ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

ADVANTAGES OF DRUID DATA STORE OVER ONLINE ANALYTICAL PROCESSING -A COMPHRENSIVE STUDY

Mutteni Nagamani*
Asst.Professor, MERI.

ABSTRACT

Druid is a open source data store which is designed for real time application analysis on large amount of data sets. It stores the data in the form of column oriented. It is distributed and an advanced indexing structure to allow for the arbitrary exploration of billion-row tables with sub-second latencies. This paper discusses about the advantages of druid over online analytical processing to.

KEYWORDS— OLAP, Client engine, Server engine, ROLAP, MOLAP, HOLAP, Druid, Real nodes, historical nodes, Storage Engine.

INTRODUCTION

OLAP (online analytical processing) is computer processing that enables a user to easily and selectively extract and view data from different points of view. OLAP allows users to analyze database information from multiple database systems at one time. OLAP data is stored in multidimensional databases.
OLAP systems have a structured architecture based on three essential components:

1. Database - the data source used for OLAP analysis. A database can use as a relational database to ensure our multidimensional storage facilities, a multidimensional database, a data warehouse, etc.

2. **OLAP server** - the one that manages multidimensional data structure and at the same time a link between the database and OLAP customer.

3. **OLAP customer** - are those that provide data mining applications but also supports the generation of results (graphs, reports, etc.).

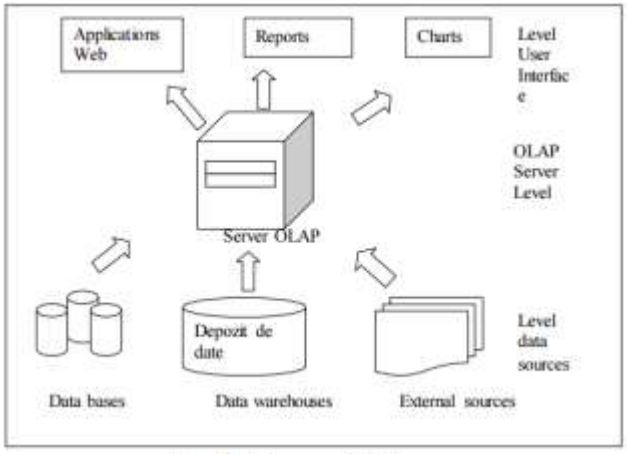

Figure 1: Architecture of OLAP Systems

There are several options in OLAP data could be stored and processed. Thus, depending on the method of organizing and storing data, there may be three options

- **Client Files -** data is stored locally on a client computer as files are organized, on which operations can be applied to analyze the processing and transformation. This organization of data has some drawbacks of which we can enumerate the amount that can be processed is indulged reduced time to processing information is quite high, the data shows a poor security, lack of advanced multidimensional analysis.
- **Relational databases** - this is used when the data comes as a relational DBMS and data warehouse is a repository be implemented virtually or using a relational model.
- **Databases multidimensional** - in this case, the data are organized into a data warehouse on a dedicated server, which is called multidimensional server. On multidimensional data can be applied by the server multidimensional operations. The data are drawn from various sources (relational databases, files), transformed and loaded into tables of facts and dimensions and units on different levels, pre-processed and prepared for analysis.

While processing three options can be chosen, namely:

- **The core of SQL** - is not the most appropriate options to perform complex multidimensional calculations because SQL language does not have facilities to perform multidimensional calculations directly and to have the same results as when using multi-dimensional functions and operations, be made several stages.
- **Multidimensional client engine** - some operations can be performed locally if the user has a relatively powerful system. If this alternative, the user must have advanced knowledge because he will have to build and apply analysis functions.
- **Multidimensional server engine** –To achieve multidimensional operations a dedicated server is used. Dedicated server provides access resources and applications competitor analysis is performed in real time. OLAP tools enable users to store data in both relational databases and multidimensional databases.

