A Comprehensive Evaluation of Big Data Management Systems

Sunit Singh
M.S. Computer Science,
Ohio State University

Ben Michalowicz
Ph.D. Computer Science,
Ohio State University

Database Management Systems (DBMS) have existed for the past several decades, with each manager providing different key performance attributes, design and implementation ideas, and optimizations during query execution. In this report, we evaluate ten different DBMSs: Apache Druid, NoisePage, Google BigQuery, Snowflake, MonetDB, Amazon Redshift, DataBricks, Google Napa, Vertica, and Leanstore. We will attempt to break down the characteristics of each DBMS, compare performance metrics where possible, and offer insights on what might be the "best" scenario for each DBMS.

This report will be broken down as follows: We will first give an overview of the history and design behind each DBMS with insights into the possible pros and cons of each of them in Part I. Following this, we will showcase some of the comparisons found in our survey of each system and provide our final conclusions and recommendations in Part II.

Part I: The DBMS Lineup

Apache Druid

Apache Druid is an open-source data analytics and persistence platform that is designed to store and retrieve large amounts of data extremely fast. It is a powerful query engine that can perform calculations on huge amounts of data very efficiently. Druid powers OLAP (Online Analytical Processing) applications well; wherein large chunks of data need to be analyzed from different perspectives. This involves performing complex analytical queries on the data without interrupting the system's data transactions. OLAP applications generally work with read-heavy workloads requiring aggregation operations like filtering etc.

Druid supports SQL which is helpful for applications using the standard CRUD operations to manage their data. Apache Druid is widely used in the industry; some examples include Netflix using Druid for real-time insights to ensure data quality, Salesforce uses Druid as an analytical tool for high quality insights, Twitter uses Druid to perform analytics that require Terabyte level querying of data. As can be seen from the examples, Druid is popular for data analytics at scale.

<u>Design and Implementation</u>

<u>Segments</u>:

At its core, Druid uses a heavily optimized columnar storage format abstraction called 'Segment' [26]. It is an immutable storage that cannot be modified but can be dropped, replaced, or combined once created. Each segment is generally 1 to million rows of data and can be created continuously as data is ingested into Druid.

Data Lifecycle:

Druid uses two server node types that interact directly with the segments, the Indexer node, and the Historical node [26]. The Indexer receives incoming data in streaming or batched form and organizes it into segments. The segments are then persisted in the 'Deep Storage' which is generally a distributed storage like HDFS, Amazon S3 or Google Storage. An important aspect of Druid's data persistence is in the way it manages the locality of data during storage with respect to high performant computes on the data. It is an efficient hybrid between the 2 storage-compute paradigms:

1. Co-locating data storage and compute resources – Important for extremely fast computations on data. Data pulled from the storage stays on the server.
2. Separating data storage and compute resources – Offers versatility among the types of analytical operations performed on the data. Data is brought to the compute resource upon being queried I.e., data pulled "on demand".

In Druid, a segment is not tied to a particular server, in that, it can end up in a different server than it was initially ingested in. The data can also be shuffled, repartitioned etc. when relocated to a different server. The Historical nodes are helpful in facilitating this handoff of segment from one data server to another and maintaining consistency in the data. Rather than caching, Druid uses pre-fetch to bring data from the deep storage onto the data server before the query, for performance reasons.

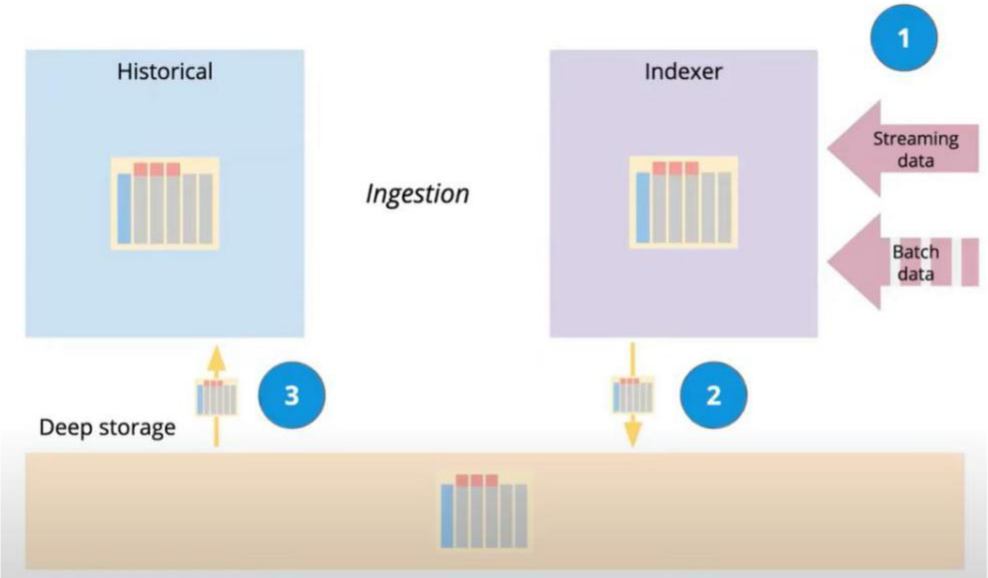


Figure: *High level schematic of data lifecycle in Apache Druid* [27]

<u>Data Server</u>:

When a query is issued to Apache Druid, it uses all available CPUs to process the query. This is achieved by using multi-threaded query processing. In the case of concurrent queries, load balancing is applied across the CPUs for optimal processing of the queries. Inspired by how Paging enhances working memory in Operating Systems, Apache Druid facilitates the hand-off of large workloads to the disk when memory is insufficient. Datasets requiring frequent updates and retrievals (hot datasets) are generally stored 50 – 80% in memory, while datasets with infrequent queries, accesses (cold datasets) are generally about 10% in memory.

A Druid data server can be broken down into 3 layers, in a bottom-up fashion:

1. Memory-mapped disk: This layer can store up to 100,000 segments on a single data server. It is designed well to work with SSD backed storage hardware systems. Apache Druid doesn't play a role in the memory mapping abstraction, it is solely handled by the operating system.
2. Per-CPU resources: This layer includes the processing thread pool, which has a single thread per CPU, and a processing memory pool, where about 0.5 - 1 GB of memory is allocated to each CPU. The thread pool management in Druid is via Java's Concurrent Util library with a minor modification in the process prioritization.
3. Per-query resources: This layer is instrumental in implementing 'Prioritization' among Druid's server resources. A server can typically have about 100 CPUs and about 100,000 segments. A single query can use every CPU in the server. Druid's server layers are designed to optimize the data workload across the different CPUs for a single query as well as concurrent queries. The Prioritization system ensures that a variety of workloads can be processed on Druid with suitable priorities. For instance, interactive workloads remain interactive (no latency) even when the server is under high load. Druid's prioritization does not use autoscaling to achieve this feature.

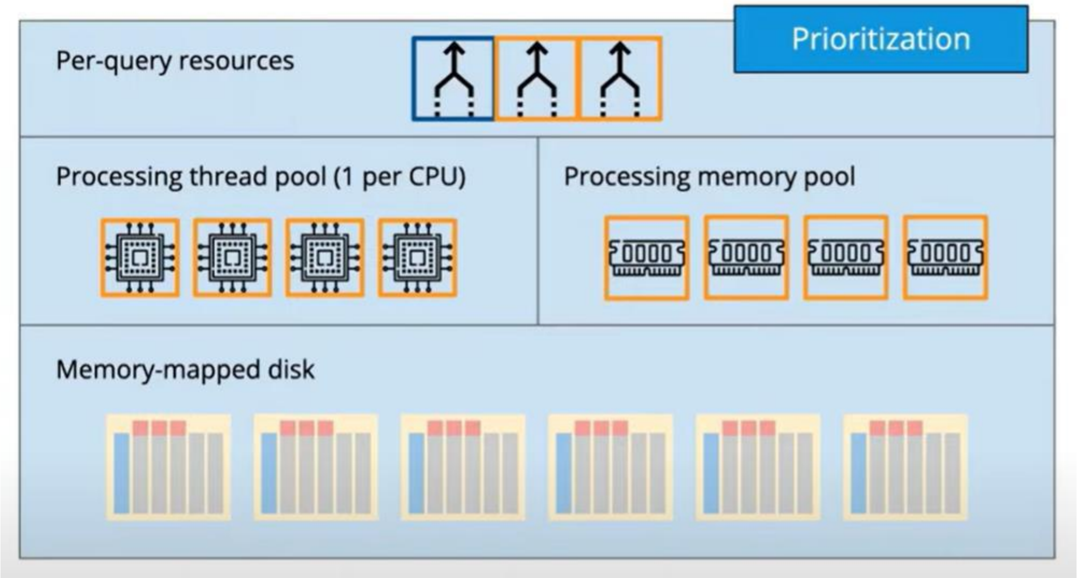


Figure: *Inside a Data Server [27]*

High priority queries (marked in orange) get priority over low priority queries (marked in blue). Druid's prioritization allows for the high priority queries to pre-empt the low priority queries until completion. This design is predicated on the observation that high priority queries in OLAP applications generally tend to be short bursts of compute requirement and have small execution time signatures. This implicitly addresses issues related to starvation and fairness for low priority queries.

Data Cluster:

Apache Druid clusters comprise of data servers and a query server called the 'Broker'. The broker serves as an interface by allowing SQL queries to be issued to the cluster. It converts the SQL queries to a lower-level native language suitable to physical operator hardware and sends them to all the relevant data servers. This process is called '*query fan-out'[31].*

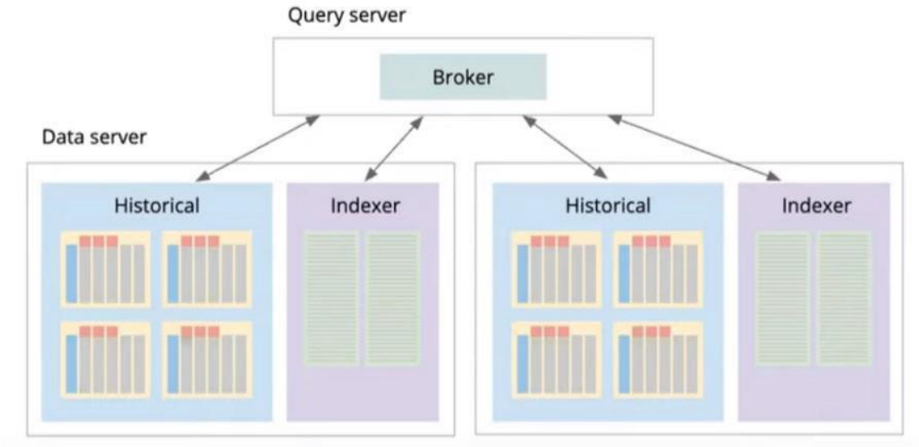


Figure: *Query fan-out in Druid's data cluster* [27]

Segment Format:

Each segment in a Druid segment is stored separately from other columns and has its own encoding and compression. That being said, the different columns are co-located on the disk and so they can be loaded from the disk together to the CPU. Druid decreases latency by scanning only those columns that contain data needed for a query. There are 3 basic column types:

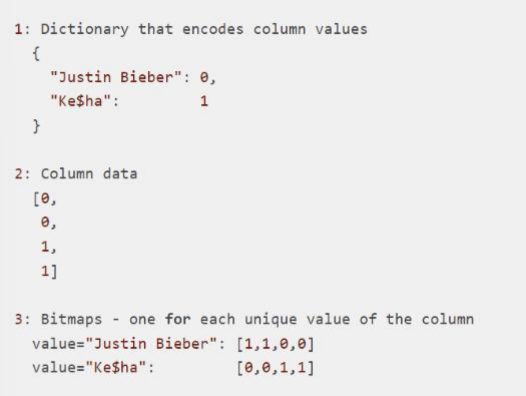
1. Timestamp
2. Dimensions
3. Metrics

Timestamp and Metrics columns use an array of integers or floating-point values with LZ4 compression applied to them. LZ4 compression is lossless and is focused on speed and compression ratio. LZ4 compression strategy is useful in Druid for queries requiring multiple aggregations that fully scan the data. It is poor for queries requiring heavy filter operations. The retrieval workflow using LZ4 compression is: once a query is issued to the server, the data is decompressed, and relevant rows are taken. Thereafter the desired aggregation operation is applied [25].

