Amazon Redshift and the Case for Simpler Data Warehouses

Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, Vidhya Srinivasan
Amazon Web Services

Abstract

Amazon Redshift is a fast, fully managed, petabyte-scale data warehouse solution that makes it simple and cost-effective to efficiently analyze large volumes of data using existing business intelligence tools. Since launching in February 2013, it has been Amazon Web Service's (AWS) fastest growing service, with many thousands of customers and many petabytes of data under management.

Amazon Redshift's pace of adoption has been a surprise to many participants in the data warehousing community. While Amazon Redshift was priced disruptively at launch, available for as little as $1000/TB/year, there are many open-source data warehousing technologies and many commercial data warehousing engines that provide free editions for development or under some usage limit. While Amazon Redshift provides a modern MPP, columnar, scale-out architecture, so too do many other data warehousing engines. And, while Amazon Redshift is available in the AWS cloud, one can build data warehouses using EC2 instances and the database engine of one's choice with either local or network-attached storage.

In this paper, we discuss an oft-overlooked differentiating characteristic of Amazon Redshift – simplicity. Our goal with Amazon Redshift was not to compete with other data warehousing engines, but to compete with non-consumption. We believe the vast majority of data is collected but not analyzed. We believe, while most database vendors target larger enterprises, there is little correlation in today's economy between data set size and company size. And, we believe the models used to procure and consume analytics technology need to support experimentation and evaluation. Amazon Redshift was designed to bring data warehousing to a mass market by making it easy to buy, easy to tune and easy to manage while also being fast and cost-effective.

1. Introduction

Many companies augment their transaction-processing database systems with data warehouses for reporting and analysis. Analysts estimate the data warehouse market segment at 1/3 of the overall relational database market segment ($14B vs. $45B for software licenses and support), with an 8-11% compound annual growth rate (CAGR)[1]. While this is a strong growth rate for a large, mature market, over the past ten years, analysts also estimate data storage at a typical enterprise growing at 30-40% CAGR. Over the past 12-18 months, new market research has begun to show an increase to 50-60%, with data doubling in size every 20 months.

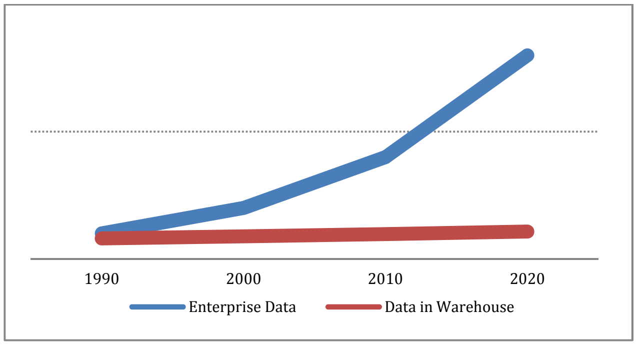

Figure 1: Data Analysis Gap in the Enterprise [10]

This implies most data in an enterprise is "dark data"[2]: data that is collected but not easily analyzed. We see this as unfortunate. If our customers didn't see this data as having value, they would not retain it. Many companies are trying to become increasingly data-driven. And yet, not only is most data already dark, the overall data landscape is only getting darker. Storing this data in NoSQL stores and/or Hadoop is one way to bridge the gap for certain use cases. However it doesn't address all scenarios.

In our discussions with customers, we found the "analysis gap" between data being collected and data available for analysis as due to four major causes.

1. Cost – Most commercial database solutions capable of analyzing data at scale require significant up-front expense. This is hard to justify for large datasets with unclear value.

2. Complexity – Database provisioning, maintenance, backup, and tuning are complex tasks requiring specialized skills. They require IT involvement and cannot easily be performed by line of business data scientists or analysts.

3. Performance – It is difficult to grow a data warehouse without negatively impacting query performance. Once built, IT teams sometimes discourage augmenting data or adding queries as a way of protecting current reporting SLAs.

4. Rigidity – Most databases work best on highly structured relational data. But a large and increasing percentage of data consists of machine-generated logs that mutate over time, audio and video, not readily accessible to relational analysis.