OLAP systems can be classified as: -

1. ROLAP systems
2. MOLAP systems
3. HOLAP systems

1. **ROLAP (Relational OLAP)**: OLAP systems that allow for appropriate data is analyzed as a relational database (data warehouse). Basic data storage and data is aggregated in a relational database, and deterioration is done. Storing multidimensional data in data warehouses or data center will be run using star or snow flake schema. It aims to ensure quick access and flexibility for handling multidimensional.
   When there are many variables, there is a risk that data cannot be stored in one table when facts and facts prove that the table is not an effective solution. Upgrading a single large fact tables proved to be an ineffective solution. Thus, the table actually used for more complex applications is divided into groups of variables, taking into account the degree of dispersion, setting the appropriate data sources and sizes.

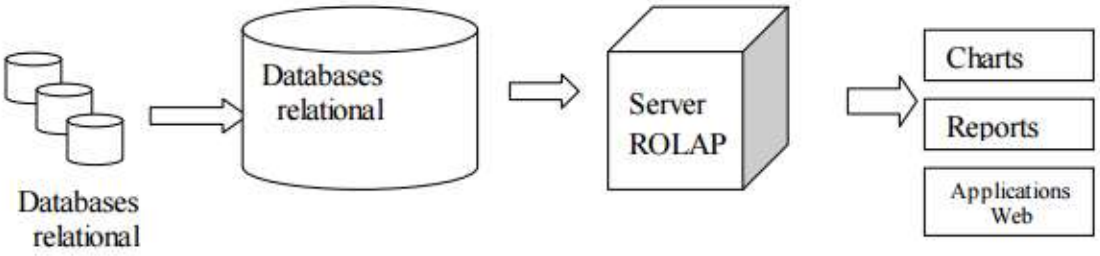

Figure 2: ROLAP Architecture

2. **MOLAP (Multi dimensional OLAP):** Multidimensional OLAP is used when we have fewer amounts of initial data and solutions are seen as classic multidimensional analysis. Stores both data base and aggregated

data in a multidimensional database, called the Cube, which are used as effective tools for operations analysis and to perform complex calculations.

Basically, MOLAP mainframe systems provide users with a multidimensional view of data. MOLAP systems have focused on optimizing the flexibility and storage techniques and the concept of transaction.

MOLAP systems are much faster in terms of data aggregation and in terms of queries, however, generate large volumes of data hedge. Response time the query is improved because of precalculates aggregations of such data and responses to queries are prepared before launching the application.

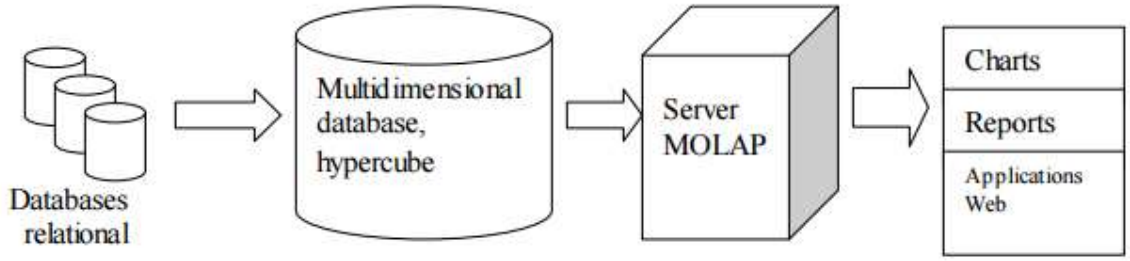

Figure 3: MOLAP Architecture

3. **HOLAP (Hybrid OLAP):** is a hybrid approach to the solution where the aggregated totals are stored in a multidimensional database while the detail data is stored in the relational database. This is the balance between the data efficiency of the ROLAP model and the performance of the MOLAP model.

Druid

Druid is an open source distributed data store which is written in java and which stores the data in column oriented. Druid retrieves the large amount of data quickly for the relevant queries. A Druid cluster consists of different types of nodes and each node type is designed to perform a specific set of things. This design separates concerns and simplifies the complexity of the overall system. The different node types operate fairly independent. of each other and there is minimal interaction among them. Hence, intra-cluster communication failures have minimal impact on data availability. To solve complex data analysis problems, the different node types come together to form a fully working system. The composition of and flow of data in a Druid cluster are shown in Figure I.

