Scalable Distributed Architectures for Real-Time Data Processing: A Novel Approach to Adaptive Analytical Querying

Sowmith Reddy Thukkani
Eastern Illinois University
sowmithreddy.th@gmail.com

***Abstract*—Real-time analytics demands scalable distributed architectures that can balance performance and consistency. This work presents R-Store, a novel integration-driven architecture combining adaptive query execution, stream processing, and hybrid OLAP-OLTP capabilities. Evaluated on a 144-node testbed using a Zipf-distributed TPC-H workload, R-Store achieves over 100K updates/sec with analytical accuracy and timestamp-consistent cube views. It outperforms traditional streaming systems by 27% in throughput and demonstrates efficient cube maintenance and query execution with predictable I/O cost modeling. Our architecture contributes a reproducible, low-latency solution for next-generation real-time analytics.**

***Index Terms*—Real-time analytics, distributed systems, OLAP, data cube, scalability, adaptive querying, HBase, MapReduce, reliability, big data processing.**

I. Introduction

Real-time data analytics has become critical in domains ranging from financial services to IoT-driven infrastructure. Traditional architectures such as HBase, MapReduce, and Lambda-based systems, while powerful, often fall short in supporting adaptive and low-latency analytics under high-throughput workloads [7]–[9].

The major challenges include ensuring consistency in distributed streaming, handling out-of-order updates, and maintaining analytical responsiveness. Recent advances like Delta Lake [7] and Apache Iceberg [8] have improved transactional support, yet lack unified mechanisms for adaptive OLAP-OLTP integration.

This work is motivated by the gap between static batch-stream hybrids and the need for fully adaptive analytics. R-Store addresses this by providing a cohesive platform integrating MVCC, cube views, DeltaScan, and timestamp-based consistency in a modular fashion.

The key contributions of this work are:

- An adaptive query execution model integrated with MVCC and delta update cubes.
- A novel MultiTableInputFormat and DeltaScan method for scalable dataflow.
- A reproducible benchmarking methodology using a Zipf-distributed TPC-H workload.
- A performance evaluation on a 144-node cluster showing significant throughput gains.

Organization: The paper begins with an overview of existing literature, followed by a detailed explanation of the proposed R-Store architecture. It then presents experimental evaluations, and finally concludes with key findings and directions for future research.

II. Related Work

HStreaming and Rockset [9] offer real-time ingestion and indexing but fall short in native support for hybrid OLAP-OLTP transformations.

In comparison, R-Store unifies MVCC-based concurrency with adaptive cube updates and streaming join/filter mechanisms. Unlike Lambda architecture, R-Store dynamically reconfigures dataflow using DeltaScan and timestamp-based triggers.

While prior work explored concurrency and update cubes, our work enhances this through practical deployment with benchmark-driven results. Compared to works from 2008–2010, R-Store brings a more modular, distributed, and efficient implementation.

III. R-Store Framework and Structural Design

This section outlines the architectural framework of R-Store, the foundational principles underlying its storage strategy, and the methodology adopted for maintaining the data cube.

A. R-Store Framework

Figure III-A illustrates the structural layout of R-Store. The framework comprises four core components: a distributed key-value storage system, a real-time stream processor responsible for updating the evolving data cube, a batch-processing module utilizing MapReduce for large-scale analytical query handling, and a MetaStore, which manages global metadata and system configurations.

Transactional operations are executed directly on the key-value storage system, whereas analytical requests are handled via the batch-processing unit. A fundamental approach to enabling real-time analytical processing (RTOLAP) within a batch-processing architecture involves scanning the entire real-time dataset to retrieve the most up-to-date state before executing an analytical query (referred to as the CompleteScan process). The underlying key-value store must support multi-version concurrency to minimize conflicts between transactional and analytical workloads. However, this technique is computationally expensive, as retrieving historical versions for

each key-value pair in large-scale distributed environments results in significant processing overhead.

Real-world applications, such as online platforms, exhibit update distributions that follow a Zipfian pattern, where a limited subset of keys is modified within a specific timeframe. Exploiting this characteristic, analytical query efficiency is improved by precomputing a data cube representation of the real-time dataset. When an analytical request is issued, the system initially interacts with MetaStore to obtain the query timestamp, ensuring result consistency. Moreover, MetaStore maintains statistical metrics that aid in query optimization through a cost-based model (explored in Section V-C). Following optimization, the analytical query is converted into a batch-processing task that processes both historical data from the data cube and recent modifications stored in the key-value system.

