbert R-Tree [39], but applying a Hilbert ordering to the MDS keys used in a PDC tree introduces new issues. In particular, MDSs in the tree are expressed at various levels, where nodes higher in the tree are likely to have keys at higher levels in the dimension hierarchy. This means that MDSs are often compared at levels different than the leaf levels for which the Hilbert indices stored in the tree were calculated. Since the breadth of various levels may vary considerably across dimensions, the Hilbert order for leaves may not provide good locality for keys higher in the tree which are expressed at higher levels in the dimension hierarchy.

To resolve this issue, 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 otherwise IDs in each dimension would occupy disjoint numerical ranges.

Figure 3 shows a simple example for an ID with two dimensions. At level 4, dimension 2 uses only two bits, but dimension 1 uses four bits. To compensate, IDs in level 4 in dimension 2 are shifted left two places, causing values to span roughly the same numerical range as those in dimension 1. This transformation is only performed on a copy of the key used to calculate the corresponding Hilbert number, the keys in the tree used for comparison during queries are unmodified. A downside to the Hilbert approach is the additional space required to store a Hilbert index for each tree node. To minimize this overhead, compact Hilbert indices [40] are stored, which use the minimum number of bits required to represent values given the range 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

Figure 3. Transforming hierarchical IDs for Hilbert mapping.

Given the Hilbert index for the item to be inserted, the tree inserts in a similar fashion to linearly ordered trees such as a B+ tree. Since no geometric calculations are performed during insertion, data ingestion is significantly faster than with a PDC tree. The node splitting algorithm also differs from the PDC tree, since the fixed ordering of children

precludes the use of R-tree-like split algorithms. 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.

E. 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. High performance requires a load balancing scheme which offers a good trade-off between load balancing overhead and effectiveness. Dynamic load balancing permits VOLAP to add, remove, or replace servers and workers dynamically in order to maintain performance in the face of changing system load.

In VOLAP, 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. The split and migration processes are designed to allow shards to split and move transparently between workers while the system continues to service both insert and aggregate query requests.

In order to support load-balancing, the shard data structures provide the following operations:

- $\mathrm{SplitQuery}(D_i, B_i)$ 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), and the corresponding $\mathrm{DeserializeShard}(b)$ which builds the data structure from such a blob.

A shard D_i stored on W_s (the *source worker*) can be migrated to another worker W_d (the *destination worker*) 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 the load balancer requires smaller shards for migration. Each worker W_k stores a *mapping table* M_j to handle operations while a split is in progress. 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. To support insertion during splits and migrations, an insertion queue is created for the relevant shard. During the operation, new items for the shard are inserted into the queue to prevent the shard from growing continuously while it is being serialized. The insertion queue uses the same data structure as shards, and is queried along with the shard itself for any incoming queries that touch the shard. Thus, query processing is not interrupted while a split is in progress. When the operation is finished, the insertion queue is drained into the shard and destroyed.

IV. Experimental Evaluation

We evaluated the performance of VOLAP with respect to the system size, *workload mix* (percentage of inserts in the operation stream; e.g. workload mix 25% is 25% inserts and 75% aggregate queries), and *query coverage* (percentage of the database that needs to be aggregated for a query). We used the TPC-DS decision support data set from the Transaction Processing Council, with $d = 8$ hierarchical dimensions as shown in Figure 1. Experiments were performed on Amazon EC2, using `c3.8xlarge` instances for servers, `c3.4xlarge` instances for workers, and `c3.2xlarge` instances for clients and the manager. At the time of writing, these instances are based on Intel Xeon E5-2680 (Ivy Bridge) processors. All instances are 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.

A. Data Structure Performance

Figure 4 shows the performance of the Hilbert PDC tree compared to the PDC tree [35] for queries with different coverage, using the TPC-DS data set with a single tree on one worker instance.

