OPEN ACCESS

DRA ANNUAL INTERNATIONAL CONFERENCE 2024 ON "SOCIO-ECONOMIC TRANSFORMATION: OPPORTUNITIES AND CHALLENGES"

Int. Jr. of Contemp. Res. in Multi. *PEER-REVIEWED JOURNAL* *Volume 4 [Special Issue 1] Year 2025*

Conference Paper

High-Performance Real-Time Data Store

Bhuvan Chandra Sarakam
Student, Doctor of Business Administration, Belhaven University, Jackson, Mississippi, USA

Corresponding Author: * Bhuvan Chandra Sarakam

DOI: https://doi.org/10.5281/zenodo.20303876

Abstract

This paper presents Druid, an open-source, distributed data store designed specifically for real-time analytical processing of massive datasets. Druid combines the efficiency of a column-oriented storage model with the flexibility and scalability of a shared-nothing architecture, enabling high-performance ingestion, exploration, and aggregation of time-series data. The system is optimized for sub-second query latency, even when dealing with large-scale data, by utilizing advanced indexing techniques such as inverted indices, bloom filters, and bitmap indexes. The multi-tiered node architecture of Druid plays a crucial role in enabling low-latency query processing and ensuring high availability across distributed environments. Real-time data ingestion is coupled with the ability to execute complex analytical queries over both real-time and historical data, making Druid particularly well-suited for applications requiring quick insights into time-sensitive data. Druid's architecture allows for horizontal scalability, where clusters can expand or contract based on workload demands, and its fault-tolerant design ensures high availability even in the event of node failures. By supporting advanced aggregation and filtering techniques, Druid caters to interactive data exploration and visualisation applications, often found in industries such as finance, telecommunications, and e-commerce, where rapid data processing and decision-making are essential. Druid offers a powerful, scalable solution for modern analytical workloads that require real-time data processing and low-latency query execution. Its combination of distributed architecture, time-series optimisation, and high performance makes it a robust platform for large-scale, data-driven applications across a wide range of industries.

Manuscript Information

- **ISSN No:** 2583-7397
- **Received:** 12-12-2024
- **Accepted:** 25-02-2025
- **Published:** 05-03-2025
- **IJCRM:**4 (SP1); 2025: 294-304
- **©2025, All Rights Reserved**
- **Plagiarism Checked:** Yes
- **Peer Review Process:** Yes

How to Cite this Article

Sarakam B C. High-Performance Real-Time Data Store. Int J Contemp Res Multidiscip. 2025;4 (6):294-304.

Access this Article Online

www.multiarticlesjournal.com

KEYWORDS: Sustainable development, online freelancing, economic aspect, social inclusion, environmental benefits, gig economy.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

INTRODUCTION

The rise of internet technology has led to a surge in machine-generated events, which are often low-value individually but hold potential in aggregate. While infrastructure like IBM's Netezza [1], HP's Vertica [2], and EMC's Greenplum [3] exists to process such data, these solutions are costly and cater primarily to large organizations.

Google's MapReduce [4] and the Hadoop project [5] have been pivotal in enabling organizations to process large-scale log data efficiently. Despite its strengths in storage and batch processing, Hadoop faces limitations in query performance, concurrent load handling, and real-time data ingestion. To address these challenges, we developed Druid: an opensource, distributed, column-oriented, real-time analytical data store. Druid draws inspiration from OLAP systems [6], interactive query systems [7], and distributed data stores [8, 9]. It is designed for environments requiring high query performance and availability under heavy concurrent usage.

This paper details Druid's architecture and design, highlighting its suitability for always-on production systems. The structure is as follows: Section II defines the problem; Section III describes the system architecture; Section IV discusses data storage; Section V outlines the query API; Section VI presents benchmarks; and Sections VII, VIII, and IX discuss production insights, related work, and conclusions, respectively.

PROBLEM DEFINITION

Druid was designed to address challenges in ingesting and analysing large volumes of transactional, append-heavy timeseries data, such as the log data in Table I. This table, based on Wikipedia edits, includes timestamps, dimension columns (e.g., page, user, location), and metric columns (e.g., characters added or removed).

The system needed to compute aggregates and drilldowns efficiently, answering queries like "How many edits were made by males in San Francisco on Justin Bieber's page?" with sub second latency. Existing open-source RDBMS and NoSQL systems lacked the capabilities for low-latency data ingestion and querying required for interactive applications [10].

Druid was built to support interactive dashboards with fast query responses, high availability, and multi-tenancy. The platform also addressed real-time analytics needs, enabling users to react quickly to critical events. Popular data warehousing systems like Hadoop failed to meet the required sub-second ingestion latency.

Since its open-sourcing in 2012, Druid has been adopted across industries for use cases like video analytics, network monitoring, operations monitoring, and online advertising.

Table 1: Sample Druid data for Wikipedia edits.

Timestamp	Page	Username	City	Characters Added	Characters Removed
2011-01-01T01:00:00Z	Justin Bieber	Boxer	San Francisco	1800	25
2011-01-01T01:00:00Z	Justin Bieber	Reach	Waterloo	2912	42
2011-01-01T02:00:00Z	Ke$ha	Helz	Calgary	1953	17
2011-01-01T02:00:00Z	Ke$ha	Xeno	Taiyuan	3194	170

