SUMMIT

TOKYO | APRIL 20-21, 2023

AWS-36

Amazon Redshift クエリパフォーマンスチューニング Deep Dive

畔勝 洋平
アマゾン ウェブ サービス ジャパン合同会社
プロフェッショナルサービス本部 シニアビッグデータコンサルタント

自己紹介

畔勝 洋平（あぜかつ ようへい）

プロフェッショナルサービス本部

前職は外資系ベンダーでデータベースコンサルタント。
現在はビッグデータコンサルタントとしてお客様の
データ基盤構築やDB・DWH移行を支援。

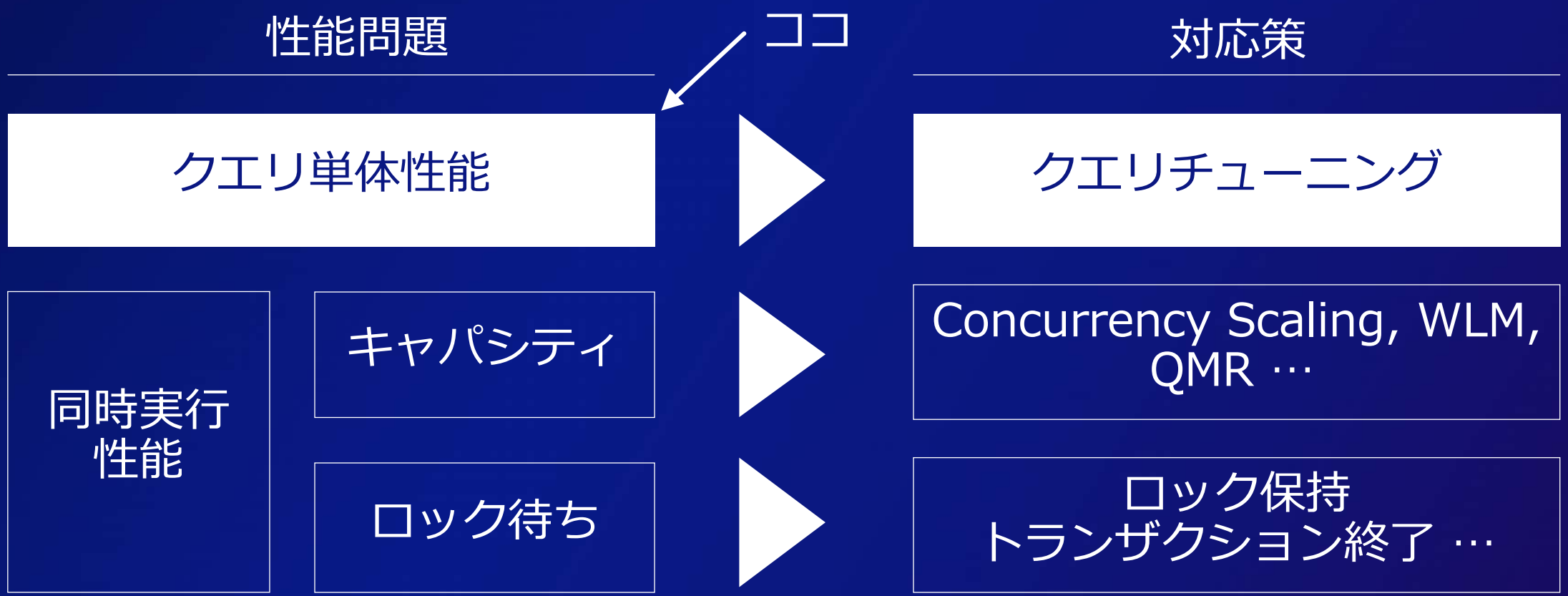
好きなAWSサービス： Amazon S3、 Performance Insights、 Amazon Redshift

特技：パフォーマンス分析・チューニング

趣味： ポートレート

本日お話しすること

本日お話しする範囲



想定する対象

- Amazon Redshift でクエリチューニングをされる方
- Amazon Redshift 以外の RDBMS(OTLP/OLAP) のエキスパート

クエリチューニングにおける課題

- 職人芸
  - 特定のエキスパートに依存する

- 試行錯誤するため時間がかかる
  - 実行計画を見るなどして思いついたチューニング案の試行錯誤を繰り返すため、時間がかかる

このセッションのゴール

- 職人芸からの脱却
  - 経験や勘ではなく、システマチックにボトルネックを特定できるようになる

- 最小限の時間で効率的にチューニング
  - ボトルネック（どこに時間がかかっているか）を特定できているため、最小限の試行錯誤でチューニングができるようになる

アジェンダ

- Amazon Redshift の進化の歴史
- Amazon Redshift のアーキテクチャ
- Amazon Redshift における分析クエリを速くする仕組み
- テーブル設計の重要性

