Let’s Get Physical

Optimizing big data for fun and profit (or at least lower costs and query times)

Keith Gregory
AWS Practice Lead, Chariot Solutions

CHARIOT SOLUTIONS

Definitions

Logical Design

Your business entities (customers, accounts, transactions, …) and the relationships between them

Physical Design

Table structure, indexes, and data placement, which together define the performance of your database system

Big Data

Anything that’s too large to fit on your laptop

Characteristics of “Big Data” queries

Table Scans

“Find me *all* of the customers that have done X”

Often restricted by date range

“What have my customers done for me *lately*”

Multi-table Joins, Outer Joins, Sub-queries

Example: transaction volume for customers who opened their first account in the past year, by geographic region

Two Approaches to “Big Data”

Star Schema

- A “fact” table that holds fine-grained aggregations by “dimensions”
- Allows “slice and dice” of facts, but only by predefined dimensions
- Example: sales by date, product, state, region, …

Relational

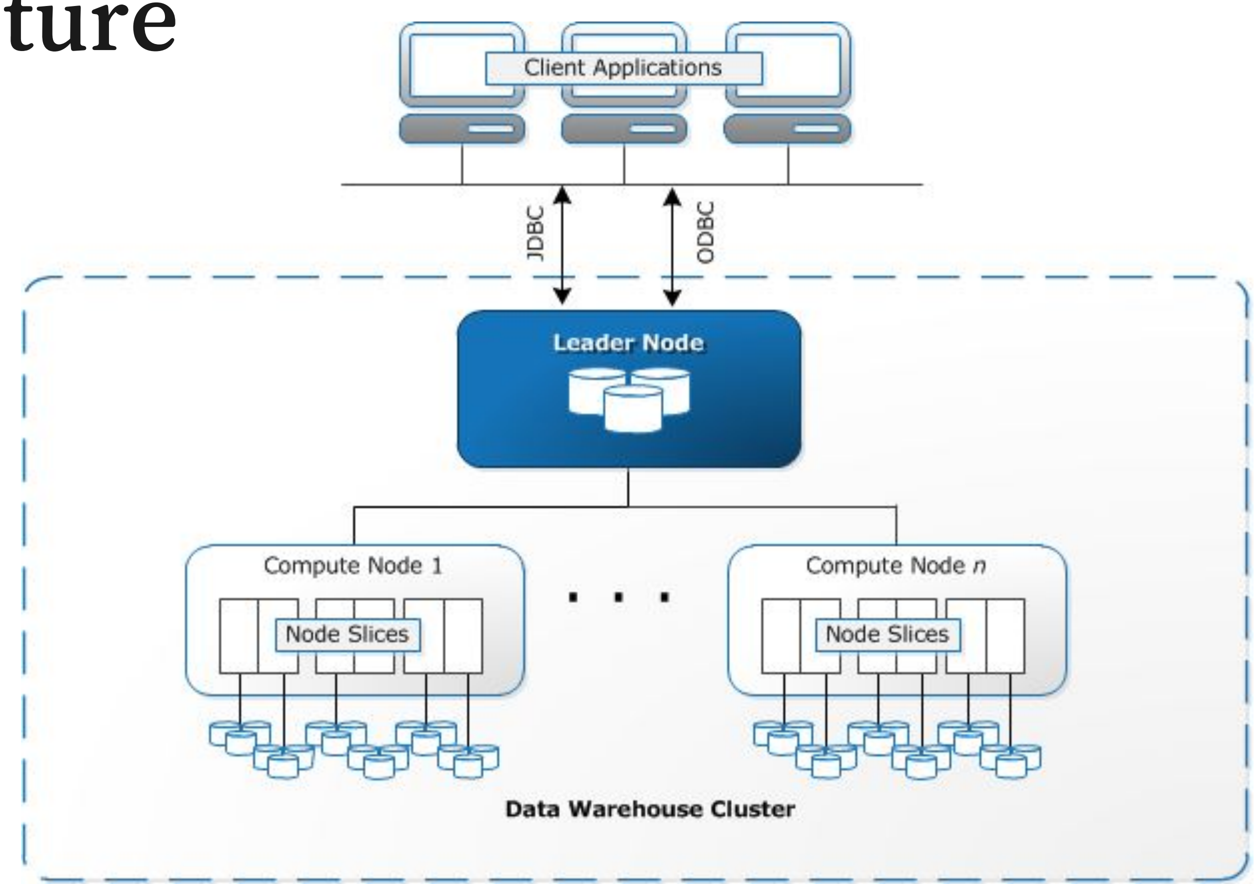
- A multi-table model that’s based on existing business entities
- Allows “exploratory” queries, still supports dimensional aggregation
- Example: customers, accounts, transactions

