The Snowflake Elastic Data Warehouse

Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes,
Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel,
Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley,
Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner

Snowflake Computing

ABSTRACT

We live in the golden age of distributed computing. Public cloud platforms now offer virtually unlimited compute and storage resources on demand. At the same time, the Software-as-a-Service (SaaS) model brings enterprise-class systems to users who previously could not afford such systems due to their cost and complexity. Alas, traditional data warehousing systems are struggling to fit into this new environment. For one thing, they have been designed for fixed resources and are thus unable to leverage the cloud's elasticity. For another thing, their dependence on complex ETL pipelines and physical tuning is at odds with the flexibility and freshness requirements of the cloud's new types of semi-structured data and rapidly evolving workloads.

We decided a fundamental redesign was in order. Our mission was to build an enterprise-ready data warehousing solution for the cloud. The result is the Snowflake Elastic Data Warehouse, or "Snowflake" for short. Snowflake is a multi-tenant, transactional, secure, highly scalable and elastic system with full SQL support and built-in extensions for semi-structured and schema-less data. The system is offered as a pay-as-you-go service in the Amazon cloud. Users upload their data to the cloud and can immediately manage and query it using familiar tools and interfaces. Implementation began in late 2012 and Snowflake has been generally available since June 2015. Today, Snowflake is used in production by a growing number of small and large organizations alike. The system runs several million queries per day over multiple petabytes of data.

In this paper, we describe the design of Snowflake and its novel multi-cluster, shared-data architecture. The paper highlights some of the key features of Snowflake: extreme elasticity and availability, semi-structured and schema-less data, time travel, and end-to-end security. It concludes with lessons learned and an outlook on ongoing work.

Categories and Subject Descriptors

Information systems [**Data management systems**]: Database management system engines

SIGMOD/PODS'16 June 26 - July 01, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2903741

Keywords

Data warehousing, database as a service, multi-cluster shared data architecture

1. INTRODUCTION

The advent of the cloud marks a move away from software delivery and execution on local servers, and toward shared data centers and software-as-a-service solutions hosted by platform providers such as Amazon, Google, or Microsoft. The shared infrastructure of the cloud promises increased economies of scale, extreme scalability and availability, and a pay-as-you-go cost model that adapts to unpredictable usage demands. But these advantages can only be captured if the *software* itself is able to scale elastically over the pool of commodity resources that is the cloud. Traditional data warehousing solutions pre-date the cloud. They were designed to run on small, static clusters of well-behaved machines, making them a poor architectural fit.

But not only the platform has changed. Data has changed as well. It used to be the case that most of the data in a data warehouse came from sources within the organization: transactional systems, enterprise resource planning (ERP) applications, customer relationship management (CRM) applications, and the like. The structure, volume, and rate of the data were all fairly predictable and well known. But with the cloud, a significant and rapidly growing share of data comes from less controllable or external sources: application logs, web applications, mobile devices, social media, sensor data (Internet of Things). In addition to the growing volume, this data frequently arrives in schema-less, semi-structured formats [3]. Traditional data warehousing solutions are struggling with this new data. These solutions depend on deep ETL pipelines and physical tuning that fundamentally assume predictable, slow-moving, and easily categorized data from largely internal sources.

In response to these shortcomings, parts of the data warehousing community have turned to "Big Data" platforms such as Hadoop or Spark [8, 11]. While these are indispensable tools for data center-scale processing tasks, and the open source community continues to make big improvements such as the Stinger Initiative [48], they still lack much of the efficiency and feature set of established data warehousing technology. But most importantly, they require significant engineering effort to roll out and use [16].

We believe that there is a large class of use cases and workloads which can benefit from the economics, elasticity, and service aspects of the cloud, but which are not well served by either traditional data warehousing technology or

by Big Data platforms. So we decided to build a completely new data warehousing system specifically for the cloud. The system is called the Snowflake Elastic Data Warehouse, or "Snowflake". In contrast to many other systems in the cloud data management space, Snowflake is not based on Hadoop, PostgreSQL or the like. The processing engine and most of the other parts have been developed from scratch.

The key features of Snowflake are as follows.

Pure Software-as-a-Service (SaaS) Experience Users need not buy machines, hire database administrators, or install software. Users either already have their data in the cloud, or they upload (or mail [14]) it. They can then immediately manipulate and query their data using Snowflake's graphical interface or standardized interfaces such as ODBC. In contrast to other Database-as-a-Service (DBaaS) offerings, Snowflake's service aspect extends to the whole user experience. There are no tuning knobs, no physical design, no storage grooming tasks on the part of users.

Relational Snowflake has comprehensive support for ANSI SQL and ACID transactions. Most users are able to migrate existing workloads with little to no changes.

Semi-Structured Snowflake offers built-in functions and SQL extensions for traversing, flattening, and nesting of semi-structured data, with support for popular formats such as JSON and Avro. Automatic schema discovery and columnar storage make operations on schema-less, semi-structured data nearly as fast as over plain relational data, without any user effort.

Elastic Storage and compute resources can be scaled independently and seamlessly, without impact on data availability or performance of concurrent queries.

Highly Available Snowflake tolerates node, cluster, and even full data center failures. There is no downtime during software or hardware upgrades.

Durable Snowflake is designed for extreme durability with extra safeguards against accidental data loss: cloning, undrop, and cross-region backups.

Cost-efficient Snowflake is highly compute-efficient and all table data is compressed. Users pay only for what storage and compute resources they actually use.

Secure All data including temporary files and network traffic is encrypted end-to-end. No user data is ever exposed to the cloud platform. Additionally, role-based access control gives users the ability to exercise fine-grained access control on the SQL level.

Snowflake currently runs on the Amazon cloud (Amazon Web Services, AWS), but we may port it to other cloud platforms in the future. At the time of writing, Snowflake executes millions of queries per day over multiple petabytes of data, serving a rapidly growing number of small and large organizations from various domains.

Outline.

The paper is structured as follows. Section 2 explains the key design choice behind Snowflake: separation of storage and compute. Section 3 presents the resulting multi-cluster, shared-data architecture. Section 4 highlights differentiating features: continuous availability, semi-structured and schema-less data, time travel and cloning, and end-to-end security. Section 5 discusses related work. Section 6 concludes the paper with lessons learned and an outlook on ongoing work.

2. STORAGE VERSUS COMPUTE

Shared-nothing architectures have become the dominant system architecture in high-performance data warehousing, for two main reasons: scalability and commodity hardware. In a shared-nothing architecture, every query processor node has its own local disks. Tables are horizontally partitioned across nodes and each node is only responsible for the rows on its local disks. This design scales well for star-schema queries, because very little bandwidth is required to join a small (broadcast) dimension table with a large (partitioned) fact table. And because there is little contention for shared data structures or hardware resources, there is no need for expensive, custom hardware [25].

In a pure shared-nothing architecture, every node has the same responsibilities and runs on the same hardware. This approach results in elegant software that is easy to reason about, with all the nice secondary effects. A pure shared-nothing architecture has an important drawback though: it tightly couples compute resources and storage resources, which leads to problems in certain scenarios.

Heterogeneous Workload While the hardware is homogeneous, the workload typically is not. A system configuration that is ideal for bulk loading (high I/O bandwidth, light compute) is a poor fit for complex queries (low I/O bandwidth, heavy compute) and vice versa. Consequently, the hardware configuration needs to be a trade-off with low average utilization.

Membership Changes If the set of nodes changes; either as a result of node failures, or because the user chooses to resize the system; large amounts of data need to be reshuffled. Since the very same nodes are responsible for both data shuffling and query processing, a significant performance impact can be observed, limiting elasticity and availability.