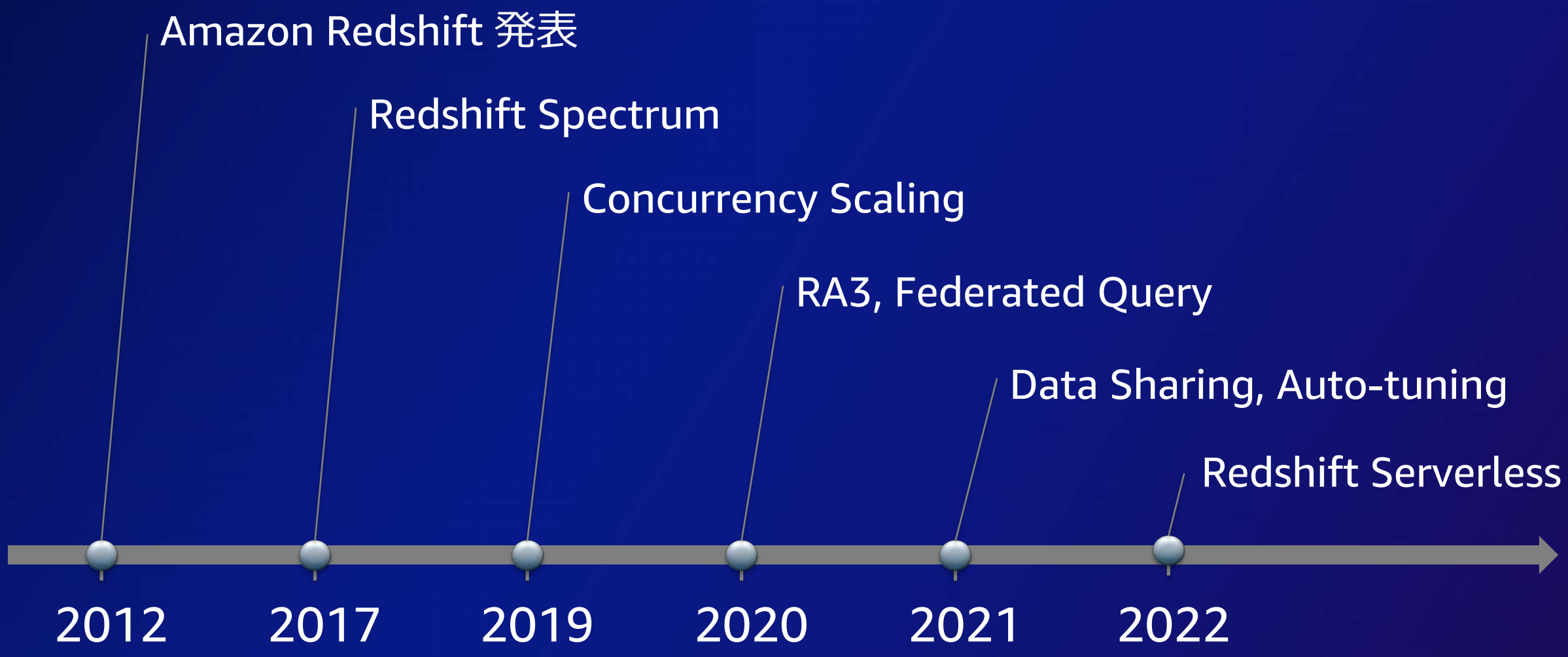
前提知識

- クエリのボトルネック分析方法
- 具体的なクエリチューニング例

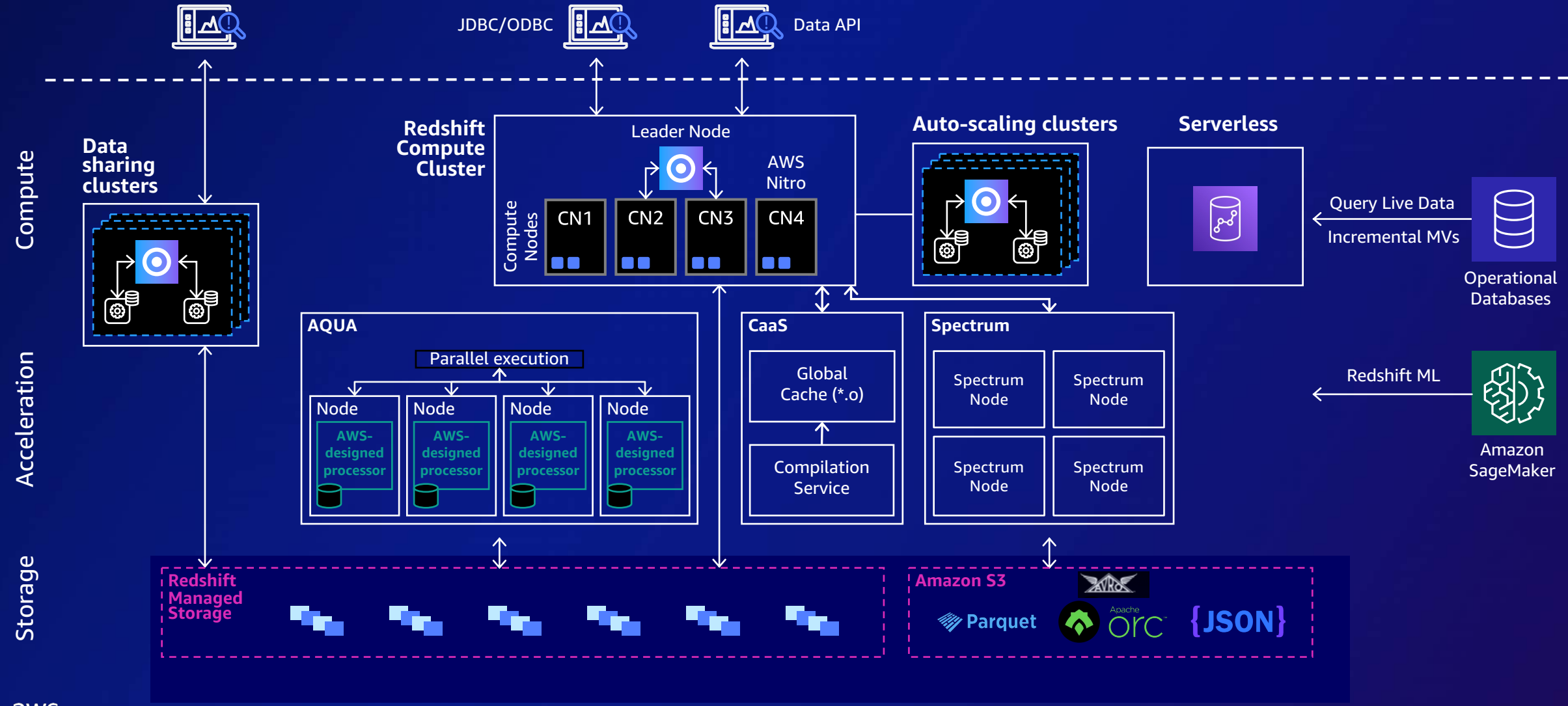
分析・チューニング

Amazon Redshift の進化の歴史

Amazon Redshift 進化の歴史



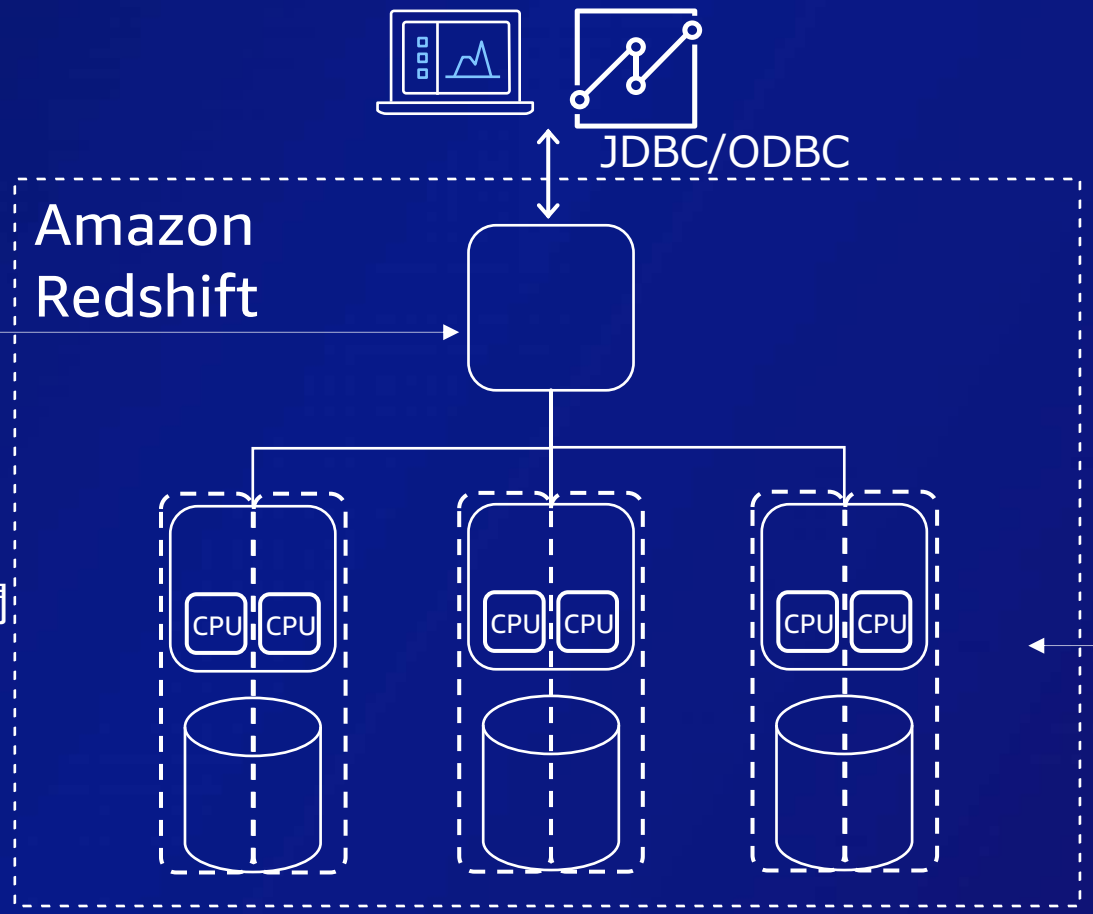
Amazon Redshift architecture 進化の歴史



Amazon Redshift のアーキテクチャ

Amazon Redshift のアーキテクチャ

ノードタイプ：DC2



リーダーノード

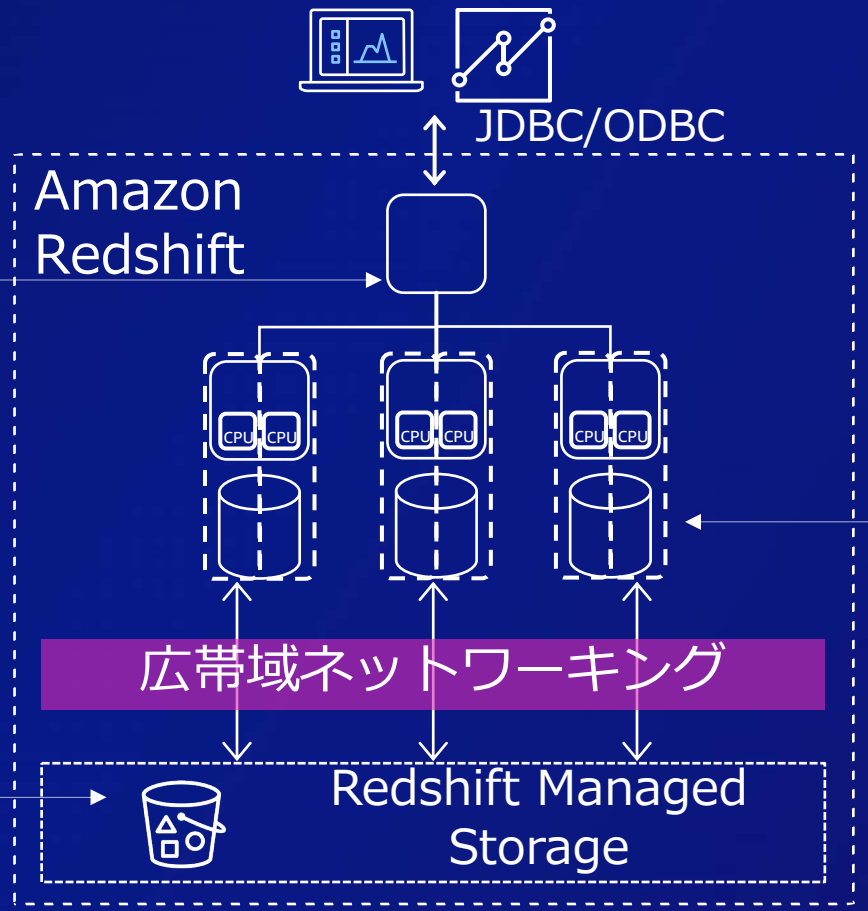
- クエリのエンドポイント
- クエリオプティマイザが SQL 処理コードを生成し、コンピュートノードに展開

コンピュートノード

- ローカル列指向ストレージ
- クエリの並列実行エンジン

Amazon Redshift のアーキテクチャ

ノードタイプ：RA3



リーダーノード

- クエリのエンドポイント
- クエリオプティマイザが SQL 処理コードを生成し、コンピュートノードに展開

コンピュートノード

- 高速ローカル SSD キャッシュ ＋大容量 RAM 搭載 ＋広帯域ネットワーキング
- クエリの並列実行エンジン

マネージドストレージ

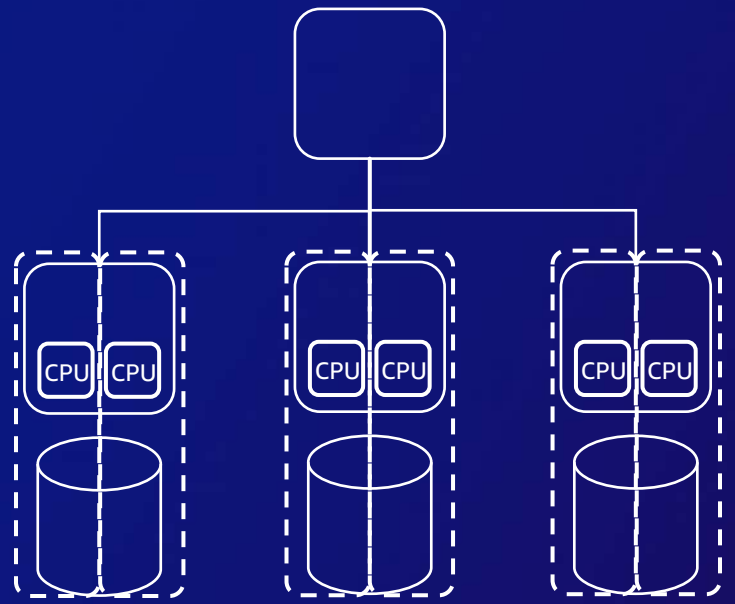
- Redshift 管理 S3 バケット
- データの永続ストレージ

Amazon Redshift のアーキテクチャ

シェアードナッシング ＋ MPP

シェアードナッシング

- ストレージをノード間で共有しない
- ノード とストレージ がセット
- マネージドストレージを持つRA3インスタンスでも処理時の考え方は同じ（処理するデータは共有しない）