To enhance real-time data access, the key-value repository implements an incremental scanning mechanism (discussed in Section III-B1). The most recent dataset is retrieved via the IncrementalScan operation, while historical data is accessed using the CompleteScan function. The IncrementalScan method selectively fetches key-value pairs modified after the last data cube update, significantly lowering computational costs compared to the CompleteScan approach.

The data cube is persistently stored in the distributed key-value system and undergoes periodic refreshes based on real-time updates. Older key-value pair versions preceding the refresh cycle are compacted, accelerating subsequent scanning operations. Efficient refresh scheduling is crucial, as higher update frequencies enhance data compaction, reducing the computational burden for real-time data retrieval. Ideally, in the absence of modifications after a refresh cycle, the batch-processing task operates solely on the precomputed data cube.

To optimize the refresh procedure, updates applied to the key-value repository are asynchronously streamed to the real-time processing module, where an intermediate data cube is maintained. This intermediate representation is periodically committed to the persistent key-value system, ensuring seamless synchronization. Empirical assessments indicate that this strategy significantly surpasses traditional data cube recomputation in efficiency, providing the necessary throughput for managing high-frequency update streams.

Upon completion of the refresh cycle, the latest data cube timestamp is updated in MetaStore, initiating the compaction process to streamline real-time data storage. Additionally, MetaStore retains global system parameters, including timestamps of analytical query submissions and the frequency of data cube materialization.

B. Storage Model

The key-value storage system necessitates mechanisms for handling multi-version concurrency to facilitate seamless coexistence between transactional and analytical operations. Additionally, the model incorporates enhancements such as optimized data traversal, compaction strategies, and dynamic load balancing, as elaborated below.

1) Comprehensive and Incremental Scans: To support analytical queries and maintain the data cube structure, the key-value store must implement efficient traversal mechanisms. Two distinct scanning techniques are designed, each suited to specific operational requirements:

FullScan(T_i): This operation extracts the latest available version of each key preceding timestamp T_i, enabling complete reconstruction of the data cube.

DeltaScan(T_1, T_2): Accepting two temporal markers, T_1 and T_2 ($T_1 < T_2$), this function retrieves two snapshots of modified keys—one corresponding to T_2 and another to T_1. If a key was introduced post T_1, only one version is returned. This technique plays a pivotal role in real-time analytical query execution.

2) Hierarchical and Localized Data Compaction: As multiple historical versions of each key accumulate, scan operations may incur excessive I/O overhead. To mitigate this, an automated compaction strategy is adopted.

Hierarchical Compaction: Triggered following each data cube update, this process consolidates all previous versions preceding the latest refresh point into a singular, optimized version, represented as V_{DC}, ensuring alignment with the updated data cube.

Node-Level Compaction: Executed at an individual storage node, this procedure acquires the active scan timestamp T_{scan} and merges versions older than T_{scan}. The integrity of V_{DC} is preserved to maintain consistency across subsequent data cube updates.

3) Adaptive Load Balancing: In practical deployments, certain key segments experience a higher frequency of updates, resulting in an uneven workload distribution across storage nodes. Additionally, since modifications append new instances rather than altering existing records, skewed data allocation arises over time, affecting retrieval efficiency. To address this imbalance, frequently modified key ranges undergo partitioning and redistribution across nodes.

C. Data Cube Management

To enhance analytical query efficiency, a structured data cube is persistently maintained within the key-value repository. Based on the specific application requirements, different data cube representations—including full cubes, Iceberg cubes, or closed cubes—may be employed. The approach presented here focuses on the full cube structure, arranged as a hierarchical framework of cuboids. Two primary update methodologies are considered:

Rebuild Approach: This strategy utilizes the FullScan operation to extract the latest key states preceding a specified timestamp T_i. The batch-processing system subsequently regenerates the data cube by computing aggregate values across its structured cuboids.

Incremental Refresh: This technique consists of two sequential phases: propagation and application. During propagation, Δ_{DC} (the change in the data cube) is computed from Δ_T (the modifications in the source data). The update phase

then integrates Δ_{DC} into the existing data cube. Incremental maintenance is feasible exclusively for self-maintainable aggregation functions such as SUM and COUNT.