Online Upgrade While the effects of small membership changes can be mitigated to some degree using replication, software and hardware upgrades eventually affect *every* node in the system. Implementing online upgrades such that one node after another is upgraded without any system downtime is possible in principle, but is made very hard by the fact that everything is tightly coupled and expected to be homogeneous.

In an on-premise environment, these issues can usually be tolerated. The workload may be heterogeneous, but there is little one can do if there is only a small, fixed pool of nodes on which to run. Upgrades of nodes are rare, and so are node failures and system resizing.

The situation is very different in the cloud. Platforms such as Amazon EC2 feature many different node types [4]. Taking advantage of them is simply a matter of bringing the data to the right type of node. At the same time, node failures are more frequent and performance can vary dramatically, even among nodes of the same type [45]. Membership changes are thus not an exception, they are the norm. And finally, there are strong incentives to enable online upgrades and elastic scaling. Online upgrades dramatically shorten the

software development cycle and increase availability. Elastic scaling further increases availability and allows users to match resource consumption to their momentary needs.

For these reasons and others, Snowflake separates storage and compute. The two aspects are handled by two loosely coupled, independently scalable services. Compute is provided through Snowflake's (proprietary) shared-nothing engine. Storage is provided through Amazon S3 [5], though in principle any type of blob store would suffice (Azure Blob Storage [18, 36], Google Cloud Storage [20]). To reduce network traffic between compute nodes and storage nodes, each compute node caches some table data on local disk.

An added benefit of this solution is that local disk space is not spent on replicating the whole base data, which may be very large and mostly cold (rarely accessed). Instead, local disk is used exclusively for temporary data and caches, both of which are hot (suggesting the use of high-performance storage devices such as SSDs). So, once the caches are warm, performance approaches or even exceeds that of a pure shared-nothing system. We call this novel architecture the *multi-cluster, shared-data architecture*.

3. ARCHITECTURE

Snowflake is designed to be an *enterprise-ready* service. Besides offering high degrees of usability and interoperability, enterprise-readiness means high availability. To this end, Snowflake is a service-oriented architecture composed of highly fault tolerant and independently scalable services. These services communicate through RESTful interfaces and fall into three architectural layers:

Data Storage This layer uses Amazon S3 to store table data and query results.

Virtual Warehouses The "muscle" of the system. This layer handles query execution within elastic clusters of virtual machines, called virtual warehouses.

Cloud Services The "brain" of the system. This layer is a collection of services that manage virtual warehouses, queries, transactions, and all the metadata that goes around that: database schemas, access control information, encryption keys, usage statistics and so forth.

Figure 1 illustrates the three architectural layers of Snowflake and their principal components.

3.1 Data Storage

Amazon Web Services (AWS) have been chosen as the initial platform for Snowflake for two main reasons. First, AWS is the most mature offering in the cloud platform market. Second (and related to the first point), AWS offers the largest pool of potential users.

The next choice was then between using S3 or developing our own storage service based on HDFS or similar [46]. We spent some time experimenting with S3 and found that while its performance could vary, its usability, high availability, and strong durability guarantees were hard to beat. So rather than developing our own storage service, we instead decided to invest our energy into local caching and skew resilience techniques in the Virtual Warehouses layer.

Compared to local storage, S3 naturally has a much higher access latency and there is a higher CPU overhead associated with every single I/O request, especially if HTTPS connections are used. But more importantly, S3 is a blob store with

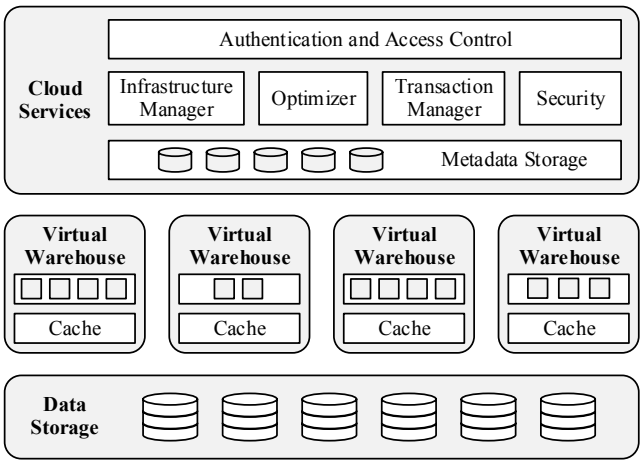

Figure 1: Multi-Cluster, Shared Data Architecture

a relatively simple HTTP(S)-based PUT/GET/DELETE interface. Objects i.e. files can only be (over-)written in full. It is not even possible to append data to the end of a file. In fact, the exact size of a file needs to be announced up-front in the PUT request. S3 does, however, support GET requests for parts (ranges) of a file.

These properties had a strong influence on Snowflake's table file format and concurrency control scheme (cf. Section 3.3.2). Tables are horizontally partitioned into large, immutable files which are equivalent to blocks or pages in a traditional database system. Within each file, the values of each attribute or column are grouped together and heavily compressed, a well-known scheme called PAX or hybrid columnar in the literature [2]. Each table file has a header which, among other metadata, contains the offsets of each column within the file. Because S3 allows GET requests over parts of files, queries only need to download the file headers and those columns they are interested in.

Snowflake uses S3 not only for table data. It also uses S3 to store temp data generated by query operators (e.g. massive joins) once local disk space is exhausted, as well as for large query results. Spilling temp data to S3 allows the system to compute arbitrarily large queries without out-of-memory or out-of-disk errors. Storing query results in S3 enables new forms of client interactions and simplifies query processing, since it removes the need for server-side cursors found in traditional database systems.

Metadata such as catalog objects, which table consists of which S3 files, statistics, locks, transaction logs, etc. is stored in a scalable, transactional key-value store, which is part of the Cloud Services layer.

3.2 Virtual Warehouses

The Virtual Warehouses layer consists of clusters of EC2 instances. Each such cluster is presented to its single user through an abstraction called a virtual warehouse (VW). The individual EC2 instances that make up a VW are called worker nodes. Users never interact directly with worker nodes. In fact, users do not know or care which or how many worker nodes make up a VW. VWs instead come in abstract "T-Shirt sizes" ranging from X-Small to XX-Large. This abstraction allows us to evolve the service and pricing independent of the underlying cloud platform.

3.2.1 Elasticity and Isolation

VWs are pure compute resources. They can be created, destroyed, or resized at any point, on demand. Creating or destroying a VW has no effect on the state of the database. It is perfectly legal (and encouraged) that users shut down *all* their VWs when they have no queries. This elasticity allows users to dynamically match their compute resources to usage demands, independent of the data volume.

Each individual query runs on exactly one VW. Worker nodes are not shared across VWs, resulting in strong performance isolation for queries. (That being said, we recognize worker node sharing as an important area of future work, because it will enable higher utilization and lower cost for use cases where performance isolation is not big concern.)

When a new query is submitted, each worker node in the respective VW (or a subset of the nodes if the optimizer detects a small query) spawns a new worker process. Each worker process lives only for the duration of its query[1]. A worker process by itself—even if part of an update statement—never causes externally visible effects, because table files are immutable, cf. Section 3.3.2. Worker failures are thus easily contained and routinely resolved by retries. Snowflake does not currently perform partial retries though, so very large, long-running queries are an area of concern and future work.

Each user may have multiple VWs running at any given time, and each VW in turn may be running multiple concurrent queries. Every VW has access to the same shared tables, without the need to physically copy data.

Shared, infinite storage means users can *share and integrate* all their data, one of the core principles of data warehousing. Simultaneously, users benefit from *private compute* resources, avoiding interference of different workloads and organizational units—one of the reasons for data marts. This elasticity and isolation enables some novel use strategies. It is common for Snowflake users to have several VWs for queries from different organizational units, often running continuously, and periodically launch on-demand VWs, for instance for bulk loading.