We see each of the above issues only increasing with data set size. To take one large-scale customer example, the Amazon Retail team collects about 5 billion web log records daily (2TB/day,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author. Copyright is held by the owner/author(s).
SIGMOD '15, May 31–June 4, 2015, Melbourne, Victoria, Australia
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2742795

[1] Forecast data combined from IDC, Gartner, and 451 Research.
[2] http://www.gartner.com/it-glossary/dark-data

growing 67% YoY). It is very valuable to combine this data with transactional data stored in their relational and NoSQL systems. Using an existing scale-out commercial data warehouse, they were able to analyze 1 week of data per hour and maintain a cap of 15 months of log, using the largest configuration available. Using much larger Hadoop clusters, they were able to analyze up to 1 month of data per hour, though these clusters were very expensive to administer.

Benchmarks

By augmenting their existing systems with a petabyte-scale Amazon Redshift cluster, the Amazon Enterprise Data Warehouse team was able to perform their daily load (5B rows) in 10 minutes, load a month of backfill data (150B rows) in 9.75 hours, take a backup in 30 minutes and restore it to a new cluster in 48 hours. Most interestingly, they were able to now run queries that joined 2 trillion rows of click traffic with 6 billion rows of product ids in less than 14 minutes, an operation that didn't complete in over a week on their existing systems.

Techniques built in Redshift

Some of the above results emerge from Amazon Redshift's use of, by now, familiar data warehousing techniques, including columnar layout, per-column compression, co-locating compute and data, co-locating joins, compilation to machine code and scale-out MPP processing. While these techniques have not fully made their way into the largest data warehousing vendors, Amazon Redshift's approach is very close to that taken by other columnar MPP data warehousing engines.

Design Goals

Beyond these techniques used to scale out query processing and to reduce CPU, disk IOs, network traffic, Amazon Redshift had a number of additional design goals:

1. Minimize cost of experimentation – It is difficult to understand whether data is worth analyzing without performing at least a few experiments. Amazon Redshift provides a free trial allowing customers to evaluate the service for 60 days using up to 160GB of compressed SSD data at no charge. For subsequent experiments, they can spin up a cluster with no commitments for $0.25/hour/node. This is inclusive of hardware, software, maintenance and management.

2. Minimize time to first report – We measure the time it takes our customers to go from deciding to create a cluster to seeing the results of their first query. As seen in Figure 2, this can be as little as 15 minutes, even to provision a multi-PB cluster.

3. Minimize administration – Amazon Redshift tries to reduce the undifferentiated heavy lifting of creating and operating a data warehouse. We automate backup, restore, provisioning, patching, failure detection and repair. Advanced operations like encryption, cluster resizing and disaster recovery require only a few clicks to enable.

4. Minimize scaling concerns - Many data warehousing engines provide parallelization of loads and queries. We additionally try to parallelize administrative operations like cluster creation, patching, backup, restore and cluster resize. Pricing is also linear or sub-linear with data set growth.

5. Minimize tuning – We avoid knobs and apply techniques to make the ones we do provide "dusty" with disuse. For most users the default setting for these knobs as determined by the system would be sufficient. However, advanced users can choose to change it as needed. For example, we automatically pick compression types based on data sampling. We also avoid the use of indexing or projections, instead favoring multi-dimensional z-curves.

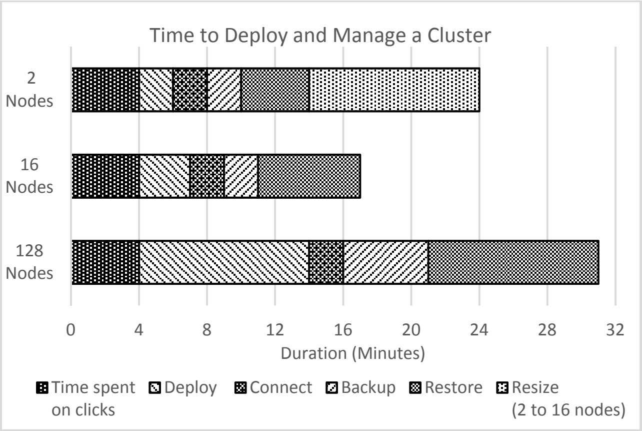

Figure 2: Common admin operation execution time by size

Benifits for being Managed Service in AWS

Amazon Redshift also obtains a number of advantages by operating as a managed service in the AWS cloud.

First and foremost, we are able to leverage the pricing and scale of Amazon EC2 for our instances, the durability and throughput of Amazon S3 for backup, and the security of Amazon VPC, to name just a few. This enables our customers to benefit from the innovation and price reductions delivered across a much larger team than ours alone.