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-D) produces less overlap at lower levels in the tree than the

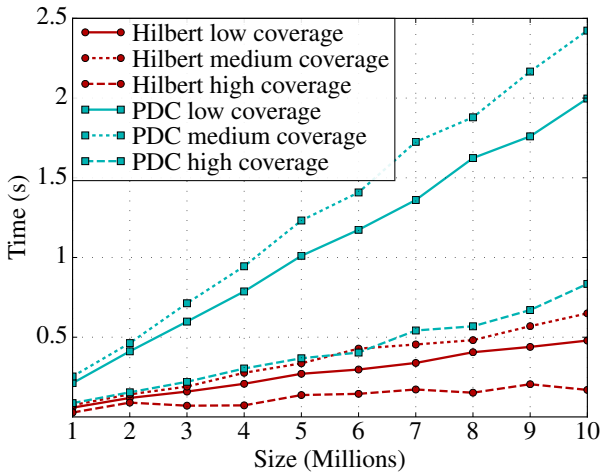


Figure 4. Query performance of Hilbert PDC tree vs. PDC tree for various query coverages.

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. The Hilbert PDC tree preserves this scalability in query performance, and significantly improves it for insertion. Figure 5 compares the insert and query performance of both conventional and Hilbert R trees and PDC trees. 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.

For the TPC-DS data set, preliminary experiments showed that the Hilbert PDC tree out-performs the PDC tree in all cases. Hence, all subsequent experiments in this section use the Hilbert PDC tree as the underlying data structure.

B. 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 6 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.

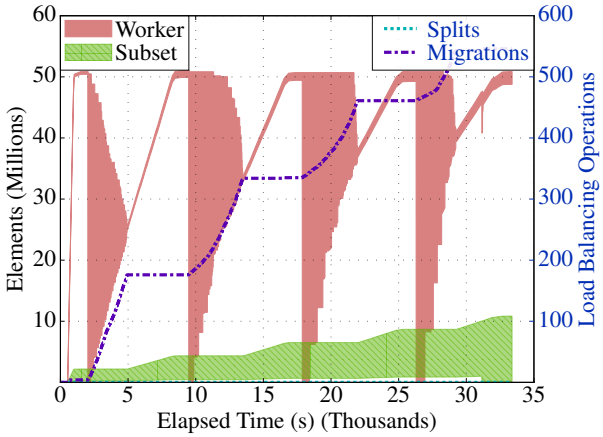


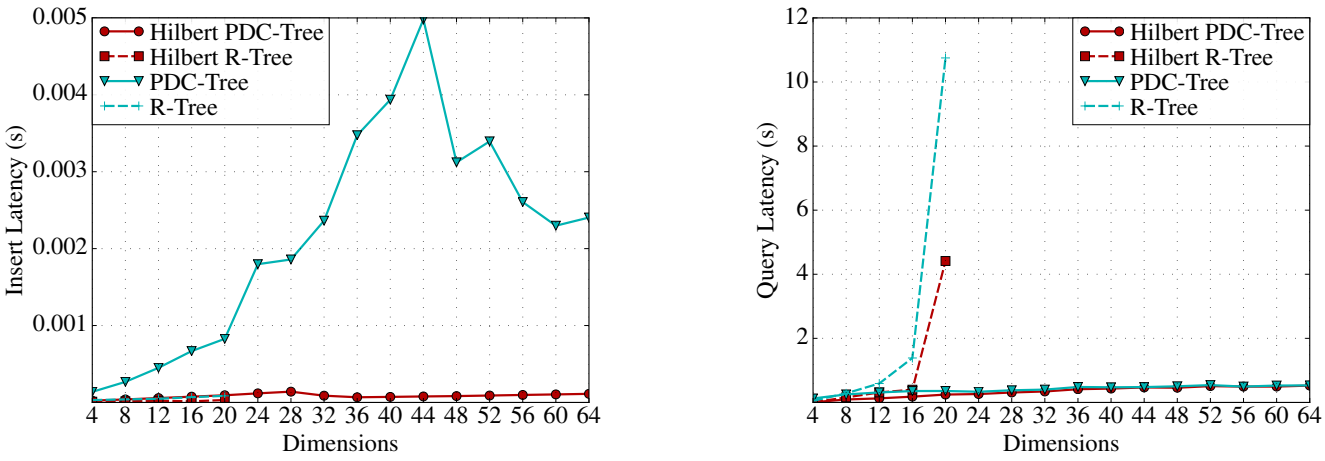
Figure 6. Load balancing data size per worker as database size N and number of workers p increases. $N \approx p \cdot 500$ million; $p = 4 \ldots 20$; $m = 2$.