Another important observation related to elasticity is that it is often possible to achieve much better performance for roughly the same price. For example, a data load which takes 15 hours on a system with 4 nodes might take only 2 hours with 32 nodes[2]. Since one pays for compute-hours, the overall cost is very similar—yet the user experience is dramatically different. We thus believe that VW elasticity is one of the biggest benefits and differentiators of the Snowflake architecture, and it shows that a novel design is needed to make use of unique capabilities of the cloud.

3.2.2 Local Caching and File Stealing

Each worker node maintains a cache of table data on local disk. The cache is a collection of table files i.e. S3 objects that have been accessed in the past by the node. To be precise, the cache holds file headers and individual columns of files, since queries download only the columns they need.

The cache lives for the duration of the worker node and is shared among concurrent and subsequent worker processes i.e. queries. It just sees a stream of file and column requests, and follows a simple least-recently-used (LRU) replacement policy, oblivious of individual queries. This simple scheme works surprisingly well, but we may refine it in the future to better match different workloads.

To improve the hit rate and avoid redundant caching of individual table files across worker nodes of a VW, the query optimizer assigns input file sets to worker nodes using consistent hashing over table file names [31]. Subsequent or concurrent queries accessing the same table file will therefore do this on the same worker node.

Consistent hashing in Snowflake is lazy. When the set of worker nodes changes—because of node failures or VW resizing—no data is shuffled immediately. Instead, Snowflake relies on the LRU replacement policy to eventually replace the cache contents. This solution amortizes the cost of replacing cache contents over multiple queries, resulting in much better availability than an eager cache or a pure shared-nothing system which would need to immediately shuffle large amounts of table data across nodes. It also simplifies the system since there is no "degraded" mode.

Besides caching, skew handling is particularly important in a cloud data warehouse. Some nodes may be executing much slower than others due to virtualization issues or network contention. Among other places, Snowflake deals with this problem at the scan level. Whenever a worker process completes scanning its set of input files, it requests additional files from its peers, a technique we call *file stealing*. If a peer finds that it has many files left in its input file set when such a request arrives, it answers the request by transferring ownership of one remaining file for the duration and scope of the current query. The requestor then downloads the file directly from S3, not from its peer. This design ensures that file stealing does not make things worse by putting additional load on straggler nodes.

3.2.3 Execution Engine

There is little value in being able to execute a query over 1,000 nodes if another system can do it in the same time using 10 such nodes. So while scalability is prime, per-node efficiency is just as important. We wanted to give users the best price/performance of any database-as-a-service offering on the market, so we decided to implement our own state-of-the-art SQL execution engine. The engine we have built is columnar, vectorized, and push-based.

Columnar storage and execution is generally considered superior to row-wise storage and execution for analytic workloads, due to more effective use of CPU caches and SIMD instructions, and more opportunities for (lightweight) compression [1, 33].

Vectorized execution means that, in contrast to MapReduce for example [42], Snowflake avoids materialization of intermediate results. Instead, data is processed in pipelined fashion, in batches of a few thousand rows in columnar format. This approach, pioneered by VectorWise (originally MonetDB/X100 [15]), saves I/O and greatly improves cache efficiency.

Push-based execution refers to the fact that relational operators *push* their results to their downstream operators, rather than waiting for these operators to *pull* data (classic Volcano-style model [27]). Push-based execution improves cache efficiency, because it removes

[1]Ephemeral processes are appropriate for analytic workloads but entail some query start-up cost. As an obvious optimization, we plan to recycle worker processes for small queries.

[2]Snowflake exposes T-shirt sizes rather than concrete node numbers, but the principle holds.

control flow logic from tight loops [41]. It also enables Snowflake to efficiently process DAG-shaped plans, as opposed to just trees, creating additional opportunities for sharing and pipelining of intermediate results.

At the same time, many sources of overhead in traditional query processing are not present in Snowflake. Notably, there is no need for transaction management during execution. As far as the engine is concerned, queries are executed against a fixed set of immutable files. Also, there is no buffer pool. Most queries scan large amounts of data. Using memory for table buffering versus operation is a bad trade-off here. Snowflake does, however, allow all major operators (join, group by, sort) to spill to disk and recurse when main memory is exhausted. We found that a pure main-memory engine, while leaner and perhaps faster, is too restrictive to handle all interesting workloads. Analytic workloads can feature extremely large joins or aggregations.

3.3 Cloud Services

Virtual warehouses are ephemeral, user-specific resources. In contrast, the Cloud Services layer is heavily multi-tenant. Each service of this layer—access control, query optimizer, transaction manager, and others—is long-lived and shared across many users. Multi-tenancy improves utilization and reduces administrative overhead, which allows for better economies of scale than in traditional architectures where every user has a completely private system incarnation.

Each service is replicated for high availability and scalability. Consequently, the failure of individual service nodes does not cause data loss or loss of availability, though some running queries may fail (and be transparently re-executed).

3.3.1 Query Management and Optimization

All queries issued by users pass through the Cloud Services layer. Here, all the early stages of the query life cycle are handled: parsing, object resolution, access control, and plan optimization.

Snowflake's query optimizer follows a typical Cascades-style approach [28], with top-down cost-based optimization. All statistics used for optimization are automatically maintained on data load and updates. Since Snowflake does not use indices (cf. Section 3.3.3), the plan search space is smaller than in some other systems. The plan space is further reduced by postponing many decisions until execution time, for example the type of data distribution for joins. This design reduces the number of bad decisions made by the optimizer, increasing robustness at the cost of a small loss in peak performance. It also makes the system easier to use (performance becomes more predictable), which is in line with Snowflake's overall focus on service experience.

Once the optimizer completes, the resulting execution plan is distributed to all the worker nodes that are part of the query. As the query executes, Cloud Services continuously tracks the state of the query to collect performance counters and detect node failures. All query information and statistics are stored for audits and performance analysis. Users are able to monitor and analyze past and ongoing queries through the Snowflake graphical user interface.

3.3.2 Concurrency Control

As mentioned previously, concurrency control is handled entirely by the Cloud Services layer. Snowflake is designed for analytic workloads, which tend to be dominated by large reads, bulk or trickle inserts, and bulk updates. Like most systems in this workload space, we decided to implement ACID transactions via Snapshot Isolation (SI) [17].

Under SI, all reads by a transaction see a consistent snapshot of the database as of the time the transaction started. As customary, SI is implemented on top of multi-version concurrency control (MVCC), which means a copy of every changed database object is preserved for some duration.

MVCC is a natural choice given the fact that table files are immutable, a direct consequence of using S3 for storage. Changes to a file can only be made by replacing it with a different file that includes the changes[3]. It follows that write operations (insert, update, delete, merge) on a table produce a newer version of the table by adding and removing whole files relative to the prior table version. File additions and removals are tracked in the metadata (in the global key-value store), in a form which allows the set of files that belong to a specific table version to be computed very efficiently.

Besides for SI, Snowflake also uses these snapshots to implement time travel and efficient cloning of database objects, see Section 4.4 for details.

3.3.3 Pruning

Limiting access only to data that is relevant to a given query is one of the most important aspects of query processing. Historically, data access in databases was limited through the use of indices, in the form of B^+-trees or similar data structures. While this approach proved highly effective for transaction processing, it raises multiple problems for systems like Snowflake. First, it relies heavily on random access, which is a problem both due to the storage medium (S3) and the data format (compressed files). Second, maintaining indices significantly increases the volume of data and data loading time. Finally, the user needs to explicitly create the indices—which would go very much against the pure service approach of Snowflake. Even with the aid of tuning advisors, maintaining indices can be a complex, expensive, and risky process.