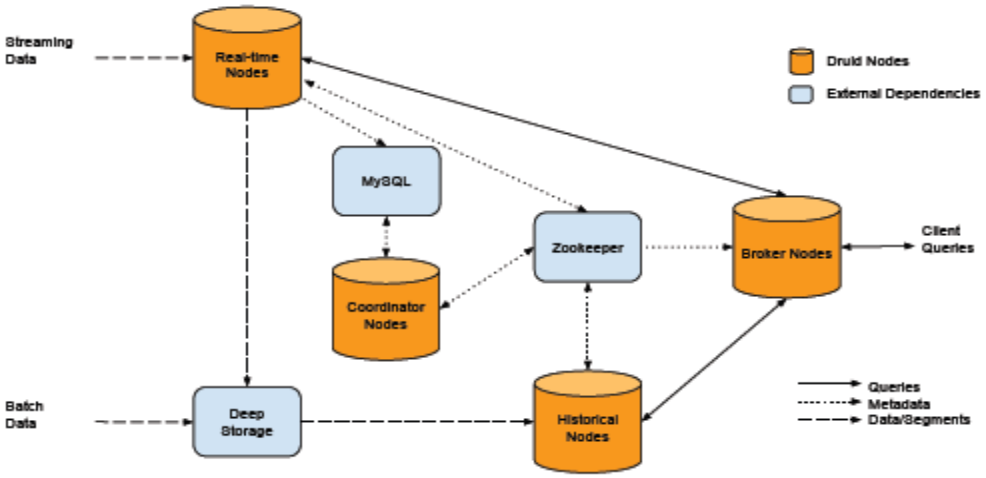

Figure 4: An overview of a Druid cluster and the flow of data through the cluster.

1. **Real-time nodes:** Real-time nodes encapsulate the functionality to ingest and query event streams. Events indexed via these nodes are immediately available for querying. The nodes are only concerned with events for some small time range and periodically hand off immutable batches of events they have collected over this small time range to other nodes in the Druid cluster that are specialized in dealing with batches of immutable events.

Real-time nodes leverage Zookeeper for coordination with the rest of the Druid cluster. The nodes announce their online state and the data they serve in Zookeeper.

Real-time nodes maintain an in-memory index buffer for all incoming events. These indexes are incrementally populated as events are ingested and the indexes are also directly queryable. Druid behaves as a row store for queries on events that exist in this JVM heap-based buffer. To avoid heap overflow problems, real-time nodes persist their in-memory indexes to disk either periodically or after some maximum row limit is reached. This persist process converts data stored in the in-memory buffer to a column oriented storage format Each persisted index is immutable and real-time nodes load persisted indexes into off-heap memory such that they can still be queried.

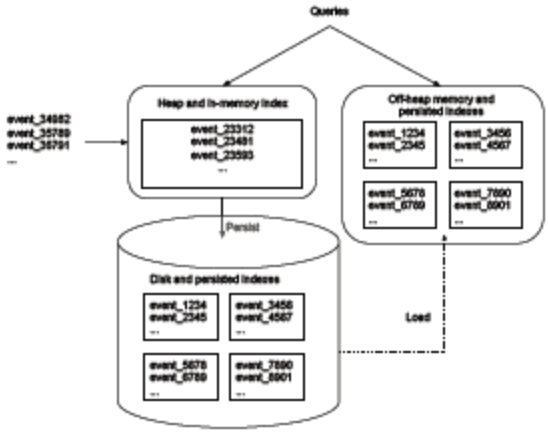

Figure 5: Real-time nodes buffer events to an in-memory index, which is regularly persisted to disk.

On a periodic basis, each real-time node will schedule a background task that searches for all locally persisted indexes. The task merges these indexes together and builds an immutable block of data that contains all the events that have been ingested by a real time node for some span of time. We refer to this block of data as a "segment". During the handoff stage, a real-time node uploads this segment to a permanent backup storage, typically a distributed file system such as which Druid refers to as "deep storage". The ingest, persist, merge, and handoff steps are fluid; there is no data loss during any of the processes.