C. Horizontal Scale-Up Performance

Figure 7 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 6, where two new empty workers are added at each scale-up step. For each system size with p workers and $N \approx p \times 50$ million data elements, benchmarks are performed for insertions as well as queries with low (below 33%), medium (between 33% and 66%), and high (above 66%) coverage. The throughput and corresponding latency are shown in Figure 7.

Figure 7 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.

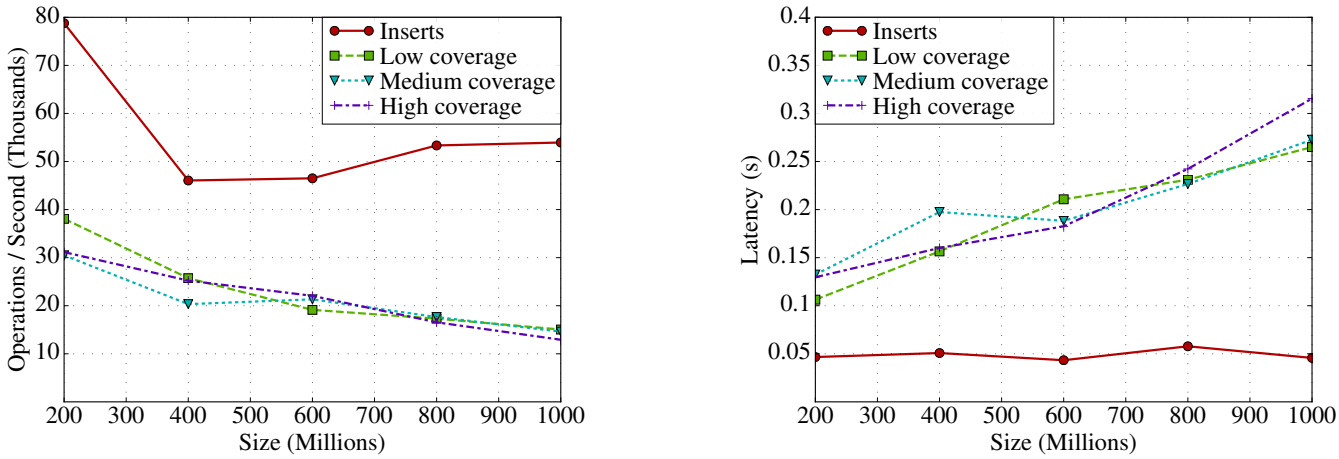
VOLAP also supports bulk ingestion which allows data to be loaded at a much higher rate than point insertion. When many records are available to be bulk inserted at once, experiments on the same system have shown VOLAP to be



(a) Insert latency

(b) Query latency

Figure 5. Performance of tree variants as the number of dimensions is increased.



(a) Throughput

(b) Latency

Figure 7. Query and insert performance with increasing system size. Database size N and number of workers $p = N/50$ million ($4 \leq p \leq 20$) both increasing. Low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.

capable of ingesting data at over 400 thousand items per second.

D. Insert and Query Performance

Figure 8 shows the throughput and latency for insertions and queries at a fixed database size of 1 billion. Performance is measured for various workload mixes (percentage of insert operations) and query coverages (percentage of the database aggregated by a query). Workload mix has a significant impact on throughput because the time spent for insertions and queries may vary considerably, and insertions may manipulate data structures that are concurrently being used to answer queries.