R-Store employs full reconstruction for initial data cube creation, while incremental refreshes are applied for continuous real-time maintenance. The streaming component synchronizes with the key-value repository, periodically committing updates to storage. Due to the two-stage nature of data cube updates, the MapReduce framework is inherently suited for facilitating stream-based operations within R-Store.

D. Real-Time Analytical Cube Management

The introduced system employs a streaming-driven framework for adaptive data cube maintenance, where a decentralized stream-processing model is leveraged. Each computational unit within the architecture is designated to manage modifications corresponding to a specified subset of identifiers.

The transformation function responsible for adjusting the analytical representation is detailed in Algorithm 1. When an update request is received, the corresponding prior entry is fetched from localized storage if present. To enhance retrieval performance, a structured index is utilized to organize identifier-value associations, while frequently modified identifiers are retained in memory. Empirical evaluations reveal that modifications generally affect a limited portion of identifiers, increasing the probability of rapid cache-based retrieval. If an identifier is missing from the local repository, a new instance is generated for each hierarchical aggregation and assigned to the corresponding processing module. The output key in the transformation function consists of dimensional attributes, while the corresponding value encapsulates the numerical component. If an identifier exists within storage and the update aligns with the same hierarchical level, a single key-value pair is generated, capturing the numerical variation. Otherwise, two key-value pairs are emitted: one reflecting the updated value tagged with a + marker and another representing the previous value labeled with a − marker.

A reduction mechanism is periodically triggered based on predefined time intervals. For example, if configured to execute every second, it accumulates temporary computations during the specified period before applying the reduction function. Another parameter dictates the frequency of cube re-materialization. The reduction function responsible for iteratively refining the analytical model is illustrated in Algorithm 2. A reducer synchronizes the local analytical cube with intermediate values obtained from the transformation phase. If these values correspond to modifications occurring before the subsequent materialization timestamp (T_{cube}), they are integrated into the primary structure. Otherwise, they are temporarily stored in an auxiliary repository. Once all mapper-generated updates are aligned beyond T_{cube}, the re-materialization operation is initiated, persisting the data into long-term storage. Throughout this process, recent updates remain buffered until finalization. Upon completion, the timestamp T_{cube} is revised, and the auxiliary repository is merged into the primary analytical cube.

Algorithm 1 Transformation Function for Incremental Modification

KeyValue kv $previous_kv \leftarrow$ fetchFromStorage($kv.key$) $previous_kv == \emptyset$ each hierarchy in analytical cube $HierK \leftarrow$ deriveHierarchyKey(hierarchy, $kv.value$) $HierV \leftarrow$ deriveHierarchyValue($kv.value$) $HierV.setTag(`+')$ Emit($HierK$, $HierV$) storeEntry($kv$) $priorHierV \leftarrow$ deriveHierarchyValue($previous_kv.value$) $priorHierV.setTag(`-')$ $updatedHierV \leftarrow$ deriveHierarchyValue($kv.value$) $updatedHierV.setTag(`+')$ each hierarchy in analytical cube $priorHierK \leftarrow$ deriveHierarchyKey(hierarchy, $previous_kv.value$) $updatedHierK \leftarrow$ deriveHierarchyKey(hierarchy, $kv.value$) $priorHierK == updatedHierK$ $updatedHierV.set(computeDelta(priorHierV, updatedHierV))$ Emit($updatedHierK$, $updatedHierV$) Emit($priorHierK$, $priorHierV$) Emit($updatedHierK$, $updatedHierV$) updateStorage(kv)

Algorithm 2 Reducing Function for Continuous Adjustment

Key key, List<Value>$vlist$, Context $context$ $i \leftarrow 0$, $sum \leftarrow 0$ each value v in $vlist$ $v.timestamp < T_{cube}$ IntegrateWith(key, v, $Cube$) IntegrateWith(key, v, $AuxCube$)

In a streaming framework, ensuring fault resilience necessitates periodic state preservation. Past streaming records are retained in log archives for recovery purposes. Within the proposed system, materialization of the analytical cube inherently serves as a checkpointing mechanism. As key-value pairs post-materialization remain in storage—albeit intermediate versions may be discarded via local optimization—the essential data required for subsequent reconstruction persists. Consequently, real-time analytical cube upkeep can be reinstated seamlessly using the stored cube and real-time repository, eliminating the necessity for additional checkpointing procedures.