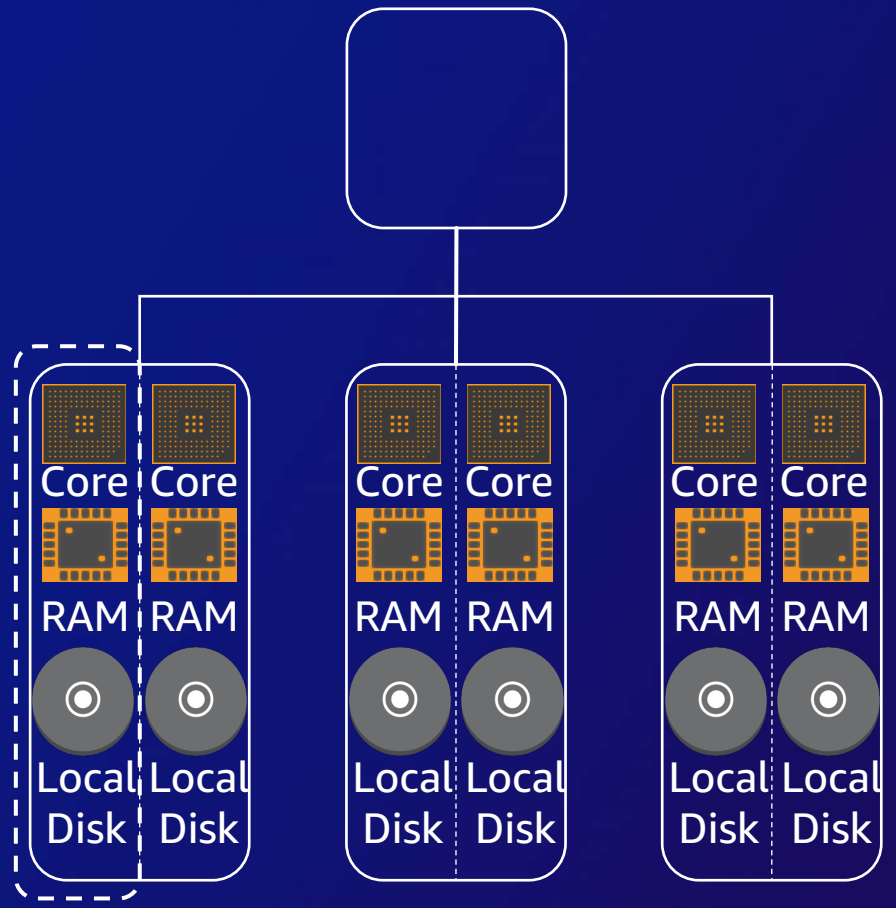
MPP：Massively Parallel Processing

- 1つのクエリを複数ノードで並列分散実行
- ノードを増やすこと（スケールアウト）でパフォーマンス向上

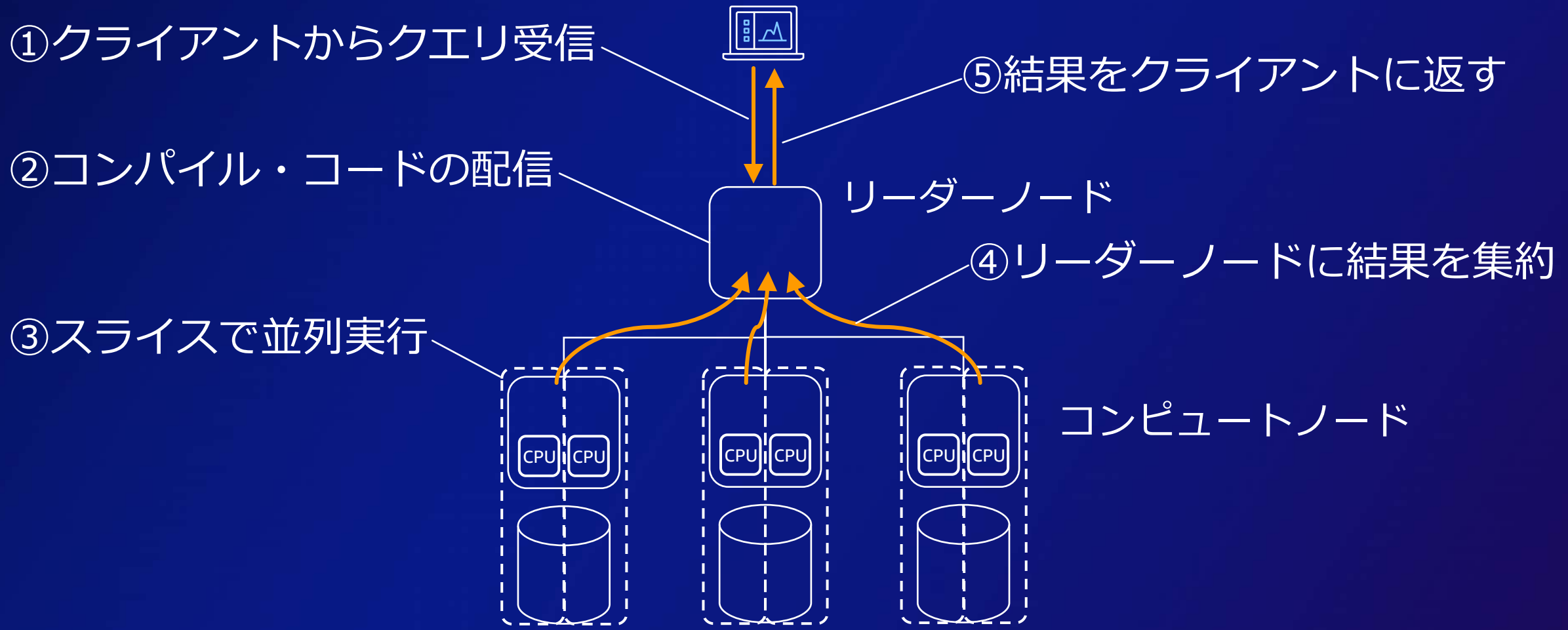
Amazon Redshift のアーキテクチャ

スライス

- ノード内でCPU、メモリ、ストレージを論理的に分割した処理単位
- 各スライスに割り当てられたリソースが、配下に分散されたデータに対して並列処理
- 各コンピュートノードはインスタンスタイプに応じてデフォルトで2、4、16のスライスを持つ

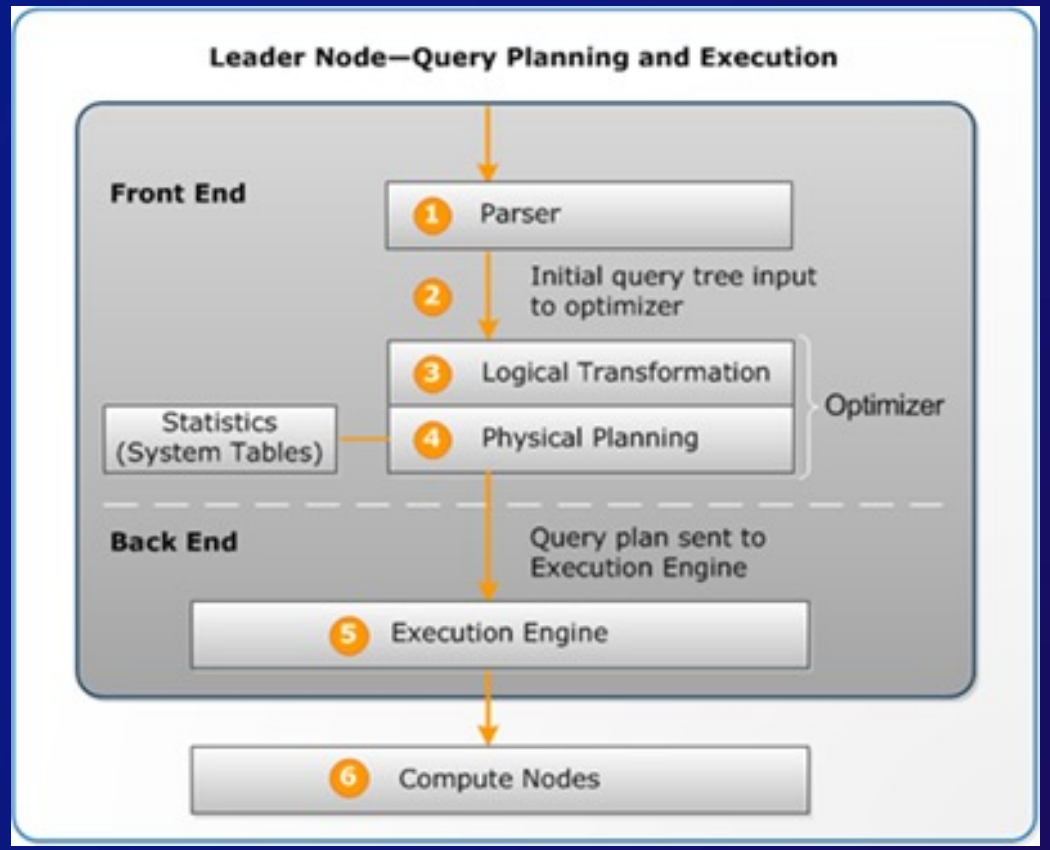


クエリ実行の流れ



クエリのコンパイルの流れ

① パーサがクエリを解析
② クエリから初期クエリツリーを作成、オプティマイザに渡す
③ 最適化のためクエリ書き換え
④ クエリプラン（実行計画）作成
⑤ 実行エンジンが実行計画をステップ・セグメント・ストリームに変換し、C++コード、オブジェクトコードにコンパイル
⑥ コンピュートノードにコンパイル済みコードが渡されて並列実行



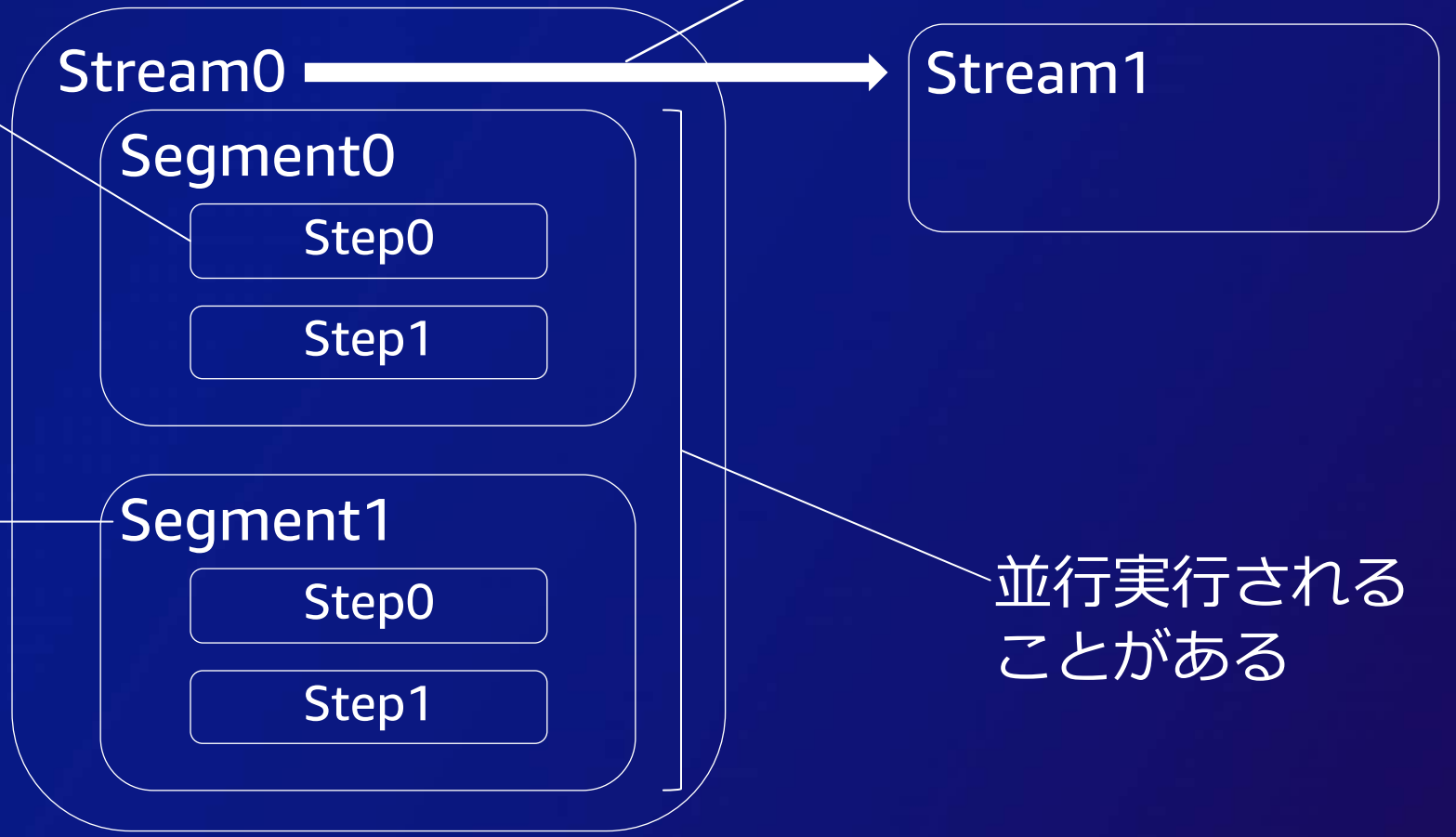
オブジェクトコードは、セグメント単位で作成される

出典: https://docs.aws.amazon.com/redshift/latest/dg/c-query-planning.html

ステップ → セグメント → ストリーム

scan、join、aggregation、
projection 等の単位

シリアル実行



オブジェクトコード作成、
コンパイルキャッシュの単位

並行実行される
ことがある

出典: https://docs.aws.amazon.com/redshift/latest/dg/c-query-planning.html
https://dl.acm.org/doi/10.1145/3514221.3526045