Second, by making deployments and patching automatic and painless for our customers, we are able to deploy software at a high frequency. We have averaged the addition of one feature per week, over the past two years. Reducing the cost of deployment and the rapid feedback mechanisms available in running a service allows us to run a more agile OODA[3] loop compared to on-premise providers, ensuring we quickly provide the features our customers care most about.

Finally, since we manage a fleet of thousands of Amazon Redshift clusters, we benefit from automating capabilities that would not make economic sense for any individual on-premise DBA. An operation requiring one minute of per-node overhead is worth many days of investment for us to remove. Removing the root cause of a defect that causes an individual cluster to fail and restart once per thousand days is well worth our investment to reduce paging. As we scale, we have to continually improve the reliability, availability and automation across our fleet to manage operations, which makes failures increasingly uncommon for our customers.

In this paper, we discuss Amazon Redshift's system architecture, simplicity as a guiding principle, and lessons we've learned over the past two years operating the service.

2. System Architecture

The Amazon Redshift architecture consists of a **data plane**, the database engine providing data storage and SQL execution, a **control plane**, providing the workflows to monitor and manage the database, and other AWS services we depend on to support data plane and control plane execution. We summarize each below.

[3] http://en.wikipedia.org/wiki/OODA_loop

2.1 Data Plane

The Amazon Redshift engine is a SQL-compliant, massively-parallel, query processing and database management system designed to support analytics workload. We consider analytic workloads those that regularly ingest incremental sets of data that can be large in size and run queries that join, scan, filter and aggregate data that can comprise a significant fraction of the total stored data. The initial version of the engine was derived from a code base licensed from ParAccel. Data storage and compute is distributed across one or more nodes, which provides near-linear scalability for maintaining & querying datasets from 100s of gigabytes on up to petabyte scale. An Amazon Redshift cluster is comprised of a leader node and one or more compute nodes. We also support a single-node design where leader and compute work is shared on a single node.

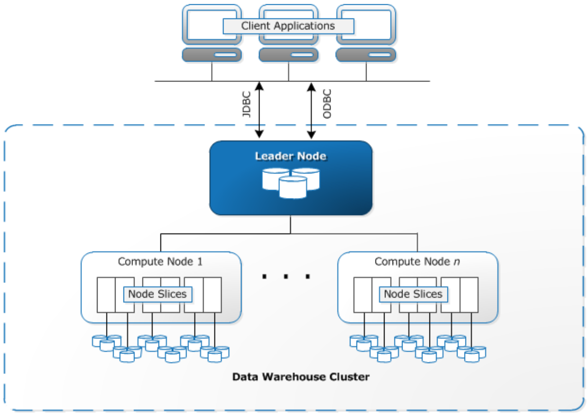

Figure 3: Amazon Redshift system architecture

The **leader node** accepts connections from client programs, parses requests, generates & compiles query plans for execution on the compute nodes, performs final aggregation of results when required, and coordinates serialization and state of transactions. The compute node(s) perform the heavy lifting inherent in both query processing and data manipulation against local data.

Distributed Data Across Nodes

Node is partioned into slices

Data stored within each Amazon Redshift table is automatically distributed both across compute nodes, to enable scale out of large data sets, and within a compute node, to reduce contention across processing cores. A compute node is partitioned into slices; one slice for each core of the node's multi-core processor. Each slice is allocated a portion of the node's memory and disk space, where it processes a portion of the workload assigned to the node. The user can specify whether data is distributed in a round robin fashion, hashed according to a distribution key, or duplicated on all slices. Using distribution keys allows join processing on that key to be co-located on individual slices, reducing IO, CPU and network contention and avoiding the redistribution of intermediate results during query execution. Within each slice, data storage is column-oriented. Each column within each slice is encoded in a chain of one or more fixed size data blocks. The linkage between the columns of an individual row is derived by calculating the logical offset within each column chain. This linkage is stored as meta-data.

Slice contains data in columnar way

Data Replication Strategy

Data blocks are replicated both within the database instance and within Amazon Simple Storage Service (S3). Each data block is synchronously written to both its primary slice as well as to at least one secondary on a separate node. Cohorting is used to limit the number of slices impacted by an individual disk or node failure. Here, we attempt to balance the resource impact of re-replication against the increased probability of correlated failures as disk and node counts increase. Data blocks are also asynchronously and automatically backed up to Amazon S3, which is designed to provide 99.9999999% durability by storing multiple copies across multiple data centers. The primary, secondary and Amazon S3 copies of the data block are each available for read, making media failures transparent. Loss of durability requires multiple faults to occur in the time window from the first fault to re-replication or backup to Amazon S3. Customers can also choose to have their backups occur to Amazon S3 in a second region for further protection against disasters.

