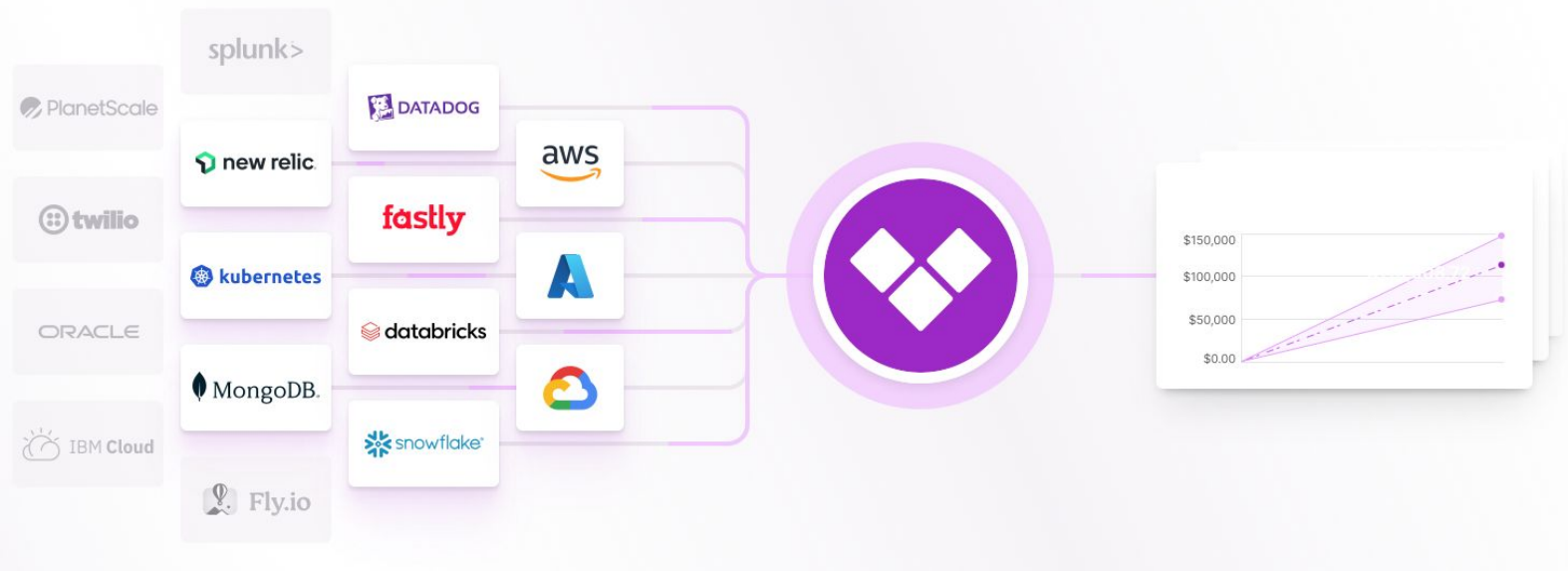




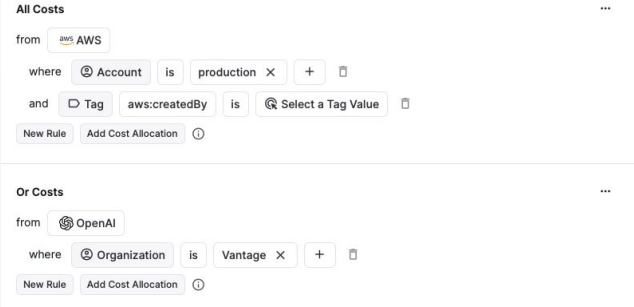
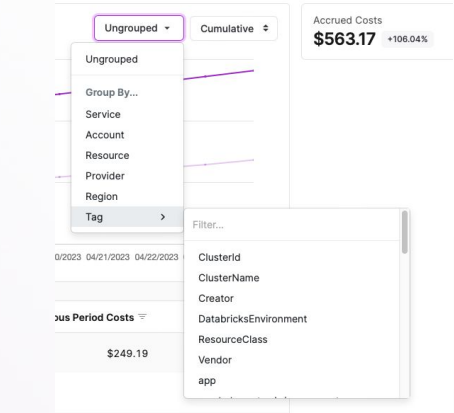
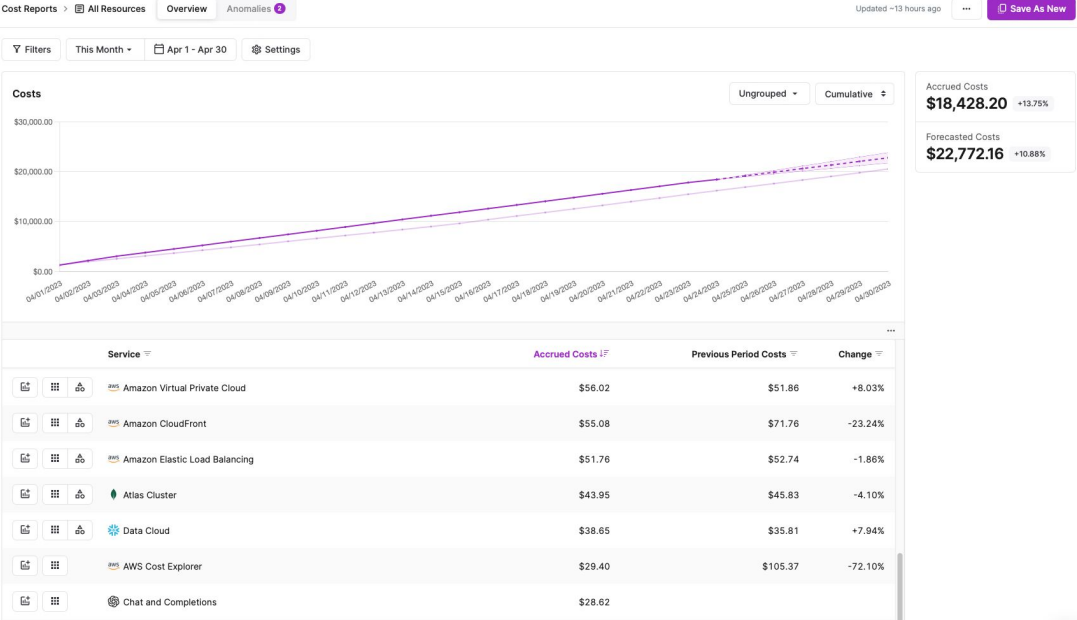
Our Journey from Redshift to Clickhouse Cloud

Vantage

Vantage is a FinOps Platform. We ingest cost data from various infrastructure providers and combine it in a single platform for analysis, debugging and optimization.



Vantage




Vantage

[All Resources](#) > [Amazon Simple Storage Service](#) > By Category ▾

...				
Service Category ▾		Accrued Costs ▾	Previous Period Costs ▾	Change ▾
	 API Request	\$255.01	\$123.27	+106.87%
	 Storage	\$252.39	\$125.92	+100.44%
	 Data Transfer	\$0.01	\$0.01	

[All Resources](#) > [Amazon Simple Storage Service](#) > By Resource ▾ > uploads.vantage.sh > Storage

...				
Service Category ▾		Accrued Costs ▾	Previous Period Costs ▾	Change ▾
	TimedStorage-ByteHrs	\$0.01	\$0.01	

Cost Data

Cost data comes in many formats, but the most common is the AWS Cost and Usage Report. These are delivered about every 8-12 hours for specific billing period and by the end of the month can be hundreds of millions of records for a single account.