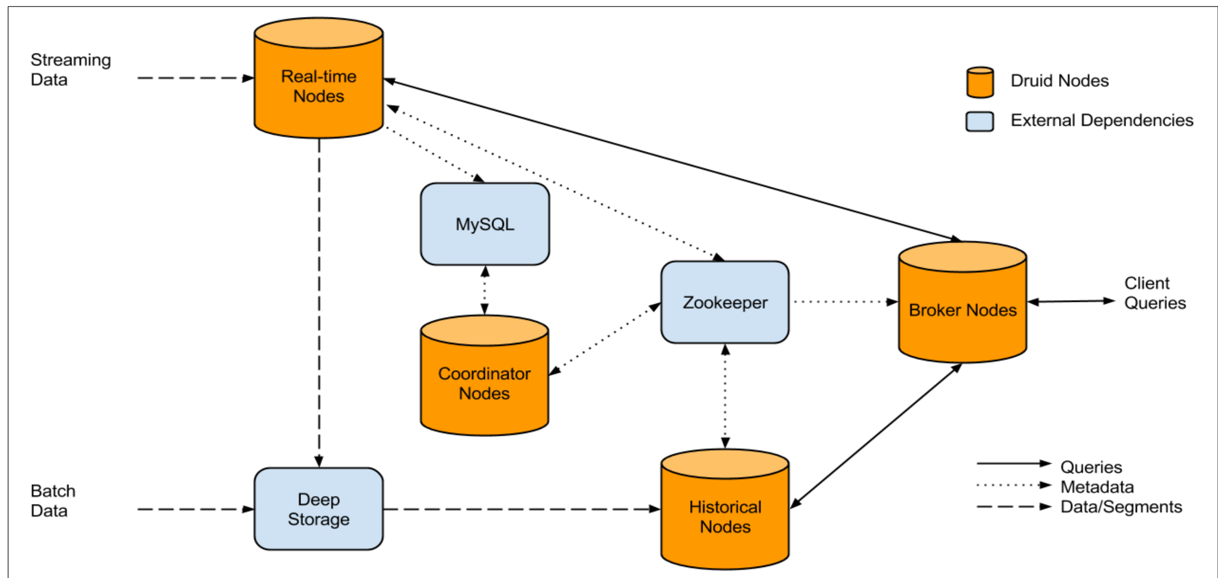

Fig 1: Overview of Druid cluster architecture and data flow.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

Architecture

A Druid cluster comprises various node types, each serving distinct roles, enabling separation of concerns and minimizing intra-cluster communication failures. This design ensures data availability while facilitating collaboration among nodes for complex data analysis. The cluster's architecture and data flow are depicted in Figure 1.

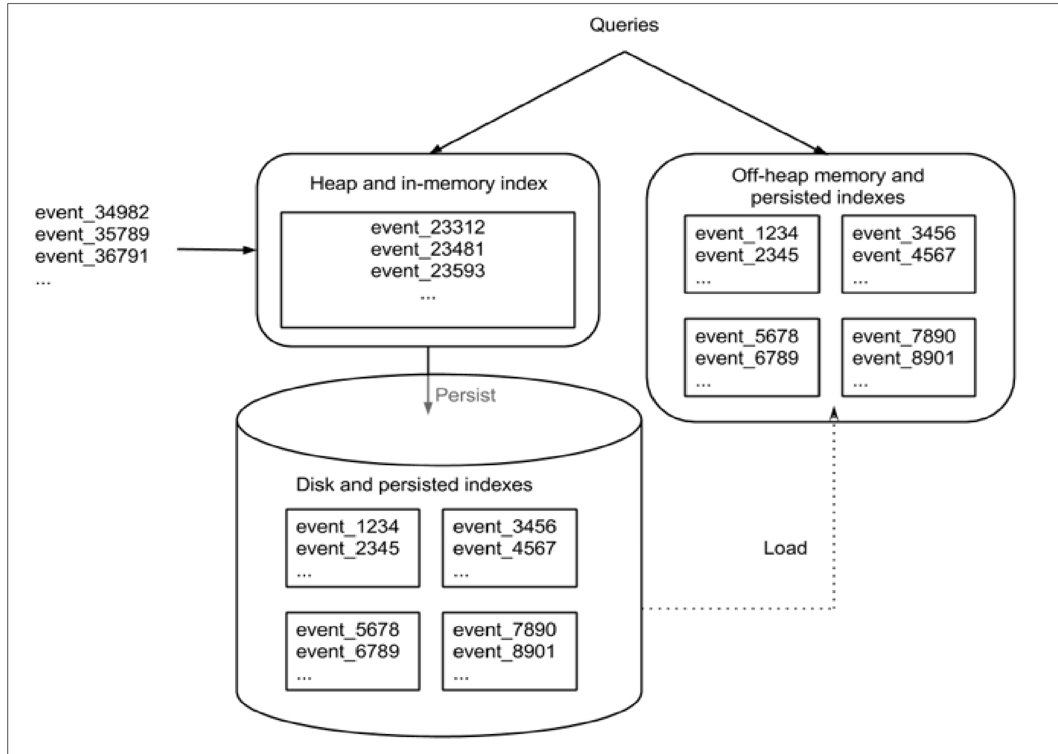

Fig 2: Real-time nodes buffer, persist, merge, and hand off data.