Query Processing

Query processing within Amazon Redshift begins with query plan generation and compilation to C++ and machine code at the leader node. The use of query compilation adds a fixed overhead per query that we feel is generally amortized by the tighter execution at compute nodes vs. the overhead of execution in a general-purpose set of executor functions. The executable and plan parameters are sent to each compute node participating in the query. At the compute nodes, the executable is run with the plan parameters and the intermediate results are sent back to the leader node for final aggregation. Each slice in the compute node may run multiple operations such as scanning, filtering, processing joins, etc., in parallel.

Data Loading

Data loading is a special case of query processing, using a modified version of the PostgreSQL COPY command. The Amazon Redshift version of COPY provides direct access to load data from Amazon S3, Amazon DynamoDB, Amazon EMR, or over an arbitrary SSH connection. COPY is parallelized across slices, with each slice reading data in parallel, distributing as needed, and sorting locally. By default, compression scheme and optimizer statistics are updated with load. While customers can override these, they represent some of the dustier knobs still remaining in the system. COPY also directly supports ingestion of JSON data as well as data that is encrypted and/or compressed.

2.2 Control Plane

Host Manager

In addition to the database engine software itself, each Amazon Redshift node has host manager software that helps with deploying new database engine bits, aggregating events and metrics, generating instance-level events, archiving and rotating logs, and monitoring the host, database and log files for errors. The host manager also has limited capability to perform actions, for example, restarting a database process on failure.

Most control plane actions are coordinated off-instance by a separate Amazon Redshift control plane fleet. These nodes are responsible for fleet-wide monitoring and alarming as well as initiating maintenance tasks based on telemetry from instance host managers or actions requested by end-customers through the console or API. Example tasks would include node replacements, cluster resize, backup, restore, provisioning, patching, etc.

2.3 Dependent AWS Services

In addition to the core Amazon Redshift software itself, we leverage multiple AWS services, most significantly Amazon Elastic Compute Cloud (EC2) for instances, Amazon S3 for backup, Amazon Simple Workflow (SWF) for control plane actions, Amazon CloudWatch for customer instance metrics, Amazon Simple Notification Service (SNS) for customer alarms, Amazon VPC for network isolation, Amazon Route53 for DNS lookup, AWS CloudTrail for audit logging, and AWS Key Management Service and AWS CloudHSM for key management.

We additionally leverage a number of internal AWS services for additional capabilities including deployment, short-term credentials, log harvesting, load balancing, metering, etc.

The use of these services has significantly accelerated Amazon Redshift development, both as we have avoided the need to build robust, fault-tolerant and secure components where available in other AWS services, as well as providing us with access to an ongoing innovation stream by our sister development teams. For example, Amazon EC2 has significantly enhanced intrusion detection, network QoS, packets per second, server health monitoring, and IO queue management over the past two years – all capabilities we have absorbed with no changes to our engine.

The overall Amazon Redshift system architecture can be seen as the integration of traditional parallel, distributed relational database architecture with service-oriented architecture for management. Our greatest differentiation occurs when we are able to leverage these together.

For example, many database providers provide backup integration to cold storage. However, they are rarely able to expect the availability of any specific provider. By operating in the AWS cloud, Amazon Redshift is able to rely upon the specific characteristics of Amazon S3 availability, durability and access APIs. This has allowed us to entirely automate backup, making it continuous, incremental and automatic, and removing the need for customer attention. More significantly, we are able to include Amazon S3 backups as part of our data availability and durability design, by doing block-level backups and "page-faulting" in blocks when unavailable on local storage. This also allowed us to implement a streaming restore capability, allowing the database to be opened for SQL operations after metadata and catalog restoration, but while blocks were still being brought down in background. Since the average working set for a data warehouse is a small fraction of the total data stored, this allows performant queries to be obtained in a small fraction of the time required for a full restore. A meaningful percentage of Amazon Redshift customers delete their clusters every Friday and restore from backup each Monday.