identity/LineItemId	identity/TimeInterval	bill/BillingPeriodStartDate	bill/BillingPeriodEndDate	lineitem/LineItemType	lineitem/UsageStartDate	lineitem/UsageEndDate	lineitem/ProductCode	lineitem/UsageType
rrlaw763zyhi6pm3dn2pgf4c62vzusrldhsetsdoiwhfc3qgfma	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	APN1-ElasticIP:IdleAddress
wyimm4yuzyyhlxazof5bsredfapiykidm4letsaox6hxiuw33dxka	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AWSCostExplorer	USE1-APIRequest
wdybwgzexehyk4dxtu4pbrraayjrprbhcu7qjpnbwpp6hhc36eb4q	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AWSDataTransfer	APS1-APN1-AWS-Out-Bytes
oferr4avt47yrw2a2dcw37c4hejpo6y4ccb7sgih4q5vdkevxyq	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AWSDataTransfer	DataTransfer-Regional-Bytes
jvzavwoqmmxidv7uafmtms2ywyunqnhwc3jf5ywpkuomwkhg5zja	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AWSDataTransfer	USW1-USE1-AWS-Out-Bytes
nznji6duhlfxqgmauwud4i5aem3u45hn7fjow3f5s5cy77rhdxa	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AWSDataTransfer	USW1-USE2-AWS-Out-Bytes
4wk2qios36zwxqizdpimdqvws2lqrit6bwt44mhjr4fiktjdbq	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	EBS:VolumeUsage:gp2
byn7r5yct25zbiz4xl32bmdzyvktog2i7f6huw556wp2345nnhbq	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	EBS:VolumeUsage:gp3
idvxxsiuenc4fxc3noif2uladuiocjpe6cr3d6gapizhr6hyzy2q	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T18:00:00Z	2023-04-02T09:00:01Z	AmazonEC2	SpotUsage:g5.xlarge
pwhnbyycdzz6yssvfk4t4ui2xkhlfbe3jzozocwmajrjimjbt	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Usage	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	USE2-EBS:VolumeUsage:gp2
wes3ikqzfdm6bvveh7q7ffirt27joa452gkrxqlnhsdioj2zdz	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Tax	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	USW1-BoxUsage:t2.micro
rk3oaaqz2wtdyqshvwddyunfa4p3afedjniay42h2yquboi3q	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Tax	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	AmazonEC2	USW1-ElasticIP:IdleAddress
sjmvizarp5xexkxn37ewbshijo3n32uv3bns5vtvdowspxumdeua	2023-04-03T00:00:00Z/2023-04-04T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	Tax	2023-04-03T22:00:00Z	2023-04-05T17:00:01Z	AmazonEC2	USE2-SpotUsage:c7g.medium
lnv5dueowp2pa24eyjafhm56fhal23zylphkrdcytaheilc56ena	2023-04-01T00:00:00Z/2023-04-02T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	SavingsPlanRecurringFee	2023-04-01T00:00:00Z	2023-04-02T00:00:00Z	ComputeSavingsPlans	ComputeSP:3yrNoUpfront
lnv5dueowp2pa24eyjafhm56fhal23zylphkrdcytaheilc56ena	2023-04-02T00:00:00Z/2023-04-03T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	SavingsPlanRecurringFee	2023-04-02T00:00:00Z	2023-04-03T00:00:00Z	ComputeSavingsPlans	ComputeSP:3yrNoUpfront
f7dhp3d3dsvmtwqhtzhlfbfwnlnzz3goc3lws22ammxluxeaa	2023-04-05T00:00:00Z/2023-04-06T00:00:00Z	2023-04-01T00:00:00Z	2023-05-01T00:00:00Z	SavingsPlanRecurringFee	2023-04-05T00:00:00Z	2023-04-06T00:00:00Z	ComputeSavingsPlans	ComputeSP:3yrNoUpfront

Cost Data

```
CREATE TABLE costs
(  
  `account_id` Integer,  
  `billing_period` String,  
  `date` Date,  
  `amount` Decimal,  
  `tags` JSON,  
  `service` String,  
  (account, region, etc..)
)
```

Just a single multi-tenant table.

No need for joins and will always be queried with `account_id`, dates and aggregated by amount.

Any of the other fields will be optionally filtered on depending on the user.

Cost Data

```
SELECT date, ..., SUM(amount)
FROM costs
WHERE account_id = ?
GROUP BY date, ...
```

Cost Data

When we import data for a billing_period we need to ensure only the latest version is available for querying.

```
BEGIN
```

```
DELETE FROM costs WHERE account_id = ? and billing_period = "2023-04"
```

```
INSERT INTO costs...
```

```
COMMIT
```


PostgreSQL

Initially started with PostgreSQL as that is our primary data store.
Worked well in the beginning, but as Vantage scaled it fell apart very quickly.

- Constant Deletes and Inserts caused a lot of issues
 - Constant Vacuums
 - `VACUUM` reclaims storage occupied by dead tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a `VACUUM` is done. Therefore it's necessary to do `VACUUM` periodically, especially on frequently-updated tables.
 - Had to run period REINDEX to remove dead tuples from the Index
 - Hard to keep the INDEX in memory
 - Going out to disk for millions of rows was just too slow
 - A query reading from memory could be <1s, but >60s otherwise.