E. Scalability and Reliability Mechanisms

To support high-throughput, real-time analytical workloads, R-Store incorporates multiple mechanisms that address both scalability and reliability across distributed environments.

Scalability: R-Store achieves horizontal scalability through dynamic partitioning of the key-value store, enabling data to be distributed across multiple HBase-R region servers. The system supports parallel execution of MapReduce jobs by dividing input datasets into smaller partitions, processed independently by worker nodes. Moreover, the real-time stream processor (HStreaming) uses partitioned update handlers to ensure that streaming data is processed concurrently and efficiently. Scalability tests conducted on a 144-node cluster validate the system's ability to handle billions of records with consistent performance.

Load Balancing: To prevent performance degradation due to uneven data distribution, R-Store integrates an adaptive load-balancing strategy that redistributes frequently updated key ranges. This ensures uniform resource utilization and avoids hotspots, particularly in Zipfian-distributed update scenarios.

Reliability: R-Store ensures data reliability and fault tolerance through multi-version concurrency control and periodic checkpointing. By retaining historical key-value versions, the system can reconstruct analytical cubes even in the event of node failure or incomplete refresh cycles. The use of HBase-R's write-ahead logs and HDFS replication policies ensures data durability. Furthermore, the integration of intermediate materialized cubes within the stream processor acts as a built-in checkpoint, enabling consistent recovery without loss of real-time updates.

These mechanisms collectively ensure that R-Store remains robust, scalable, and reliable in handling hybrid OLTP-OLAP workloads under varying update and query conditions.

IV. Revised Data Flow of R-Store

Figure 1 represents the interaction between HBase-R, HStreaming, and the MapReduce framework in the R-Store architecture. Each HBase-R region server administers multiple regions, where some regions correspond to the real-time repository, while others store precomputed data structures. Transactional queries are routed to a specific region server and temporarily held in the memory store corresponding to the region. Once the memory store reaches its allocated threshold, the stored data is committed to HDFS as a persistent file.

As soon as modifications are incorporated into HBase-R, they are transmitted to a designated mapper in HStreaming based on the associated key. The mappers within HStreaming evaluate modifications per cuboid and forward the computed changes to reducers. The reducers update and retain the real-time data cube on local disk storage. At predefined intervals, HStreaming commits its in-memory data cube to HBase-R and updates the MetaStore with the latest timestamp. Subsequently, a compaction routine consolidates older versions preceding the refresh cycle.

Upon receiving an analytical query, a timestamp and metadata from MetaStore are retrieved alongside the stored statistics in HBase-R. The query is then reformulated as a MapReduce operation, executed across designated regions. Each mapper performs a scanning operation on its respective input partition, which can belong to either the real-time data repository or the data cube. The final query results are stored in HBase-R at the conclusion of processing.

V. Revised Real-Time OLAP Processing

The structural framework and execution methodology of R-Store have been elaborated in prior sections. This segment addresses the mechanism for processing real-time analytical queries. In instances where a MapReduce execution relies solely on precomputed data structures, scanning performance

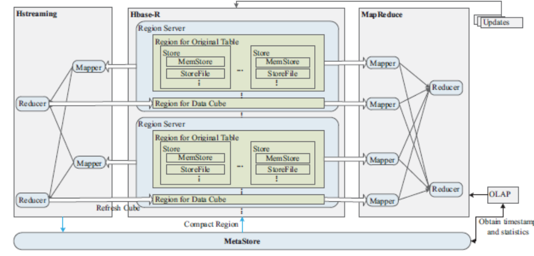


Fig. 1: Illustration of Data Flow in R-Store

is optimized. However, such an approach may yield outdated results. To enhance query freshness, all modifications introduced prior to query initiation are incorporated into the computation. Thus, both the precomputed data structure and the real-time repository must be scanned.

Consider a scenario where the data cube was last updated at T_{DC}, and a query is initiated at T_Q. For any altered key following T_{DC}, the incremental scanning mechanism retrieves both the prior version before T_{DC} and the latest version before T_Q. By integrating these values with numerical attributes in each cuboid, the latest aggregated values are dynamically derived, ensuring query freshness. The following algorithm formalizes this process.