HISTORICAL NODES

Historical nodes encapsulate the functionality to load and serve the immutable blocks of data (segments) created by real-time nodes. In many real-world workflows, most of the data loaded in a Druid cluster is immutable and hence, historical nodes are typically the main workers of a Druid cluster. Historical nodes follow shared nothing architecture and there is no single point of contention among the nodes. The nodes have no knowledge of one another and are operationally simple; they only know how to load, drop, and serve immutable segments.

Similar to real-time nodes, historical nodes announce their online state and the data they are serving in Zookeeper. Instructions to load and drop segments are sent over Zookeeper and contain information about where the segment is located in deep storage and how to decompress and process the segment. Before a historical node downloads a particular segment from deep storage, it first checks a local cache that maintains information about what segments already exist on the node. If information about a segment is not present in the cache, the historical node will proceed to download the segment from deep storage. This process is shown in Figure 7. Once processing is complete, the

segment is announced in Zookeeper. At this point, the segment is queryable. The local cache also allows for historical nodes to be quickly updated and restarted. On startup, the node examines its cache and immediately serves whatever data it finds.

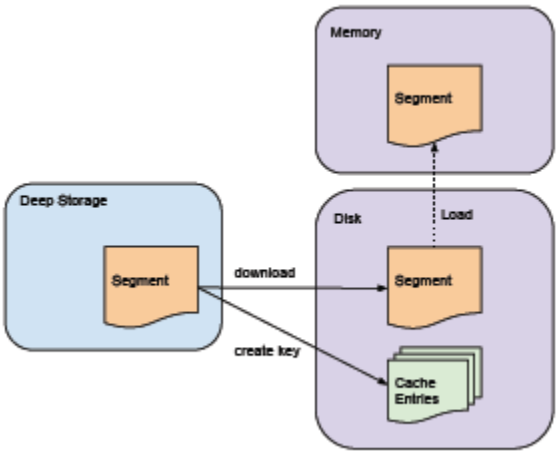

Figure 6: Historical nodes download immutable segments from deep storage. Segments must be loaded in memory before they can be queried.

Historical nodes can support read consistency because they only deal with immutable data. Immutable data blocks also enable a simple parallelization model. Historical nodes can concurrently scan and aggregate immutable blocks without blocking.

2.1 Tiers

Historical nodes can be grouped in different tiers, where all nodes in a given tier are identically configured. Different performance and fault-tolerance parameters can be set for each tier. The purpose of tiered nodes is to enable higher or lower priority segments to be distributed according to their importance.

2.2 Availability

Historical nodes depend on Zookeeper for segment load and unload instructions. Should Zookeeper become unavailable, historical nodes are no longer able to serve new data or drop outdated data, however, because the queries are served over HTTP, historical nodes are still able to respond to query requests for the data they are currently serving. This means that Zookeeper outages do not impact current data availability on historical nodes.

2.3 Broker Nodes

Broker nodes act as query routers to historical and real-time nodes. Broker nodes understand the metadata published in Zookeeper about what segments are queryable and where those segments are located.

Broker nodes route incoming queries such that the queries hit the right historical or real-time nodes. Broker nodes also merge partial results from historical and real-time nodes before returning a final consolidated result to the caller.

2.4 Caching

Broker nodes contain a cache with a LRU invalidation strategy. The cache can use local heap memory or an external distributed key/value store such as Memory cached Each time a broker node receives a query, it first maps the query to a set of segments. Results for certain segments may already exist in the cache and there is no need to recompute them. For any results that do not exist in the cache, the broker node will forward the query to the correct historical and real-time nodes. Once historical nodes return their results, the broker will cache these results on a per segment basis for future use. This process is illustrated in Figure 6. Real-time data is never cached and hence requests for real-time data will always be forwarded to real-time nodes. Real-time data is perpetually changing and caching the results is

unreliable. The cache also acts as an additional level of data durability. In the event that all historical nodes fail, it is still possible to query results if those results already exist in the cache

2.5 Availability

In the event of a total Zookeeper outage, data is still queryable. If broker nodes are unable to communicate to Zookeeper, they use their last known view of the cluster and continue to forward queries to real-time and historical nodes. Broker nodes make the assumption that the structure of the cluster is the same as it was before the outage. In practice, this availability model has allowed our Druid cluster to continue serving queries for a significant period of time while we diagnosed Zookeeper outages.