An alternative technique has recently gained popularity for large-scale data processing: min-max based pruning, also known as small materialized aggregates [38], zone maps [29], and data skipping [49]. Here, the system maintains the data distribution information for a given chunk of data (set of records, file, block etc.), in particular minimum and maximum values within the chunk. Depending on the query predicates, these values can be used to determine that a given chunk of data might not be needed for a given query. For example, imagine files f_1 and f_2 contain values `3..5` and `4..6` respectively, in some column `x`. Then, if a query has a predicate `WHERE x >= 6`, we know that only f_2 needs to be accessed. Unlike traditional indices, this metadata is usually orders of magnitude smaller than the actual data, resulting in a small storage overhead and fast access.

Pruning nicely matches the design principles of Snowflake: it does not rely on user input; it scales well; and it is easy to maintain. What is more, it works well for sequential access of large chunks of data, and it adds little overhead to loading, query optimization, and query execution times.

Snowflake keeps pruning-related metadata for every indi-

[3]It would certainly be possible to defer changes to table files through the introduction of a redo-undo log, perhaps in combination with a delta store [32], but we are currently not pursuing this idea for reasons of complexity and scalability.

vidual table file. The metadata not only covers plain relational columns, but also a selection of auto-detected columns inside of semi-structured data, see Section 4.3.2. During optimization, the metadata is checked against the query predicates to reduce ("prune") the set of input files for query execution. The optimizer performs pruning not only for simple base-value predicates, but also for more complex expressions such as `WEEKDAY(orderdate) IN (6, 7)`.

Besides this *static* pruning, Snowflake also performs *dynamic* pruning during execution. For example, as part of hash join processing, Snowflake collects statistics on the distribution of join keys in the build-side records. This information is then pushed to the probe side and used to filter and possibly skip entire files on the probe side. This is in addition to other well-known techniques such as bloom joins [40].

4. FEATURE HIGHLIGHTS

Snowflake offers many features expected from a relational data warehouse: comprehensive SQL support, ACID transactions, standard interfaces, stability and security, customer support, and—of course—strong performance and scalability. Additionally, it introduces a number of other valuable features rarely or never-before seen in related systems. This section presents a few of these features that we consider technical differentiators.

4.1 Pure Software-as-a-Service Experience

Snowflake supports standard database interfaces (JDBC, ODBC, Python PEP-0249) and works with various third-party tools and services such as Tableau, Informatica, or Looker. However, it also provides the possibility to interact with the system using nothing but a web browser. A web UI may seem like a trivial thing, but it quickly proved itself to be a critical differentiator. The web UI makes it very easy to access Snowflake from any location and environment, dramatically reducing the complexity of bootstrapping and using the system. With a lot of data already in the cloud, it allowed many users to just point Snowflake at their data and query away, without downloading any software.

As may be expected, the UI allows not only SQL operations, but also gives access to the database catalog, user and system management, monitoring, usage information, and so forth. We continuously expand the UI functionality, working on aspects such as online collaboration, user feedback and support, and others.

But our focus on ease-of-use and service experience does not stop at the user interface; it extends to every aspect of the system architecture. There are no failure modes, no tuning knobs, no physical design, no storage grooming tasks. It is all about the data and the queries.

4.2 Continuous Availability

In the past, data warehousing solutions were well-hidden back-end systems, isolated from most of the world. In such environments, downtimes—both planned (software upgrades or administrative tasks) and unplanned (failures)—usually did not have a large impact on operations. But as data analysis became critical to more and more business tasks, continuous availability became an important requirement for any data warehouse. This trend mirrors the expectations on modern SaaS systems, most of which are always-on, customer-facing applications with no (planned) downtime.

Snowflake offers continuous availability that meets these expectations. The two main technical features in this regard are fault resilience and online upgrades.

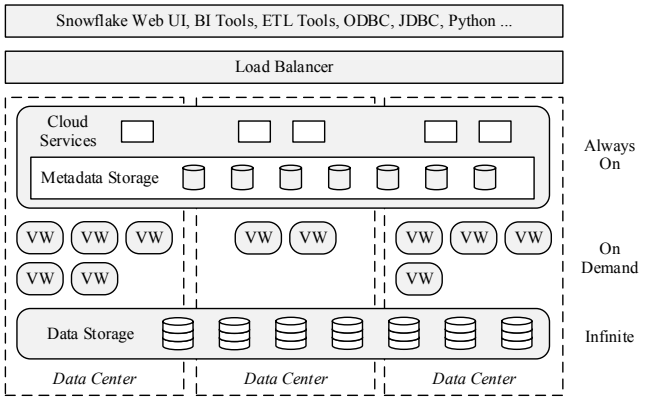

Figure 2: Multi-Data Center Instance of Snowflake

4.2.1 Fault Resilience

Snowflake tolerates individual and correlated node failures at all levels of the architecture, shown in Figure 2. The Data Storage layer of Snowflake today is S3, which is replicated across multiple data centers called "availability zones" or AZs in Amazon terminology. Replication across AZs allows S3 to handle full AZ failures, and to guarantee 99.99% data availability and 99.999999999% durability. Matching S3's architecture, Snowflake's metadata store is also distributed and replicated across multiple AZs. If a node fails, other nodes can pick up the activities without much impact on end users. The remaining services of the Cloud Services layer consist of stateless nodes in multiple AZs, with a load balancer distributing user requests between them. It follows that a single node failure or even a full AZ failure causes no system-wide impact, possibly some failed queries for users currently connected to a failed node. These users will be redirected to a different node for their next query.

In contrast, Virtual Warehouses (VWs) are not distributed across AZs. This choice is for performance reasons. High network throughput is critical for distributed query execution, and network throughput is significantly higher within the same AZ. If one of the worker nodes fails during query execution, the query fails but is transparently re-executed, either with the node immediately replaced, or with a temporarily reduced number of nodes. To accelerate node replacement, Snowflake maintains a small pool of standby nodes. (These nodes are also used for fast VW provisioning.)

If an *entire* AZ becomes unavailable though, all queries running on a given VW of that AZ will fail, and the user needs to actively re-provision the VW in a different AZ. With full-AZ failures being truly catastrophic and exceedingly rare events, we today accept this one scenario of partial system unavailability, but hope to address it in the future.

4.2.2 Online Upgrade

Snowflake provides continuous availability not only when failures occur, but also during software upgrades. The system is designed to allow multiple versions of the various services to be deployed side-by-side, both Cloud Services components and virtual warehouses. This is made possible by the fact that all services are effectively stateless. All hard state is kept in a transactional key-value store and is ac-

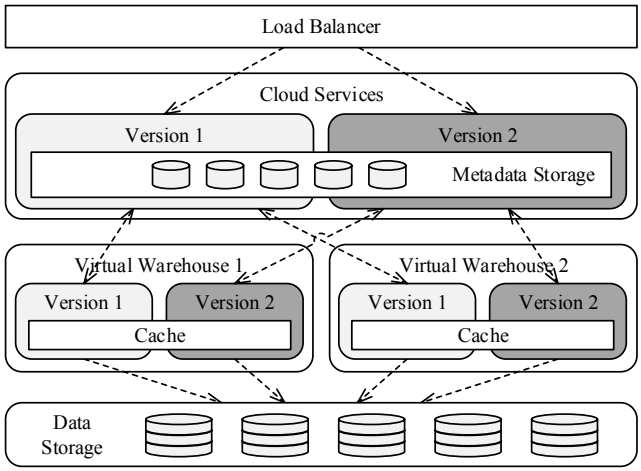

Figure 3: Online Upgrade

cessed through a mapping layer which takes care of metadata versioning and schema evolution. Whenever we change the metadata schema, we ensure backward compatibility with the previous version.

To perform a software upgrade, Snowflake first deploys the new version of the service alongside the previous version. User accounts are then progressively switched to the new version, at every which point all new queries issued by the respective user are directed to the new version. All queries that were executing against the previous version are allowed to run to completion. Once all queries and users have finished using the previous version, all services of that version are terminated and decommissioned.