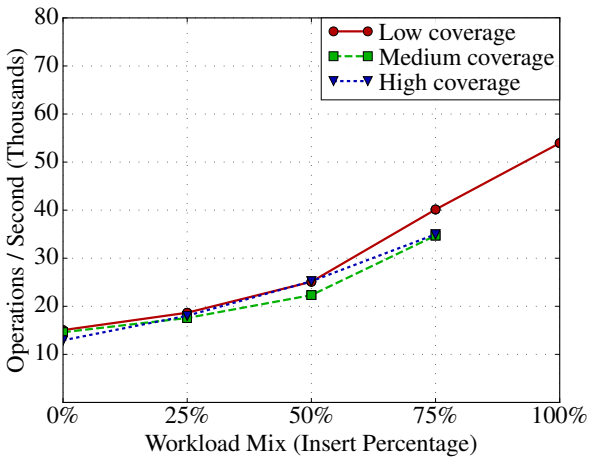
Figure 8 shows that the "coverage resilience" of the Hilbert PDC tree carries through to VOLAP as a whole: query performance is nearly identical regardless of coverage. As discussed in Sections III-D and IV-A, the Hilbert PDC tree stores aggregate values at all levels in the tree which speeds up aggregate queries significantly by preventing large aggregations from needing to scan the entire database.

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.

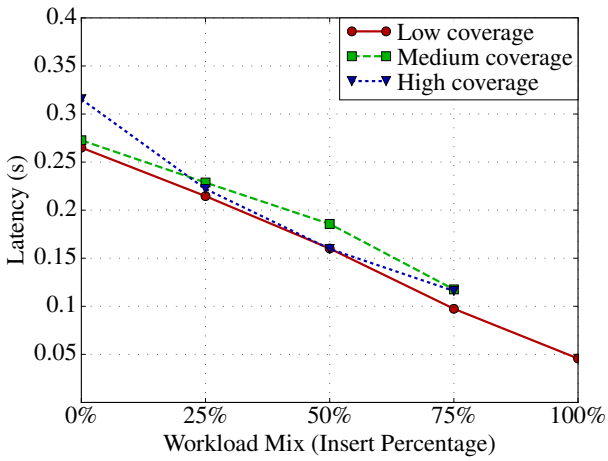
E. Coverage Impact

A more detailed analysis of the impact of query coverage on performance is shown in Figure 9. Both the impact on individual query time and the number of shards searched are shown as a heat map.

As shown in Figure 9(a), 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

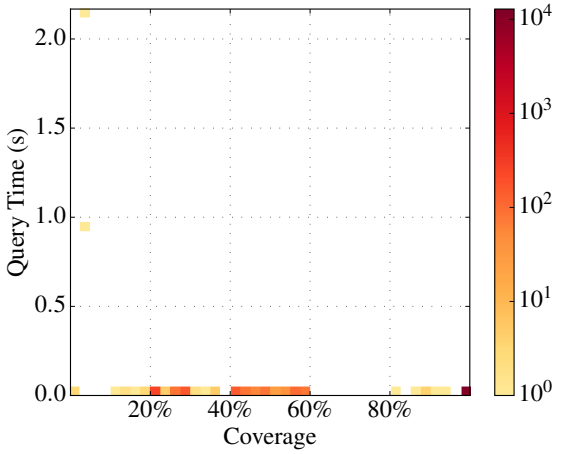


(a) Query Throughput

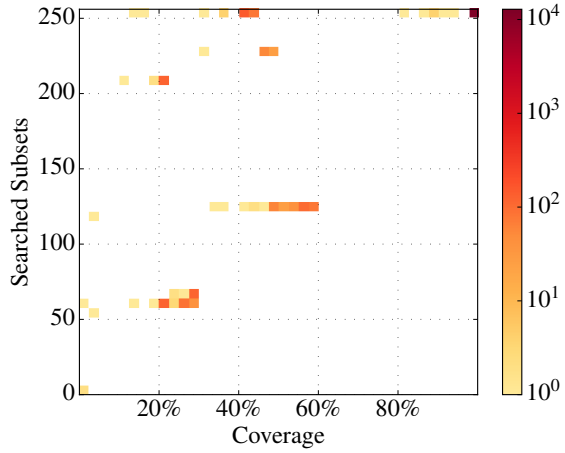


(b) Query Latency

Figure 8. Performance for various workload mixes and query coverages. TPC-DS; $N = 1$ billion; $p = 20$; $m = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.



(a) Query time vs. coverage.



(b) Searched shards vs. coverage.