3. The Case for Simplicity

We believe the success of SQL-based databases came in large part from the significant simplifications they brought to application development through the use of declarative query processing and a coherent model of concurrent execution. The introduction of data warehousing and later columnar layouts extended this by simplifying schema design and reducing the expense of join processing. And while customers will always demand more powerful capabilities (e.g., consider the rise in popularity of the MapReduce paradigm), our approach to this customer pull is to balance it with the belief that additional power generally requires additional education and understanding. As one of our customers put it, 'I want a relationship with my data, not my database.'

3.1 Simplifying the Purchase Decision Process

'Time to first report' is a key metric for our team. We see the clock starting at the point a customer first looks at our website to evaluate our service and stopping when they are first able to issue a query and obtain a result from their database instance. We take a retail mindset to this, understanding one of our customers looking for data management technology on AWS is the same person looking for a laptop or CD on Amazon.com. Many of the same techniques in ecommerce applications apply. For example, one might look at click trails through web pages and evaluate where abandonment occurs. These may not feel like software development concerns, but we have found that they impact product decisions. It helped us to have launched using the standard PostgreSQL ODBC/JDBC drivers, providing confidence that our customers existing tools ecosystem would largely work. Our linear pricing model (with discounts for commitment) has informed how we scale out to support larger databases. Reducing the up-front steps required to create and configure a database has reduced abandonment.

Looking at the specific process of cluster creation, our analogue to "package delivery", we've limited the information required to number and type of nodes, basic network configuration and administrative account information. We are working to reduce this further. At launch time, cluster creation times averaged 15 minutes, which we viewed as a significant benefit compared to on-premise database provisioning times. Some months later, we introduced support for preconfigured Amazon Redshift nodes available for faster creations and supporting standbys for node failure replacements. These reduced provisioning time to 3 minutes, and meaningfully reduced abandonment.

Reduced cluster provisioning time by providing pre-configured node templates

We have found reducing the cost of an error to be as important as improving delivery time. Our customers can more readily experiment if they can easily "return" or "exchange" their database. Customers creating their first database cluster will automatically get enough free hours for their first two months to continually run a database supporting 160GB of compressed SSD data. Beyond this level, the use of hourly pricing enables experimentation and trial of the service by reducing commitment. While pricing may seem orthogonal to product design, a service involves hardware capital expense and operations personnel and software design can meaningfully impact both.

For example, at any time, customers can resize their clusters up or down or to a different instance type, removing the need for up-front capacity and performance estimation. Underneath the covers, we provision a new cluster, put the original cluster in read-only mode, and run a parallel node-to-node copy from source cluster to target. The source cluster is available for reads until the operation completes, at which time, we move the SQL endpoint and decommission the source.

3.2 Simplifying Database Administration

Most Amazon Redshift customers do not have a designated DBA to administer their database. This reduces their costs and enables them to allocate resources towards higher value work with their data. Our service manages much of the undifferentiated heavy lifting involved in database administration, including provisioning, patching, monitoring, repair, backup and restore.

We believe database administration operations should be as declarative as queries, with the database determining parallelization and distribution. Amazon Redshift operations are data-parallel within the cluster, as well cluster-parallel for fleet-wide actions such as patching and monitoring. For example, the time required to backup an entire cluster is proportional to the data changed on a single node. System backups are taken automatically and are automatically aged out. User backups leverage the blocks already backed up in system backups and are kept until explicitly deleted. Disruptions to cloud infrastructure get wide publicity so some customers ask for disaster recovery by storing backups in a second region. In Amazon Redshift, that only requires setting a checkbox and specifying the region. Within the system, we will initiate backing up data blocks to both the local and the remote region. Disaster recovery backups have the same streaming restore capabilities as local backups, allowing customers to start issuing queries within minutes of starting a cluster in the remote region.

Encryption is similarly straightforward. Enabling encryption requires setting a checkbox in our console and, optionally, specifying a key provider such as a hardware security module (HSM). Under the covers, we generate block-specific encryption keys (to avoid injection attacks from one block to another), wrap these with cluster-specific keys (to avoid injection attacks from one cluster to another), and further wrap these with a master key, stored by us off-network or via the customer-specified HSM. All user data, including backups, is encrypted. Key rotation is straightforward as it only involves re-encrypting block keys or cluster keys, not the entire database. Repudiation is equally straightforward, as it only involves losing access to the customer's key or re-encrypting all remaining valid cluster keys with a new master. We also benefit from security features in the core AWS platform. For example, we use Amazon VPC to provide network isolation of the compute nodes providing cluster storage, isolating them from general-purpose access from the leader node, which is accessible from the customer's VPC.