Seymour Cray Was Wrong

Sometimes, 1024 chickens *are* better than two strong oxen

Redshift

Architecture



source: https://web.archive.org/web/20130220201730/https://docs.aws.amazon.com/redshift/latest/dg/c_high_level_system_architecture.html

Table Distribution Styles

EVEN

Rows are distributed across nodes in a round-robin manner

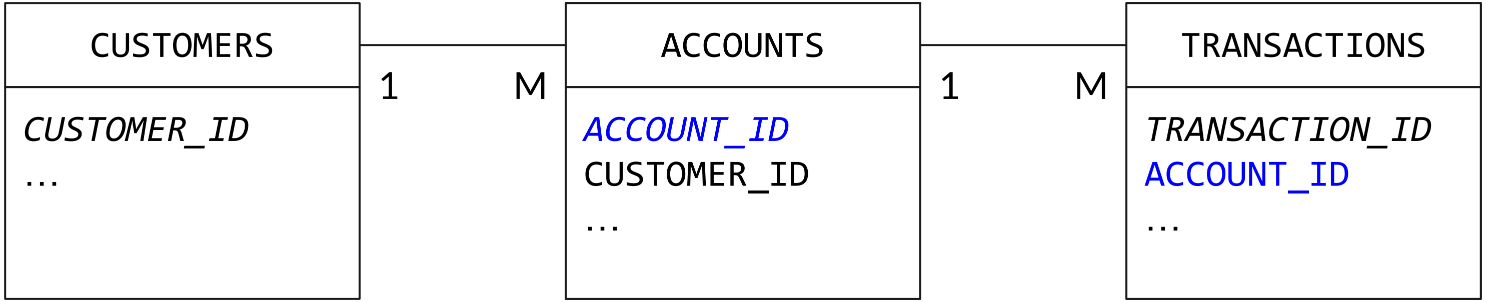
ALL

All rows are replicated on all nodes

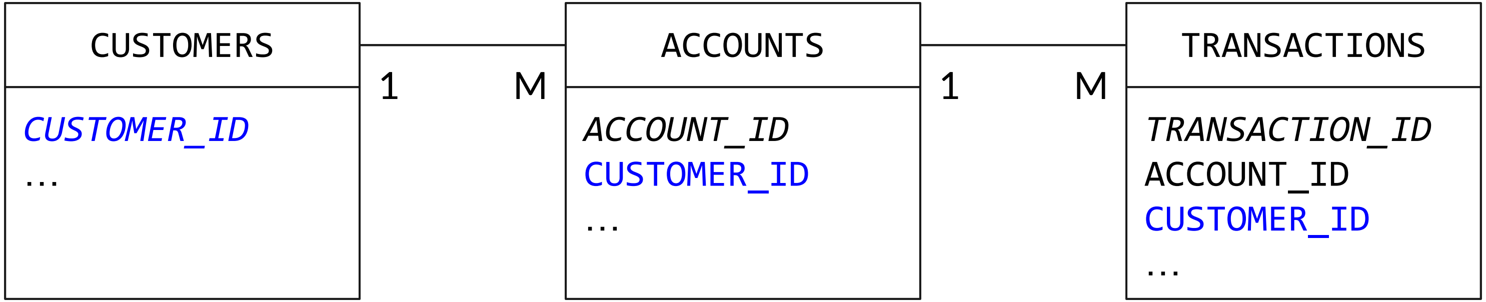
KEY

Rows are distributed based on the hash of a single column

Distribute on Join Column, not Primary Key



Denormalization Can Be Your Friend



CUSTOMERS

CUSTOMER_ID
...

1 M

ACCOUNTS

ACCOUNT_ID
CUSTOMER_ID
...

1 M

TRANSACTIONS

TRANSACTION_ID
ACCOUNT_ID
CUSTOMER_ID
...

Query-time Redistribution

DS_DIST_NONE

No distribution; joins can be performed in parallel

DS_DIST_OUTER / DS_DIST_INNER

One table is redistributed to match the other

DS_DIST_BOTH

Both tables are redistributed by join key – usually happens because tables creation defaults to EVEN distribution.