Amazon Redshift における分析クエリを速くする仕組み

分析クエリは仕事量が多い

```
select * from t1
  where a='a5'
```

OLTP
行指向

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5
a6	b6	c6
a7	b7	c7
a8	b8	c8
a9	b9	c9
a10	b10	c10

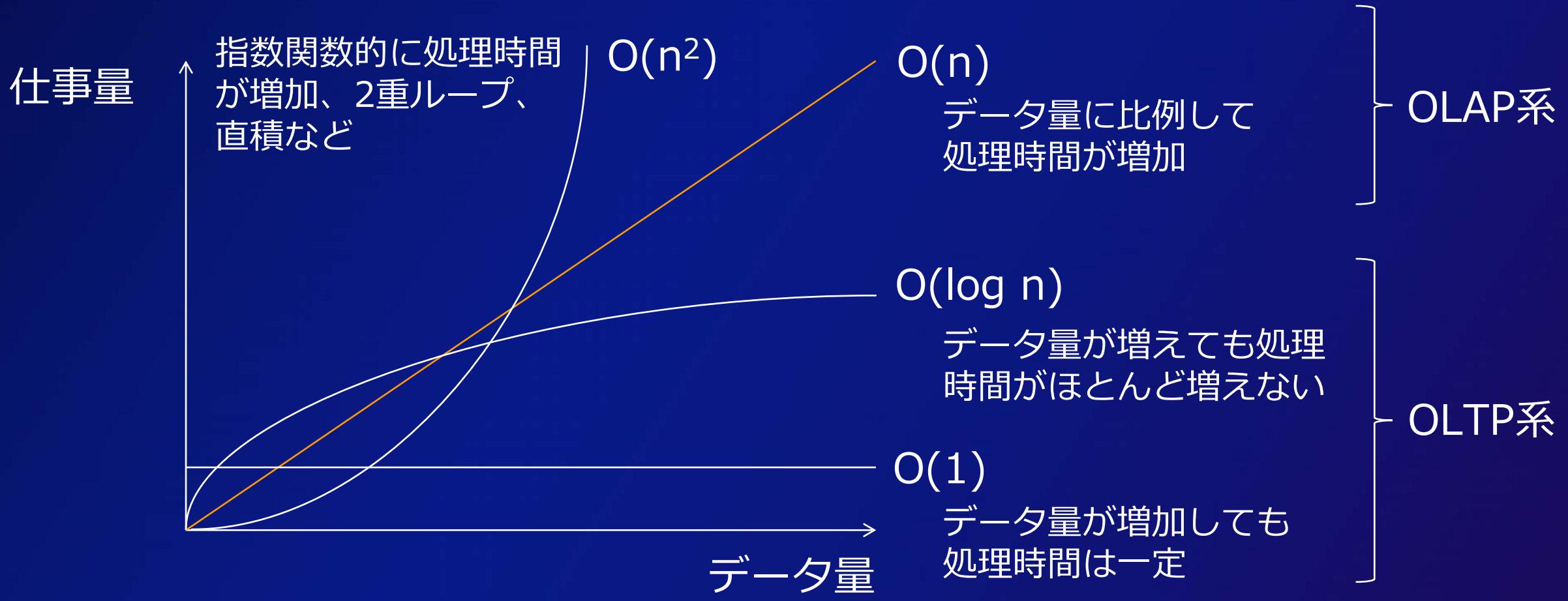
```
select sum(*) from t1
  b between b3 and b9
```

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5
a6	b6	c6
a7	b7	c7
a8	b8	c8
a9	b9	c9
a10	b10	c10

OLAP
列指向

分析クエリは仕事量が多い

テーブルのデータ量に比例して仕事量（CPU計算量 + I/ O量）が増加



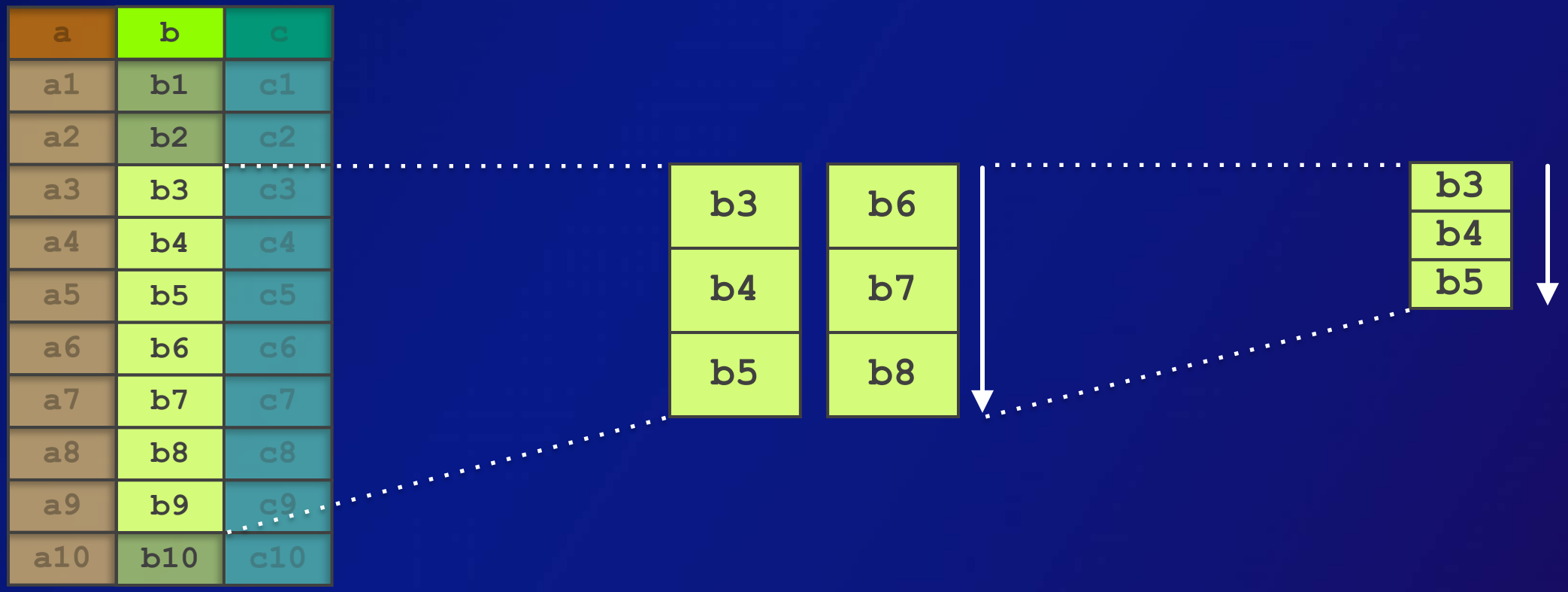
＊O（オーダー）記法：入力データのサイズに対して、計算に要する実行時間（時間計算量）を表したもの