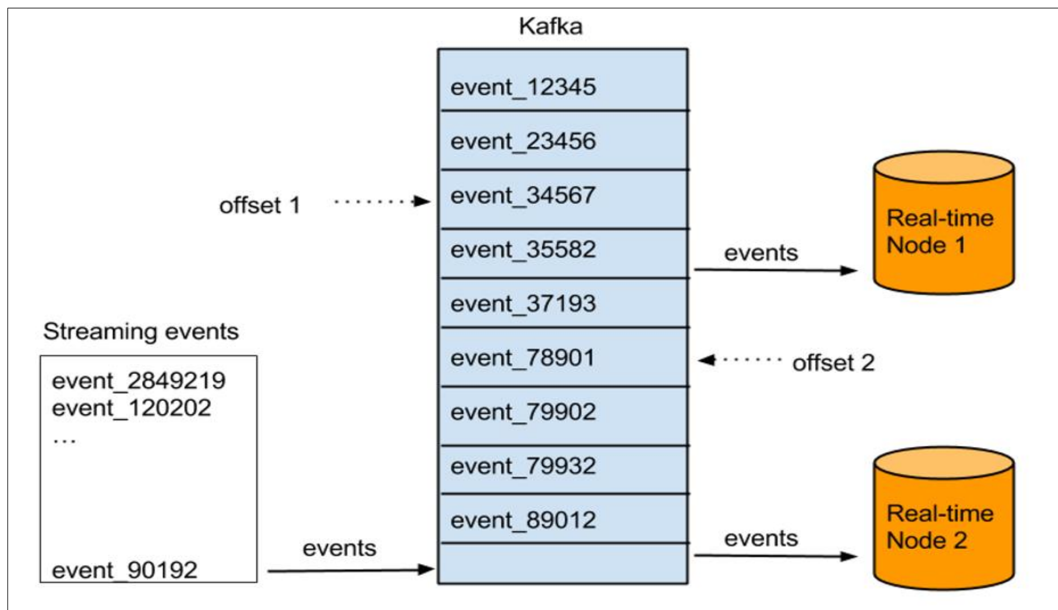

Fig 3: Real-time nodes ingest data via a message bus.\

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

A. Real-time Nodes

Real-time nodes ingest and query event streams, immediately making data available for querying. Events are buffered in-memory and periodically persisted to disk in an immutable columnar format (Figure 2). Persisted data is periodically merged into immutable "segments" and handed off to deep storage (e.g., S3 or HDFS).

Data ingestion leverages a message bus like Kafka to ensure durability and scalability (Figure 3). This model enables high ingestion rates and fault-tolerance through replication.

B. Historical Nodes

Historical nodes load and serve immutable segments created by real-time nodes. They operate in a shared-nothing architecture, ensuring scalability and independence. Nodes download segments from deep storage and announce their state in Zookeeper (Figure 4).

Historical nodes support tiering for prioritizing data access, allowing segregation into" hot" and "cold" clusters based on data importance.

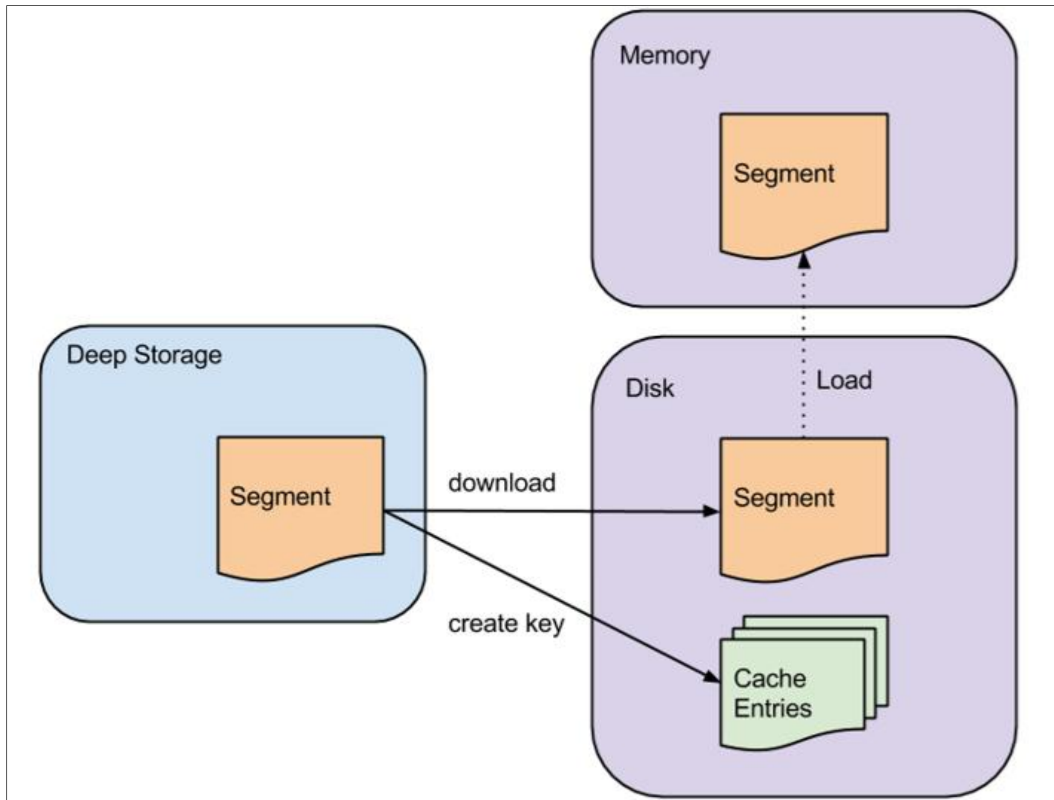

Fig 4: Historical nodes retrieve and process segments from deep storage.

C. Broker Nodes

Broker nodes route queries to real-time and historical nodes based on segment metadata stored in Zookeeper. They merge partial results and optionally cache query results using LRU based strategies (Figure 5). Cached data enhances durability and reduces query latency.