Future work will remove the need for user-initiated table administration operations, making them closer to backup in operation. The database should be able to determine when data access performance is degrading and take action to correct itself when load is otherwise light.

3.3 Simplifying Database Tuning

Amazon Redshift has few tuning knobs in comparison to other database engines. We view this as a plus, removing burden from our customers in favor of putting responsibility on ourselves. The main things set by a customer are instance type and number of nodes for the database cluster, and sort and distribution model used for individual tables.

While customers can set other parameters, such as column compression type, we strive to make such knobs dusty with disuse, by simply setting them accurately ourselves. The database generally has as much or more information as available to the customer to set these well, including query patterns, data distribution and cost of compression.

We are striving to make other settings, such as sort column and distribution key equally dusty. One technique we are applying is to reduce the cost of a suboptimal decision. For example, a missing projection can result in a full table scan while an additional one can greatly impact load time. By comparison, a multidimensional index [JAO] using z-curves degrades more gracefully with excess participation and still provides utility if leading columns are not specified. Similarly, z-curves can reduce the span of nodes involved in a join, rather than making them entirely local or fully distributed.

Similarly, relaxation of how we map data and query computation to nodes and slices would allow a more elastic system, which could grow and shrink as load required.

4. Customer Use Cases

Below, we touch on some of the ways customers are using Amazon Redshift. What connects many of these and other cases is SQL. The ability to declaratively state one's intent and have it automatically converted into an optimized execution plan that is resilient to changes in access patterns and data distribution is a very significant benefit compared to other data-intensive computing techniques. This is only yet more important when computation needs to be distributed and parallelized across many nodes, and resources distributed across many concurrent queries. A common theme across these use cases for forward work is data movement and transformation, which today seems to lack the simplicity and power of in-database processing using declarative SQL. A standardized declarative model for data structure identification and transformation would be valuable.

Enterprise data warehousing: Many customers utilize Amazon Redshift for what we consider to be the traditional enterprise data warehousing use case. They populate data from a set of source relational databases, ingest at an hourly or nightly cadence, and access data through BI tools. These customers have appreciated the simplicity and transparency of the procurement process, the ability to evaluate the service at minimal cost, and the ability to use their existing ecosystem of BI and ETL tools. They are generally struggling with the maintenance overhead for their existing systems and see the value of managed systems that take care of undifferentiated heavy lifting.

When migrating from other systems, we find that our customers like their existing databases, but do not always have great relationships with their database vendors. They often come to us when faced with a decision point on their existing system, either being forced to upgrade hardware for a managed system, coming to the end of a term license and about to enter a protracted negotiation cycle, or reaching the scaling limit on one engine and being forced to migrate to a second from the same vendor with somewhat different SQL semantics and maintenance. These have guided us towards reducing "sharp edges" in our own service, by making scaling linear and straightforward, increasing compatibility with PostgreSQL and growing the tools ecosystem we support.

Semi-structured "Big Data" analysis: Many customers also use Amazon Redshift for the integrated analysis of log and transaction data. We see a number of customers migrating away from HIVE on Hadoop and obtain much better performance at a much lower cost. They are also able to make their systems directly available to business analysts in their organizations using SQL or BI tools, rather than burdening their engineers and data scientists with the responsibility to generate these reports. For these customers, simplicity is a key driver, as they generally are not staffed with DBAs to manage and maintain their systems. We see this use case as where the bulk of "dark data" resides, and see many opportunities to further simplify their lives. For example, we could support transient data warehouses on a source 'data lake' or automatically 'relationalizing' source semi-structured data into tables for efficient query execution.

Data Transformation: An increasing number of Amazon Redshift customers use the service as part of a data processing pipeline, taking large amounts of raw data, dropping it into the data warehouse to run large SQL jobs that generate output tables that they can then use in their online business. An example would be in ad-tech, where many billion ad impressions may be distilled into lookup tables that informs an ad exchange online service. We have also seen customers begin to directly integrate Amazon Redshift into their customer facing screens under analytic reports and graphs. We see these customers moving towards SQL for the benefits of being able to straightforwardly and declaratively indicate intent and have the underlying system perform the parallel query decomposition. We see a similar trend in the SQL on Hadoop community, with SQL being used to reduce the labor involved in writing Map Reduce jobs. Speed and expressibility are key attributes here, for example, guiding our work on approximate functions. In time, we would like to build distributed approximate equivalents for all non-linear exact operations within our engine.