DS_BCAST_INNER

Inner table is replicated on all nodes – almost always indicates a bad query

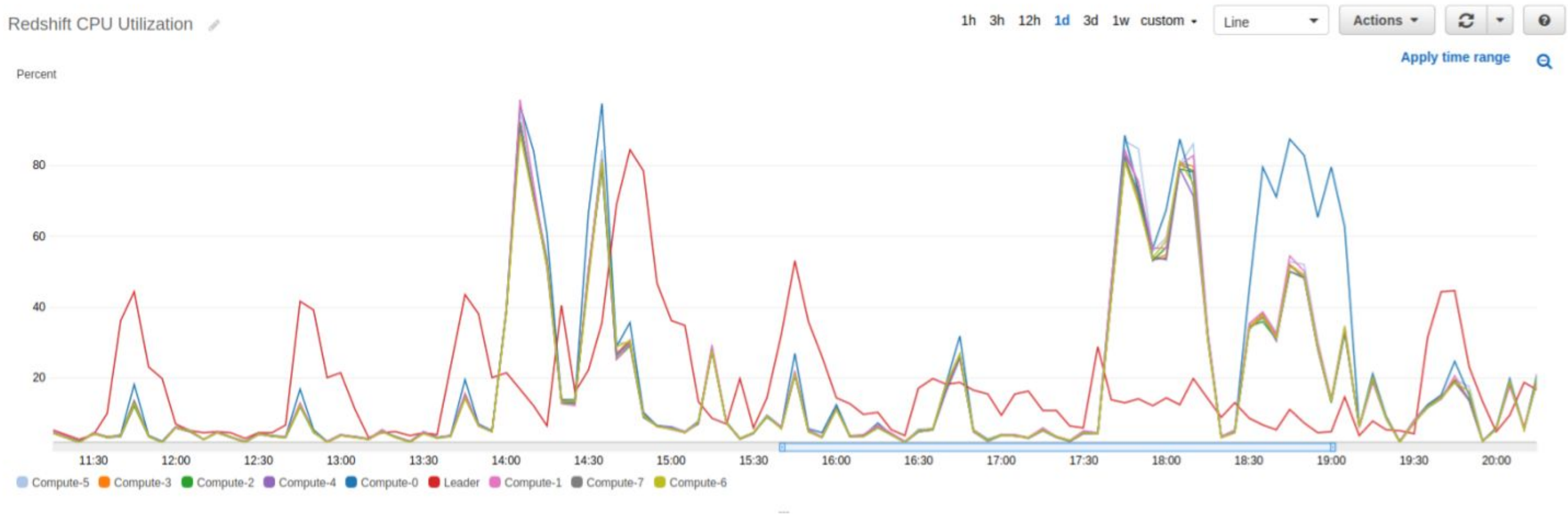
Redshift, you got some ’splaining to do!

```
XN Limit
   ->  XN Merge
       Merge Key: (count(DISTINCT pp.eventid) - count(DISTINCT atc.eventid))
       ->  XN Network
           Send to leader
           ->  XN Sort
               Sort Key: (count(DISTINCT pp.eventid) - count(DISTINCT atc.eventid))
               ->  XN HashAggregate
                   ->  XN Hash Left Join DS_DIST_NONE
                       Hash Cond: ((("outer".userid)::text = ("inner".userid)::text)
                                  AND (("outer".productid)::text = ("inner".productid)::text))
                       ->  XN Seq Scan on product_page pp
                       ->  XN Hash
                           ->  XN Seq Scan on add_to_cart atc
```