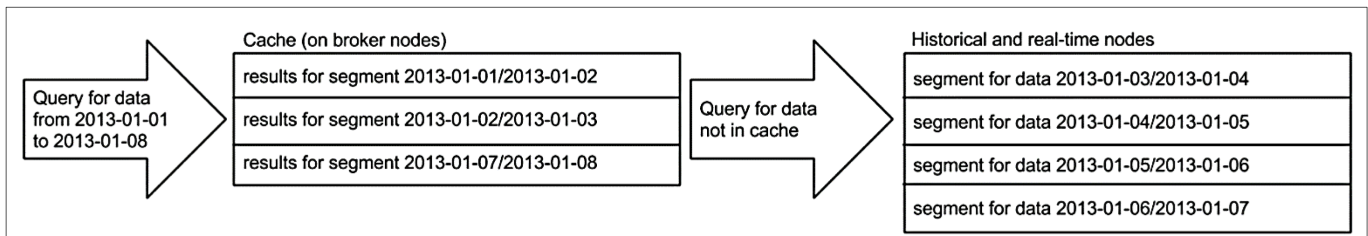

Fig 5: Broker nodes cache results per segment and combine them for queries.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

D. Coordinator Nodes

Coordinator nodes manage segment distribution and replication on historical nodes, utilizing rules defined in a MySQL database. These rules govern segment assignment, replication, and retention. Load balancing is achieved through cost-based optimization, considering query patterns and segment characteristics.

Coordinator nodes ensure cluster stability during failures. While Zookeeper or MySQL outages limit coordination, data availability remains unaffected.

1) Replication and Availability: Replication ensures fault tolerance, allowing seamless node upgrades without downtime. Coordinator nodes leverage Zookeeper and MySQL for metadata but maintain the cluster's status quo during outages. Druid's architecture emphasizes modularity, fault-tolerance, and scalability, leveraging specialized nodes for ingestion, storage, and query handling. This ensures robust performance for real-time analytics at scale.

Storage Format

Druid stores timestamped events in data tables called *data sources*, partitioned into *segments*, each spanning a time interval (e.g., an hour or a day) and containing 5–10 million rows. Segments are uniquely identified by a data source ID, time interval, and version string, enabling concurrency control and ensuring queries access the latest data.

Data is stored in a columnar format, optimized for aggregation and query efficiency [11]. Columns are compressed using encoding techniques such as dictionary encoding for strings and raw compression (e.g., LZF) [12] for numeric data. For example, a string column may map values to integer identifiers to reduce storage and enable efficient compression.

A. Indices for Filtering Data

To enable fast filtering, Druid builds inverted indices for dimension columns. For instance, a binary array indicates row positions for specific dimension values. Boolean operations (e.g., OR) on these arrays identify matching rows. Bitmap compression, such as the Concise algorithm [13], reduces index size significantly compared to integer arrays, with up to 50% compression.

Figure ?? shows that Concise compression reduces storage requirements compared to integer arrays. In an unsorted dataset, Concise compressed sets were 42% smaller, while sorting improved compression only slightly. As dimension cardinality approaches the total number of rows, integer arrays may become more space-efficient.

B. Storage Engine

Druid supports pluggable storage engines, including in memory or memory-mapped options, allowing configurations based on performance or cost needs. The default memory mapped engine relies on the OS for paging. Recent segments remain in memory, while less-used segments are paged out. However, paging constraints can degrade performance when memory limits are exceeded.

Query API

Druid accepts queries via HTTP POST requests, using a JSON object to specify parameters such as data source, time range, granularity, query type, and aggregations. Query results are returned as JSON objects containing aggregated metrics over the specified time range.

Filters in queries allow Boolean expressions of dimension value pairs to restrict scanned data. Nested filters enable complex drill-downs. A sample query for counting rows in the Wikipedia data source over a week, filtered by the page dimension, is shown below:

"queryType"	: "timeseries",
"dataSource"	: "wikipedia",
"intervals"	: "2013-01-01/2013-01-08",
"filter"	: {
"type"	: "selector",
"dimension"	: "page",
"value" },	: "Ke$ha"
"granularity"	: "day",

```
"aggregations" : [ {
        "type"    : "count",
        "name"  : "rows"
} ]
}
```

This query returns daily row counts matching the filter. The result is a JSON array, e.g.:

```
[ {
"timestamp": "2013-01-01T00:00:00.000Z",
"result": { "rows": 393298 }
},...
]
```

Druid supports various aggregations (e.g., sums, minimums, maximums) and complex operations like cardinality estimation and approximate quantiles. Aggregation results can be combined using mathematical expressions. The API is highly flexible, allowing filters and grouping on arbitrary conditions.

While Druid's storage format supports joins, the query language does not. This decision prioritizes sub-second query performance for user-facing workloads. Future work aims to implement joins and extend API functionality to support SQL.

EXPERIMENTAL RESULTS

To evaluate Druid's performance, I conducted experiments measuring its query and data ingestion capabilities.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

A. Query Performance

A test cluster was set up with 6TB of uncompressed data (tens of billions of rows) containing over a dozen dimensions with cardinalities ranging from tens to millions. Each row included metrics for counts, sums, and averages. The data was sharded on timestamps and dimensions, with shards of approximately 8 million rows each.

The cluster comprised 100 historical nodes (16 cores, 60GB RAM, 1TB disk, 10 GigE), totalling 1600 cores and 6TB RAM. Queries in Table II represent typical workloads, expressed in SQL for clarity. Key details:

- Queries spanned all data and used memory-mapped storage.