"Small Data" - A large number of Amazon Redshift customers have never previously used a data warehouse, instead directly running their reports against their source transaction systems. Amazon Redshift's cost structure and removal of overhead administration has enabled these customers to create data warehouses and obtain the benefits of improved performance, OLTP system offload, and retention of history. These customers are used to a short time lag to source data changes, so automated change data capture and automatic schema creation and maintenance are important capabilities.

5. Lessons Learned

Amazon Redshift has grown rapidly since it became generally available in February 2013. While our guiding principles have served us well over the past two years, we now manage many thousands of database instances and below offer some lessons we have learned from operating databases at scale.

Design escalators, not elevators: Failures are common when operating large fleets with many service dependencies. A key lesson for us has been to design systems that degrade on failures rather than losing outright availability. These are a common design pattern when working with hardware failures, for example, replicating data blocks to mask issues with disks. They are less common when working with software or service dependencies, though still necessary when operating in a dynamic environment. Amazon overall (including AWS) had 50 million code deployments over the past 12 months. Inevitably, at this scale, a small number of regressions will occur and cause issues until reverted. It is helpful to make one's own service resilient to an underlying service outage. For example, we support the ability to preconfigure nodes in each data center, allowing us to continue to provision and replace nodes for a period of time if there is an Amazon EC2 provisioning interruption. One can locally increase replication to withstand an Amazon S3 or network interruption. We are adding similar mitigation strategies for other external dependencies that can fail independently from the database itself.

Continuous delivery should be to the customer: Many engineering organizations now use continuous build and automated test pipelines to a releasable staging environment. However, few actually push the release itself at a frequent pace. While customers would prefer small patches to large ones for the same reasons engineering organizations prefer to build and test continuously, patching is an onerous process. This often leads to special-case, one-off patches per customer that are limited in scope – while necessary, they make patching yet more fragile.

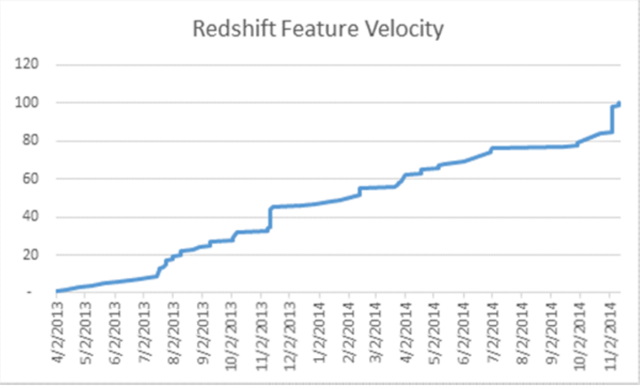

Figure 4: Cumulative features deployed over time

Amazon Redshift is set up to automatically patch customer clusters on a weekly basis in a 30-minute window specified by the customer. Patches are reversible and will automatically be reversed if we see an increase in errors or latency in our telemetry. At any point, a customer will only be on one of two patch versions, greatly improving our ability to reproduce and diagnose issues. We typically push new database engine software, including both features and bug fixes, every two weeks. We have found reducing this pace, for example to every four weeks, meaningfully increased the probability of a failed patch.

Use Pareto analysis to schedule work: In a rapidly growing service, operational load can easily overwhelm development capacity. We page ourselves on each database failure, understanding that, even if not a widespread concern, each issue is meaningful to the customer experiencing it. In Figure 5, Sev 2 refers to a severity 2 alarm that causes an engineer to get paged. This means operational load roughly correlates to business success. Within Amazon Redshift, we collect error logs across our fleet and monitor tickets to understand top ten causes of error, with the aim of extinguishing one of the top ten causes of error each week.

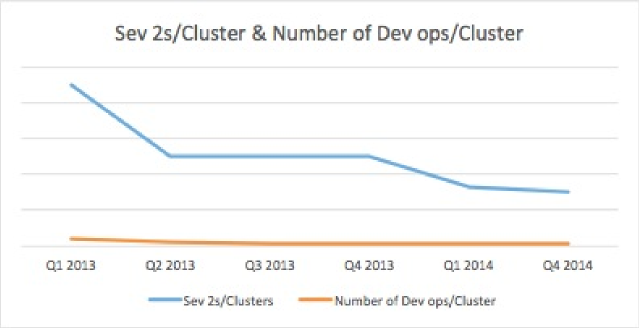

Figure 5: Tickets per cluster over time

Pareto analysis is equally useful in understanding customer functional requirements. However, it is more difficult to collect. We manage this by simply conducting over 1000 direct one-to-one conversations with customers each year. This provides a clear sample of customer needs and service gaps, providing us with actionable telemetry towards scheduling feature development. In future, we would like to add automated collection of usage statistics by feature, query plan shapes, etc. across our fleet.