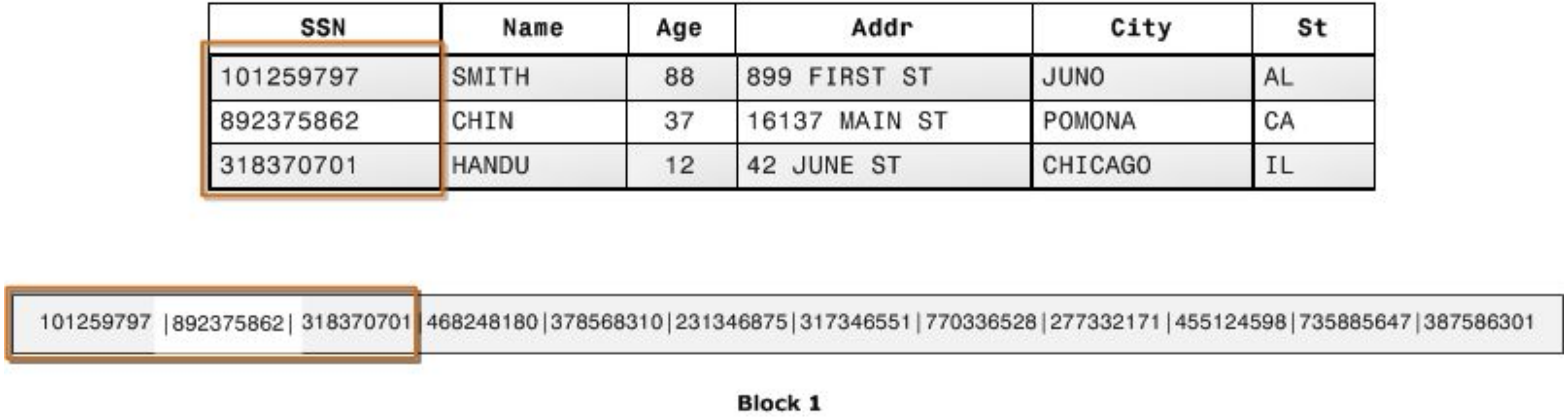
Unbalanced Data

One node has more data than the others

Often happens when distribution column contains nulls



Row-oriented versus Column-oriented



SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

```
101259797 |892375862| 318370701|468248180|378568310|231346875|317346551|770336528|277332171|455124598|735885647|387586301
```

Block 1

source: https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html

Sort Keys

A list of columns that defines the physical order of rows in the table

When sort key used in the WHERE clause, Redshift can ignore blocks that don’t contain relevant data

Two forms of multi-column sort keys:

- Compound: hierarchical sorting based on order of columns
- Interleaved: all columns (up to 8) have equal representation

Sorting on timestamp (by itself) is usually best

Data Compression

Redshift can compress each column individually

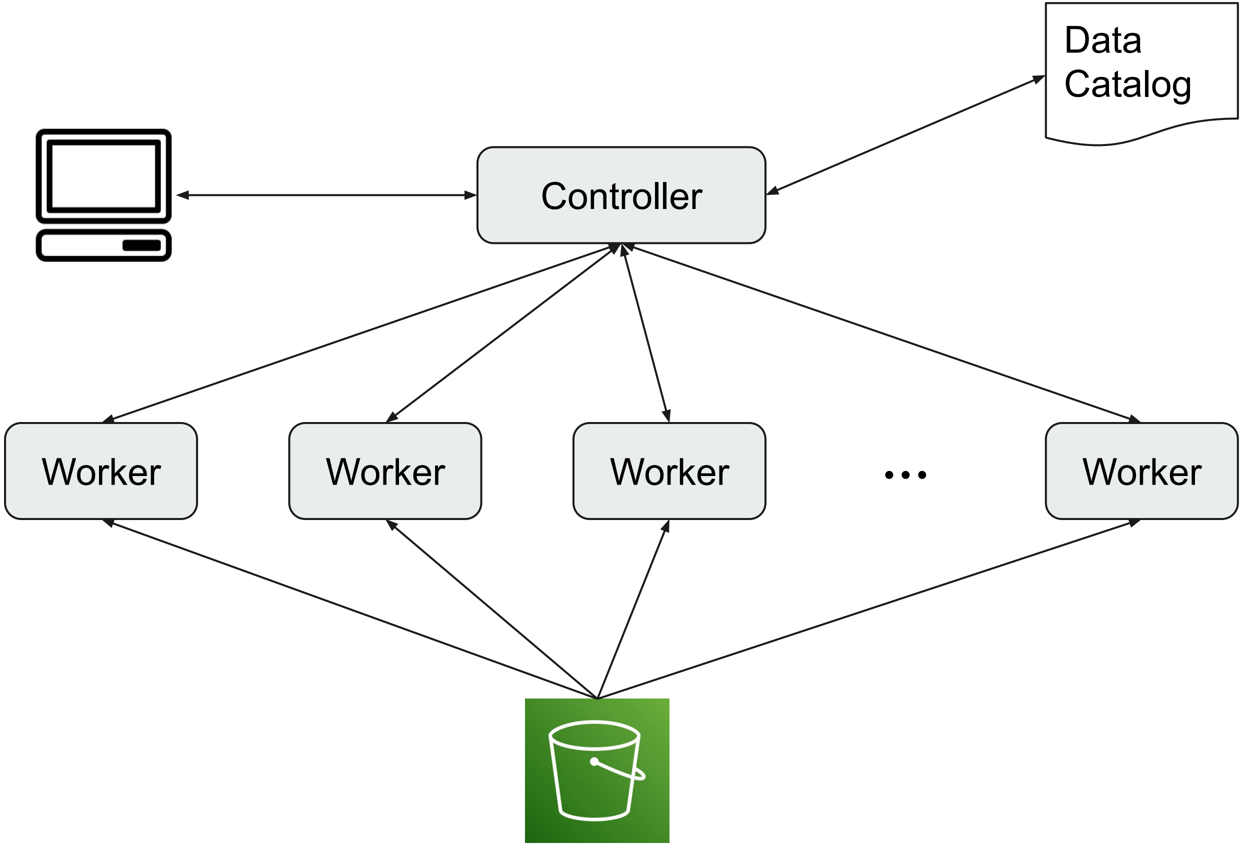
By default, Redshift auto-configures compression