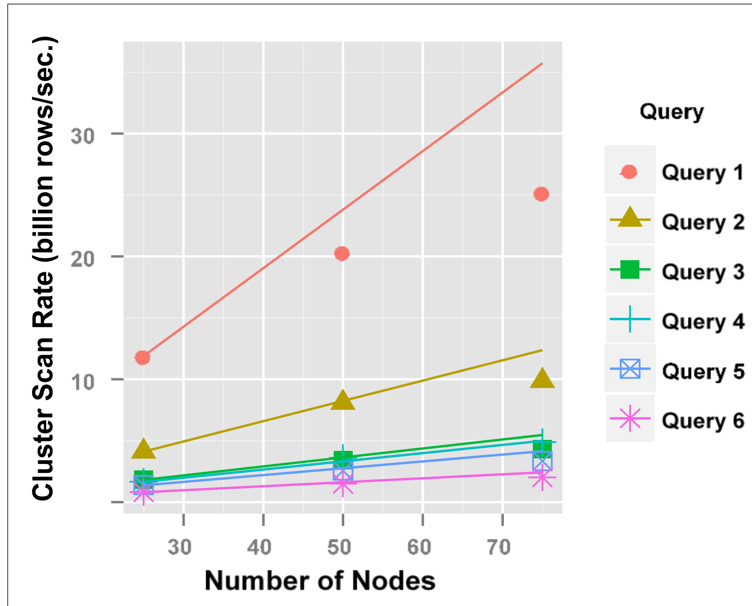

Fig 6: Druid cluster scan rate with lines indicating linear scaling from 25 nodes.

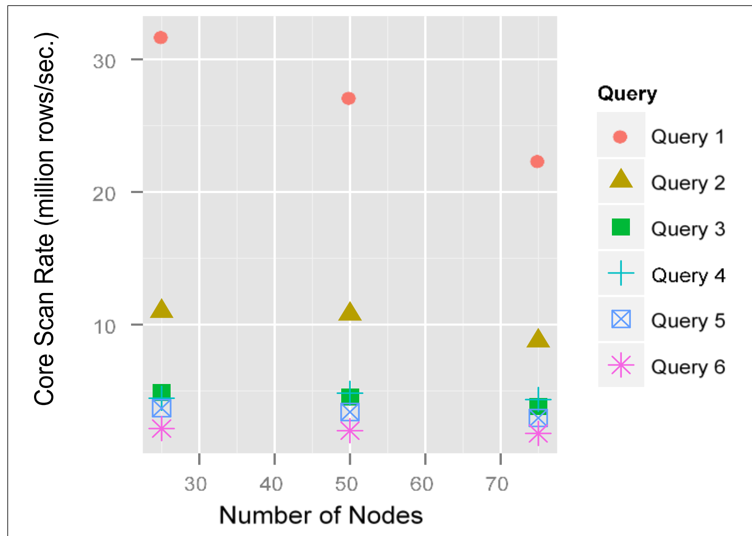

Fig 7: Druid core scan rate.

- Machines were configured with 15 threads for query processing.

Figures 6 and 7 show cluster and core scan rates, respectively. For Query 1, scan rates reached 33M rows/second/core. The cluster achieved 26 billion

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

rows/second for 75 nodes, though performance diminished beyond 50 nodes due to Amdahl's law [14]. Adding metrics in queries degraded performance due to Druid's columnar storage, which required additional data loads for each metric.

B. Data Ingestion Performance

Ingestion latency was tested on a real-time node (2.3GHz CPU, 2GB JVM heap). A basic setup (timestamp-only data) achieved 800k events/sec/node. Ingesting realistic data sets showed performance decreases with increasing dimension cardinality (Figure 8), number of dimensions (Figure 9), and number of metrics (Figure 10).

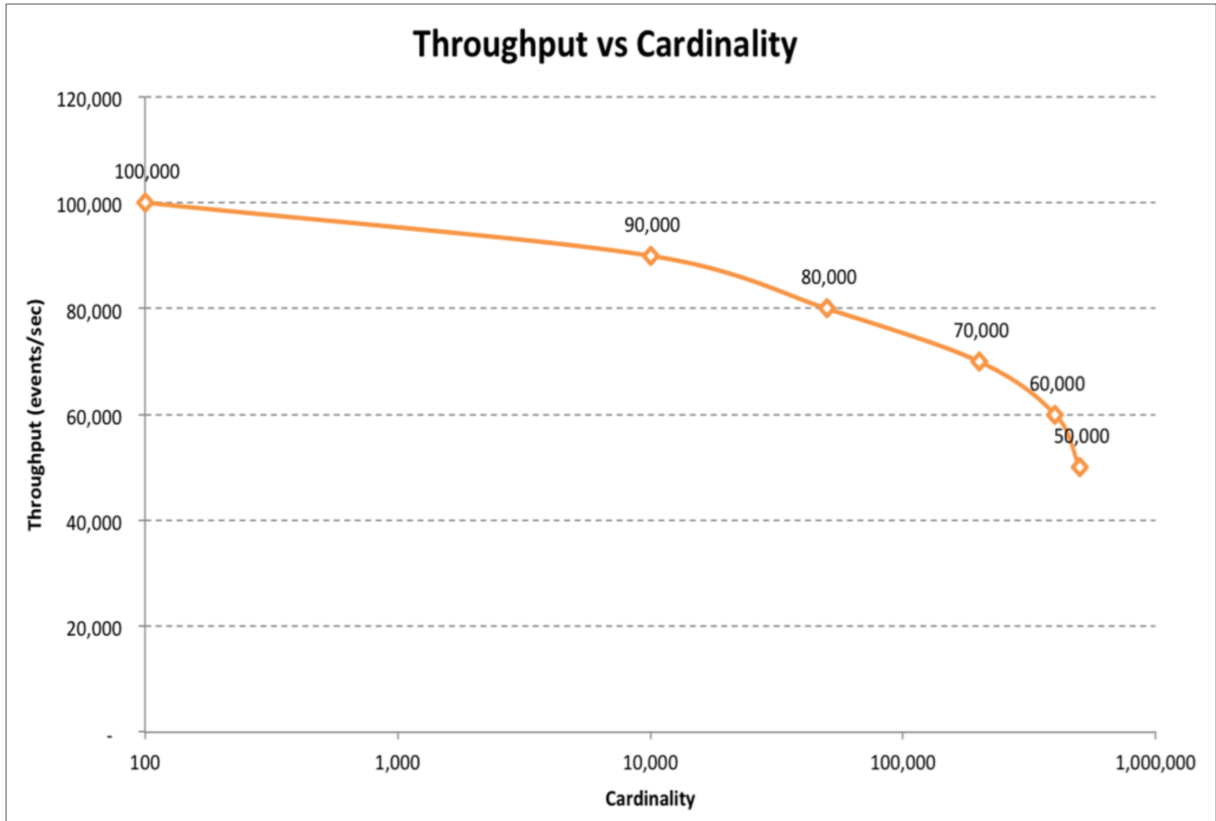

Fig 8: Throughput decreases with higher dimension cardinality.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