COORDINATOR NODES

Druid coordinator nodes are primarily in charge of data management and distribution on historical nodes. The coordinator nodes tell historical nodes to load new data, drop outdated data, replicate data, and move data to load balance. Druid uses a multi-version concurrency control swapping protocol for managing immutable segments in order to maintain stable views. If any immutable segment contains data that is wholly obsolete by newer segments, the outdated segment is dropped from the cluster. Coordinator nodes undergo a leader-election process that determines a single node that runs the coordinator functionality. The remaining coordinator nodes act as redundant backups.

A coordinator node runs periodically to determine the current state of the cluster. It makes decisions by comparing the expected state of the cluster with the actual state of the cluster at the time of the run. As with all Druid nodes, coordinator nodes maintain a Zookeeper connection for current cluster information. Coordinator nodes also maintain a connection to a MySQL database that contains additional operational parameters and configurations. One of the key pieces of information located in the MySQL database is a table that contains a list of all segments that should be served by historical nodes. This table can be updated by any service that creates segments, for example, real-time nodes. The MySQL database also contains a rule table that governs how segments are created, destroyed, and replicated in the cluster.

3.1 Rules

Rules govern how historical segments are loaded and dropped from the cluster. Rules indicate how segments should be assigned to different historical node tiers and how many replicates of a segment should exist in each tier. Rules may also indicate when segments should be dropped entirely from the cluster. Rules are usually set for a period of time. For example, a user may use rules to load the most recent one month's worth of segments into a "hot" cluster, the most recent one year's worth of segments into a "cold" cluster, and drop any segments that are older.

The coordinator nodes load a set of rules from a rule table in the MySQL database. Rules may be specific to a certain data source and/or a default set of rules may be configured. The coordinator node will cycle through all available segments and match each segment with the first rule that applies to it.

3.2 Load Balancing

In a typical production environment, queries often hit dozens or even hundreds of segments. Since each historical node has limited resources, segments must be distributed among the cluster to ensure that the cluster load is not too imbalanced. Determining optimal load distribution requires some knowledge about query patterns and speeds. Typically, queries cover recent segments spanning contiguous time intervals for a single data source. On average, queries that access smaller segments are faster. These query patterns suggest replicating recent historical segments at a higher rate, spreading out large segments that are close in time to different historical nodes, and co-locating segments from different data sources. To optimally distribute and balance segments among the cluster, we developed a cost-based optimization procedure that takes into account the segment data source, regency, and size.

3.3 Replication

Coordinator nodes may tell different historical nodes to load a copy of the same segment. The number of replicates in each tier of the historical compute cluster is fully configurable. Setups that require high levels of fault tolerance can be configured to have a high number of replicas. Replicated segments are treated the same as the originals and follow the same load distribution algorithm. By replicating segments, single historical node failures are transparent in the

Druid cluster. We use this property for software upgrades. We can seamlessly take a historical node offline, update it, bring it back up, and repeat the process for every historical node in a cluster.

3.4 Availability

Druid coordinator nodes have Zookeeper and MySQL as external dependencies. Coordinator nodes rely on Zookeeper to determine what historical nodes already exist in the cluster. If Zookeeper becomes unavailable, the coordinator will no longer be able to send instructions to assign, balance, and drop segments. However, these operations do not affect data availability at all. The design principle for responding to MySQL and Zookeeper failures is the same: if an external dependency responsible for coordination fails, the cluster maintains the status quo. Druid uses MySQL to store operational management information and segment metadata information about what segments should exist in the cluster. If MySQL goes down, this information becomes unavailable to coordinator nodes. However, this does not mean data itself is unavailable. If coordinator nodes cannot communicate to MySQL, they will cease to assign new segments and drop outdated ones. Broker, historical, and real-time nodes are still queryable during MySQL outages.

STORAGE FORMAT

Data tables in Druid (called data sources) are collections of time stamped events and partitioned into a set of segments, where each segment is typically 5–10 million rows. Formally, we define a segment as a collection of rows of data that span some period of time. Segments represent the fundamental storage unit in Druid and replication and distribution are done at a segment level.