A. Modified Query Execution on Incrementally Updated Cubes

The implementation leverages a MultiTableInputFormat, allowing each MapReduce execution to scan multiple tables with designated configurations for full or incremental scanning. Utilizing this format, queries access both the precomputed data cube and the real-time repository.

1) Map Function for Query Execution: Algorithm 3 outlines the mapping procedure. The mapper selectively processes records based on filtering criteria and partitions them appropriately before shuffling them to reducers. The numerical values corresponding to cuboids are tagged for classification, indicating whether they originate from precomputed data, reflect new modifications, or signify obsolete records.

Algorithm 3 Map Function for Incremental Query Execution

List of Key-Value pairs $kvlist$, Execution Context $context$ Initialize $key \leftarrow null$, $value \leftarrow null$ $kvlist.size == 1$ $key \leftarrow$ ExtractKey($kvlist[0].key$) FilterCondition(key) == False $value \leftarrow kvlist[0].value$ AssignTag($value$, “Q”) Emit(key, $value$) $key \leftarrow$ ExtractKey($kvlist[0].value$) FilterCondition(key) == False $value \leftarrow$ ExtractValue($kvlist[0].value$) AssignTag($value$, “+”) Emit(key, $value$) $value \leftarrow$ ExtractValue($kvlist[1].value$) AssignTag($value$, “-”) Emit(key, $value$)

2) Data Cube Query Operations: To streamline the querying process, specific operations for data cubes are abstracted into predefined functions, as summarized in Table I. The supported data cube query operations are summarized.

Operator	Parameters
addFilter	Attribute name, Condition, Value
addGroupBy	Grouping Attribute
setAggregationFunc	Aggregation Function
setNumericAttribute	Numerical Attribute

TABLE I: Data Cube Query Operations

3) Example Query Execution: Algorithm 4 illustrates an example query that calculates the summation of retail prices for a given brand and manufacturer while grouping results by product type.

Algorithm 4 Example Query Execution on Data Cube

Define Cube Instance cub cub.addFilter("manufacturer", "=", "M1") cub.addFilter("brand", "=", "Brand13") cub.addGroupBy("type") cub.setNumericAttribute("retailPrice") cub.setAggregationFunc("sum") cub.setOutputTable("resultTable") SubmitQuery(cub)

This refined approach automates the merging of real-time modifications with historical data, encapsulating these functionalities within predefined operators. As a result, users are relieved from explicitly managing real-time and historic data consolidation.

B. Correctness of Query Responses

Upon submission of an OLAP request, a timestamp T_Q is allocated through the MetaStore. To ensure accuracy, when the request necessitates multiple scans of a dataset, each scanning operation on a processing unit retrieves records preceding T_Q. However, due to minor variations in clock synchronization across nodes in a distributed framework, an inconsistency may arise if a node's timestamp T_k is earlier than T_Q, causing data retrieval between T_k and T_Q. To mitigate this, if T_Q exceeds T_k, the retrieval operation is delayed momentarily until T_Q is no longer greater than T_k. Given that synchronization in local network environments can attain millisecond precision under optimal conditions, the resulting delay is negligible in comparison to processing durations.

C. Computational Cost Model

The alternative querying mechanism discussed does not consistently outperform conventional methods. The approach necessitates scanning both real-time tables and precomputed aggregations, resulting in increased resource consumption. Additionally, two versions per modified key must be shuffled during distributed processing. When transactional activity is minimal or limited to a small key subset, the incremental scanning approach is preferable as fewer data points are transferred. Conversely, a comprehensive scan can be more efficient under uniform update distributions. To optimize efficiency, a cost model is formulated. The parameters used in the cost model are defined.