Dimensions columns support filter and group-by operations and made using the following data structures:

1. Dictionary - Maps string values to integer IDs
2. List - Column values encoded using the Dictionary
3. Bitmap - Each column value has a bitmap index associated with it to indicate the rows containing the column value. These are stored in compressed form and can be combined (intersection, union etc.) in their compressed form.

Lists of column values are helpful in performing group-by and top-N queries. Bitmaps are useful here because they facilitate fast filtering operations and are useful when applying AND and OR operations. Data can be reconstructed using bitmap indexes, while already being stored in Druid's segments. Druid stores data in 2 different ways in order to allow different operations on the same data such as filtering (index efficient) versus grouping (segment efficient).



```
1: Dictionary that encodes column values
  {
    "Justin Bieber": 0,
    "Ke$ha":         1
  }

2: Column data
  [0,
   0,
   1,
   1]

3: Bitmaps - one for each unique value of the column
  value="Justin Bieber": [1,1,0,0]
  value="Ke$ha":         [0,0,1,1]
```

Figure: *Sample column types in Druid segment, courtesy of [26]*

Lists of column values are helpful in performing group-by and topN queries. Bitmaps are useful here because they facilitate fast filtering operations and are useful when applying AND and OR operations. Data can be reconstructed using bitmap indexes, while already being stored in Druid's segments. Druid stores data in 2 different ways to allow different operations on the same data such as filtering (index efficient) versus grouping (segment efficient).

A segment is split into blocks. Typically, a block in a Druid segment has about 1000 rows and is individually compressed using LZ4 compression. Upon decompressing, an entire block must be brought from disk to memory and into the CPU before applying LZ4 decompression. Furthermore, LZ4 compression prevents random access in compressed data payload. Data layout techniques like sorted clusters of entities help in performance gains as it offers better data locality and minimizes number of blocks to read.

<u>**Noteworthy Points**</u>:

Apache Druid spreads the values of each row across columns. This design is not meant for systems that need frequent single row retrievals. Instead, it is optimized for scanning large data chunks and storing rows in column-fragmented format as data chunks can be loaded into memory separately and so rows likely to be queried together can be clustered together. Another point to note is the immutability of the data segments and the manner in which Druid segments are not tied to a data server, making it difficult to rectify chunks of the data. In such cases the entire data payload needs to be re-processed before being re-indexed and partitioned.

NoisePage

Introduction and Motivation

NoisePage is designed to be a self-driving database management system, where the system automatically manages itself, without requiring manual interventions by human database administrators. This autonomous approach is designed to reduce the burden placed on human database administrators, as database management systems are challenging to administer and deploy. This complexity makes such systems expensive to manage – staffing is estimated to be almost half of the total cost of DBMS and database administrators report spending most of their time tuning system performance [18].

Design and Implementation

NoisePage predicts the behavior of the DBMS using machine learning algorithms. The SQL queries used in the DBMS deployment are used to forecast the workload on-line. The forecasting framework utilizes unsupervised learning and ensemble machine learning models to predict query arrival rates under different DB workload patterns. The self-driving DBMS behavior is predicted offline by separating the DBMS architecture into components to evaluate the system's behavior under different conditions. NoisePage uses the forecasted workload and behavior models as inputs to complete the action planning on-line. The goal of the action-planning framework is to generate explainable decisions to optimize system performance based on the workload forecast and predicted system behavior [14, 18].

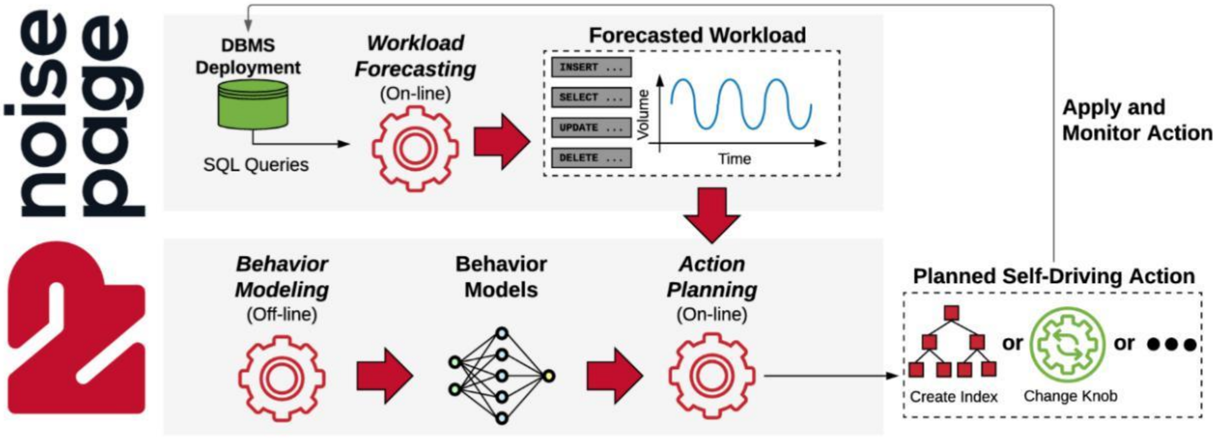


Figure: *NoisePage Architecture – NoisePage's self-driving architecture consists of on an on-line workload forecasting off-line behavior modeling component, and an on-line action planning component.* [14]

Challenges

One common challenge in autonomic or self-organizing systems is that the design presumes that the designers can accurately articulate the utility function to represent the system's high-level objectives. However, it is often the case that humans cannot accurately describe the utility function that will generate their desired outcomes, particularly in highly complex systems. NoisePage leverages recent advancements in ML techniques to overcome these challenges, but more research and development is required to build a fully autonomous database system.

Google BigQuery

Introduction and Motivation:

BigQuery is the backing data warehouse behind Google Cloud, allowing users to run analytics on data at the petabyte scale. It is another RDBMS and given that it appears to have less of a research background than, e.g., MonetDB or NoisePage, it focuses that much more on customer satisfaction with different pricing models and scales for different cloud images in the Google Cloud Platform (GCP) [7]. It approaches being an RDBMS with a "software as a service" (SaaS) perspective alongside being a more closed-source design/implementation [8]. The authors of [8] presented one of the first papers on Google BigQuery in 2015, explaining its features and where it sits in the fields of Big Data and Cloud Computing.

The prime reasons for its nascence revolved around fixing issues they believed existed with Hadoop's MapReduce procedure as well as a successor to the initial solution, Dremel, that they had released in 2008. The authors explain that MapReduce is not a real-time solution. That is, it could work efficiently on large amounts of data, but not while data was, e.g., being streamed in real-time. Focusing on the aspect of Big Data analytics versus standard database querying, BigQuery's initial paper explained how its ability to handle "millions of information not indexed" in seconds are thanks to three main aspects.

[9] brings up other aspects of BigQuery's abilities to make it stand out in the realm of big data, such as the designs behind its scalability (more on this later), the fact that it's an interactive system, and the fact that it is an ad-hoc system – that is, as mentioned earlier, it is able to do full-table scans per query without prior knowledge, indexes, aggregation, etc. While this is great – similar to doing any sort of on-the-fly functions – the use of caching and indexing, especially for repeated queries – might do well to serve better in the long run, such as what's seen in MonetDB (see below).

Design and Implementation:

[8] Breaks BigQuery's ability to handle the "velocity" aspect of data coming in down to two main factors: Columnar storage and a tree-based setup.

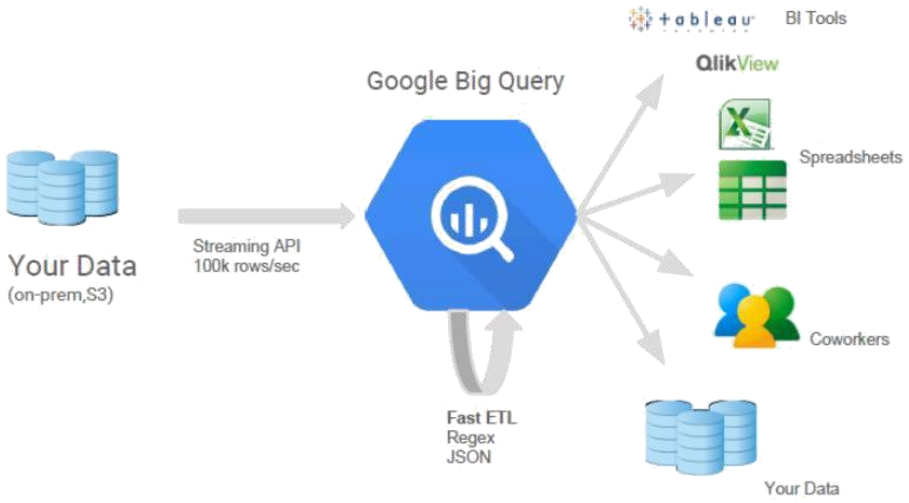


Figure: *Google BigQuery Overview on Streaming (Courtesy of [8])*

Firstly, like MonetDB (see below), BigQuery also orients its data in a columnar fashion for similar reasons – in some fields of data analytics, only a subset of the columns may be analyzed, and so it would behoove a DBMS architect to utilize this over row-based storage. Furthermore, BigQuery utilizes this in conjunction with their trees. Accessible records are placed at the leaves of each tree. This, coupled with the previous statement on columnar-based storage, allows for substantial reductions in query latency. Furthermore, with attributes being placed contiguously in memory/storage, it becomes far easier to compress and search the needed records compared to a standard, row-oriented approach, like the efficient storage of attributes/tables in SSDs mentioned during the week of September 4th and September 11th in class.

Secondly, we note that, like most present-day DBMSs, BigQuery takes a few pages from distributed and parallel computing [9]. The authors of [9] describe BigQuery's execution model as follows: "… it costs the same if *x* CPUs work for *y* amount of time or if one CPU works for *x**y* amount of time…"; dynamically selecting the number of nodes/CPUs for a given query is certainly resourceful, though like anything, it comes with ups and downs. The positives of dynamically using resources in a datacenter allows for easier access of resources by other users, even if it means sharing resources on the same server; furthermore, if the resources are disaggregated (read: not all in one box per server), then it makes the allocations even easier (distributed resources).

The authors of [9] go further to explain BigQuery's scalability through a "shared multi-tenant architecture." What this means is that, since BigQuery is also a service, it allows people to get their money's worth by fetching their query in as little time as possible. Since the query could be assigned to any number of cores or resources, slicing it up in a heuristic manner allows it to still move to completion, even when the servers involved are under a heavy load.

Snowflake

Introduction and Motivation:

Snowflake is a data warehouse offered as a service (like BigQuery), with emphasis on elastic scalability and the fact that it is hosted in the cloud [9]. What "elastic scalability" means is that a service or software can scale up/out or down/in as it sees fit. Something like this is extremely useful in cloud computing-based applications and services as it allows for dynamic allocation and freeing of resources to allow other services and software to run on a given system without (ideally) too much resource contention. Like BigQuery, Snowflake uses a multi-cluster/shared data architecture – not only does it decouple resources (CPU, I/O, storage, etc.), but it also becomes a cost-effective measure. Like with each of the RDBMSs here, their documentation provides plenty of room for optimizations, cost management, security, etc. [10,11].