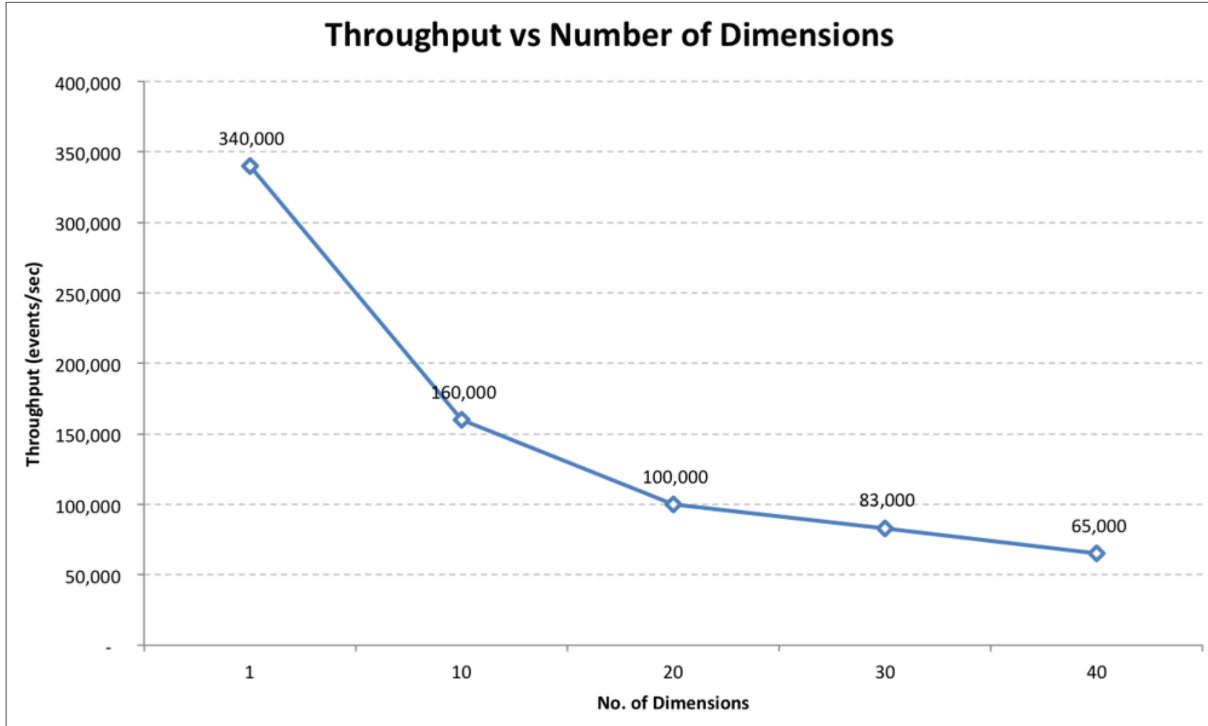

Fig 9: Increased number of dimensions reduces throughput.

Druid in Production

Using Druid in production has highlighted key lessons about operational monitoring, system integration, and handling failures. Despite extensive testing, production environments reveal unanticipated issues, often stemming from the overlooked impact of small features.

A. Key Observations

- **Exploratory Dashboards:** Many users issue inefficient queries without understanding system impacts, such as repeated dashboard refreshes for large data sets. Druid has improved defences against repetitive expensive queries.
- **Multitenancy Benefits:** Hosting multiple data sources in the same cluster improves parallelization and scalability as nodes are added.
- **Caching Importance:** A high percentage of queries hit the broker cache, as users often have repetitive queries.
- **Memory-Mapped Storage:** Disk paging can severely impact query performance; SSDs mitigate this issue.
- **Approximate Algorithms:** Leveraging approximations reduces storage costs and improves performance, with most users tolerating minor errors.

B. Operational Monitoring

Effective monitoring is critical for large clusters. Druid nodes emit metrics covering system performance (e.g., CPU, memory, disk), JVM stats (e.g., garbage collection, heap usage), and query-specific metrics (e.g., filters, intervals).