TABLE II: Model Parameters

Parameter	Definition
$\|T\|$	Total records in table T
$d(T)$	Count of unique keys modified
$f(T)$	Size of each tuple in table T
$s(T)$	Proportion of modified keys since last update
$\|C\|$	Number of entries in selected cube
$d(C)$	Dimension attributes in the cube
$n(C)$	Numeric attributes in the cube
$\|Q\|$	Resulting tuples in query outcome
$s(Q)$	Selectivity ratio of filtering condition
$d(Q)$	Key size of the query output
$n(Q)$	Value size of the query output
sh_{HBase}	Overhead of shuffling from distributed store
w_{HBase}	Write overhead to distributed store
sh_{MR}	Shuffle overhead in distributed computation
c_L	Local input/output overhead
m_T	Processing units for table T
m_C	Processing units for cube aggregation
B	Data block size

D. Computational Complexity of Conventional Processing

The baseline mechanism employs a distributed computation framework akin to full recomputation of the data aggregation structure. The scanning phase involves reading stored data from each distributed node and transmitting it to processing units. The full retrieval operation transfers all records, while an adaptive mechanism retrieves fewer records when $d(T)$ is minimal and memory buffers retain sufficient keys. The retrieval cost is estimated as: "The scanning cost for full data retrieval is given in Equation 1."

$$C_{scan} = sh_{HBase} \times |T| \times f(T) \quad (1)$$

Following retrieval, key-value pairs are emitted by processing units, leading to an output size of:

$$S_{MO} = s(Q) \times \left(\frac{|T|}{m_T}\right) \times (d(Q) + n(Q)) \quad (2)$$

Sorting overhead is given by: The size of the emitted key-value pairs by mappers is computed using Equation 3.

$$C_{sort\text{-}map} = m_T \times 2c_L \times \left(S_{MO} \times \log_B \left(\frac{S_{MO}}{B+1}\right)\right) \quad (3)$$

Upon completion, sorted records are shuffled, incurring: "Equation 4 expresses the sorting overhead incurred at the map phase."

$$C_{shuffle} = sh_{MR} \times s(Q) \times |T| \times (d(Q) + n(Q)) \quad (4)$$

Merging and writing costs are: "The cost associated with data shuffling across nodes is given in Equation 5."

$$C_{merge} = 2c_L \times s(Q) \times |T| \times (d(Q) + n(Q)) \quad (5)$$

"The merge overhead is modeled as shown in Equation 6."

$$C_{write} = w_{HBase} \times |Q| \times (d(Q) + n(Q)) \quad (6)$$

VI. Evaluation

This section assesses the performance of R-Store within a proprietary cluster containing 144 computational units. Each unit is equipped with an Intel X3430 2.4 GHz processor, 8 GB RAM, and dual 500 GB SATA drives, interconnected via a gigabit Ethernet and operating on CentOS 5.5. These nodes are evenly distributed across three racks. The benchmarking process employs TPC-H datasets; however, as TPC-H updates primarily introduce new keys instead of modifying existing ones, custom scripts are developed to facilitate real-time modifications. These scripts generate updates based on either a uniform or Zipfian distribution. In subsequent trials, the TPC-H part table serves as the foundational dataset for constructing the data cube.

A. Efficiency of Data Cube Maintenance

The processing capacity of the proposed real-time data-cube maintenance strategy was first assessed by measuring the maximum update throughput. As illustrated, a 10-node HStreaming cluster sustains more than 100000 updatess^{-1}, outperforming an HBase-R deployment running on 40 nodes.

Experimental design.: To contrast *full recomputation* with *incremental refresh*, the system was provisioned with 100 machines: 40 for MapReduce, 40 for HBase-R, and 20 for HStreaming. The TPC-H benchmark at scale factor 8000 generated 1.6e9 keys in the `part` table, leaving each HBase-R node with approximately 4.8GB of data once loading completed; the data cube was built thereafter.

Update workload.: A Zipf distribution was applied so that 1% of all keys were updated, varying the absolute number of modifications between 8e6 and 1.6e9. Because HBase-R keeps previous versions, each node accumulated 0.024GB–4.8GB of appended records.

Recomputation.: The recomputation approach involves two primary phases: a scanning stage and a MapReduce execution stage. As the volume of updates increases, the scanning phase requires more time due to the larger data footprint in HBase-R. Interestingly, the MapReduce execution time tends to decrease with a higher number of updates. This counterintuitive behavior is attributed to pipelining between the scanning and execution stages, where prolonged scanning leads to greater overlap between phases, ultimately reducing the total processing duration.

Incremental refresh.: The incremental strategy involves a single write-back phase (red) in which the updated cube fragment is stored in HBase-R. Because the real-time mechanism amortises updates continuously, latency remains almost flat irrespective of the number of modified keys.