PostgreSQL

Quick tips for working with time series data in PostgreSQL

- Use INCLUDE on aggregated columns in the index
 - *However, an index-only scan can return the contents of non-key columns without having to visit the index's table, since they are available directly from the index entry. Thus, addition of non-key columns allows index-only scans to be used for queries that otherwise could not use them.*
 - `CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);`
- Use ORDER BY on the datetime column
 - `CREATE INDEX costs_date_sorted ON costs (date DESC)`

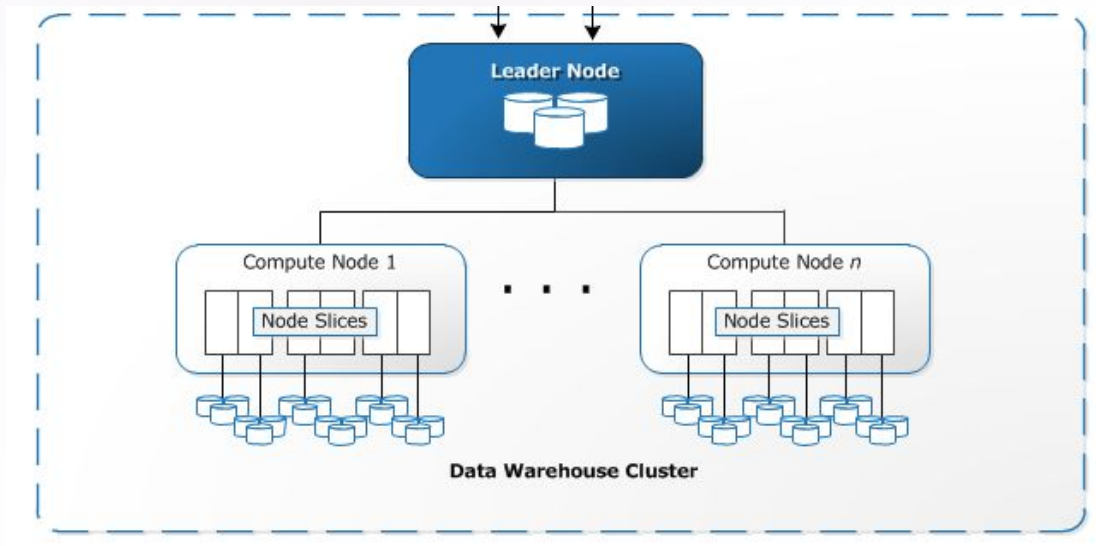
Redshift

Redshift is a columnar store and is similar to Clickhouse in concept. When creating a table you specify a SORTKEY (similar to Primary Key in Clickhouse) and a DISTSTYLE which dictates how your data will be spread across the nodes in your cluster.

```
CREATE TABLE costs
(
  `account_id` Integer,
  `billing_period` String,
  `date` Date,
  `amount` Decimal,
  `tags` SUPER,
  `service` String,
  (account, region, etc..)
)
DISTSTYLE even
SORTKEY(account_id, date, billing_period)
```

Redshift

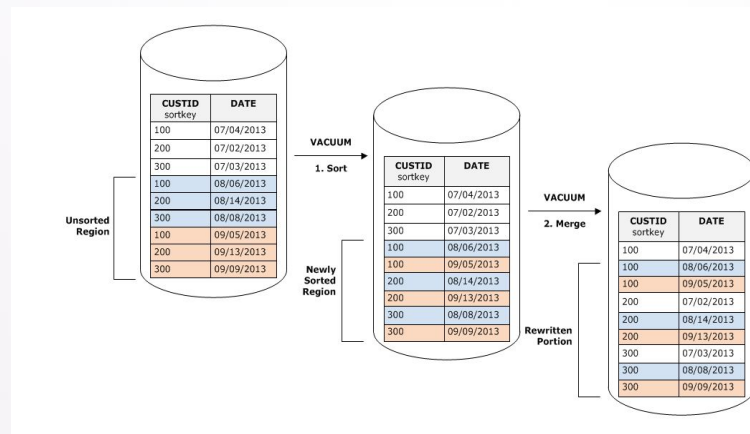
You can scale up and out.
Data is distributed by DISTSTYLE and sorted per slice by SORTKEY.



Redshift

Redshift could handle queries on much larger datasets a lot better than Postgres.
However, the constant DELETE/INSERT pattern still caused issues.