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

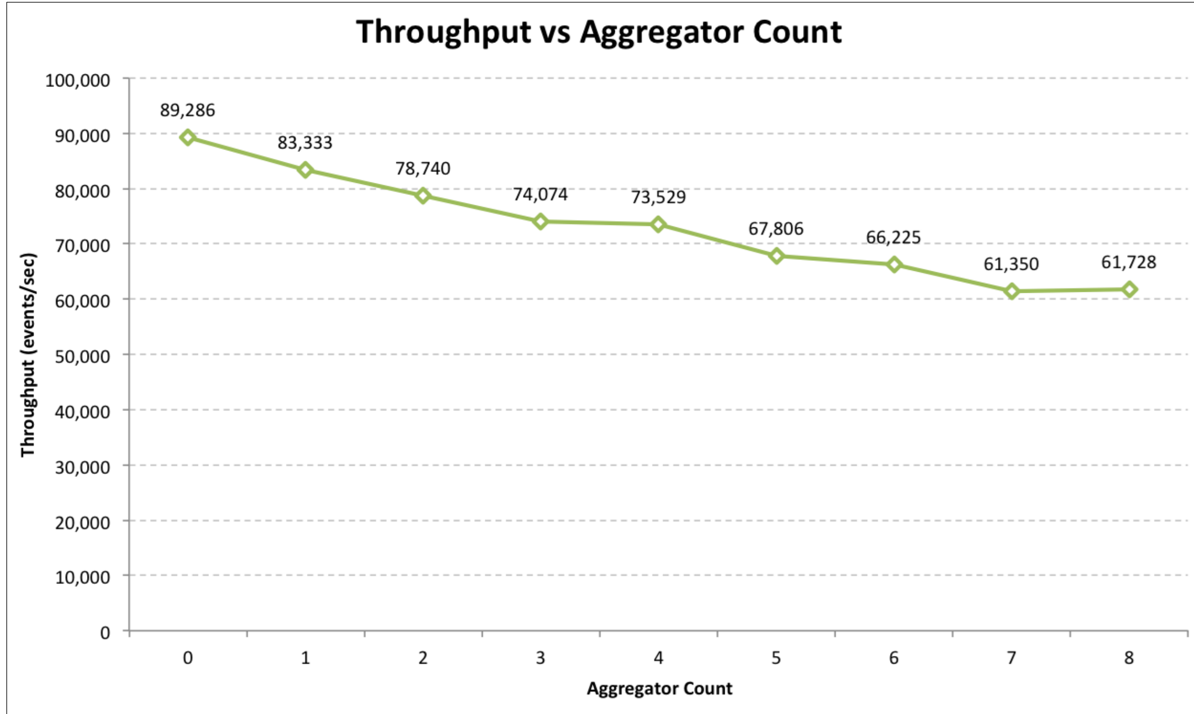

Fig 10: Adding metrics reduces ingestion latency, though effects are less severe than adding dimensions.

Table 2: Druid Queries

Query #	Query
1	SELECT count (*) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ?
2	SELECT count (*), sum (metric1) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ?
3	SELECT count (*), sum (metric1), sum (metric2), sum (metric3), sum (metric4) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ?
4	SELECT high_card_dimension, count (*) AS cnt FROM _table_ WHERE timestamp ≥ ? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt LIMIT 100
5	SELECT high_card_dimension, count (*) AS cnt, sum (metric1) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt LIMIT 100
6	SELECT high_card_dimension, count (*) AS cnt, sum (metric1), sum (metric2), sum (metric3), sum (metric4) FROM _table_ WHERE timestamp ≥ ? AND timestamp < ? GROUP BY high_card_dimension ORDER BY cnt LIMIT 100

Metrics can be ingested into a dedicated metrics Druid cluster, enabling insights into cluster stability and user behavior. This setup has uncovered query slowdowns, hardware inefficiencies, and bottlenecks, informing system tuning and roadmap decisions.

C. Pairing Druid with Stream Processors

Druid operates on fully denormalized data, making it suitable for pairing with stream processors like Apache Storm [15]. Storm handles transformations, business logic, and multi-stream joins, forwarding processed data to Druid for real-time and historical query handling.

D. Multi-Data Center Distribution

Druid supports segment replication across multiple tiers and data centers. This configuration enables primary clusters in one data center (preferred for queries) with redundant clusters in others for disaster recovery. Such setups are particularly useful when proximity to users is critical.

Related Work

Cattell [16] and Hu [17] provide comprehensive surveys of scalable SQL/NoSQL data stores and streaming databases, respectively. Feature-wise, Druid is positioned between Google's Dremel [7] and PowerDrill [18], sharing Dremel's core features (though with limited nesting) and some compression techniques from PowerDrill.

Compared to distributed columnar stores [19], Druid focuses on computation within the storage layer, unlike generic key value systems like Cassandra [9]. In-memory databases such as SAP HANA [20] and VoltDB [21] lack Druid's low-latency ingestion, while Druid also supports rolling updates with no downtime, like systems like ParAccel [22].

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

Druid combines a read-optimized subsystem (historical nodes) with a write-optimized subsystem (real-time nodes), akin to [23, 24], but is tailored for OLAP rather than OLTP workloads. Its low-latency ingestion is comparable to Trident/Storm [15] and Streaming Spark [25], though these focus on stream processing, complementing Druid's ingestion and aggregation capabilities.
Systems like Shark [26] and Impala [27] optimize queries on cluster frameworks, but Druid's historical nodes leverage native indexes for faster query latencies. Additionally, Druid's use of inverted indices for fast filtering mirrors techniques in other data stores [28].
Druid's unique combination of ingestion, aggregation, and query optimisations distinguishes it within the ecosystem of distributed data stores.

CONCLUSIONS

In this paper, I introduced Druid, a distributed, column oriented, real-time analytical data store, specifically designed to support high-performance, large-scale applications that require low-latency query responses. Druid is optimized to handle both real-time streaming data ingestion and fast, efficient querying, making it suitable for use cases that demand quick insights from massive datasets. I discussed Druid's architecture in depth, emphasizing key components such as its distributed storage format, which is highly efficient for large-scale data storage and fast retrieval. Druid's ability to ingest and query streaming data sets it apart from many traditional databases, offering fault tolerance and seamless scaling to handle growing data volumes. By leveraging a columnar storage model and advanced indexing techniques, Druid can process large datasets in real-time, delivering fast results for analytical queries. Furthermore, I presented a set of benchmarks that illustrate Druid's performance in various scenarios, including query execution speed and data ingestion rates. These benchmarks demonstrate Druid's ability to efficiently process complex queries over vast datasets, showing significant improvements in query performance with scaling, and highlighting the importance of using parallelism to maintain low latency even in large clusters. The query language, while SQL-like, is tailored to Druid's underlying architecture, supporting high-performance, time-series-based analytical queries. Additionally, Druid's query execution engine optimizes for aggregations, groupings, and filtering, enabling interactive data exploration even in high-throughput environments. In conclusion, Druid provides a unique combination of low latency data ingestion and high-performance querying, making it a compelling choice for modern analytical applications that require fast insights from large-scale datasets. Its architecture, designed to scale horizontally, provides fault tolerance and robust support for real-time analytics, positioning Druid as a powerful tool for handling the demands of contemporary data intensive workloads.