Design and Implementation:

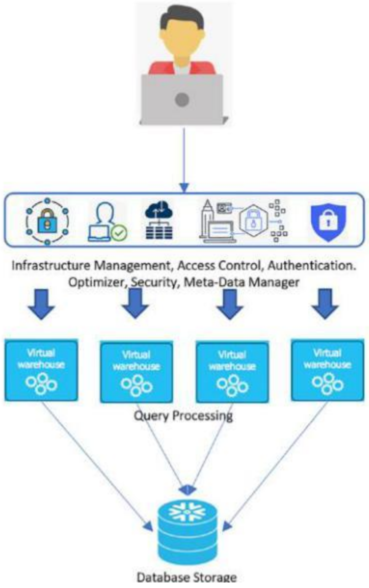


Figure: *Architecture behind Snowflake (Courtesy of [12])*

The above figure shows a high-level view of Snowflake's architecture, and the authors behind [12] further elaborate upon how Snowflake takes in data, its query functionality, etc.

Starting with the architecture itself, Snowflake uses these somewhat disaggregated MPP (Massively Parallel Processing) clusters for both distributed storage AND distributed computing for every query being accessed. This and the previously mentioned capabilities (ideally) will allow for high availability and reliability of the underlying system for end users.

Snowflake already acts upon some form of the notion of “Confidential Computing” – that is, every user’s data is encrypted and secured (though not 100% partitioned, perhaps) after being input into the database. For example, each input is encrypted via 256-bit AES encryption. For some background, AES-256 is currently virtually uncrackable, and this becomes extremely important for data security. The only caveat about automatic encryption is the need to decrypt it later. In general, encryption and decryption are somewhat expensive tasks, though it seems that it gets hidden in some way when querying data via Snowflake.

Being a service, Snowflake needs to be flexible. Its base is routed in being a relational DBMS – structured tables, security, ACID properties, etc. – it also offers the ability to have semi-structured data [10]. Given the rigidity of relational databases, data structured in formats like JSON and XML can also be stored efficiently with their own schema, though this comes with the drawbacks of lessened flexibility and even data loss. Even with dynamic tables being brought up as of recent [11, software release made in June 2023 according to Snowflake’s website].

To continue Snowflake’s flexibility, it can also become integrated with other storage/cloud/DBMS services. The figure below mentions a small subset that includes PowerBI, Azure, and Databricks (another of the DBMS’ reported on)[12].

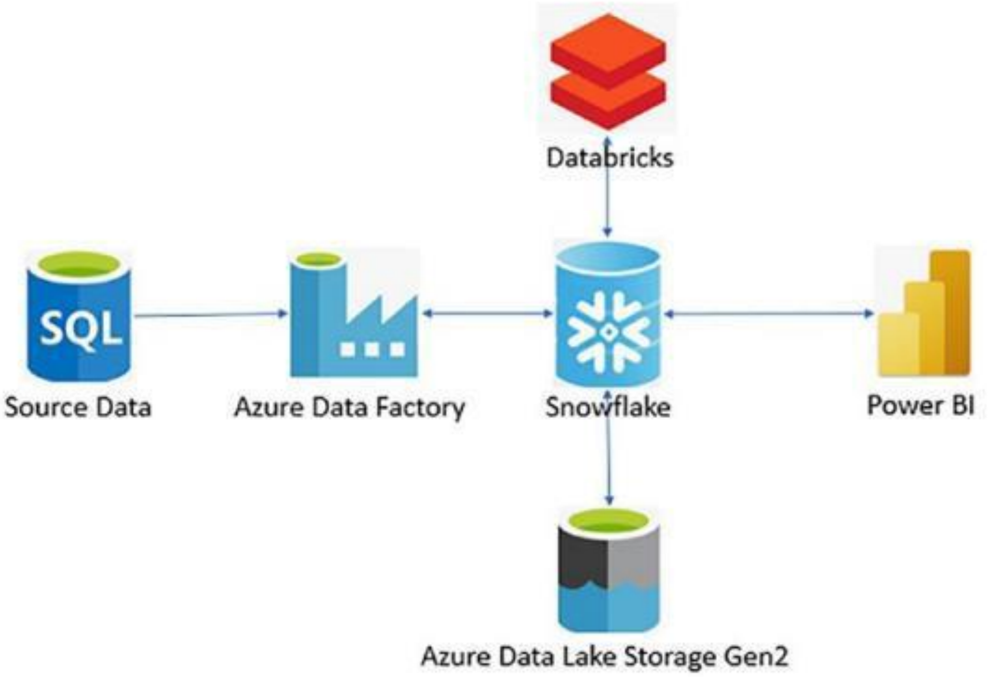


Figure: *Snowflake’s integration with PowerBI, Databricks, Azure’s Data Factory, and ADLS Gen2 (Courtesy of [12])*

MonetDB

Introduction and Motivation:

MonetDB is an open-source DBMS designed for numerous high-performance applications across a wide variety of fields from data mining and business intelligence to XML Queries and multimedia retrieval [5]. It is a relational DBMS (RDBMS), meaning that it comes with all the inherent pros and cons that appear with one: enhanced security and integrity at the cost of scalability and (generally) the ability to execute queries in parallel, among others. MonetDB was developed in 1993 with large databases in mind. Their approach and design, which we will explain next, has certainly worked – as of 2012, over ten thousand monthly downloads of MonetDB took place. Their website [1] mentions its substantial accolades and numbers to showcase its popularity and reach in the database community. It also shows some of its more technical features for developers. and MonetDB adheres both to more recent SQL standards as well as being back portable to a wide range from SQL:1999 to SQL:2011 [1].

Like other libraries and software in other communities, research and development is poured into MonetDB, such as [6], where members of the MonetDB team redesigned their database architecture to exploit modern hardware and avoid the "memory wall" – that is, how to avoid the bottleneck surrounding accessing main memory.

Design and implementation:

[5] Explains most of the design behind MonetDB as of 2012, and further work and documentation will explain elsewhere through MonetDB's website [1]. We will start with the physical data model as explained by [5]. MonetDB uses columnal stores for their records. While this normally brings more overhead than a traditional row-store approach, they use what they call Binary Association Tables (BAT) for their keys and attributes – in other words, BATs are the basic building block for MonetDB. All keys are in the leftmost column acting as a "key" if it were a key-value store. The reasoning behind this approach allows it to behave like typed C-based arrays. This, coupled with dictionary-like encodings and late-tuple reconstruction to help optimize computation with columnar-storage, allows MonetDB to take advantage of a CPU's cache hierarchy and vector-like operators. All of this is impressive, though it wasn't until more recently that they were able to incorporate multi-core/multi-thread parallelism, which normally isn't possible with traditional RDBMSs. One caveat that we can observe from this is potential memory footprints for persistent storage.

Moving onto its execution model: MonetDB's execution kernel is programmed in its own "assembly" entitled "MonetDB Assembly Language" (MAL), which is built on top of the relational algebra operations issued to the BATs mentioned earlier. Its "operator-at-a-time" approach to queries allow each operation the be evaluated to completion following ACID properties across the entire dataset before specific operations are issued such as specific selections, cascading updates, and other manipulations. This also allows more complex operations (e.g., nested selections and projections) to be evaluated in a stepwise fashion efficiently. BAT operations get transformed internally to array operations, and with modern day compiler optimizations – vectorization, loop unrolling if used in the case of iterative/nested operations, and parallelism if enough internal operations are independent of each other.

MonetDB relies on “hardware-conscious algorithms” [5] – that is, these tend to be cache-aware and are fine-tuned to avoid any sort of cache misses (such as what’s presented in partitioned hash-joined and radix-cluster algorithms [6]). This type of approach leaves very few spots for degradations in algorithm thanks to the use of cost models, though having to potentially re-tune a library for any new system – like tuning an MPI library – tends to be a nontrivial, time-consuming task. Another enhancement mentioned in [5] is thanks to what they call adaptive indexing and database cracking. At a high level, this allows MonetDB to be used in scientific databases and other workloads where there is not much knowledge given ahead of application execution or data storage time.

In-depth, database cracking performs a partial sort of the data. Given the columnar nature of MonetDB, it avoids the hassle that comes with performing a full sort on a column-store RDBMS, but it also *exploits* that feature using incremental indexing. This further helps adapt indexes to the given workload to avoid the memory wall mentioned previously.

To elaborate on the algorithms as explained by [6]:

1. The Radix-Cluster algorithm takes a relation U and breaks it down into H clusters over some number of passes P. The lower B bits (the “radix”). Each cluster H has some randomly accessed regions $H_{x.}$ Keeping this smaller than the number of cache lines and TLB entries for a given system prevents cache/TLB thrashing from taking place.
2. This is used within the partitioned hash-join algorithm – a hashing algorithm that assumes a random-access pattern to the data needed. [6] explains it in brief compared to the radix cluster algorithm but explains it as a scan to enter each tuple of data into some cluster. A naïve partitioned hash-join will potentially cause cache and TLB thrashing to take place, which is why radix-cluster becomes a more memory-efficient alternative.

Further enhancements include the ability to perform compression and decompression as a method of trading disk space for extra CPU-cycles required to decompress the data – though this latter point is common across nearly all compression libraries such as LZ4 and ZFP.

Amazon Redshift

<u>Introduction and Motivation</u>:

Amazon Redshift is a cloud based large scale data warehousing product offered by Amazon Web Services. It has a Postgres compatible querying layer which enables it to connect to most SQL based tools used extensively in business intelligence applications. It also encompasses Redshift Spectrum which allows users to run SQL queries against Amazon S3 data. Redshift is a fully managed platform, which means there is no administrative overhead in the form of provisioning compute capacity, monitoring, logging, creating backups and performing upgrades.

Redshift was created to store, retrieve and perform complex queries on large datasets and deliver high performance analytical operations in a cost-effective manner. This is achieved through Redshift’s concurrency scaling which automatically provision the available clusters when there is a spike in the concurrent workload and decreases the cluster capacity when the demand subsides.

Implementation:

Amazon Redshift's architecture has evolved radically since 2013. The current architecture is a truly distributed system with several disaggregated components specialized for compute, query processing for large scale data operations, persistent storage etc.

Redshift Compute Cluster:

Redshift compute cluster comprises of two types of nodes:

1. Leader Node - The leader node is the entry point to the Redshift cluster. It facilitates connections by using JDBC or other data APIs. The leader node performs crucial operations like query formation, query planning and optimization, process scheduling among the compute nodes. The incoming request over the API is parsed, tallied with a statistics catalog with the help of which it is logically rewritten and cost optimization is performed on it.
2. Compute Nodes - These nodes store data and execute queries. Compute nodes are assigned data partitions from the Redshift Managed Storage. A distributed query execution plan furnished after the cost optimization from the leader node generates a C++ code (*shown query1.cpp in Figure*) for the query. The C++ executable object is run against the compute node's data slices and it is processed rapidly.