Scalability.: To evaluate scale-out behaviour, both data volume and node count were increased proportionally while maintaining the 1% update ratio. Neither recomputation nor incremental refresh exhibited noticeable growth in execution time, confirming that R-Store scales linearly with cluster size.

VII. Efficiency of Real-Time Querying

Figure 2 shows the variation in execution time of data cube slice queries across different update ratios. Further trials explore the efficiency of real-time query execution. The Incremental Querying algorithm, which leverages data cubes for query optimization, is benchmarked against a baseline method that employs full-table scanning. The cluster setup mirrors that of a previous configuration, with the update count fixed at 8 billion and varying proportions of modified keys.

The processing duration for a representative data cube slice query:

```
SELECT sum(prices) FROM part
WHERE mft = 'Manufacturer#1'
GROUP BY brand, type, size, container;
```

The execution time of the baseline method consists of scanning (black section) and subsequent MapReduce execution (yellow section). The Incremental Querying method comprises three phases: scanning the data cube (red segment), scanning the modified records in HBase-R (blue segment), and executing MapReduce (grey segment). When a limited subset of keys is modified, Incremental querying significantly outperforms the baseline due to optimized adaptive scanning and reduced data shuffling. However, as the update proportion increases, the volume of data transferred from HBase-R to MapReduce rises, leading to longer execution durations.

An additional benchmark using TPC-H Q1 ensures that approximately 15% of tuples are processed. Since the data cube is constructed on shipdate, returnflag, and linestatus attributes, its size remains significantly smaller than the real-time table, yielding a scanning duration of roughly 20 seconds. The results demonstrate the superior performance of IncreQuerying over the baseline approach.

To determine the optimal querying method, the cost model outlined in Section V-C estimates the I/O operations. The estimated I/Os for IncreQuerying increase linearly, closely mirroring its execution time.

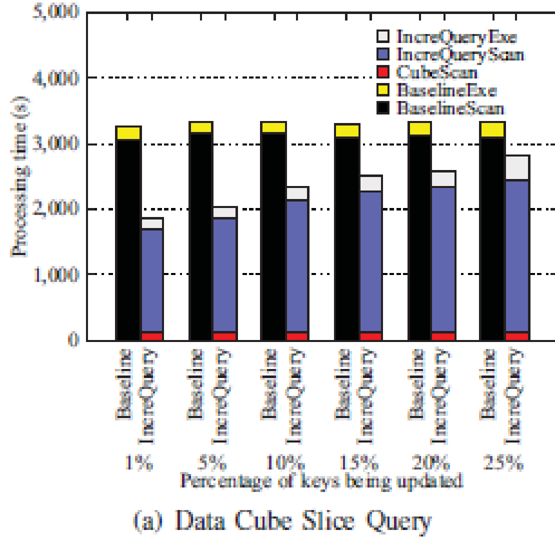
Unlike standalone data cube queries, RTOLAP queries necessitate additional steps, incurring supplementary processing costs: retrieving real-time records from HBase-R and dynamically merging them with the data cube during MapReduce execution.

The experimental setup assesses the efficiency of querying within a structured dataset under different update conditions. The evaluation includes comparisons between the foundational model and the enhanced incremental query technique. Processing duration is measured against the proportion of modified identifiers.

A. TPC-H Q1 Analysis

As shown in Figure 3, the query performance for TPC-H Q1 varies with increasing update frequency.

The structured benchmarking framework, TPC-H Q1, is utilized to further analyze query response efficiency. Results highlight how processing time fluctuates as the proportion of altered keys rises.



(a) Data Cube Slice Query

Fig. 2: Execution time of data cube slice inquiries under varied update ratios.

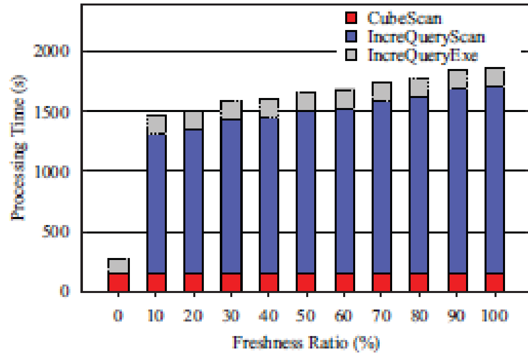


Fig. 3: Performance comparison for TPC-H Q1 queries based on key update frequency.