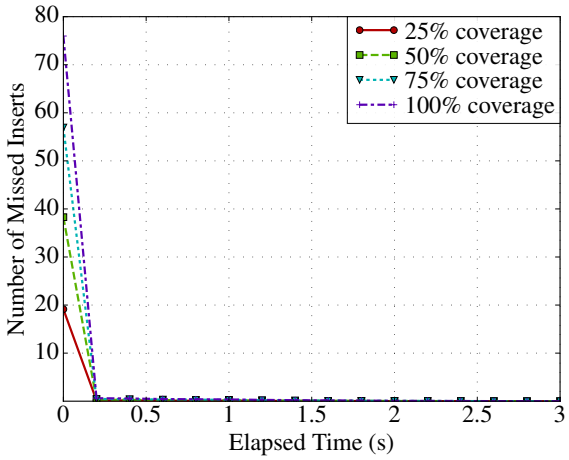
Figure 9. Effect of coverage on query performance; $N = 1$ billion; $p = 20$.

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 is more likely that higher level directory nodes do not precisely cover the query region. Thus, it may be necessary to descend to deep levels of the tree to find directory nodes that cover a small enough region, or even descend all the way to the leaf level to find individual values.

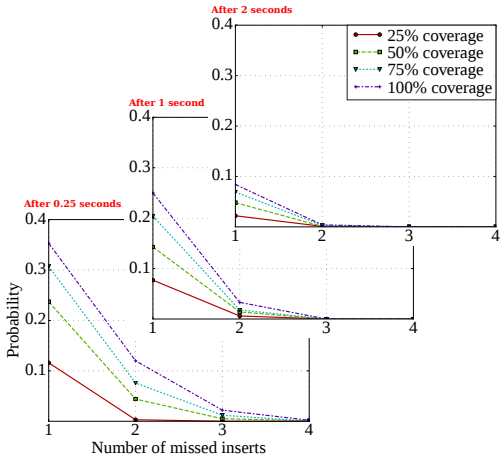
As shown in Figure 9(b), the relationship between coverage and number of shards searched is approximately linear, where increasing coverage requires an increasing number of shards to be searched. There are some outlier points at around 50% coverage where many more shards must be searched, however. This is due to queries that intersect many boundaries of the shard partitions, requiring a larger number of shards to be queried.

F. Query Freshness

Each user session is attached to one of the server nodes. 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. These experiments use the default rate of 3 seconds. 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 order to estimate the number of possibly missed inserts in an aggregate query result relative to elapsed time, a simulation was performed using TPC-DS data and the query and insert latency distributions observed for VOLAP in these experiments.



(a) Avg. number of missed inserts rel. to elapsed time.



(b) Probability of missed inserts after 0.25, 1 and 2 seconds elapsed time.

Figure 10. Serialization between user sessions attached to different servers.

Figure 10(a) shows the average number of missed inserts relative to elapsed time. The number of missed inserts drops to close to zero after 0.25 seconds. Figure 10(b) examines in more detail the probability of 1 to 4 missed inserts after an elapsed time of 0.25, 1, and 2 seconds. The same behaviour is observed for any database size $n \geq 500,000$, including the experiments shown in Figure 7 and Figure 8, because only the most recent three seconds of inserted data contain items that are ever missed. For example, for a database with n = 1 billion data items, a large query with 50% coverage (500 million reported data items) and 1 second elapsed time, there is approximately a 1% probability of missing 2 data items (0.0000004% of the result). In the many experiments performed with VOLAP, consistency between insert and query operations executed on different servers was always observed in under 3 seconds.

V. Conclusion

VelocityOLAP (VOLAP) is a scalable OLAP system which allows high velocity data to be queried in real-time. A novel underlying data structure, the Hilbert PDC tree, exploits dimension hierarchies to allow aggregating large portions of the database quickly without materializing views, and supports a very high rate of data ingestion. VOLAP uses a fully decentralized architecture which supports horizontal scaling, allowing the system to scale up to very large sizes and maintain performance using only commodity hardware or modest cloud instances. Experiments confirm that VOLAP performs well for queries ranging from aggregations of a few elements to aggregations of nearly the entire database.

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

[4] 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.

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

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

[7] 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.

[8] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Quantifying eventual consistency with pbs," *The VLDB Journal*, vol. 23, no. 2, pp. 279–302, 2014.

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

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

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

[40] 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.