REFERENCES

1. Singh M, Leonhardi B. Introduction to the IBM Netezza warehouse appliance. In: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp.; 2011. p. 385-386.
2. Bear C, Lamb A, Tran N. The Vertica database: SQL RDBMS for managing big data. In: Proceedings of the 2012 Workshop on Management of Big Data Systems. ACM; 2012. p. 37-38.
3. Miner D. Unified analytics platform for big data. In: Proceedings of the WICSA/ECSA 2012 Companion Volume. ACM; 2012. p. 176.
4. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM. 2008;51(1):107-113.
5. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies. IEEE; 2010. p. 1-10.
6. Oehler K, Gruenes J, Ilacqua C, Perez M. IBM Cognos TM1: The Official Guide. New York: McGraw-Hill; 2012.
7. Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, *et al*. Dremel: Interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment. 2010;3(1-2):330-339.
8. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, *et al*. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems. 2008;26(2):4.
9. Lakshman A, Malik P. Cassandra—A decentralized structured storage system. Operating Systems Review. 2010;44(2):35.
10. Tschetter E. Introducing Druid: Real-time analytics at a billion rows per second. Available from: http://druid.io/blog/2011/04/30/introducing-druid.html
11. Abadi DJ, Madden SR, Hachem N. Column-stores vs. row-stores: How different are they really? In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM; 2008. p. 967-980.
12. Liblzf [Internet]. 2013 Mar. Available from: http://freecode.com/projects/liblzf
13. Colantonio A, Di Pietro R. Concise: Compressed 'n' composable integer set. Information Processing Letters. 2010;110(16):644-650.
14. Amdahl GM. Validity of the single processor approach to achieving large-scale computing capabilities. In: Proceedings of the April 18-20, 1967 Spring Joint Computer Conference. ACM; 1967. p. 483-485.
15. Marz N. Storm: Distributed and fault-tolerant realtime computation. 2013 Feb. Available from: http://storm-project.net/
16. Cattell R. Scalable SQL and NoSQL data stores. ACM SIGMOD Record. 2011;39(4):12-27.
17. Hu B. Stream database survey. 2011.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/

18. Hall A, Bachmann O, Bussow R, Gănceanu S, Nunkesser M. Processing a trillion cells per mouse click. Proceedings of the VLDB Endowment. 2012;5(11):1436-1446.
19. Fink B. Distributed computation on Dynamo-style distributed storage: Riak pipe. In: Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop. ACM; 2012. p. 43-50.
20. Farber F, Cha SK, Primsch J, Bornhövd C, Sigg S, Lehner W. SAP HANA database: Data management for modern business applications. ACM SIGMOD Record. 2012;40(4):45-51.
21. VoltDB L. VoltDB technical overview. 2010. Available from: https://voltdb.com/
22. ParAccel analytic database. 2013 Mar. Available from: http://www.paraccel.com/resources/Datasheets/ParAccel-Core-Analytic-Database.pdf
23. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, *et al*. C-Store: A column-oriented DBMS. In: Proceedings of the 31st International Conference on Very Large Databases. VLDB Endowment; 2005. p. 553-564.
24. Cipar J, Ganger G, Keeton K, Morrey CB III, Soules CA, Veitch A. LazyBase: Trading freshness for performance in a scalable database. In: Proceedings of the 7th ACM European Conference on Computer Systems. ACM; 2012. p. 169-182.
25. Zaharia M, Das T, Li H, Shenker S, Stoica I. Discretised streams: An efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing. USENIX Association; 2012. p. 10.
26. Engle C, Lupher A, Xin R, Zaharia M, Franklin MJ, Shenker S, *et al*. Shark: Fast data analysis using coarse-grained distributed memory. In: Proceedings of the 2012 International Conference on Management of Data. ACM; 2012. p. 689-692.
27. Cloudera Impala. 2013 Mar. Available from: http://blog.cloudera.com/blog
28. MacNicol R, French B. Sybase IQ multiplex-designed for analytics. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB Endowment; 2004. p. 1227-1230.

Creative Commons (CC) License

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY 4.0) license. This license permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

CONFERENCE ORGANIZERS

- Desert Research Association (DRA), Headquarters – Jodhpur
- Nehru Study Centre, Jai Narain Vyas University, Jodhpur
- Government Girls College, Jhalamand (Jodhpur)
- Department of Geography, Dr. Bhim Rao Ambedkar Government College, Sri Ganganagar

In Collaboration with Kalinga University, Raipur (Chhattisgarh)

Disclaimer: The views, opinions, statements, and conclusions expressed in the papers, abstracts, presentations, and other scholarly contributions included in this conference are solely those of the respective authors. The organisers and publisher shall not be held responsible for any loss, harm, damage, or consequences — direct or indirect — arising from the use, application, or interpretation of any information, data, or findings published or presented in this conference. All responsibility for the originality, authenticity, ethical compliance, and correctness of the content lies entirely with the respective authors.

© 2025 Bhuvan Chandra Sarakam. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY NC ND).https://creativecommons.org/licenses/by/4.0/