分析クエリを速くする3つのアプローチ

①I/O量削減

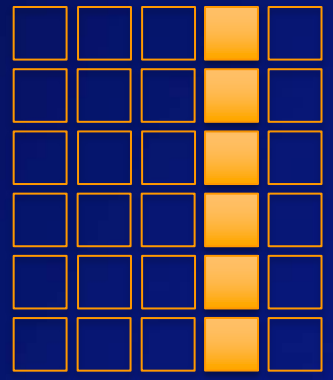
②並列化

③高速化



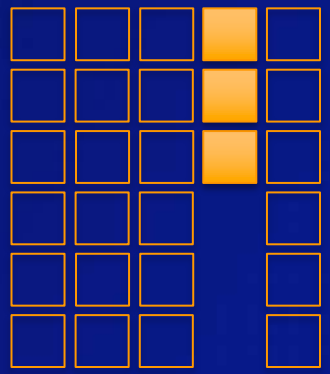
Amazon Redshift におけるI/O量削減

①列指向
ストレージ

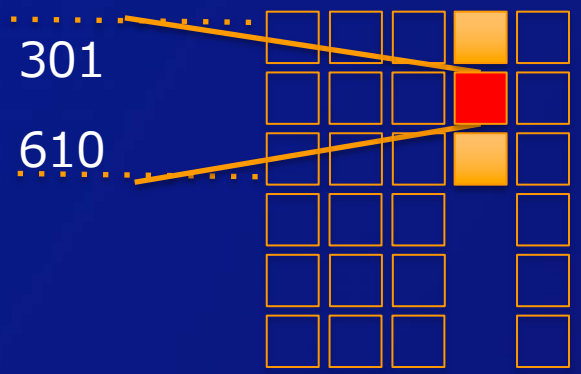


ブロック（1MB）

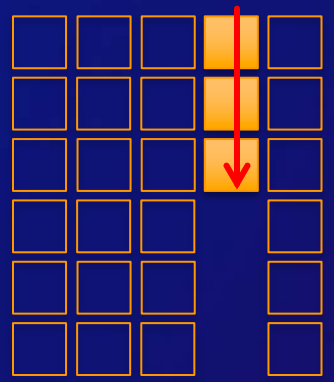
②圧縮
エンコード



③ゾーンマップ



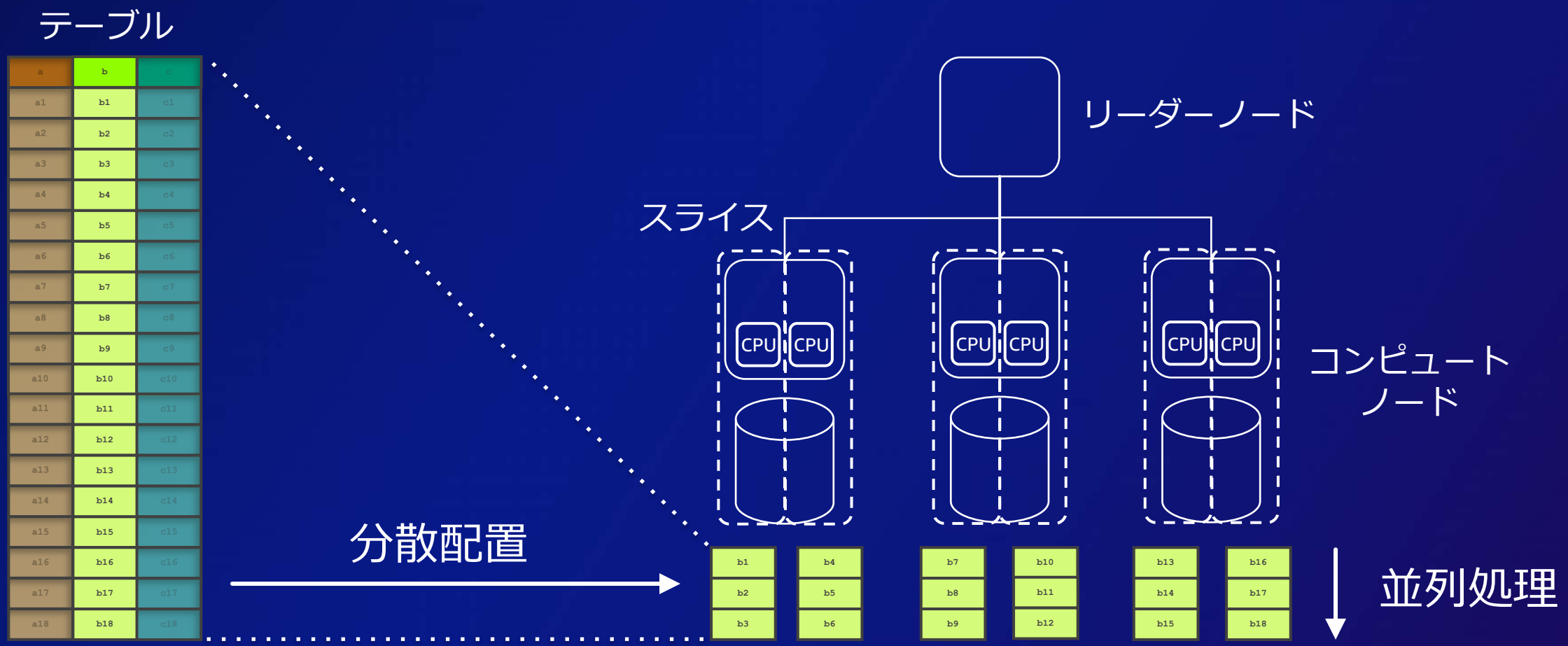
④ソートキー



Amazon Redshift における並列化

シェアードナッシング + MPP* による並列分散処理

* MPP: Massively Parallel Processing



Amazon Redshift における高速化

- SIMD Vectorized Scans
- AWS Nitro ASIC
- FPGA
- SSD

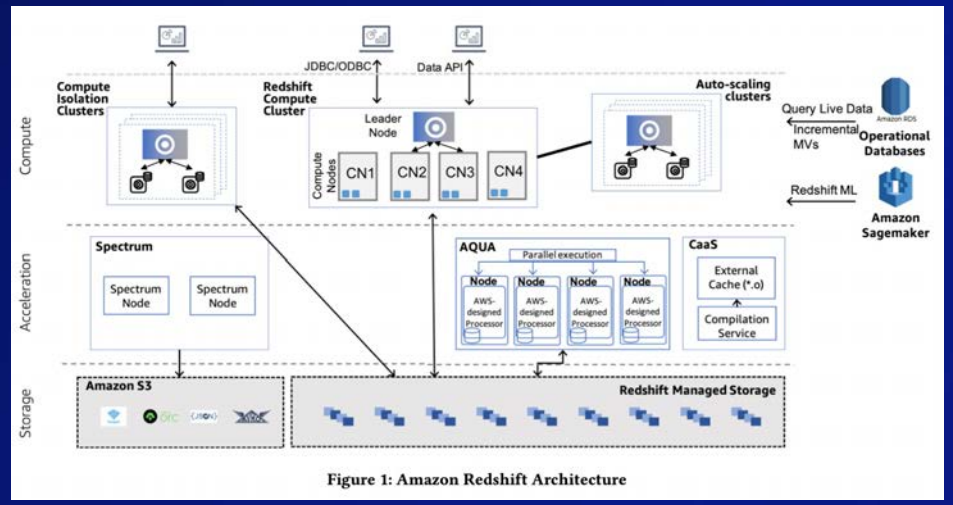


Figure 1: Amazon Redshift Architecture

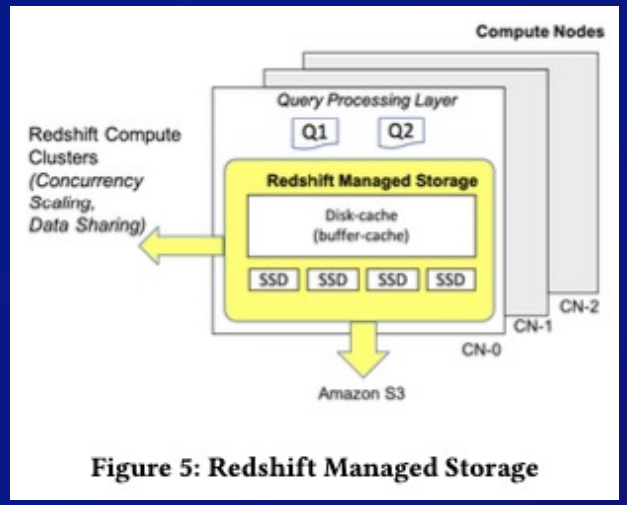


Figure 5: Redshift Managed Storage

Amazon Redshift Re-invented

Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta
Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger
Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas
Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan
Sriram Subramanian, Doug Terry
Amazon Web Services
USA
redshift-paper@amazon.com

ABSTRACT

In 2013, Amazon Web Services revolutionized the data warehousing industry by launching Amazon Redshift, the first fully-managed, petabyte-scale, enterprise-grade cloud data warehouse. Amazon Redshift made it simple and cost-effective to efficiently analyze large volumes of data using existing business intelligence tools. This cloud service was a significant leap from the traditional on-premise data warehousing solutions, which were expensive, not elastic, and required significant expertise to tune and operate. Customers embraced Amazon Redshift and it became the fastest growing service in AWS. Today, tens of thousands of customers use Redshift in AWS's global infrastructure to process exabytes of data daily.

In the last few years, the use cases for Amazon Redshift have evolved and in response, the service has delivered and continues to deliver a series of innovations that delight customers. Through architectural enhancements, Amazon Redshift has maintained its industry-leading performance. Redshift improved storage and compute scalability with innovations such as tiered storage, multi-cluster auto-scaling, cross-cluster data sharing and the AQUA query acceleration layer. Autonomics have made Amazon Redshift easier to use. Amazon Redshift Serverless is the culmination of autonomics effort, which allows customers to run and scale analytics without the need to set up and manage data warehouse infrastructure. Finally, Amazon Redshift extends beyond traditional data warehousing workloads, by integrating with the broad AWS ecosystem with features such as querying the data lake with Spectrum, semistructured data ingestion and querying with PartiQL, streaming ingestion from Kinesis and MSK, Redshift ML, federated queries to Aurora and RDS operational databases, and federated materialized views.

CCS CONCEPTS

• **Information systems → Database design and models**; **Database management system engines**.

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526045

KEYWORDS

Cloud Data Warehouse, Data Lake, Redshift, Serverless, OLAP, Analytics, Elasticity, Autonomics, Integration

ACM Reference Format:

Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, Doug Terry . 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3514221.3526045

1 INTRODUCTION

Amazon Web Services launched Amazon Redshift [13] in 2013. Today, tens of thousands of customers use Redshift in AWS's global infrastructure of 26 launched regions and 84 availability zones (AZs) to process exabytes of data daily. The success of Redshift inspired innovation in the analytics segment [3, 4, 9, 21], which in turn has benefited customers. The service has evolved at a rapid pace in response to the evolution of the customers' use cases. Redshift's development has focused on meeting the following four main customer needs.

First, customers demand high-performance execution of increasingly complex analytical queries. Redshift provides industry-leading data warehousing performance through innovative query execution that blends database operators in each query fragment via code generation. State-of-the-art techniques like prefetching and vectorized execution, further improve its efficiency. This allows Redshift to scale linearly when processing from a few terabytes of data to petabytes.

Second, as our customers grow, they need to process more data and scale the number of users that derive insights from data. Redshift disaggregated its storage and compute layers to scale in response to changing workloads. Redshift scales up by elastically changing the size of each cluster and scales out for increased throughput via multi-cluster autoscaling that automatically adds and removes compute clusters to handle spikes in customer workloads. Users can consume the same datasets from multiple independent clusters.

Third, customers want Redshift to be easier to use. For that, Redshift incorporated machine learning based autonomics that fine-tune each cluster based on the unique needs of customer workloads.