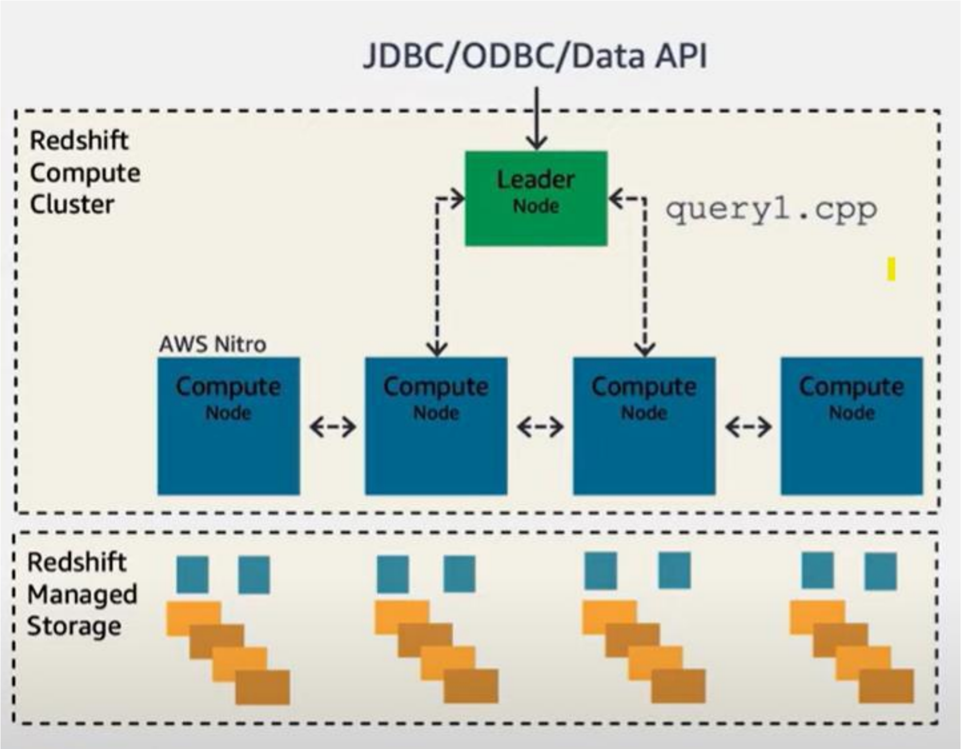
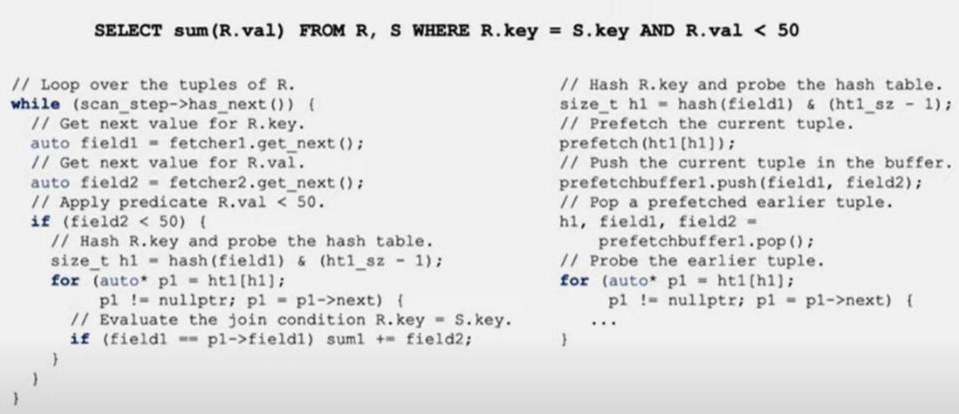


Figure: *Redshift's cluster comprises of leader node which the user interfaces with. The leader node schedules jobs among the compute nodes.* [32]

Redshift is a purely columnar database. The data slices are optimized using techniques like min/max pruning, vectorized scanning and encoding of data to get the most performance gains possible. It uses cache line data prefetching to eliminate data access latency. A L1 Cache Buffer is queried for a few lookahead tuples instead of fetching tuples on demand. These optimizations help incrementally in gaining a 350% increase in the Query Per Hour execution rate over the span of a few months.



```
SELECT sum(R.val) FROM R, S WHERE R.key = S.key AND R.val < 50
```

```
// Loop over the tuples of R.
while (scan_step->has_next()) {
  // Get next value for R.key.
  auto field1 = fetcher1.get_next();
  // Get next value for R.val.
  auto field2 = fetcher2.get_next();
  // Apply predicate R.val < 50.
  if (field2 < 50) {
    // Hash R.key and probe the hash table.
    size_t h1 = hash(field1) & (ht1_sz - 1);
    for (auto* p1 = ht1[h1];
         p1 != nullptr; p1 = p1->next) {
      // Evaluate the join condition R.key = S.key.
      if (field1 == p1->field1) sum1 += field2;
    }
  }
}
```

```
// Hash R.key and probe the hash table.
size_t h1 = hash(field1) & (ht1_sz - 1);
// Prefetch the current tuple.
prefetch(ht1[h1]);
// Push the current tuple in the buffer.
prefetchbuffer1.push(field1, field2);
// Pop a prefetched earlier tuple.
h1, field1, field2 =
    prefetchbuffer1.pop();
// Probe the earlier tuple.
for (auto* p1 = ht1[h1];
     p1 != nullptr; p1 = p1->next) {
  ...
}
```

Figure: *C++ code for query processing prepared by the Leader node and cached. Optimizations like `prefetch' help reduce access latency overheads by fetching lookahead data into the `prefetch buffer'.* [32]

<u>Global Cache</u>:

Generating the C++ query code and compiling it for every query that is issued to the leader node can be a time-consuming process. In earlier versions of Redshift, to mitigate the latency code compilation, a cache was maintained by the leader node which stored the query signatures of the past queries and their associated C++ code executables. This caching scheme gave a cache hit rate of about 99.5%, meaning that only 50 out of 10,000 queries needed code compilation.

To cater to applications demanding extremely low latency SLAs, a global cache scheme was built into the Redshift. This scheme uses a compilation module for any unseen query and caches its corresponding compiled executable in a global cache which can be accessed by all the Redshift clusters. This simple change of disaggregating the cache from individual clusters increased the cache hit ratio by an order of magnitude, to over 99.96%, meaning only 4 queries out of 10,000 queries needed compilation. The underlying idea here was the “Pigeonhole principle”, by letting all the clusters access the global cache, the caching could exploit common query fragments processed on different clusters altogether.

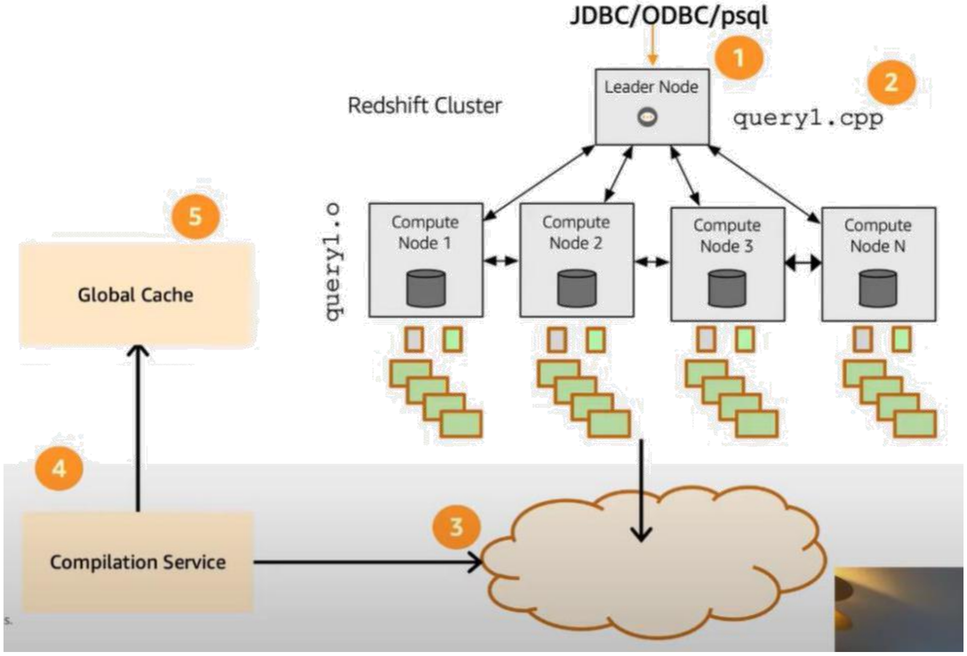


Figure: *Global cache to store executables from all clusters in a unified manner to increase cache hit ratio.* [32]

<u>Advanced Query Accelerator</u>:

Advanced SSD based storage performance has overpassed the CPU performance and as data warehouses continue to grow the limiting factor in performance turns out to be CPU's bandwidth. Redshift pushes computation close to data warehousing in terms of locality. One of the ways it achieves this is by integrating the Advanced Query Accelerator(or, AQUA for short) component. AQUA provides a distributed and hardware accelerated cache that simulates a compute-at-the-storage layer for Amazon Redshift. Under the hood it uses AWS designed analytics processors that dramatically accelerate data compression, encryption and data processing on queries that scan, filter and aggregate large data slices. This integration is available with the ra3.16xl and ra3.4xl compute nodes and does not require a code change to execute operations.

Instead of plainly caching and fetching the data from remote persistent storage like Amazon S3 onto the compute node, the AQUA layer receives the data reducing part of the queries like filters and aggregations and executes them. This approach works especially well as data scans are parallelizable and data reduction operations like filters and aggregations remove the network bottleneck as only the results need to be sent back to the cluster instead of the accompanying data.

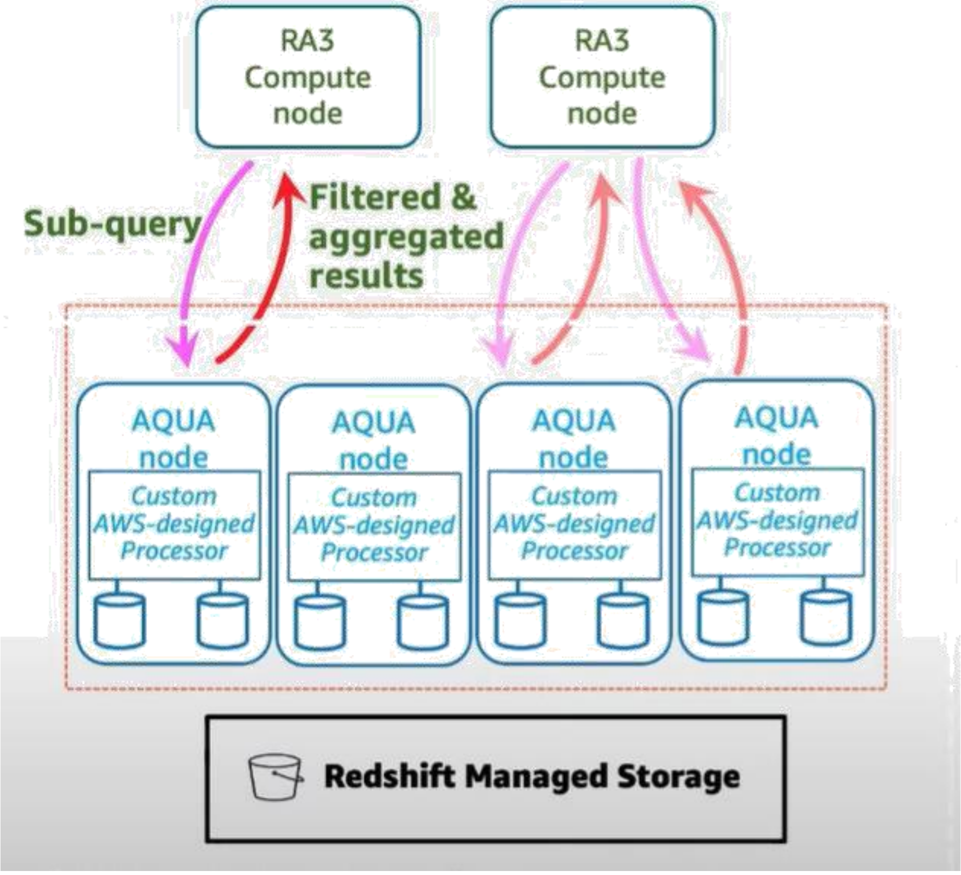

RA3 Compute node
RA3 Compute node
Sub-query
Filtered & aggregated results
AQUA node
Custom AWS-designed Processor
AQUA node
Custom AWS-designed Processor
AQUA node
Custom AWS-designed Processor
AQUA node
Custom AWS-designed Processor
Redshift Managed Storage

Figure: *Overview of AQUA nodes in retrieval from Redshift Managed Store. AQUA nodes perform the necessary computations close to the persistent data and boost data processing and retrieval.* [32]

Cross AZ Cluster Recovery:

Redshift clusters are easily ported between availability zones. This has the benefit of avoiding throttling due to resource constraints on the current cluster, especially during cluster resize operations. The cluster relocation is seamless and non-lossy, with the Recovery Point Objective (RPO) index at 0, meaning no data is lost in the interval of time during which the cluster is being relocated to a different zone. The cluster migration across zones is carried out on-demand and so no additional cost is incurred due to a standby replica of the same cluster. This recovery scheme thereby increases cluster availability and considerably increases the fault tolerance of the system in a seamless and expeditious manner.