- Redshift was built on top of PostgreSQL 8.0.2 and has the same VACUUM concept.
- When data was perfectly sorted performance was decent, however when a new import came in it would be appended to the unsorted region.
 - Needed to constantly run VACUUM SORT
 - Would often take a couple hours and gets worse as the table grows.
 - We receive a higher volume of data towards the end of the month and the current month is usually what the user is querying on.
- Could not achieve a consistent level of performance required to power real time user needs.
- Created materialized views for different queries, but they often had trouble keeping up with the constant INSERTS and DELETES



I don't think this works...

Amazon Redshift introduces Automatic Table Sort, an automated alternative to Vacuum Sort

Posted On: Nov 25, 2019

Amazon Redshift now provides an efficient and automated way to maintain sort order of the data in Redshift tables to continuously optimize query performance. The new automatic table sort capability offers simplified maintenance and ease of use without compromising performance and access to Redshift tables. Automatic table sort complements [Automatic Vacuum Delete](#) and [Automatic Analyze](#) and together these capabilities fully automate table maintenance. Automatic table sort is now enabled by default on Redshift tables where a sort key is specified.

With Redshift automatic table sort, an administrator no longer needs to worry about tracking what to sort and when to sort. Redshift does it automatically. Redshift runs the sorting in the background and re-organizes the data in tables to maintain sort order and provide optimal performance. This operation does not interrupt query processing and reduces the compute resources required by operating only on frequently accessed blocks of data. It prioritizes which blocks of table to sort by analyzing query patterns using machine learning. Automatic table sort is most useful for use cases with continuous ingestion of data and ETL/Batch processing operations such as incremental daily updates. Redshift will provide a recommendation if there is a benefit to explicitly run vacuum sort on a given table.

Redshift + PostgreSQL

We ended up with an intermediary where Redshift was used as the warehouse and data would be pre-aggregated into a series of perfectly indexed Postgres tables.

```
CREATE TABLE costs_service_rollup
(  
  `report_id` Integer,  
  `date` Date,  
  `amount` Decimal,  
  `service` String,  
)
```

```
CREATE TABLE costs_resource_rollup
(  
  `report_id` Integer,  
  `date` Date,  
  `amount` Decimal,  
  `resource_id` String,  
)
```

... several others

When a user is querying we would query Redshift with a limited view and aggregation.

The user could then save the report and we would pre-aggregate the data to power the different views the user could click into.

Resulted in moving billions of records between Redshift and PostgreSQL every day. Which required high Provisioned IOPS on the Postgres instance.

Poor user experience as the user had to wait for the report to finish generating. Also we would populate data for a view which may never be looked at.

Clickhouse

Clickhouse gave us similar performance as querying directly from a well-indexed Postgres table.

You can tell Clickhouse is more purpose built for powering user facing applications vs data analysis use cases.

```
CREATE TABLE costs
(
  `account_id` Integer,
  `billing_period` String,
  `date` Date,
  `amount` Decimal,
  `tags` Map(String, String),
  `service` String,
  (account, region, etc..)
)
ENGINE = MergeTree
ORDER BY (account_id, billing_period, import_version, date)

CREATE MATERIALIZED VIEW costs_import_latest_versions
ENGINE = ReplacingMergeTree(import_version)
ORDER BY billing_period
  POPULATE AS
    SELECT billing_period,
      MAX(import_version) as import_version
FROM costs
GROUP BY billing_period
```

Clickhouse automatically sorts the data after INSERT.

Deletes are avoided on import and instead we version the imports and include that in the query.

Able to deprecate rollups and just query Clickhouse directly.

Data cleanup is done periodically using the new Lightweight Delete functionality.

As Clickhouse is OSS it has far more comprehensive and technically deeper documentation.

Clickhouse Cloud

A few more benefits of Clickhouse Cloud

- Automatic no downtime version updates
 - AWS would reboot the Redshift cluster and it would be offline while it was updated.
- No downtime scaling
 - Redshift had several different upgrade strategies depending on where you were going and where you were coming from, but all of them involved entering read-only mode. Potentially for “days”.
- Rate on innovation is much higher for Clickhouse over Redshift
- Clickhouse Cloud stores the data in S3 so leaving around old data is very cheap

We're Hiring

Engineering, GTM & Design.
NYC + Remote.