6. Related Work

While Amazon Redshift, when launched, was the first widely available data warehouse-as-a-service, its core database technology (parser, optimizer, engine, storage organization, MPP architecture) was derived from technology licensed from ParAccel. ParAccel belongs to a group of column-oriented DBMS products that appeared from the middle through the end of the 2000s: Vertica, Ingres VectorWise, Infobright, Kickfire, and many others [1]. These systems had several similarities in their design philosophy and list of features, with many of them influenced by two pioneering modern column-store systems: C-Store [8] and MonetDB/X100 [3].

Redshift's compression techniques are similar to those used by Vertica, and their performance tradeoffs are well understood [2]. Redshift foregoes traditional indexes (or projections in C-Store/Vertica) and instead focuses on sequential scan speed through compiled code execution and column-block skipping based on value-ranges stored in memory. Infobright's Knowledge Grid and Netezza's Zone Maps also rely on block skipping which as a technique was first discussed in [5]. Code compilation techniques in query execution have received renewed attention in academia [6][9] and are also used in other systems such as Microsoft's Hekaton [4].

7. Conclusion

Amazon Redshift's price, performance, and simplicity extend the use cases for data warehousing beyond traditional enterprise data warehousing, into big data and software-as-service applications with embedded analytics. Unlike traditional data warehouse systems, which often require large upfront payments and months of vendor negotiations, hardware procurement and deployments, Amazon Redshift clusters can be provisioned in minutes, enabling customers to get started with no commitments and scale up to a petabyte scale cluster. Redshift also offers automated patching, provisioning, scaling, securing, backup, restore, and a comprehensive set of security features such as encryption at rest, in transit, HSM integration and audit logging.

By dramatically lowering the cost and effort associated with deploying data warehousing systems without compromising on features and performance, Amazon Redshift is not only changing how traditional enterprises think about data warehousing but also making data warehousing technology available to segments that had not previously considered it. This is evident in Redshift's customer base, which ranges from enterprises like NTT DOCOMO and Amazon.com with multi-petabyte systems, to high-scale startups like Pinterest and Flipboard with hundreds of terabytes, to small startups with hundreds of gigabytes in their data warehouses.

8. Acknowledgements

We first would like to acknowledge the contributions of each engineer on the Amazon Redshift team, who worked hard to deliver the first version, and continue to drive a fast pace of innovation while maintaining operational excellence of the service for our customers. We would also like to thank Raju Gulabani for his guidance through the launch of Amazon Redshift and its subsequent operation.

Amazon Redshift has lineage in the database engine work done by the ParAccel team, now at Actian, which in turn benefited from work from the PostgreSQL community. Their work was a very significant accelerant in the delivery of Amazon Redshift.

We also thank Christopher Olston and Sihem Amer-Yahia for shepherding this paper and all the reviewers for their valuable comments.

9. References

[1] Abadi D, Boncz P, Harizopoulos S, Idreos S, Madden S. The Design and Implementation of Modern Column-Oriented Database Systems. Foundations and Trends in Databases. 2013;5(3):197-280.

[2] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Proceedings of the ACM SIGMOD Conference on Management of Data, pages 671–682, 2006.

[3] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper- pipelining query execution. In Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR), 2005.

[4] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. SIGMOD Conference 2013: 1243-1254.

[5] Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. VLDB '98 Proceedings of the 24rd International Conference on Very Large Data Bases. Pages 476-487.

[6] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 2011, Seattle, USA.

[7] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. PODS*, pages 181–190, 1984.

[8] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexan- der Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. In Proceedings of the International Conference on Very Large Data Bases (VLDB), pages 553–564, 2005.

[9] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. compilation in query execution. In DaMoN, pages 33–40, 2011.

[10] Gartner : User Survey Analysis: Key Trends Shaping the Future of Data Center Infrastructure Through 2011
IDC: Worldwide Business Analytics Software 2012–2016 Forecast and 2011 Vendor Shares.