Compute Elasticity:

Redshift's compute elasticity is crucial in managing the cost-to-performance tradeoff. It is responsible for requisitioning of compute clusters so as to meet the latency SLA's during times of bursts in data querying and processing activity and removal of clusters when the query load is low in order to save cost.

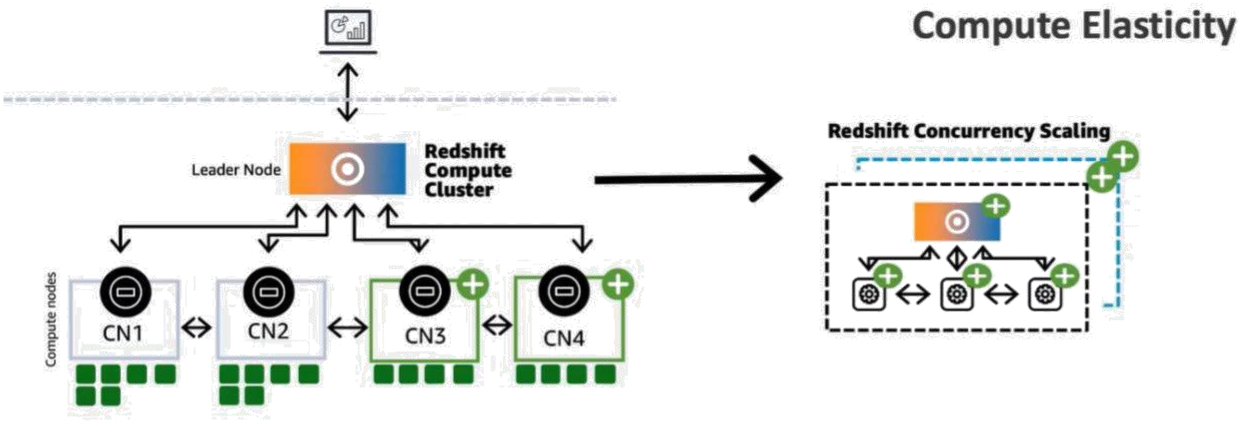


Figure: *Compute elasticity helps in resource allocation by requisitioning compute clusters under high query processing loads and releasing them during low workloads.* [32]

As processes running on a compute cluster start approaching capacity, Redshift acquires an equal sized cluster (could be from a different cluster zone) and concurrently scales the process across the newly acquired and previous clusters. Now the new cluster is able to acquire data from and persist data to the Redshift Management Storage and it is released when the activity load goes down, before which the sparse processes running on the cluster are reverted to the previous cluster. Compute elasticity improves scalability of the system by increasing Redshift's throughput almost linearly.

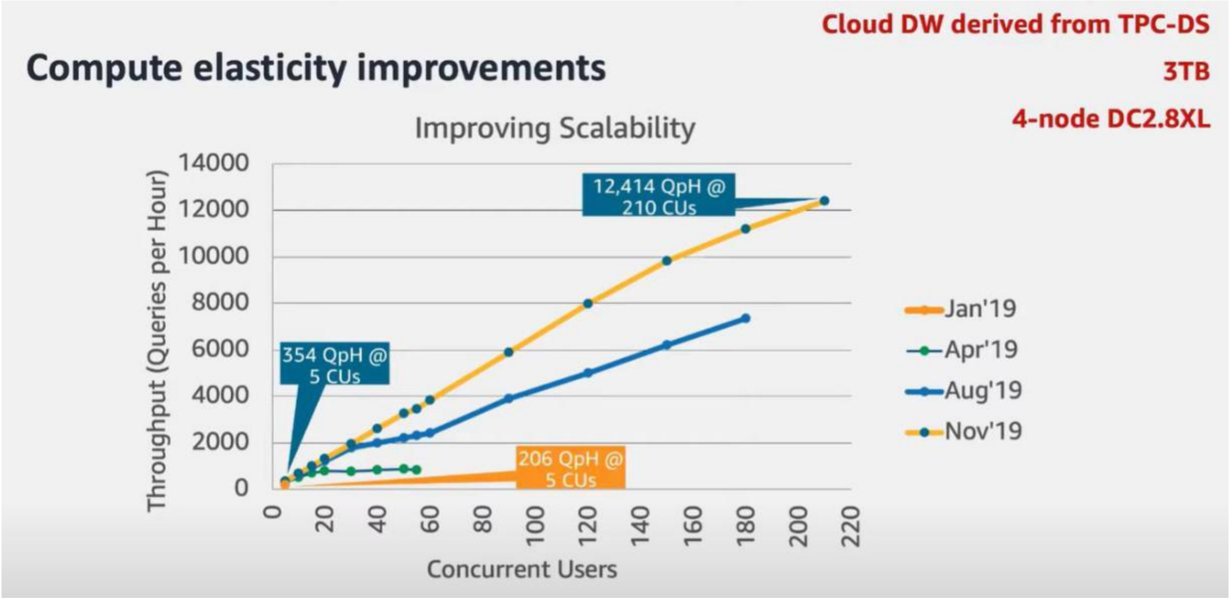


Figure: *Performance boost achieved by compute elasticity in the form of higher throughput.* [32]

DataBricks

Databricks is a unified data processing and analytics platform built on top of Apache Spark. It is a ubiquitous tool that unifies data science solutions across engineering and business. Databricks runtime is powered by cloud-based Spark clusters; it offers developer support, data security and compliance, and integrated workspace to run production jobs and workflow automation. [29]

Apache Spark is designed to handle large scale real-time streaming data workloads. It supports a wide range of integration compliance with other frameworks and a variety of workloads such as real-time analytics, graph processing, machine learning. It also supports multitude of programming languages like Java, Python, Scala, R, SQL thereby making it widely usable. Apart from the range of frameworks it supports, a characteristic feature of Spark is its in-memory optimization for large workloads that offers it an impressive speed-up over other big data analytic applications.

Apache Spark is disruptive in the way it breaks down workload-specific processing specialization by offering a unified stack of components that address diverse workloads under a single fast, distributed engine. There are four main components of Spark:

1. Spark SQL: This module is ideal for processing of structured data stored in relational database formats such as RDBM tables or file formats such as JSON, CSV, Parquet etc. After data ingestion, SQL-like queries can be run to create the native *Spark DataFrame.*
2. Spark MLlib: Spark contains an in-built library of commonly used machine learning algorithms. These algorithms are tailor made to be compatible with Spark DataFrames.
3. Real Time Streaming: Spark supports computations on streaming data from sources like Kafka, Kinesis, HDFS. The real time data is viewed as new rows of data appended to data source that can be queried by the Spark SQL module like queries to a static table.
4. GraphX: Spark is built with graph processing techniques useful in manipulating graphs like social networks, network topology graphs etc.

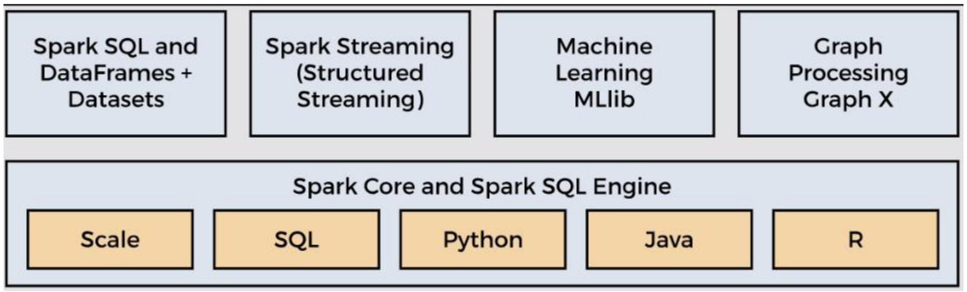

Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scale
SQL
Python
Java
R

Figure: *The Spark stack* [29]

<u>Design and Implementation</u>

Spark has a leader-worker architecture, similar to MapReduce framework. The leader process (driver) distributes and coordinates the workload among worker processes (executors). The driver manages the execution of a Spark job. It is responsible for maintaining the global state of the Spark application which comprises taking user inputs and distributing and scheduling jobs among executors.

The driver converts Spark operations into DAG computations and distributes and schedules the computation tasks across the executors. The driver uses a single point of entry called *SparkSession* to access the distributed runtimes on clusters. Through the SparkSession, data can be read from the underlying sources, written to DataFrames or Datasets.

The executors, each, work on a subset of the data located closest to them in the cluster. The data proximity consideration complies with the notion of *data locality* and helps reduce the consumption of network bandwidth.

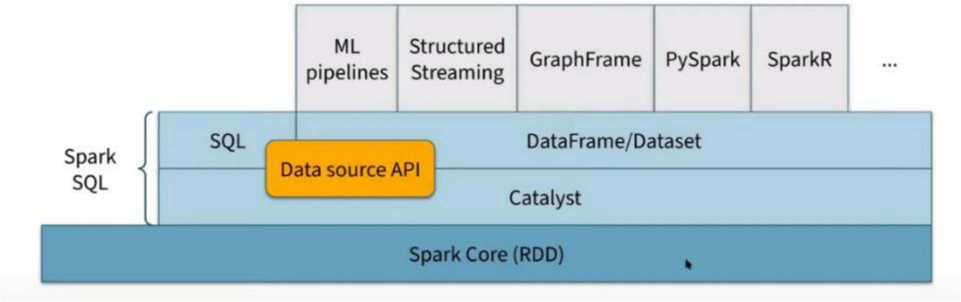


Figure: *The Spark SQL layer with Catalyst query optimizer* [29]

<u>Catalyst</u>:

Spark SQL is one of the major components of Apache Spark, useful for working with structured data workloads. As mentioned earlier, it powers both SQL queries and DataFrame API. At the core of Spark SQL is the Catalyst optimizer which leverages functional programming paradigms of Scala to build an efficient query optimizer. Catalyst is made up of a tree composed of Spark node objects. Each Spark node has a type and child nodes. Spark nodes are immutable objects that undergo functional transformations which form the basis of optimizations to Spark's data processing pipelines. Spark uses Scala's pattern matching feature to transform trees by replacing subtrees with specific structures. Catalyst offers the capability to extract values from nested structures recursively, which is the case when the pattern matches with a subset of all possible input trees and applying transformations to them. Catalyst groups the transformation rules into batches and executes each batch of rules until it reaches a fixed point, which is when a tree stops changing after applying rules to it.