- This only “works” for long-lived tables

`ANALYZE COMPRESSION` to identify best mechanism

Athena SQL

Architecture



Supported File Formats

Avro: a row-oriented format that includes a schema

CSV: the old standby, albeit not well-defined

JSON: the new hotness

ORC: a columnar format used by Hadoop

Parquet: a columnar format used by Hadoop

Text: if it can be turned into fields via regex

Pick the Right File Size

Tradeoff:

Bigger files reduce overhead

More files allow more workers to run in parallel

Some numbers: counting CloudTrail events

Raw CloudTrail logs (1,637,376 files):	3 minutes 15 seconds
Aggregated by date (684 files):	6.3 seconds
One 1.76 GB file:	1 minute 44 seconds

Partition Data

Allows Athena to read only some of your files

Incorporates information into S3 prefix

Two formats:

```
s3://BUCKET/TABLE/VALUE/VALUE/FILENAME

s3://BUCKET/TABLE/COLNAME=VALUE/COLNAME=VALUE/FILENAME
```

Example:

```
s3://clickstream-data/add_to_cart/2023/08/...
```

Partitioned Table Definition

```
CREATE EXTERNAL TABLE `cloudtrail_projected`
(
    `eventtime` string COMMENT 'from deserializer',
    `eventname` string COMMENT 'from deserializer',
    `awsregion` string COMMENT 'from deserializer',
    `recipientaccountid` string COMMENT 'from deserializer',
    ...
)
PARTITIONED BY (
  `account_id` string,
  `region` string,
  `ingest_date` string
)
LOCATION
  's3://com-chariotsolutions-cloudtrail/AWSLogs/o-x72e8b2quf'
TBLPROPERTIES (
  ...
'storage.location.template'='s3://com-chariotsolutions-cloudtrail/AWSLogs/o-x72e8b2quf/${account_id}/CloudTrail/${region}/${ingest_date}',
)
```

Querying with Partitions

Must specify partition values in WHERE clause

```
SELECT  count(*) as event_count
FROM    "default"."cloudtrail_projected"
where   account_id = '123456789012'
and     region = 'us-east-1'
and     ingest_date >= '2022/09/01'
and     ingest_date < '2022/10/01'
```

Run time:	2.038 sec
Data scanned:	4.30 MB
Count:	18,706

```
SELECT  count(*) as event_count
FROM    "default"."cloudtrail_projected"
where   recipientaccountid = '123456789012'
and     awsregion = 'us-east-1'
and     eventtime >= '2022-09-01'
and     eventtime < '2022-10-01'
```

Run time:	6 min 27.47 sec
Data scanned:	13.73 GB
Count:	18,705

Querying with Partitions, part 2

For performance *and* accuracy, combine partitions and internal field values

```
SELECT  count(*) as event_count
FROM    "default"."cloudtrail_projected"
where   ingest_date >= '2022/08/29'
and     ingest_date < '2022/10/03'
and     recipientaccountid = '123456789012'
and     awsregion = 'us-east-1'
and     eventtime >= '2022-09-01'
and     eventtime < '2022-10-01'
```