Figure 3 shows a snapshot of an ongoing upgrade process. There are two versions of Snowflake running side-by-side, version 1 (light) and version 2 (dark). There are two versions of a single incarnation of Cloud Services, controlling two virtual warehouses (VWs), each having two versions. The load balancer directs incoming calls to the appropriate version of Cloud Services. The Cloud Services of one version only talk to VWs of a matching version.

As mentioned previously, both versions of Cloud Services share the same metadata store. What is more, VWs of different versions are able to share the same worker nodes and their respective caches. Consequently, there is no need to repopulate the caches after an upgrade. The entire process is transparent to the user with no downtime or performance degradation.

Online upgrade also has had a tremendous effect on our speed of development, and on how we handle critical bugs at Snowflake. At the time of writing, we upgrade all services once per week. That means we release features and improvements on a weekly basis. To ensure the upgrade process goes smoothly, both upgrade and downgrade are continuously tested in a special pre-production incarnation of Snowflake. In those rare cases where we find a critical bug in our production incarnation (not necessarily during an upgrade), we can very quickly downgrade to the previous version, or implement a fix and perform an out-of-schedule upgrade. This process is not as scary as it may sound, because we continuously test and exercise the upgrade/downgrade mechanism. It is highly automated and hardened at this point.

4.3 Semi-Structured and Schema-Less Data

Snowflake extends the standard SQL type system with three types for semi-structured data: `VARIANT`, `ARRAY`, and `OBJECT`. Values of type `VARIANT` can store any value of native SQL type (`DATE`, `VARCHAR` etc.), as well as variable-length `ARRAY`s of values, and JavaScript-like `OBJECT`s, maps from strings to `VARIANT` values. The latter are also called *documents* in the literature, giving rise to the notion of document stores (MongoDB [39], Couchbase [23]).

`ARRAY` and `OBJECT` are just restrictions of type `VARIANT`. The internal representation is the same: a self-describing, compact binary serialization which supports fast key-value lookup, as well as efficient type tests, comparison, and hashing. `VARIANT` columns can thus be used as join keys, grouping keys, and ordering keys, just like any other column.

The `VARIANT` type allows Snowflake to be used in an ELT (Extract-Load-Transform) manner rather than a traditional ETL (Extract-Transform-Load) manner. There is no need to specify document schemas or to perform transformations on the load. Users can load their input data from JSON, Avro, or XML format directly into a `VARIANT` column; Snowflake handles parsing and type inference (cf. Section 4.3.3). This approach, aptly called "schema later" in the literature, allows for schema evolution by decoupling information producers from information consumers and any intermediaries. In contrast, any change in data schemas in a conventional ETL pipeline requires coordination between multiple departments in an organization, which can take months to execute.

Another advantage of ELT and Snowflake is that later, if transformation is desired, it can be performed using the full power of a parallel SQL database, including operations such as joins, sorting, aggregation, complex predicates and so forth, which are typically missing or inefficient in conventional ETL toolchains. On that point, Snowflake also features procedural user-defined functions (UDFs) with full JavaScript syntax and integration with the `VARIANT` data type. Support for procedural UDFs further increases the number of ETL tasks that can be pushed into Snowflake.

4.3.1 Post-relational Operations

The most important operation on documents is extraction of data elements, either by field name (for `OBJECT`s), or by offset (for `ARRAY`s). Snowflake provides extraction operations in both functional SQL notation and JavaScript-like path syntax. The internal encoding makes extraction very efficient. A child element is just a pointer inside the parent element; no copying is required. Extraction is often followed by a cast of the resulting `VARIANT` value to a standard SQL type. Again, the encoding makes these casts very efficient.

The second common operation is flattening, i.e. pivoting a nested document into multiple rows. Snowflake uses SQL lateral views to represent flattening operations. This flattening can be recursive, allowing the complete conversion of the hierarchical structure of the document into a relational table amenable to SQL processing. The opposite operation to flattening is aggregation. Snowflake introduces a few new aggregate and analytic functions such as `ARRAY_AGG` and `OBJECT_AGG` for this purpose.

4.3.2 Columnar Storage and Processing

The use of a serialized (binary) representation for semi-structured data is a conventional design choice for integrating semi-structured data into relational databases. The row-

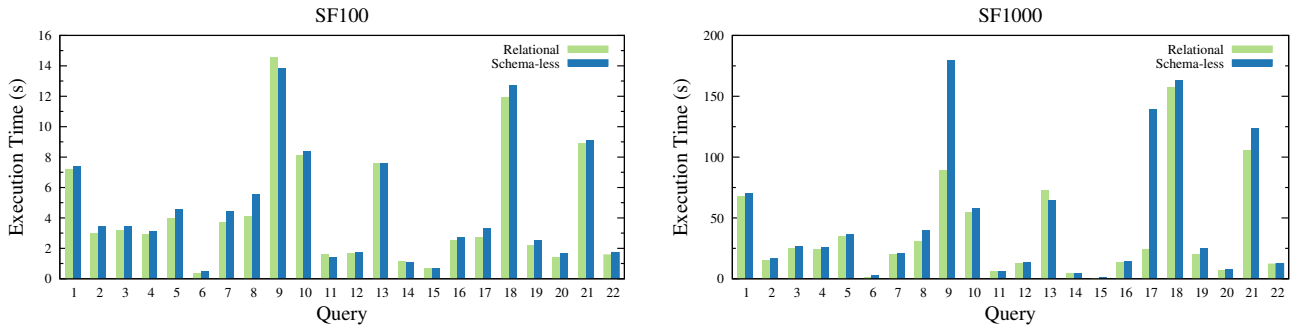

Figure 4: TPC-H SF100 and SF1000 Performance: Relational vs. Schema-less Row Format

wise representation, unfortunately, makes storage and processing of such data less efficient than that of columnar relational data—which is the usual reason for transforming semi-structured data into plain relational data.

Cloudera Impala [21] (using Parquet [10]) and Google Dremel [34] have demonstrated that columnar storage of semi-structured data is possible and beneficial. However, Impala and Dremel (and its externalization BigQuery [44]) require users to provide complete table schemas for columnar storage. To achieve *both* the flexibility of a schema-less serialized representation and the performance of a columnar relational database, Snowflake introduces a novel automated approach to type inference and columnar storage.

As mentioned in Section 3.1, Snowflake stores data in a hybrid columnar format. When storing semi-structured data, the system automatically performs statistical analysis of the collection of documents within a single table file, to perform automatic type inference and to determine which (typed) paths are frequently common. The corresponding columns are then removed from the documents and stored separately, using the same compressed columnar format as native relational data. For these columns, Snowflake even computes materialized aggregates for use by pruning (cf. Section 3.3.3), as with plain relational data.

During a scan, the various columns can be reassembled into a single column of type `VARIANT`. Most queries, however, are only interested in a subset of the columns of the original document. In those cases, Snowflake pushes projection and cast expressions down into the scan operator, so that only the necessary columns are accessed and cast directly into the target SQL type.

The optimizations described above are performed *independently* for every table file, which allows for efficient storage and extraction even under schema evolution. However, it does raise challenges with respect to query optimization, in particular pruning. Suppose a query has a predicate over a path expression, and we would like to use pruning to restrict the set of files to be scanned. The path and corresponding column may be present in most files, but only frequent enough to warrant metadata in some of the files. The conservative solution is to simply scan all files for which there is no suitable metadata. Snowflake improves over this solution by computing Bloom filters over all *paths* (not values!) present in the documents. These Bloom filters are saved along with the other file metadata, and probed by the query optimizer during pruning. Table files which do not contain paths required by a given query can safely be skipped.