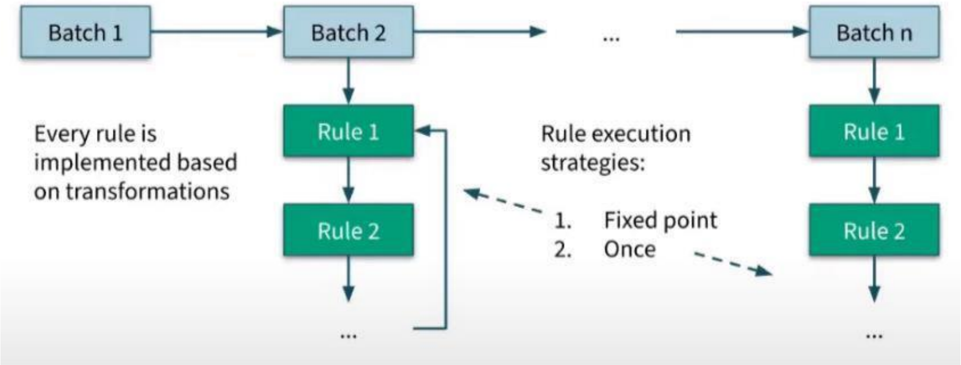

Batch 1
Batch 2
...
Batch n
Every rule is implemented based on transformations
Rule 1
Rule 2
...
Rule execution strategies:
1. Fixed point
2. Once
Rule 1
Rule 2
...

Figure: *Transformations applied to trees by repeatedly applying multiple rules in different batches.* [29]

Functional transformations on immutable trees offer the optimizer explainability and offer more opportunities to parallelize the optimizer. Below is a figure depicting the different stages of Spark's Catalyst optimizer.

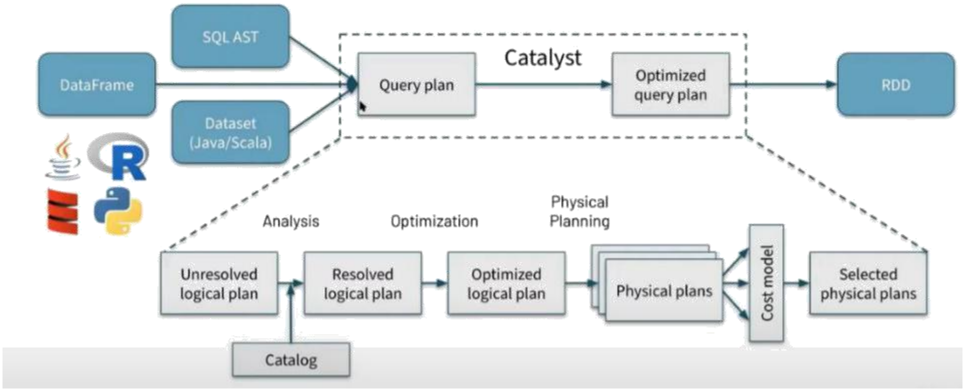


Figure: *Catalyst query optimizer's stages* [29]

1) Analysis: This is the first stage of the Catalyzer wherein an unresolved logical plan tree is constructed from relations that are either computed from a DataFrame object or from an abstract syntax tree(AST) returned by the SQL parser. It contains unresolved attributes, that is, attributes for which the type is not known or attributes which have not been matched to an input table.

2) Logical Optimizations: In this stage rule-based optimizations are applied to the logical plan. These are cost based optimizations like constant folding, predicate pushdown, null propagation, projection pruning among others, which result in multiple plans with their respective costs.

3) Physical Planning: In this phase, Spark SQL takes a logical plan and generates one or more physical plans. These plans are made to match the Spark executor engine. A physical plan is selected using a cost model. At this point, cost optimizations are applied to select and join operations. Some rule-based physical optimizations, such as pipelining projection or filter application in a Spark map operation, are also performed in the physical planning phase.

4) Code Generation: This is the final stage of query optimization. It involves generating Java bytecode to be run on the Spark nodes. Spark performs operations on in-memory datasets. Since this processing is performed on CPUs, code generation engines would need a compiler which leads to a drop in execution speed. To simplify code generation, Catalyst uses a Scala feature called quasiquotes which allows programmatic construction of abstract syntax tree(AST) nodes that can be fed to the Scala compiler at runtime to generate bytecode. This essentially saves the compilation time to get bytecode.

Adaptive Query Execution

Spark uses cost-based optimization to choose the most optimal query plan but it presents several challenges like stale or missing data statistics leading to inaccurate estimates, costly statistic collection eg. Column histograms. Adaptive query optimization is dynamic in nature and happens during the query execution based on runtime statistics.

Spark is a distributed computation system which works on the Spark SQL distributed database. Spark is optimized to execute computations parallelly while also facilitating data `exchanges' that occur from one node to another/all other nodes. The data exchanges are typically shuffle(on node) or broadcast(all nodes) operations that are scheduled between query stages. The operations inside a query stage such as scan, aggregate etc. are executed in parallel. These intermediate points between query stages are referred to as `pipeline break points'. Pipeline break points are optimal for adaptive query execution as they are inherent breakpoints for operational pipelines, which means at these points, operations across different query stages can anyway not be pipelined. Another feature of pipeline break points is that an accurate runtime statistic can be generated at these points because Spark writes intermediate results to disk during these pipeline breaks.

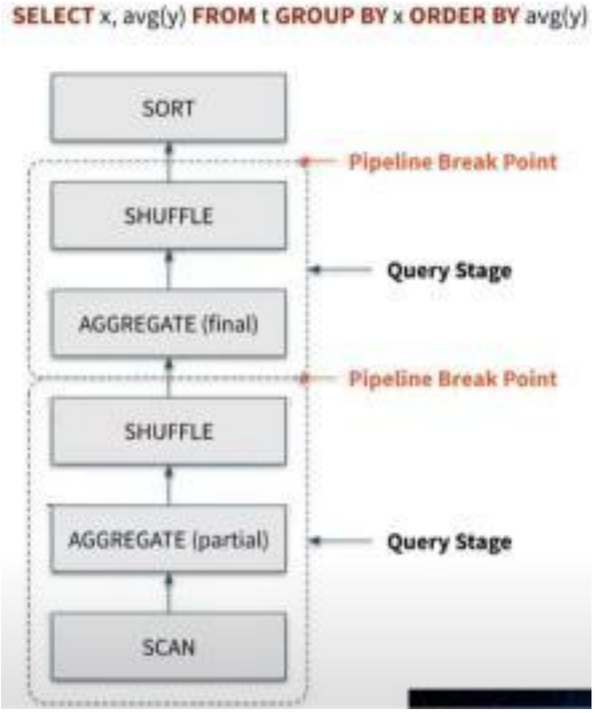


Figure: *Pipeline break points between Spark's query stages.* [29]

Adaptive Query Execution works iteratively in stages. It first runs on leaf stages, which don't depend on the execution result of other stages. On the completion of a stage, the workflow gathers accurate statistics which are used in query optimization.

During optimization query stages are shuffled as per the logical plan tree and the leaf nodes of the tree are executed first. These leaf-node query stages are not dependent on the result of a different query stage. Once these stages are complete, more statistics are available for optimization. Next, query stages dependent on the result previously finished stages will run until there are no more unexecuted query stages.

Google Napa

Introduction and Motivation

Napa replaced Mesa, an earlier Google database system, to support Google's data warehousing for its advertising and payment clientele. There is a vast range of different business firms that utilize Google's payment and advertising services, so Napa was designed to be flexible to meet different customer needs and requirements around cost, performance, and data freshness. In addition, Napa supports fast query performance that is not variable. Napa provides fast query performance to its customers by maintaining the materialized views consistently as new data is received from multiple data centers [20].

Design and Implementation

Napa was designed to be flexible to fit different customer needs. Their approach identifies three design tradeoffs that can be tuned to meet customer specifications: data freshness, resource costs, and query performance (i.e., query latency). Fast query performance requires the generation of index and material views before the query operation is performance which bundles storage and ingestion together. This approach can adversely affect data freshness since slow ingestion prevents data from being refreshed. If data freshness is the priority, query and storage operations can be bundled together.

Napa's architecture can be deployed across multiple datacenters to handle replication. The diagram below illustrates the two main components of Napa's architecture: the control plane and the data plane. The data plane handles ingestion, storage, and query serving. The control plane coordinates and synchronizes the metadata transactions across the multiple data centers. The second figure in this section details the Queryable Timestamp (QT) which decouples query performance from storage performance. The QT allows Napa to provide low latency queries to provide fresh data to users. Napa maintains each data center's QT and the system's global minimum QT to provide an up-to-date latency performance metric [19, 20].

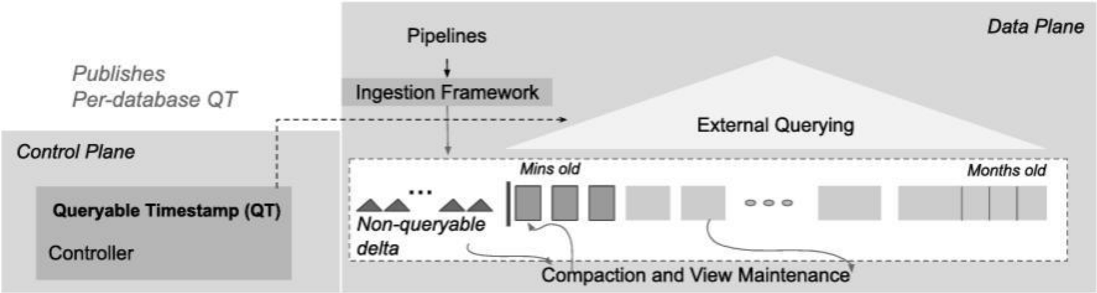


Figure 3: Napa architecture showing the major system components.

.

Figure: *Napa architecture showing major system components [19]*

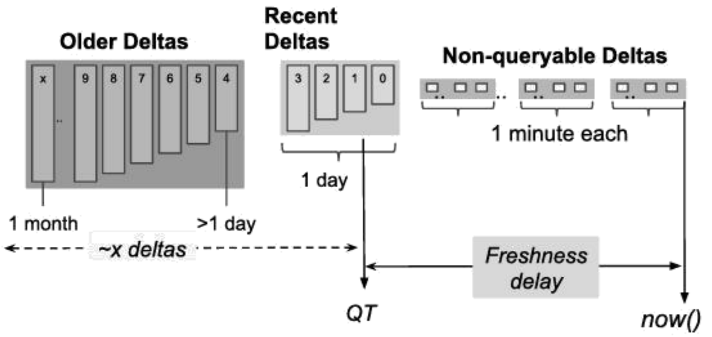


Figure 5: Queryable timestamp decouples query performance from storage performance.

Figure: *Queryable timestamp decouples query and storage performance [19]*

Challenges

Napa is designed by Google to provide a powerful database system to a wide variety of clientele. Users can choose which metrics to optimize – data freshness, cost, and query performance. The need to provide a one-size-fits-all may prevent Napa from optimizing performance for specific use-cases. For example, an application that does not have a strict latency constraint would not benefit from the QT aspect of the control plane since decoupling query and storage performance would not advance that use-case objectives.