出典: https://www.amazon.science/publications/amazon-redshift-re-invented

Amazon Redshift で分析クエリを速くする仕組み

①I/O量削減

- 列指向ストレージ
- 圧縮エンコード
- ゾーンマップ
- ソートキー

②並列化

- シェアードナッシング
- MPP

③高速化

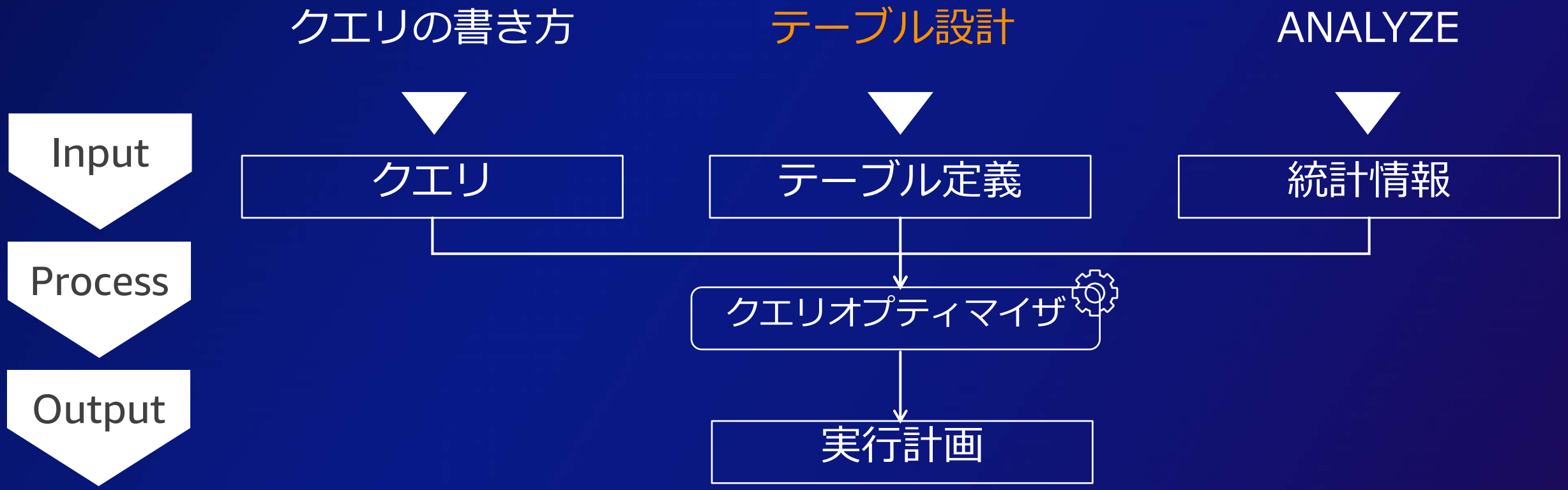
- SIMD
- ASIC
- FPGA
- SSD

…

テーブル設計の重要性

テーブル設計の重要性

クエリ性能を引き出すためにはテーブル設計が重要
適切なテーブル設計あっての、クエリの書き方、ANALYZE



クエリ性能におけるテーブル設計のカバー範囲

①I/O量削減

- 列指向ストレージ
- 圧縮エンコード
- ゾーンマップ
- ソートキー

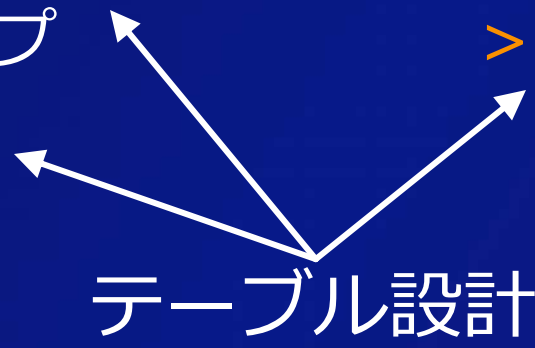
②並列化

- MPP
- シェアードナッシング
  > 分散スタイル

③高速化

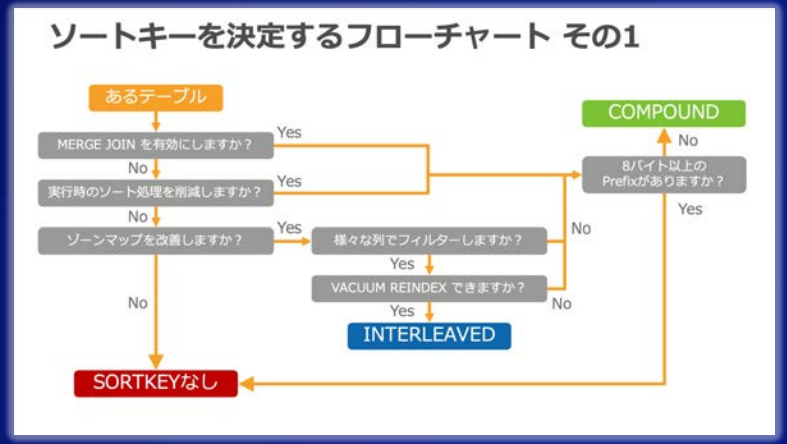
- SIMD
- AQUA
- CaaS

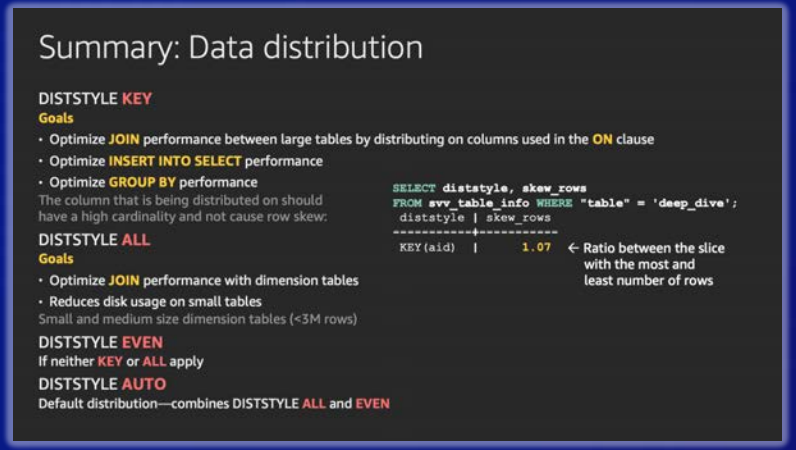
…

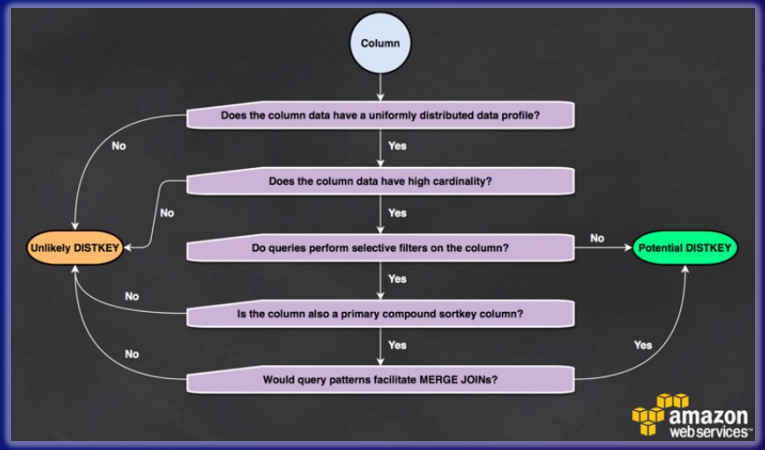


テーブル設計指針

- Amazon Redshift テーブル設計詳細ガイド
  - https://d1.awsstatic.com/events/jp/2017/summit/slide/D3T1-5.pdf
- re:Invent 2019 Deep dive and best practice for Amazon Redshift
  - https://d1.awsstatic.com/events/reinvent/2019/Deep_dive_and_best_practices_for_Amazon_Redshift_ANT418.pdf
- Amazon Redshift Engineering’s Advanced Table Design Playbook
  - https://aws.amazon.com/jp/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-preamble-prerequisites-and-prioritization/







クエリのボトルネック分析方法

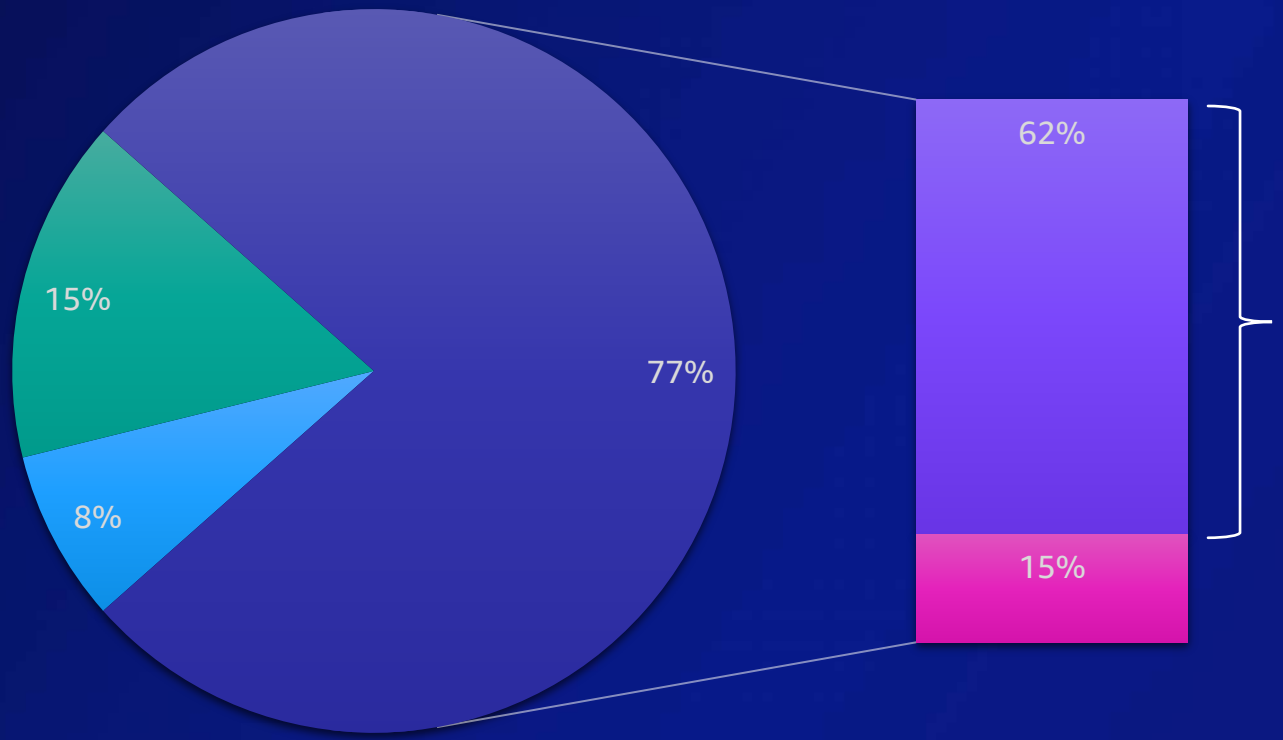
クエリのボトルネック分析でよくある誤解

実行計画（STL_EXPLAIN）を読めばよい

コストが大きいオペレーションがボトルネック	▶	コストは見積（予測）、ボトルネックでないことも
非効率とされるオペレーションがボトルネック	▶	非効率なオペレーションでもボトルネックでないことも

クエリチューニングで大切なこと：時間ベース分析

どこに時間がかかっているか計測、ボトルネックを特定



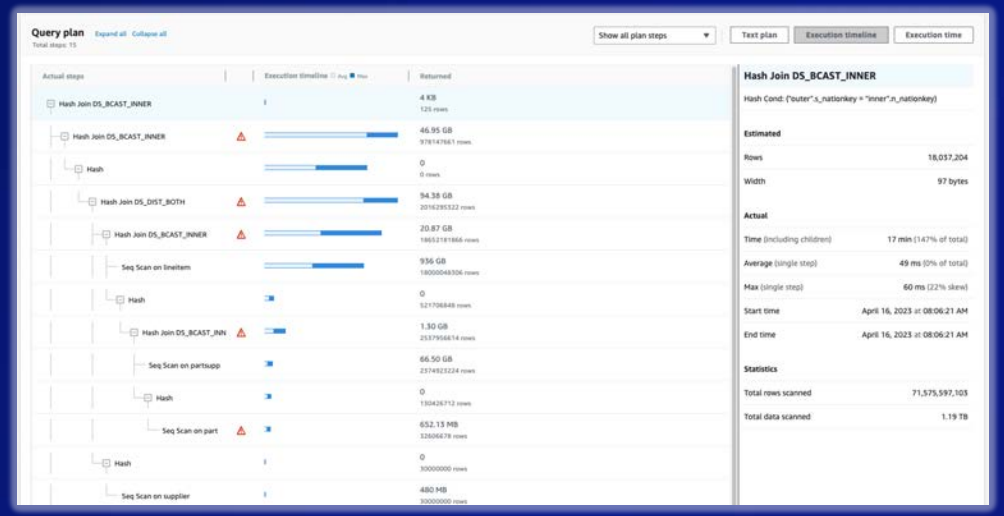
①ボトルネック分析
どこに時間がかかっているかを特定

②チューニング
時間がかかっている箇所をチューニング

Amazon Redshift の性能情報

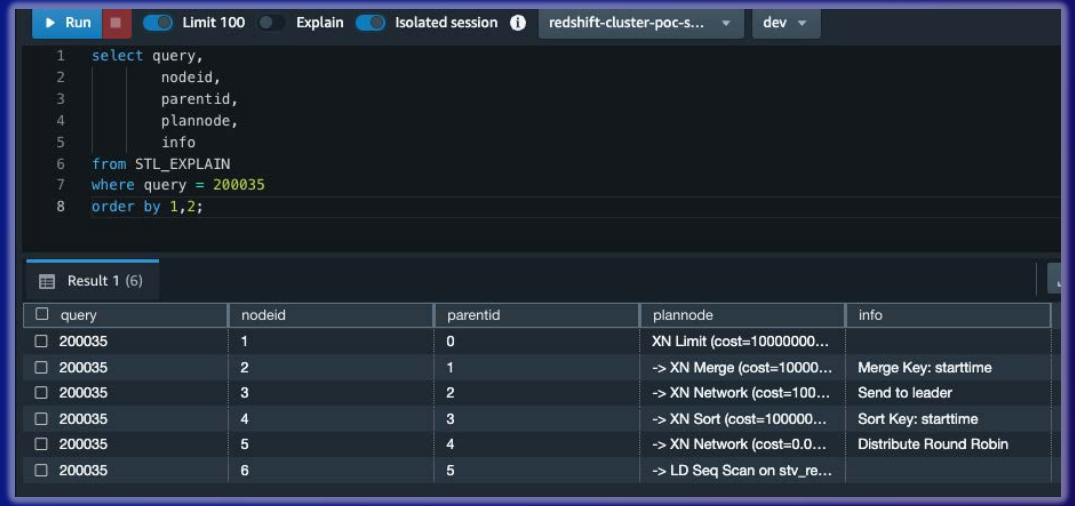
AWS マネジメントコンソール

[Amazon Redshift] – [Cluster performance / Query Monitoring]



システムテーブル・ビュー

Amazon Redshift に接続して性能情報を保持している
システムテーブル・ビューに問合せ



query	nodeid	parentid	plannode	info
200035	1	0	XN Limit (cost=10000000...	
200035	2	1	-> XN Merge (cost=10000...	Merge Key: starttime
200035	3	2	-> XN Network (cost=100...	Send to leader
200035	4	3	-> XN Sort (cost=100000...	Sort Key: starttime
200035	5	4	-> XN Network (cost=0.0...	Distribute Round Robin
200035	6	5	-> LD Seq Scan on stv_re...	