4.3.3 *Optimistic Conversion*

Because some native SQL types, notably date/time values, are represented as strings in common external formats such as JSON or XML, these values need to be converted from strings to their actual type either at write time (during insert or update) or at read time (during queries). Without a typed schema or equivalent hints, these string conversions need to be performed at read time, which, in a read-dominated workload, is less efficient than doing the conversions once, during the write. Another problem with untyped data is the lack of suitable metadata for pruning, which is especially important in case of dates. (Analytical workloads frequently have range predicates on date columns.)

But applied at write time, automatic conversions may lose information. For example, a field containing numeric product identifiers may actually not be a number but a string with significant leading zeros. Similarly, what looks like a date could really be the content of a text message. Snowflake solves the problem by performing optimistic data conversion, and preserving both the result of the conversion and the original string (unless a fully reversible conversion exists), in separate columns. If a query later requires the original string, it is easily retrieved or reconstructed. Because unused columns are not loaded and accessed, the impact of any double storage on query performance is minimal.

4.3.4 *Performance*

To assess the combined effects of columnar storage, optimistic conversion, and pruning over semi-structured data on query performance, we conducted a set of experiments using TPC-H-like[4] data and queries.

We created two types of database schemas. First, a conventional, relational TPC-H schema. Second, a "schema-less" database schema, where every table consisted of a single column of type `VARIANT`. We then generated clustered (sorted) SF100 and SF1000 data sets (100 GB and 1 TB respectively), stored the data sets in plain JSON format (i.e., dates became strings), and loaded that data into Snowflake, using both the relational and schema-less database schemas. No hints of any kind regarding the fields, types, and clustering of the schema-less data were given to the system, and no other tuning was done. We then defined a few views on top of the schema-less databases, in order to be able to run the

[4]Experimental results were obtained using a faithful implementation of TPC-H data generation and queries. Nonetheless, the data, queries, and numbers have not been verified by the TPC and do not constitute official benchmark results.

exact same set of TPC-H queries against all four databases. (At the time of writing, Snowflake does not use views for type inference or other optimizations, so this was purely a syntactic convenience.)

Finally, we ran all 22 TPC-H queries against the four databases, using a *medium standard* warehouse[5]. Figure 4 shows the results. Numbers were obtained over three runs with warm caches. Standard errors were insignificant and thus omitted from the results.

As can be seen, the overhead of schema-less storage and query processing was around 10% for all but two queries (Q9 and Q17 over SF1000). For these two queries, we determined the reason for the slow-down to be a sub-optimal join order, caused by a known bug in distinct value estimation. We continue to make improvements to metadata collection and query optimization over semi-structured data.

In summary, the query performance over semi-structured data with relatively stable and simple schemas (i.e. the majority of machine-generated data found in practice), is nearly on par with the performance over conventional relational data, enjoying all the benefits of columnar storage, columnar execution, and pruning—without the user effort.

4.4 Time Travel and Cloning

In Section 3.3.2, we discussed how Snowflake implements Snapshot Isolation (SI) on top of multi-version concurrency control (MVCC). Write operations (insert, update, delete, merge) on a table produce a newer version of the table by adding and removing whole files.

When files are removed by a new version, they are retained for a configurable duration (currently up to 90 days). File retention allows Snowflake to read earlier versions of tables very efficiently; that is, to perform *time travel* on the database. Users can access this feature from SQL using the convenient `AT` or `BEFORE` syntax. Timestamps can be absolute, relative with respect to current time, or relative with respect to previous statements (referred to by ID).

```
SELECT * FROM my_table AT(TIMESTAMP =>
  'Mon, 01 May 2015 16:20:00 -0700'::timestamp);
SELECT * FROM my_table AT(OFFSET => -60*5); -- 5 min ago
SELECT * FROM my_table BEFORE(STATEMENT =>
  '8e5d0ca9-005e-44e6-b858-a8f5b37c5726');
```

One can even access different versions of the same table in a single query.

```
SELECT new.key, new.value, old.value FROM my_table new
JOIN my_table AT(OFFSET => -86400) old -- 1 day ago
ON new.key = old.key WHERE new.value <> old.value;
```

Based on the same underlying metadata, Snowflake introduces the `UNDROP` keyword to quickly restore tables, schemas, or whole databases that have been dropped accidentally.

```
DROP DATABASE important_db; -- whoops!
UNDROP DATABASE important_db;
```

Snowflake also implements a functionality we call *cloning*, expressed through the new keyword `CLONE`. Cloning a table creates a new table with the same definition and contents quickly and without making physical copies of table files. The clone operation simply copies the metadata of the source table. Right after cloning, both tables refer to the same set of files, but both tables can be modified independently thereafter. The clone feature also supports whole schemas or databases, which allows for very efficient snapshots. Snapshots are good practice before a large batch of updates, or when performing lengthy, exploratory data analysis. The `CLONE` keyword can even be combined with `AT` and `BEFORE`, allowing such snapshots to be made after the fact.

[5]We currently do not disclose pricing and hardware details, but a medium standard warehouse consists of a very small number of inexpensive EC2 instances.

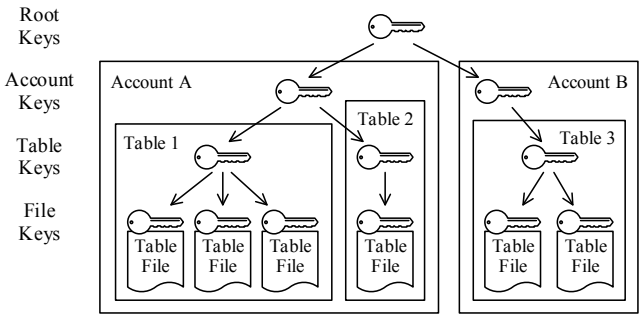

Figure 5: Encryption Key Hierarchy

```
CREATE DATABASE recovered_db CLONE important_db BEFORE(
  STATEMENT => '8e5d0ca9-005e-44e6-b858-a8f5b37c5726');
```

4.5 Security

Snowflake is designed to protect user data against attacks on all levels of the architecture, including the cloud platform. To this end, Snowflake implements two-factor authentication, (client-side) encrypted data import and export, secure data transfer and storage, and role-based access control (RBAC [26]) for database objects. At all times, data is encrypted before being sent over the network, and before being written to local disk or shared storage (S3). Thus, Snowflake provides full end-to-end data encryption and security.

4.5.1 Key Hierarchy

Snowflake uses strong AES 256-bit encryption with a hierarchical key model rooted in AWS CloudHSM [12]. Encryption keys are automatically rotated and re-encrypted ("rekeyed") to ensure that keys complete the full NIST 800-57 cryptographic key-management life cycle [13]. Encryption and key management are entirely transparent to the user and require no configuration or management.

The Snowflake key hierarchy, shown in Figure 5, has four levels: root keys, account keys, table keys, and file keys. Each layer of (parent) keys encrypts i.e. *wraps* the layer of (child) keys below. Each account key corresponds to one user account, each table key corresponds to one database table, and each file key corresponds to one table file.

Hierarchical key models are good security practice because they constrain the amount of data each key protects. Each layer reduces the scope of keys below it, as indicated by the boxes in Figure 5. Snowflake's hierarchical key model ensures isolation of user data in its multi-tenant architecture, because each user account has a separate account key.