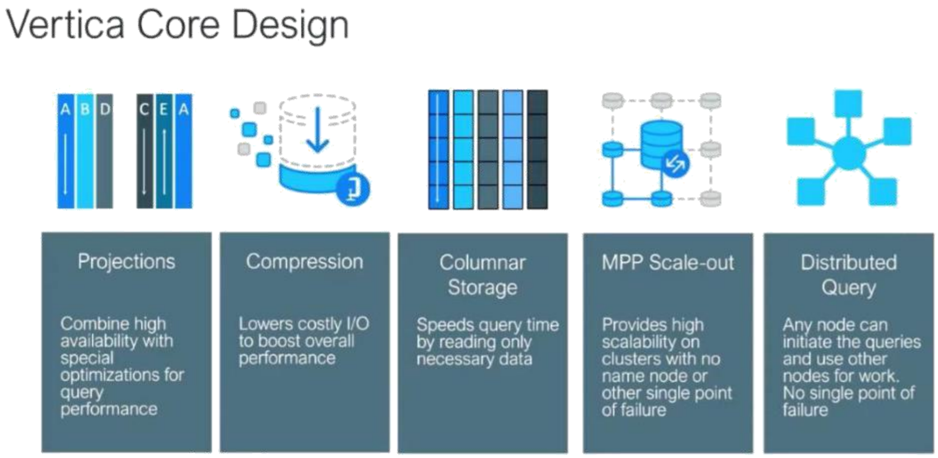
Vertica

Introduction and Motivation

Vertica aims to provide high-performance over varying domain – both in terms of data scale and deployment. Vertica is a distributed SQL Analytics database and caters to industry customers who are looking for flexibility and fast analytical performance [16, 17].

Design and Implementation

There are five main pillars of Vertica's design, as illustrated below. The first pillar is projections, which combine high data availability with special optimizations for query performance. Query performance optimization is important to Vertica's industry customers who need to quickly query their database to conduct data analysis of sales trends, inventory availability, and other measures critical to managing their business. The second pillar is compression. This lowers the amount of input and output operations (I/O). I/O operations are costly on the system as they slow down the overall performance. The third pillar is columnar storage, which also speeds query time by only reading the data that is required to complete the query instead of scanning the entire row. The fourth pillar is massively parallel processing (MPP) scale-out. This feature allows the database to scale to meet customer's needs and does require a name node so there is no single point of failure across the database system. Finally, the fifth pillar is distributed query capability across all nodes. This feature allows any node in the system to begin queries, or offload work to another node. This also provides a safeguard against a single point of failure [16, 17].



Projections

Combine high availability with special optimizations for query performance

Compression

Lowers costly I/O to boost overall performance

Columnar Storage

Speeds query time by reading only necessary data

MPP Scale-out

Provides high scalability on clusters with no name node or other single point of failure

Distributed Query

Any node can initiate the queries and use other nodes for work. No single point of failure

Figure: *Vertica Core Design [16]*

Challenges

Vertica originated as the C-Store database, which was published by researchers in 2005. The original work was published in 2005 which proposed a columnar-storage approach to optimize data queries [17]. This approach was very effective at the time, and outperformed traditional relational database systems. However, it is not clear that the underlying architecture can keep up with new innovative approaches, such as NoisePage's autonomous approach to database system management.

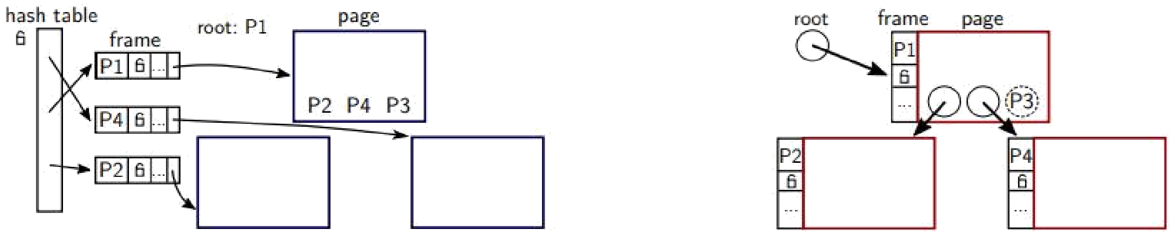
LeanStore

Introduction and Motivation

Leanstore, compared to the other RDBMSs on this list, describes themselves as a "high-performance OLTP storage engine" designed and developed by a team at TU München in Germany [24]. Their aim is to optimize transactions for the latest and greatest many/multi-core CPUs and upcoming non-volatile memory. Like MonetDB and NoisePage, research and development innovations are funneled into LeanStore before getting released into some sort of major/minor version update. Their seminal paper [23] is the first paper mentioning its design and implementation. Their website currently explains that LeanStore is largely a research prototype, with the end goal of it being able to be deployed. It initially held some resemblance to MonetDB with restrictions on concurrency (Optimistic Concurrency Control), though design and implementation from [23] will be discussed below.

Design and Implementation

The first item in Section III of [23] is the notion of "Pointer Swizzling." What this involves is using pointer-tagging to determine whether a page is resident in memory or in background storage. This allows a single conditional state to determine a page's location, and turns out to be very efficient, as comparison can be optimized down to a CPU's architecture, as shown in the figure taken from [23] below:



(a) traditional buffer manager

(b) swizzling-based buffer manager

Fig. 2. Tree structure consisting of a root page (P1) and three leaf pages (P2, P4, P3), two of which are in RAM (P2 and P4).

The second point of order in LeanStore's initial design is the notion of a more efficient page-replacement policy. Their process is akin to branch prediction and speculative execution at the architectural level: a page will speculatively get unswizzled, and if it's a page that's been frequently accessed, then it quickly gets re-swizzled to avoid incurring further disk I/O, as shown by Figure 3 from [23]. This works in conjunction with FIFO queues to efficiently move pages either from their "Cold" to "Hot" stage, or more importantly, "Hot" to "Cooling".

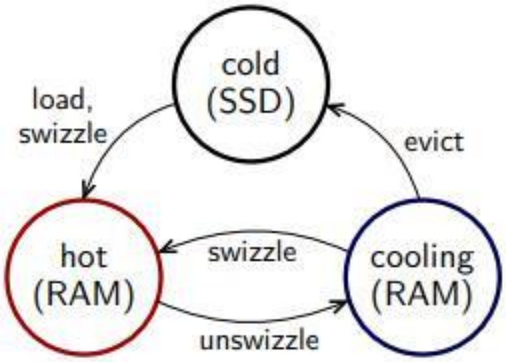


Fig. 3. The possible states of a page.

The third point of order is the ability to synchronize at scale. Use of timestamps and page pinning/unpinning allow for better scalability than what could be allowed by standard "hot" latches and cache coherency on modern multi-core CPUs.

Heading into their data-structure designs, LeanStore utilizes advanced schemes to aid in the previous aspects mentioned. Their base level involves a hashtable pointing to different buffer frames – according to their initial paper, LeanStore mentions that these frames are used to house three primarily bits of metadata: a memory pointer for tagging/un/swizzling, a state (e.g. LRU, Second Chance, etc.), and information regarding the page's state on disk. Assuming this structure remains lightweight, this allows for easy management of page-to-pointer mapping, replacement in memory/on-disk, etc.

Part II: Comparisons

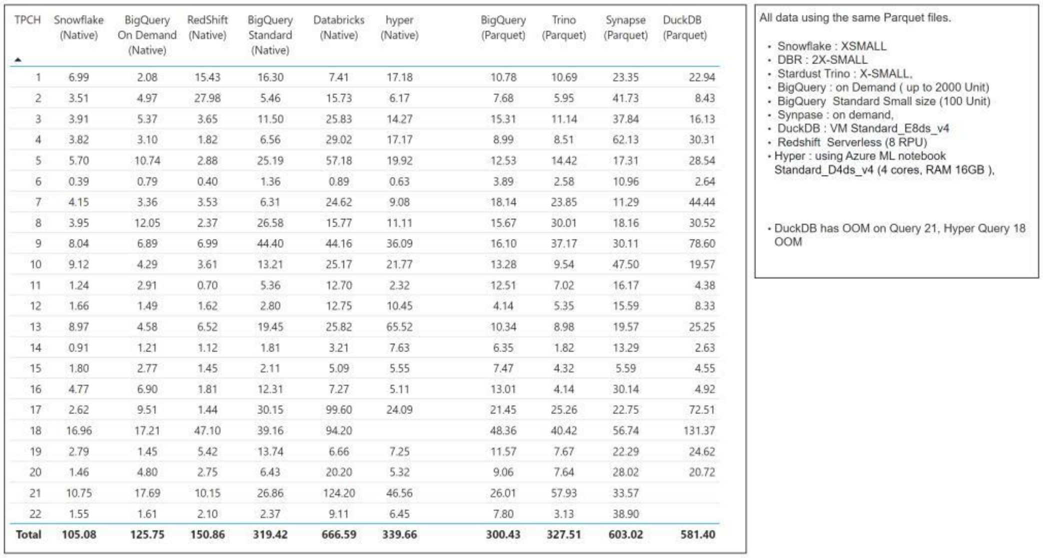
In this section of the report, we conclude with a comparative evaluation of the systems based on the technological innovations they offer. We provide a cost analysis of the systems (where data was available), a qualitative analysis of the intended use cases, benefits, and disadvantages for each system design. Finally, we provide a recommendation, based on the cost analysis, benefits, and maturity of the technology.

Cost Analysis

[13] Provides a price and performance comparison of Snowflake, Databricks, BigQuery, and Redshift among others, but we focus on the four here. Using TPCH-SF100 – a base table of some 600 million rows – the author does this as a response to a comment someone mentioned to them about Snowflake allegedly cheating with how fast a prior experiment ran on a smaller dataset.

Using an external table variant and approximately 535 GB of data, BigQuery was able to process this in 5:18 (five minutes, eighteen seconds). Using an internal format and apparent caching, the author was reportedly able to store 470.3 GB in 2:16 without any warmup of sorts, and this one wasn't charged, but the first run was at a grand total of $2.50. The bottom chart on the article shows the price per hour of these databases with a few noteworthy points: BigQuery does not charge for data loading, and the prices mentioned for Snowflake, RedShift, and Databricks do NOT include a cost for data loading – apparently this gets charged separately.

The figure below shows the runtime comparison in terms of time to completion for TPCH's separate 22 benchmark tests.



TPCH	Snowflake (Native)	BigQuery On Demand (Native)	RedShift (Native)	BigQuery Standard (Native)	Databricks (Native)	hyper (Native)	BigQuery (Parquet)	Trino (Parquet)	Synapse (Parquet)	DuckDB (Parquet)
1	6.99	2.08	15.43	16.30	7.41	17.18	10.78	10.69	23.35	22.94
2	3.51	4.97	27.98	5.46	15.73	6.17	7.68	5.95	41.73	8.43
3	3.91	5.37	3.65	11.50	25.83	14.27	15.31	11.14	37.84	16.13
4	3.82	3.10	1.82	6.56	29.02	17.17	8.99	8.51	62.13	30.31
5	5.70	10.74	2.88	25.19	57.18	19.92	12.53	14.42	17.31	28.54
6	0.39	0.79	0.40	1.36	0.89	0.63	3.89	2.58	10.96	2.64
7	4.15	3.36	3.53	6.31	24.62	9.08	18.14	23.85	11.29	44.44
8	3.95	12.05	2.37	26.58	15.77	11.11	15.67	30.01	18.16	30.52
9	8.04	6.89	6.99	44.40	44.16	36.09	16.10	37.17	30.11	78.60
10	9.12	4.29	3.61	13.21	25.17	21.77	13.28	9.54	47.50	19.57
11	1.24	2.91	0.70	5.36	12.70	2.32	12.51	7.02	16.17	4.38
12	1.66	1.49	1.62	2.80	12.75	10.45	4.14	5.35	15.59	8.33
13	8.97	4.58	6.52	19.45	25.82	65.52	10.34	8.98	19.57	25.25
14	0.91	1.21	1.12	1.81	3.21	7.63	6.35	1.82	13.29	2.63
15	1.80	2.77	1.45	2.11	5.09	5.55	7.47	4.32	5.59	4.55
16	4.77	6.90	1.81	12.31	7.27	5.11	13.01	4.14	30.14	4.92
17	2.62	9.51	1.44	30.15	99.60	24.09	21.45	25.26	22.75	72.51
18	16.96	17.21	47.10	39.16	94.20		48.36	40.42	56.74	131.37
19	2.79	1.45	5.42	13.74	6.66	7.25	11.57	7.67	22.29	24.62
20	1.46	4.80	2.75	6.43	20.20	5.32	9.06	7.64	28.02	20.72
21	10.75	17.69	10.15	26.86	124.20	46.56	26.01	57.93	33.57	
22	1.55	1.61	2.10	2.37	9.11	6.45	7.80	3.13	38.90	
Total	**105.08**	**125.75**	**150.86**	**319.42**	**666.59**	**339.66**	**300.43**	**327.51**	**603.02**	**581.40**

All data using the same Parquet files.