マネジメントコンソール

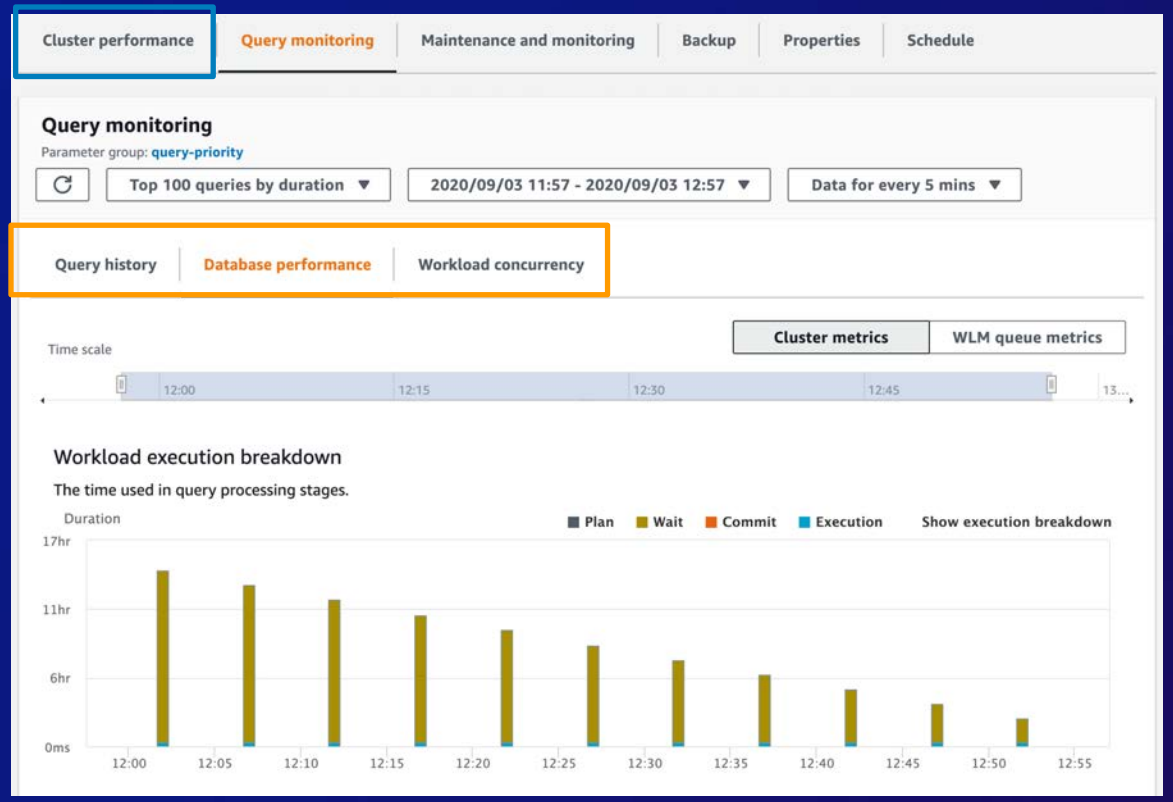
マネジメントコンソール → Amazon Redshift → クラスター → [クラスター名]

クラスターのパフォーマンス
CPU、ディスクI/O、ネットワークI/Oなど

クエリの履歴
クエリやロードの実行時間やステータスなど

データベースパフォーマンス
スループット、 レイテンシーなど

ワークロードコンカレンシー
クエリの同時実行状況 (キュー待ちや実行時間)



システムテーブル・ビューの種類

- 4 種類のシステムテーブル/ビューを保持
  - 発生元や用途によって異なる
  - いずれも DB 領域外に保持されるためスナップショットには含まれない
  - 最大で過去 7 日間のデータが保存されている(一時停止/再開しても残る)

列名	STL	STV	SVL	SVV
タイプ	テーブル	仮想テーブル	ビュー	ビュー
生成方法	ディスク上のログから生成	オンメモリーデータから生成	STLへの参照	STVへの参照
用途	過去の実行記録の参照	現在進行中の処理の参照	STL/STVデータを組み合わせて別軸で分析	

出典: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/c_types-of-system-tables-and-views.html

システムテーブル・ビューの可視化

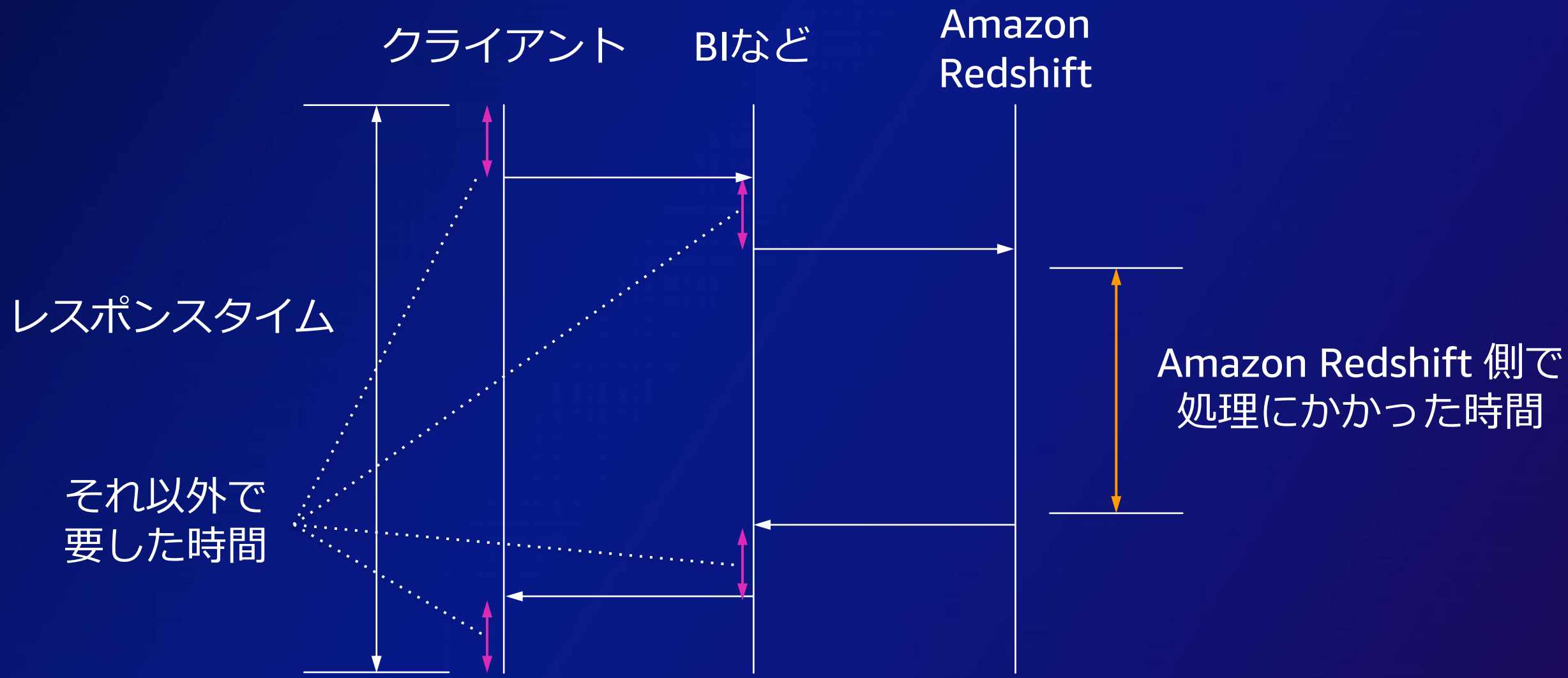
Amazon Managed Grafana を利用したシステムテーブル・ビューの参照

参考: https://yohei-a.hatenablog.jp/entry/20220720/1658268585

クエリのボトルネック分析の俯瞰図

Amazon Redshift 側で時間がかかっているか

Amazon Redshift 側で時間がかかっているか



Amazon Redshift 側でのクエリ実行時間

マネジメントコンソール – Amazon Redshift – [クラスター] – [Query Monitoring] – [Query History]

経過時間

Queries and loads (19)

Terminate query

Filter queries

1

Start time	Query	Status	Duration ▼	SQL
Apr 16th, 2023 07:31:39 AM 6 hours ago	1800236,1800401,1800391	Completed	16 min	copy partsupp from 's3://redshift-downloads/TPC-H/2.18/3TB/partsupp/' iam_role '' delimiter '|' region 'us-east-1' --RequestID=b0d58826-63ad-4808-ab7c-51bcf0117ee1; TraceID=1-643b2596-42e504f22907c8674d5533f0
Apr 16th, 2023 08:06:20 AM 5 hours ago	1800683	Completed	9 min	select /* TPC-H Q9 */ nation, o_year, sum(amount) as sum_profit from (select n_name as nation, extract(year from o_orderdate) as o_year, l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount from part, supplier, lineitem, partsupp, orders, nation where s_ ...
Apr 16th, 2023 08:06:07 AM 5 hours ago	1800681	Completed	7 min	select /* TPC-H Q9 */ nation, o_year, sum(amount) as sum_profit from (select n_name as nation, extract(year from o_orderdate) as o_year, l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount from part, supplier, lineitem, partsupp, orders, nation where s_ ...
Apr 16th, 2023 07:47:37 AM 6 hours ago	1800466,1800481,1800480...	Completed	5 min	copy customer from 's3://redshift-downloads/TPC-H/2.18/3TB/customer/' iam_role '' delimiter '|' region 'us-east-1' --RequestID=b0d58826-63ad-4808-ab7c-51bcf0117ee1; TraceID=1-643b2596-42e504f22907c8674d5533f0
Apr 16th, 2023 08:27:42 AM 5 hours ago	1800914	Completed	4 min	select /* TPC-H Q5 */ n_name, sum(l_extendedprice * (1 - l_discount)) as revenue from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n ...
Apr 16th, 2023 08:36:02 AM 5 hours ago	1801030	Completed	4 min	select /* TPC-H Q5 */ n_name, sum(l_extendedprice * (1 - l_discount)) as revenue from customer, orders, lineitem, supplier, nation, region where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n ...
Apr 16th, 2023 07:55:52 AM 5 hours ago	1800542	Completed	3 min	select /* TPC-H Q10 */ c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment from customer, orders, lineitem, nation where c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= date '199 ...
Apr 16th, 2023 07:55:55 AM 5 hours ago	1800544	Completed	3 min	select /* TPC-H Q10 */ c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment from customer, orders, lineitem, nation where c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= date '199 ...