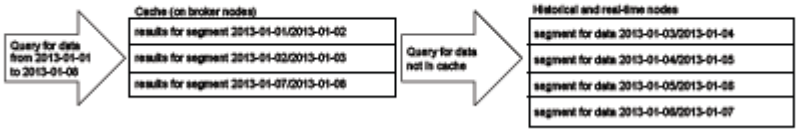

Figure 7: Results are cached per segment. Queries combine cache results with results computed on historical and real-time nodes.

Druid always requires a timestamp column as a method of simplifying data distribution policies, data retention policies, and first level query running. Druid partitions its data sources into well defined time intervals, typically an hour or a day, and may further partition on values from other columns to achieve the desired segment size. The time granularity to partition segments is a function of data volume and time range. A data set with timestamps spread over a year is better partitioned by day, and a data set with timestamps spread over a day is better partitioned by hour.

Segments are uniquely identified by a data source identifier, the time interval of the data, and a version string that increases whenever a new segment is created. The version string indicates the freshness of segment data; segments with later versions have newer views of data (over some time range) than segments with older versions. This segment metadata is used by the system for concurrency control; read operations always access data in a particular time range from the segments with the latest version identifiers for that time range.

STORAGE ENGINE

Druid's persistence components allows for different storage engines to be plugged in, similar to Dynamo. These storage engines may store data in an entirely in-memory structure such as the
JVM heap or in memory-mapped structures. The ability to swap storage engines allows for Druid to be configured depending on a particular application's specifications. An in-memory storage engine may be operationally more expensive than a memory-mapped storage engine but could be a better alternative if performance is critical. By default, a memory-mapped storage engine is used. When using a memory-mapped storage engine, Druid relies on the operating system to page segments in and out of memory. Given that segments can only be scanned if they are loaded in memory, a memory-mapped storage engine allows recent segments to retain in memory whereas segments that are never queried are paged out.

QUERY API

Druid has its own query language and accepts queries as POST requests. Broker, historical, and real-time nodes all share the same query API. The body of the POST request is a JSON object containing key value pairs specifying various query parameters. A typical query will contain the data source name, the granularity of the result data, time range of interest, the type of request, and the metrics to aggregate over. The result will also be a JSON object containing the aggregated metrics over the time period. Most query types will also support a filter set. A filter set is a Boolean expression of dimension name and value pairs. Any number and combination of dimensions and values may be specified. When a filter set is provided, only the subset of the data that pertains to the filter set will be scanned. The ability to handle complex nested filter sets is what enables Druid to drill into data at any depth. The exact query syntax depends on the query type and the information requested.

CONCLUSION

Druid is a data store which is written in java language. Druid can be run in production at several organizations. Druid has its own query language and accepts queries as POST requests and often used to explore data and generate reports on data.

REFERENCES

[1] Druid A Real-time Analytical Data Store by Fangjin Yang Metamarkets Group, Inc. fangjin@metamarkets.com Eric Tschetter echeddar@gmail.com Xavier Léauté Metamarkets Group, Inc. xavier@metamarkets.com Nelson Ray ncray86@gmail.com Gian Merlino Metamarkets Group, Inc. gian@metamarkets.com Deep Ganguli Metamarkets Group, Inc. deep@metamarkets.com
[2] OLAP Adrienne H. Slaughter
[3] ON-LINE ANALYTICAL PROCESSING Alberto Abell´o and Oscar Romero Universitat Polit`ecnica de Catalunya
[4] Oracle Express Server Datasheet, Oracle Corporation. 1997.
[5] Oracle OLAP Products: Adding Value to the Data Warehouse. Dave Menninger, Oracle. 1997.
[6] Oracle OLAP Technology: Enabling Better Business Decisions. [ORC1] Oracle(r) Express(r) Server Blazes to New OLAP Benchmark Record on Sun Servers. Press Release, Oracle Corporation. May 18, 1998.
[7] http://www.seas.gwu.edu/
[8] www.srmuniv.ac.in
[9] http://www.cse.hcmut.edu.vn/