4.5.2 Key Life Cycle

Orthogonal to constraining the *amount* of data each key protects, Snowflake also constrains the *duration of time* during which a key is usable. Encryption keys go through four

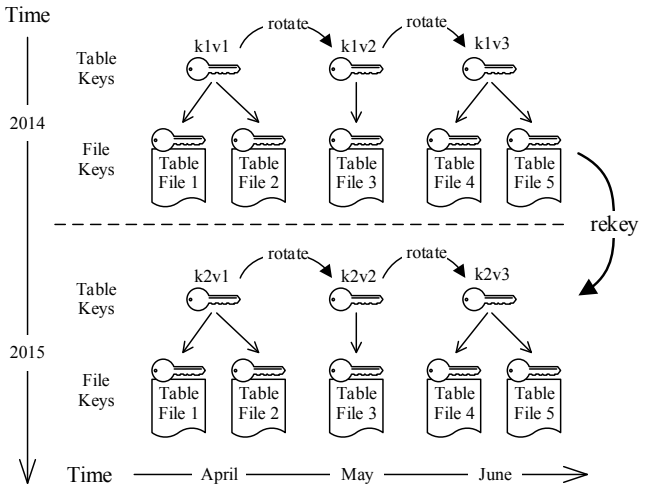

Figure 6: Table Key Life Cycle

phases: (1) a pre-operational creation phase, (2) an operational phase where keys are used to encrypt (originator-usage period) and decrypt (recipient-usage period), (3) a post-operational phase where keys are no longer in use, and (4) a destroyed phase. Phases 1, 3, and 4 are trivial to implement. Phase 2 requires one to limit the originator-usage and recipient-usage periods. Only when a key no longer encrypts any required data, it can be moved on to phases 3 and 4. Snowflake limits the originator-usage period using *key rotation* and the recipient-usage period using *rekeying*.

Key rotation creates new versions of keys at regular intervals (say, one month). After each such interval, a new version of a key is created and the previous version of the key is "retired". The retired version is still usable, but only to decrypt data. When wrapping new child keys in the key hierarchy, or when writing to tables, only the latest, active version of the key is used to encrypt the data.

Rekeying is the process of re-encrypting old data with new keys. After a specific time interval (say, one year), data that has been encrypted with a retired key is re-encrypted with an active key. This rekeying is orthogonal to key rotation. While key rotation ensures that a key is transferred from its active state (originator usage) to a retired state (recipient usage), rekeying ensures that a key can be transferred from its retired state to being destroyed.

Figure 6 shows the life cycle of a single table key. Assume keys are rotated once per month, and data is rekeyed once per year. Table files 1 and 2 are created in April 2014, using key 1 version 1 (k1v1). In May 2014, key 1 is rotated to version 2 (k1v2), and table file 3 is created using k1v2. In June 2014, key 1 is rotated to version 3 (k1v3), and two more table files are created. No more inserts or updates are made to the table after June 2014. In April 2015, k1v1 becomes one year old and needs to be destroyed. A new key, key 2 version 1 (k2v1), is created, and all files associated with k1v1 are rekeyed using k2v1. In May 2015, the same happens to k1v2 and table file 3 is rekeyed using k2v2. In June 2015, table files 4 and 5 are rekeyed using k2v3.

An analogous scheme is implemented between account keys and table keys, and between the root key and account keys. Each level of the key hierarchy undergoes key rotation and rekeying, including the root key. Key rotation and rekeying of account keys and the root key do not require re-encryption of files. Only the *keys* of the immediate lower level need to be re-encrypted.

The relationship between table keys and file keys is different though. File keys are not wrapped by table keys. Instead, file keys are cryptographically derived from the combination of table key and (unique) file name. It follows that whenever a table key changes, all of its related file keys change, so the affected table files need to be re-encrypted. The big benefit of key derivation, however, is that it removes the need to create, manage, and pass around individual file keys. A system like Snowflake that handles billions of files would have to handle many gigabytes of file keys otherwise.

We chose this design also because Snowflake's separation of storage and compute allows it to perform re-encryption without impacting user workloads. Rekeying works in the background, using different worker nodes than queries. After files are rekeyed, Snowflake atomically updates the metadata of database tables to point to the newly encrypted files. The old files are deleted once all ongoing queries are finished.

4.5.3 End-to-End Security

Snowflake uses AWS CloudHSM as a tamper-proof, highly secure way to generate, store, and use the root keys of the key hierarchy. AWS CloudHSM is a set of hardware security modules (HSMs) that are connected to a virtual private cluster within AWS. The root keys never leave the HSM devices. All cryptographic operations using root keys are performed within the HSMs themselves. Thus, lower-level keys cannot be unwrapped without authorized access to the HSM devices. The HSMs are also used to generate keys at the account and table levels, including during key rotation and rekeying. We configured AWS CloudHSM in its high-availability configuration to minimize the possibility of service outages.

In addition to data encryption, Snowflake protects user data in the following ways:

1. Isolation of storage through access policies on S3.
2. Role-based access control within user accounts for fine-grained access control to database objects.
3. Encrypted data import and export without the cloud provider (Amazon) ever seeing data in the clear.
4. Two-factor- and federated authentication for secure access control.

In summary, Snowflake provides a hierarchical key model rooted in AWS CloudHSM and uses key rotation and rekeying to ensure that encryption keys follow a standardized life cycle. Key management is entirely transparent to the user and requires no configuration, management, or downtime. It is part of a comprehensive security strategy that enables full end-to-end encryption and security.

5. RELATED WORK

Cloud-based Parallel Database Systems. Amazon has a number of DBaaS products with Amazon Redshift being the data warehousing product among these. Having evolved from the parallel database system ParAccel, Redshift was arguably the first real data warehouse system offered as a service [30]. Redshift uses a classic shared-nothing architecture. Thus, while being scalable, adding or removing

compute resources requires data redistribution. In contrast, Snowflake's multi-cluster, shared data architecture allows users to instantly scale up, scale down, or even pause compute independently from storage without data movement—including the ability to integrate data across isolated compute resources. Also, following a pure service principle, Snowflake requires no physical tuning, data grooming, manual gathering of table statistics, or table vacuuming on the part of users. Although Redshift can ingest semi-structured data such as JSON as a `VARCHAR`, Snowflake has native support for semi-structured data, including important optimizations such as columnar storage.

Google's Cloud Platform offers a fully managed query service known as BigQuery [44], which is the public implementation of Dremel [34]. The BigQuery service lets users run queries on terabytes of data at impressive speeds, parallelized across thousands of nodes. One of the inspirations for Snowflake was BigQuery's support for JSON and nested data, which we find necessary for a modern analytics platform. But while BigQuery offers a SQL-like language, it has some fundamental deviations from the ANSI SQL syntax and semantics, making it tricky to use with SQL-based products. Also, BigQuery tables are append-only and require schemas. In comparison, Snowflake offers full DML (insert, update, delete, merge), ACID transactions, and does not require schema definitions for semi-structured data.

Microsoft SQL Data Warehouse (Azure SQL DW) is a recent addition to the Azure cloud platform and services based on SQL Server and its Analytics Platform System (APS) appliance [35, 37]. Similar to Snowflake, it separates storage from compute. Computational resources can be scaled through data warehouse units (DWUs). The degree of concurrency is capped though. For any data warehouse, the maximum number of concurrently executing queries is 32 [47]. Snowflake, in contrast, allows fully independent scaling of concurrent workloads via virtual warehouses. Snowflake users are also released from the burden of choosing appropriate distribution keys and other administrative tasks. And while Azure SQL DW does support queries over non-relational data via PolyBase [24], it does not have built-in support for semi-structured data comparable to Snowflake's `VARIANT` type and related optimizations.

Document Stores and Big Data. Document stores such as MongoDB [39], Couchbase Server [23], and Apache Cassandra [6], have become increasingly popular among application developers in recent years, because of the scalability, simplicity, and schema flexibility they offer. However, one challenge that has resulted from the simple key-value and CRUD (create, read, update, and delete) API of these systems is the difficulty to express more complex queries. In response, we have seen the emergence of several SQL-like query languages such as N1QL [22] for Couchbase or CQL [19] for Apache Cassandra. Additionally, many "Big Data" engines now support queries over nested data, for example Apache Hive [9], Apache Spark [11], Apache Drill [7], Cloudera Impala [21], and Facebook Presto [43]. We believe that this shows a real need for complex analytics over schema-less and semi-structured data, and our semi-structured data support is inspired by many of these systems. Using schema inference, optimistic conversions, and columnar storage, Snowflake combines the flexibility of these systems with the storage efficiency and execution speed of a relational, column-oriented database.