Run time:	49.149 sec
Data scanned:	247.55 MB
Count:	18,705

Managing Partitions

- Explicit partition list in Glue Data Catalog
  - Glue Crawler will update automatically, otherwise must use SQL or SDK to add/discover partitions
- Projection
  - Defines partitions based on combinations of explicit values, ranges of dates/numbers
- Injection
  - Used for high cardinality partitions (eg: user ID)
  - *All* queries must include predicate on partition column

Performance Comparison

Sample Data

PRODUCT_PAGE

59,693,900 rows

ADD_TO_CART

18,523,255 rows

CHECKOUT_COMPLETE

9,853,549 rows

```
{
  "eventType": "checkoutComplete",
  "eventId": "aa243032-cffd-4fd7-ab9b-994e69567a76",
  "timestamp": "2023-04-24 19:16:42.581",
  "userId": "c5362ccc-7355-433d-9322-9b9b564276a5",
  "itemsInCart": 4,
  "totalValue": 6.00
}
```

Single-Table Aggregation

```
select  productid,
        sum(quantity) as units_added
from    "public"."add_to_cart"
group   by productid
order   by units_added desc
limit   10;
```

Athena	0.88
Provisioned, 4 dc2.large	0.49
Provisioned, 8 dc2.large	0.31
Serverless, 8 RPU	0.34
Postgres, db.m6g.xlarge	18.01

Join on Distribution Column

```
select  count(distinct user_id)
            as users_with_abandoned_carts
from    (
        select  atc.userid as user_id,
                max(atc."timestamp") as max_atc_timestamp,
                max(cc."timestamp") as max_cc_timestamp
        from    "public"."add_to_cart" atc
        left join "public"."checkout_complete" cc
        on      cc.userid = atc.userid
        group   by atc.userid
        )
where   max_atc_timestamp > max_cc_timestamp
or      max_cc_timestamp is null;
```

Athena	4.441
Provisioned, 4 dc2.large	5.805
Provisioned, 8 dc2.large	4.469
Serverless, 8 RPU	1.828
Postgres, db.m6g.xlarge	1:41.74

Join on Multiple Columns

```
select  productid,
        (views - adds) as diff
from    (
        select  pp.productid  as productid,
                count(distinct pp.eventid) as views,
                count(distinct atc.eventid) as adds
        from    "public"."product_page" pp
        left join "public"."add_to_cart" atc
        on      atc.userid = pp.userid
        and     atc.productid = pp.productid
        group   by pp.productid
        )
order   by 2 desc
limit   10;
```

Athena	4.70
Provisioned, 4 dc2.large	31.11
Provisioned, 8 dc2.large	23.63
Serverless, 8 RPU	13.40
Postgres, db.m6g.xlarge	1:29.22

Summary

Architecture

Redshift

- “Traditional” database, based on Postgres 8 SQL
- Fixed number of nodes, each with its own attached disk
- Joins performed in parallel; rows must have same distribution

Athena

- Reads structured text from S3, using Presto database engine
- Variable number of workers; probably dependent on #/files

Knobs You Can Turn

- Redshift
    - Distribution Key
    - Sort Key
    - Compression
- Athena
    - File Type
    - File Size
    - Partitioning

For More Information

Chariot Blog Posts

Right-sizing Data for Athena

Athena File-type Comparison (Avro, JSON, Parquet)

Athena/Redshift Performance Comparison

AWS Docs / Blog Posts

Top 10 Performance Tuning Tips for Amazon Athena

Query Optimization Techniques (Athena User Guide)

Partitioning data in Athena

Amazon Redshift best practices for designing tables

Amazon Redshift best practices for designing queries

(Redshift) Analyzing and improving queries

Office Hours

Sign up for a one hour one-on-one to discuss Redshift, Athena, or general data engineering practices.

Technology in the Service of Business.

Chariot Solutions is the Greater Philadelphia region’s top IT consulting firm specializing in software development, systems integration, mobile application development and training.

Our team includes many of the top software architects in the area, with deep technical expertise, industry knowledge and a genuine passion for software development.

Visit us online at chariotsolutions.com.

CHARIOT
SOLUTIONS

Leader Node

Client communication

Parsing SQL queries, creating execution tasks, gathering results

Cross-cluster aggregations