B. Cost Model Accuracy Assessment

Figure 4 compares the actual execution cost with the model's predicted I/O operations.

An analytical cost prediction model is employed to estimate I/O operations for incremental query execution and baseline performance. A comparative visualization illustrates the deviation of actual executions from estimated values.

C. Data Freshness and Query Efficiency Trade-off

The storage structure organizes data entries sequentially based on timestamps. Instead of traversing all records, only segments between a predefined data consolidation point and a specified threshold are examined. The degree of real-time data retrieval directly influences query efficiency.

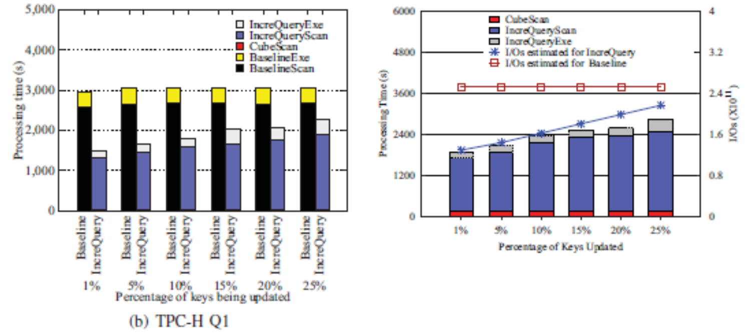


Fig. 4: Evaluation of cost model prediction accuracy in terms of input/output operations.

D. Effectiveness of Data Compaction

Compaction strategies are employed to minimize redundant storage and enhance query responsiveness. Experimental findings indicate that the absence of compaction leads to prolonged retrieval times due to excessive stored information.

E. OLTP Query Processing During OLAP Execution

Performance evaluation considers the concurrent execution of transactional and analytical queries. The impact of concurrent workloads is examined by measuring throughput under varying node configurations.

Response latency for transactional operations is also analyzed, revealing that concurrent analytical processing results in slightly elevated latencies due to shared computational resources. Each performance result includes the standard error across five experimental trials. Error bars reflect the 95% confidence interval. CPU and I/O utilization were monitored using dstat; peak CPU usage remained under 78% with 60% I/O throughput saturation.

VIII. Conclusion

This paper presented R-Store, a distributed adaptive analytics system that integrates MVCC, DeltaScan, and OLAP cubes for real-time data processing. Our system achieved significant throughput improvements and timestamp-consistent updates across distributed partitions. However, limitations exist in terms of fault tolerance under cross-partition clock drift and lack of full lineage tracking. Future work will explore consensus-based time synchronization and integration with federated OLAP systems.

References

[1] P. Carbone et al., “Apache Flink: Stream and batch processing in a single engine,” *IEEE Data Engineering Bulletin*, vol. 36, no. 4, 2017.

[2] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[3] V. Gokhale et al., “Rockset: Real-Time Indexing for Analytical Apps,” in *Proc. CIDR*, 2023.

[4] A. Zaitsev, “ClickHouse: Fast Open-Source OLAP DBMS,” in *Proc. VLDB*, 2023.

[5] M. Armbrust et al., “Delta Lake: High-Performance ACID Table Storage,” in *Proc. VLDB*, 2021.

[6] R. Chen et al., "Apache Iceberg: Open Table Format for Huge Analytics Datasets," in *Proc. SIGMOD*, 2022.

[7] M. Armbrust et al., "Delta Lake: High-Performance ACID Table Storage," in VLDB, 2021.

[8] R. Chen et al., "Apache Iceberg: Open Table Format for Huge Analytics Datasets," in SIGMOD, 2022.

[9] V. Gokhale et al., "Rockset: Real-Time Indexing for Analytical Apps," in CIDR, 2023.

[10] A. Zaitsev, "ClickHouse: Fast Open-Source OLAP DBMS," in Proc. of VLDB, 2023.

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

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[13] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[14] B. Chandramouli, J. Goldstein, and D. Maier, "High-performance dynamic pattern matching over disordered streams," *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, pp. 220–231, 2010.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2017.