6. LESSONS LEARNED AND OUTLOOK

When Snowflake was founded in 2012, the database world was fully focused on *SQL on Hadoop*, with over a dozen systems appearing within a short time span. At that time, the decision to work in a completely different direction, to build a "classic" data warehouse system for the cloud, seemed a contrarian and risky move. After 3 years of development we are confident that it was the right one. Hadoop has not replaced RDBMSs; it has complemented them. People still want a relational database, but one that is more efficient, flexible, and better suited for the cloud.

Snowflake has met our hopes of what a system built for the cloud can provide to both its users and its authors. The elasticity of the multi-cluster, shared data architecture has changed how users approach their data processing tasks. The SaaS model not only has made it easy for users to try out and adopt the system, but has also dramatically helped our development and testing. With a single production version and online upgrades, we are able to release new features, provide improvements, and fix problems much faster than we would possibly be able to do under a traditional development model.

While we had hoped that the semi-structured extensions would prove useful, we were surprised by the speed of adoption. We discovered a very popular model, where organizations would use Hadoop for two things: for storing JSON, and for converting it to a format that can be loaded into an RDBMS. By providing a system that can efficiently store and process semi-structured data as-is—with a powerful SQL interface on top—we found Snowflake replacing not only traditional database systems, but also Hadoop clusters.

It was not a painless journey of course. While our team has over 100 years of database-development expertise combined, we did make avoidable mistakes along the way, including overly simplistic early implementations of some relational operators, not incorporating all datatypes early on in the engine, not early-enough focus on resource management, postponing work on comprehensive date and time functionality etc. Also, our continuous focus on avoiding tuning knobs raised a series of engineering challenges, ultimately bringing about many exciting technical solutions. As a result, today, Snowflake has only one tuning parameter: how much performance the user wants (and is willing to pay for).

While Snowflake's performance is already very competitive, especially considering the no-tuning aspect, we know of many optimizations that we have not had the time for yet. Somewhat unexpected though, core performance turned out to be almost never an issue for our users. The reason is that elastic compute via virtual warehouses can offer the performance boost occasionally needed. That made us focus our development efforts on other aspects of the system.

The biggest technical challenges we face are related to the SaaS and multi-tenancy aspects of the system. Building a metadata layer that can support hundreds of users concurrently was a very challenging and complex task. Handling various types of node failures, network failures, and supporting services is a never-ending fight. Security has been and will continue to be a big topic: protecting the system and the users' data from external attacks, the users themselves, and our internal users as well. Maintaining a live system of hundreds of nodes running millions of queries per day, while bringing a lot of satisfaction, requires a highly integrated approach to development, operations, and support.

Snowflake users continue to throw increasingly bigger and more complex problems at the system, influencing its evolution. We are currently working on improving the data access performance by providing additional metadata structures and data re-organization tasks—with a focus on minimal to no user interaction. We continue to improve and extend core query processing functionality, both for standard SQL and semi-relational extensions. We plan to further improve the skew handling and load balancing strategies, whose importance increases along with the scale of our users' workloads. We work on solutions simplifying workload management to the users, making the system even more elastic. And we work on integration with external systems, including issues such as high-frequency data loading.

The biggest future challenge for Snowflake is the transition to a full self-service model, where users can sign up and interact with the system without our involvement at any phase. It will bring a lot of security, performance, and support challenges. We are looking forward to them.

7. ACKNOWLEDGMENTS

Snowflake is the work of far too many people to list here. We would like to thank the entire Snowflake engineering team for their contributions to the product, and for all the hard work, effort, and pride they put into it. We also would like to thank all the other "Snowflakes" for their amazing work in bringing this product to the users and building a great company together. We are continuously impressed and humbled to work with such an excellent team.

8. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. SIGMOD*, 2008.
[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. VLDB*, 2001.
[3] S. Alsubaiee et al. AsterixDB: A scalable, open source DBMS. *PVLDB*, 7(14):1905–1916, 2014.
[4] Amazon Elastic Compute Cloud (EC2). aws.amazon.com/ec2/instance-types.
[5] Amazon Simple Storage Service (S3). aws.amazon.com/s3.
[6] Apache Cassandra. cassandra.apache.org.
[7] Apache Drill. drill.apache.org.
[8] Apache Hadoop. hadoop.apache.org.
[9] Apache Hive. hive.apache.org.
[10] Apache Parquet. parquet.apache.org.
[11] Apache Spark. spark.apache.org.
[12] AWS CloudHSM. aws.amazon.com/cloudhsm.
[13] E. Barker. *NIST SP 800-57 – Recommendation for Key Management – Part 1: General (Revision 4)*, chapter 7. 2016.
[14] J. Barr. AWS Import/Export Snowball – Transfer 1 petabyte per week using Amazon-owned storage appliances. aws.amazon.com/blogs/aws/aws-importexport-snowball-transfer-1-petabyte-per-week-using-amazon-owned-storage-appliances/, 2015.
[15] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. CIDR*, 2005.
[16] V. R. Borkar, M. J. Carey, and C. Li. Big data platforms: What's next? *ACM Crossroads*, 19(1):44–49, 2012.
[17] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. SIGMOD*, 2008.
[18] B. Calder et al. Windows Azure Storage: A highly available storage service with strong consistency. In *Proc. SOSP*, 2011.
[19] Cassandra Query Language (CQL). cassandra.apache.org/doc/cql3/CQL.html.
[20] Cloud Storage – Google Cloud Platform. cloud.google.com/storage.
[21] Cloudera Impala. impala.io.
[22] Couchbase N1QL. couchbase.com/n1ql.
[23] Couchbase Server. couchbase.com.
[24] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *Proc. SIGMOD*, 2013.
[25] D. J. DeWitt, S. Madden, and M. Stonebraker. How to build a high-performance data warehouse. db.csail.mit.edu/madden/high_perf.pdf, 2006.
[26] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control*. Artech House Publishers, 2003.
[27] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE TKDE*, 6(1), 1994.
[28] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.
[29] G. Graefe. Fast loads and fast queries. In *Data Warehousing and Knowledge Discovery*, volume 5691 of *LNCS*. Springer, 2009.
[30] A. Gupta et al. Amazon Redshift and the case for simpler data warehouses. In *Proc. SIGMOD*, 2015.
[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. STOC*, 1997.
[32] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber. Optimizing write performance for read optimized databases. In *Proc. DASFAA*, 2010.
[33] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.
[34] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.
[35] Microsoft Analytics Platform System. www.microsoft.com/en-us/server-cloud/products/analytics-platform-system.
[36] Microsoft Azure Blob Storage. azure.microsoft.com/en-us/services/storage/blobs.
[37] Microsoft Azure SQL DW. azure.microsoft.com/en-us/services/sql-data-warehouse.
[38] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. VLDB*, 1998.
[39] MongoDB. mongodb.com.
[40] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE TSE*, 16(5):558–560, 1990.
[41] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
[42] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. SIGMOD*, 2009.
[43] Presto. prestodb.io.
[44] K. Sato. An inside look at Google BigQuery. cloud.google.com/files/BigQueryTechnicalWP.pdf, 2012.
[45] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.
[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. MSST*, 2010.
[47] SQL DW Concurrency. azure.microsoft.com/en-us/documentation/articles/sql-data-warehouse-develop-concurrency.
[48] Stinger.next: Enterprise SQL at Hadoop scale. hortonworks.com/innovation/stinger.
[49] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proc. SIGMOD*, 2014.