- Snowflake : XSMALL
- DBR : 2X-SMALL
- Stardust Trino : X-SMALL,
- BigQuery : on Demand (up to 2000 Unit)
- BigQuery Standard Small size (100 Unit)
- Synpase : on demand,
- DuckDB : VM Standard_E8ds_v4
- Redshift Serverless (8 RPU)
- Hyper : using Azure ML notebook Standard_D4ds_v4 (4 cores, RAM 16GB),

- DuckDB has OOM on Query 21, Hyper Query 18 OOM

Figure: *Runtime results on the TPCH benchmark across Snowflake, BigQuery, and more (Courtesy of [13]). Numbers are in seconds for each runtime result.*

From this blog alone, Snowflake is a clear winner. Not only does it have the smallest price per unit, but it also executes queries the quickest among it, RedShift, BigQuery (both in on-demand and in "standard" modes), and Databricks.

Comprehensive Qualitative Evaluation

Price comparison data was not available for a portion of the DBMSs analyzed in this report. To bridge this gap, we provide a high-level qualitative comparison of each design, with its intended use-case, strengths and weaknesses.

DBMS	Intended Use-Case	Strengths	Weaknesses
Apache Druid	Online Analytics Processing platform that generates real-time analytics and delivers high speed for query execution on streaming and batch data	Reduced compute and storage costs, connects to multitude of cloud-based data sources, enhanced data security	Only supports Inverted Index, lacks support for joins, difficult to recover if metadata is corrupted
NoisePage	Open-source proof of concept for autonomous DB system management	Leverages cutting-edge AI technology to reduce the burden of DB system management	Not a production system; not fully autonomous
Google BigQuery	Production model for Big Data analytics by Google – customer oriented	Enhanced parallelism for any/all queries; columnar/tree-based storage for quickly extracting information	Time-based limit on concurrency (workaround with a query queue), price-based queries like Redshift
Snowflake	Production-level RDBMS as a “Self-Managed Service.”	Relatively inexpensive compared to other paid services, scalable, and highly parallel compared to some of the others	Rigid given its relational model compared to a non-relational DBMS; Documentation not super-accessible, but helpful for some items.
MonetDB	Open source, relational DB, actively being researched, designed, and developed	Cutting edge vectorization, relational database properties, columnar storage optimized	Lack of parallelism to act according to ACID properties, rigidity
Amazon Redshift	Cloud based data warehouse capable of handling enormous volumes of data and extracting crucial insights from it.	Useful data insights, seamlessly integrates with AWS services, highly parallelized application offering high speeds	Lack of implementation detail in documentation, lacks data visualization tools, accumulative cost can be high
DataBricks	Cloud based large scale data platform that unifies data storage capability with extensive business analytics and machine learning frameworks.	Highly efficient query processing, data streaming capabilities, enhanced data science and data engineering frameworks, highly optimized for cloud environments	Expensive as a large-scale data processing tool, relatively small open-source community and code support, no unit testing
Google Napa	Google’s ad and payment business	Flexible to align with different customer’s needs: cost, query speed, data freshness	One-size-fit-all, not optimized to specific use-case
Vertica	Production DBMS for retailer customers	Optimized for very fast querying for business analysis	Slightly dated technology (columnar storage)
LeanStore	Research-oriented relational DB management system with a focus on lightweight execution and system-level enhancements to avoid unnecessary overhead	System-level enhancements such as pointer-tagging, scalable synchronization, advanced buffer management	Not production-ready – primarily research oriented; scalable, but cannot execute write/update/merge queries in parallel

Recommendations

NoisePage, MonetDB, and LeanStore are all compelling designs, as they are research orientated, and explore cutting-edge database management strategies. NoisePage emphasizes an autonomous approach to database management to reduce the burden of managing complex databases for human database administrators. However, this design is still a work in progress, and is not yet fully autonomous. Similarly, MonetDB and LeanStore are works in progress, and are actively being researched. LeanStore does not explicitly deal with concurrency in the form of threads. Instead, it uses "Optimistic Concurrency Control," dealing primarily with read-only transactions. This approach appears to be very restrictive since not only must queries be executed on separate tables, but they also adhere very strictly to ACID properties in their relational model [23]. The latter involves more of a research theme than the former and has not been put into large-scale production according to their website. NoisePage is the only DBMS that explicitly seeks to solve the end-user's pain-points, that is, reduce the burden of complex database management using AI-driven autonomous system management. This autonomous approach to system design is currently being designed and implemented in other technology areas in addition to database management, including autonomous driving and edge computing technologies, demonstrating the promise of such an approach. If we were a venture capital firm, ***we would choose to invest in this technology if* NoisePage *created a company based on this academic prototype***.

Of the production-ready database management systems, including Apache Druid, Google's BigQuery and Napa, Snowflake, Amazon Redshift, DataBricks, and Vertica, ***we would recommend* Snowflake *for most companies.*** As demonstrated in the Cost Analysis section above, Snowflake provides the smallest price per unit for storage, but it also executes queries the quickest among it, RedShift, BigQuery and Databricks, for both the both in on-demand and in "standard" modes [13]. In addition to being relatively inexpensive compared to other paid DBMS services evaluated here, Snowflake is scalable, and highly parallel compared to some of the others. Snowflake's design allows for high availability and reliability of the underlying system for end users. In addition, Snowflake's design scale up/out or down/in as it sees fit. This approach is useful in cloud computing-based applications and services as it allows for dynamic allocation and freeing of resources to allow other services and software to run on a given system without too much resource contention [11]. Snowflake is compatible with DataBricks, a cloud-based large scale data platform that unifies data storage capability with extensive business analytics and machine learning frameworks [11]. As more companies compete to scale-up their ML-driven analytics capabilities, ensuring their DBMS is compatible with new technologies is vital. Snowflake is also compatible with PowerBI, a popular Microsoft product for interactive data visualization [11]. Snowflake does not provide much detailed documentation on its public website; however, one author has first-hand experience implementing Snowflake on a production system for major nation-wide retailer and found the implementation engineering team provided sufficiently detailed documentation for the DBAs.

References

[1] MonetDB website: https://www.monetdb.org

[2] CMU Database Group Presentation. "MonetDB: Scale Up Before You Scale Out (Martin Kersten)": https://www.youtube.com/watch?v=vjWRE0UnJDQ

[3] CMU Database Group Presentation: “Query Processing in Google BigQuery (Hossein Ahmadi + Aleksandras Surna)”: https://www.youtube.com/watch?v=Zk5_RcRg3nA

[4] CMU Database Group Presentation: “Snowflake Iceberg Tables, Streaming Ingest, and Unistore! (N. Shingte, T. Jones, A. Motivala)”: https://www.youtube.com/watch?v=Kr-Vzvkyabw

[5] S. Idreas, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, “MonetDB: Two Decades of Research in Column-oriented Database Architectures”, Bulletin of the Technical Committee on Data Engineering, 2012

[6] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. Commun. ACM 51, 12 (December 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[7] Google BigQuery Documentation: https://cloud.google.com/bigquery/docs

[8] Sérgio Fernandes and Jorge Bernardino. 2015. What is BigQuery? In Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15). Association for Computing Machinery, New York, NY, USA, 202–203. https://doi.org/10.1145/2790755.2790797

[9] Krishnan, S.P.T., Gonzalez, J.L.U. (2015). Google BigQuery. In: Building Your Next Big Thing with Google Cloud Platform. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-1004-8_10

[10] mParticle: “How does Snowflake work? A simple explanation of the popular data warehouse” - https://www.mparticle.com/blog/how-does-snowflake-work/

[11] Snowflake documentation: https://docs.snowflake.com/

[12] L’Esteve, R. (2022). Snowflake. In: The Azure Data Lakehouse Toolkit. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-8233-5_2

[13] datamonkeysite.com, “Benchmarking Snowflake, Databricks, Synapse, BigQuery, Redshift, Trino, DuckDB, and Hyper using TPCH-SF100:” https://datamonkeysite.com/2023/03/09/benchmarking-snowflake-databricks-synapse-bigquery-and-duckdb-using-tpch-sf100/

[14] Pavlo, Andrew et al. “Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation.” *Proc. VLDB Endow.* 14 (2021): 3211-3221.

[15] 2021. NoisePage - Database Management System Project. https://noise.page.

[16] Walkauskas, Stephen. “Vertica – High Performance Over Varying Terrain.” CMU Vaccination Database (Second Dose) Tech Talk Seminar Series.

[17] Lamb, Andrew, et al. "The vertica analytic database: C-store 7 years later." *arXiv preprint arXiv:1208.4173* (2012).

[18] Ma, Lin. “NoisePage: The Self-Driving Database Management System.” CMU Vaccination Database Tech Talk Seminar Series. (2021).

[19] Agiwal, Ankur, et al. "Napa: Powering scalable data warehousing with robust query performance at Google." (2021).

[20] Jagan Sankaranarayanan and Indrajit Roy. “Google Napa: Powering Scalable Data Warehousing with Robust Query Performance.” Google Napa: Powering Scalable Data Warehousing with Robust Query Performance. CMU Vaccination Database Tech Talk Seminar Series. (2021).

[21] Amazon RedShift documents: Concurrency Scaling: https://aws.amazon.com/redshift/features/concurrency-scaling/

[22] Amazon RedShift pricing (section: Concurrency Scaling): https://aws.amazon.com/redshift/pricing/

[23] V. Leis, M. Haubenschild, A. Kemper and T. Neumann, "LeanStore: In-Memory Data Management beyond Main Memory," 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 185-196, doi: 10.1109/ICDE.2018.00026

[24] Leanstore website and documentation: https://www.cs.cit.tum.de/dis/research/leanstore/

[25] David Li, “Compressing Longs in Druid”, https://imply.io/blog/compressing-longs

[26] Apache Druid Design Documents: Segments, https://druid.apache.org/docs/0.18.0/design/segments.html

[27] CMU Database Group Lecture. “Inside Apache Druid’s Storage and Query Engine (Gian Merlino)”: https://db.cs.cmu.edu/events/vaccination-2021-inside-apache-druids-storage-and-query-engine-gian-merlino/

[28] Databricks, “Comparing Apache Spark^{TM} and Databricks,” https://www.databricks.com/spark/comparing-databricks-to-apache-spark

[29] Michael Armburst, Yin Huai, Cheng Liang, Reynold Xin, and Matei Zaharia, “Deep Dive into Spark SQL’s Catalyst Optimizer,” https://www.databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

[30] CMU Quarantine Database Tech Talks – 2020: https://db.cs.cmu.edu/seminar2020/

[31] Sergio Ferragut, “Learn how to achieve sub-second responses with Apache Druid,” https://imply.io/blog/learn-how-to-achieve-sub-second-responses-with-apache-druid/

[32] CMU Database Group Lecture “Reinventing Amazon Redshift (Ippokratis Pandis)” https://db.cs.cmu.edu/events/vaccination-2021-reinventing-amazon-redshift-ippokratis-pandis