＊ Duration は Queue time/Exec time 以外にロック待ち時間も含む

STL_QUERY

クエリの開始・終了時刻やクエリ文字列、Concurrency Scaling実行状況など

列名	説明
query	クエリID
querytxt	クエリテキスト ※4000文字を超える場合は STL_QUERYTEXT 参照
starttime	クエリの開始時間 ※キュー待ち時間含む
endtime	クエリの終了時間 ※キュー待ち時間含む
aborted	0 – 最後まで実行された / 1 – キャンセルされた
concurrency_scaling_status	0 or 1より大きい - メインクラスターで実行 1 - 同時実行スケーリングクラスターで実行

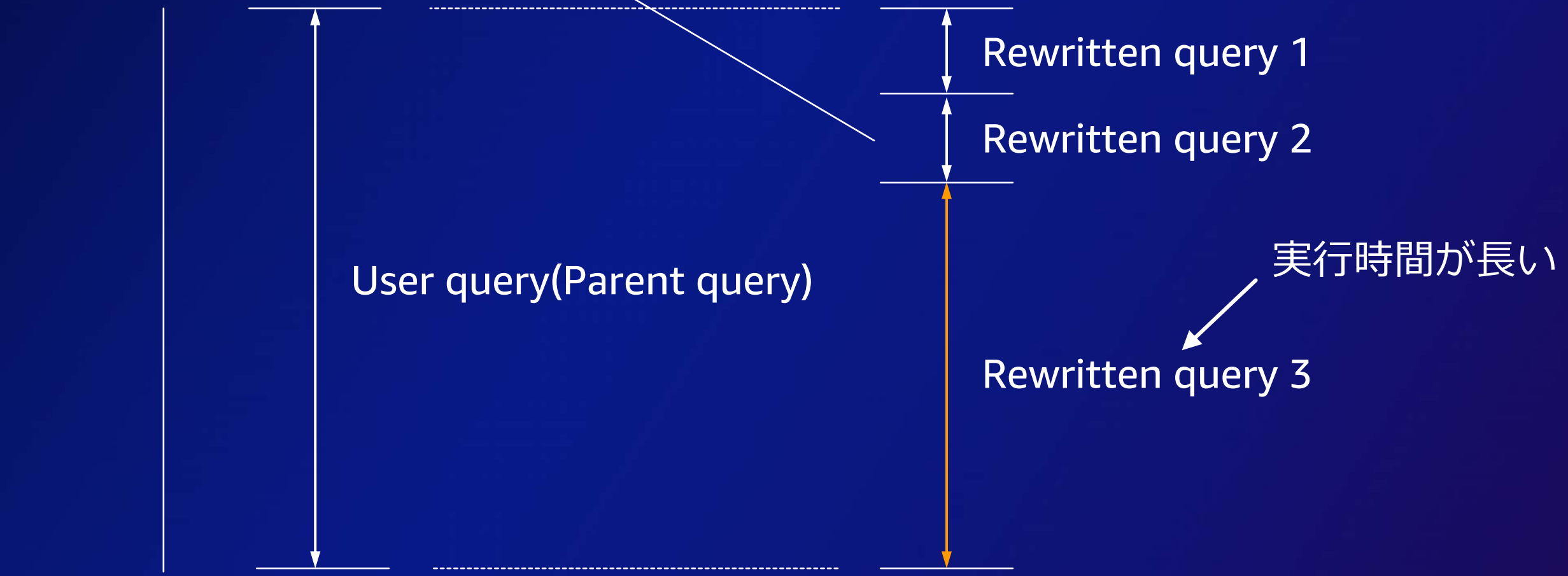
* starttime、endtime はロック待ち時間は含まない

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_STL_QUERY.html

どの Rewritten query で時間がかかっているか

ユーザークエリは分割実行されることがある

ユーザーが実行したクエリは最適化のため書換えられ、
複数の Rewritten query として実行されることがある



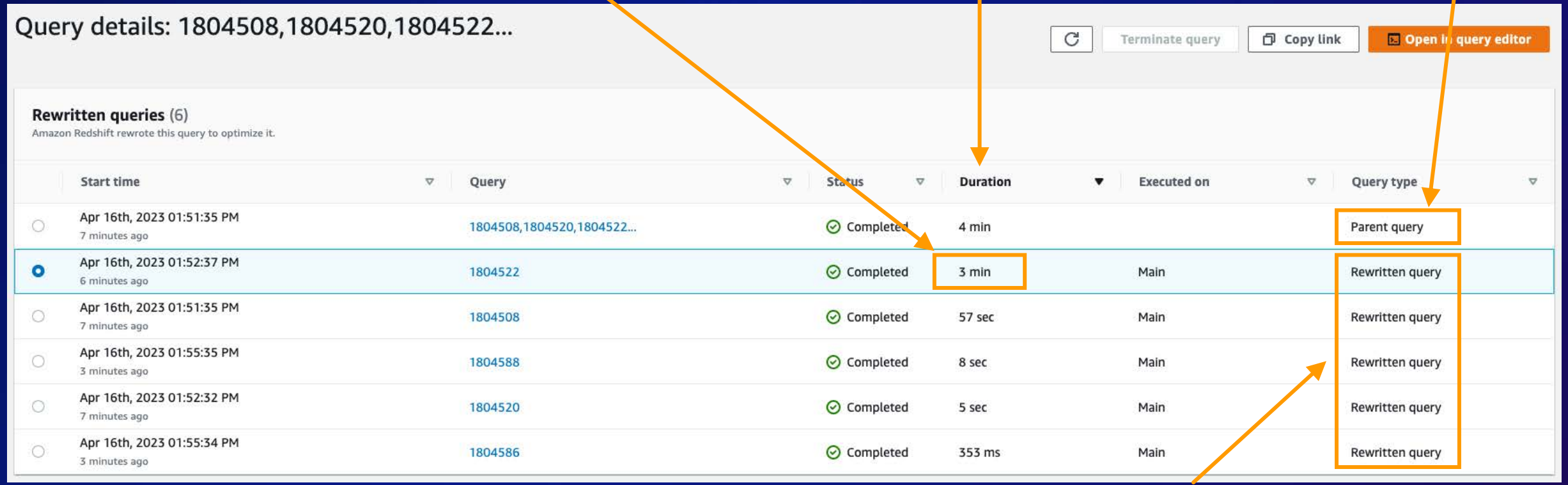
参考: https://docs.aws.amazon.com/redshift/latest/mgmt/performance-metrics-query-execution-details.html

どの Rewritten query が遅いか

[Query History] – [Query details]

実行時間の長い Rewritten query　Duration で降順ソート

Parent



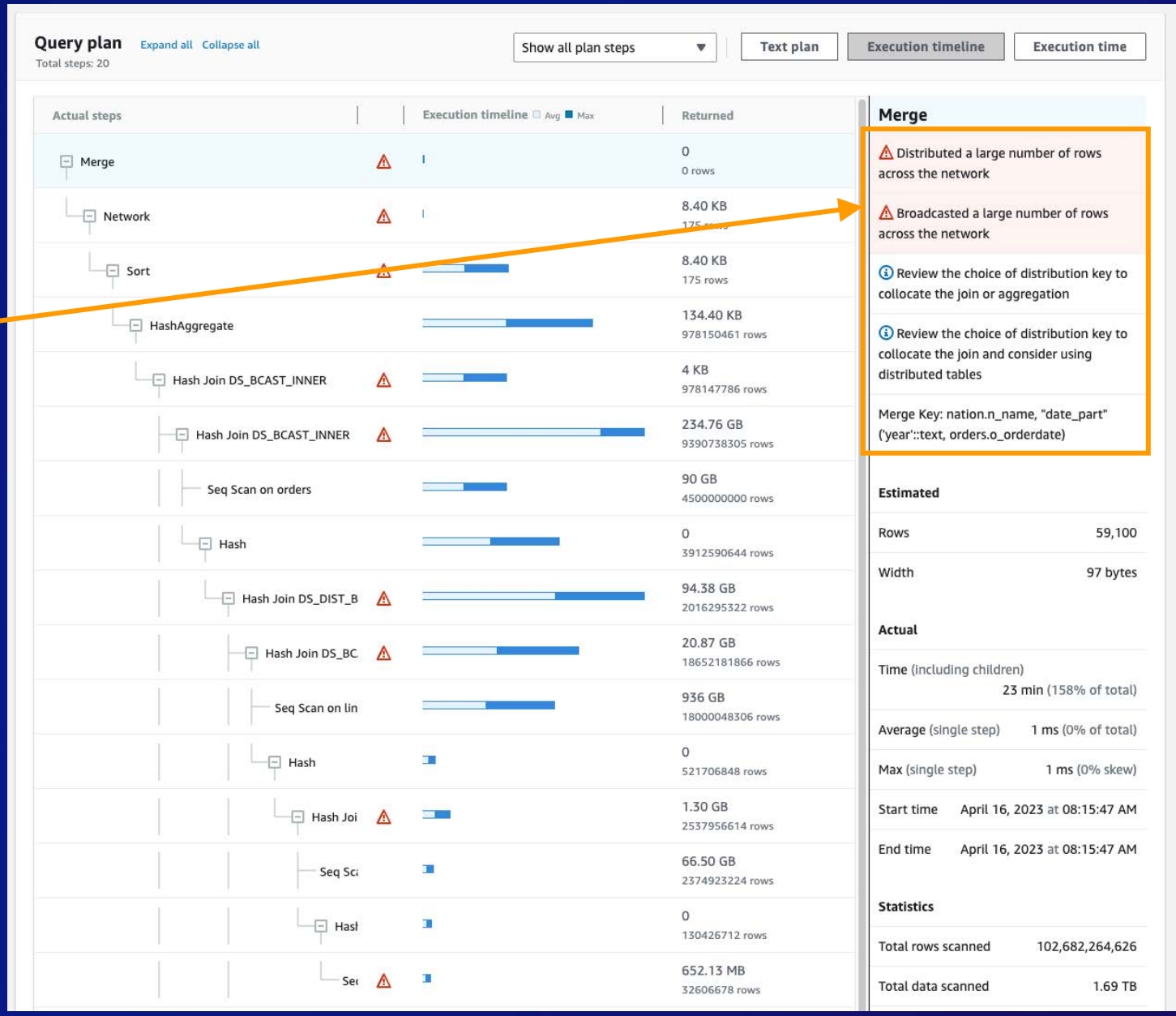
Rewritten

参考: https://docs.aws.amazon.com/redshift/latest/mgmt/analyzing-query-execution.html
https://aws.amazon.com/jp/blogs/big-data/monitor-and-optimize-queries-on-the-new-amazon-redshift-console/

（参考）チューニングアドバイスの確認

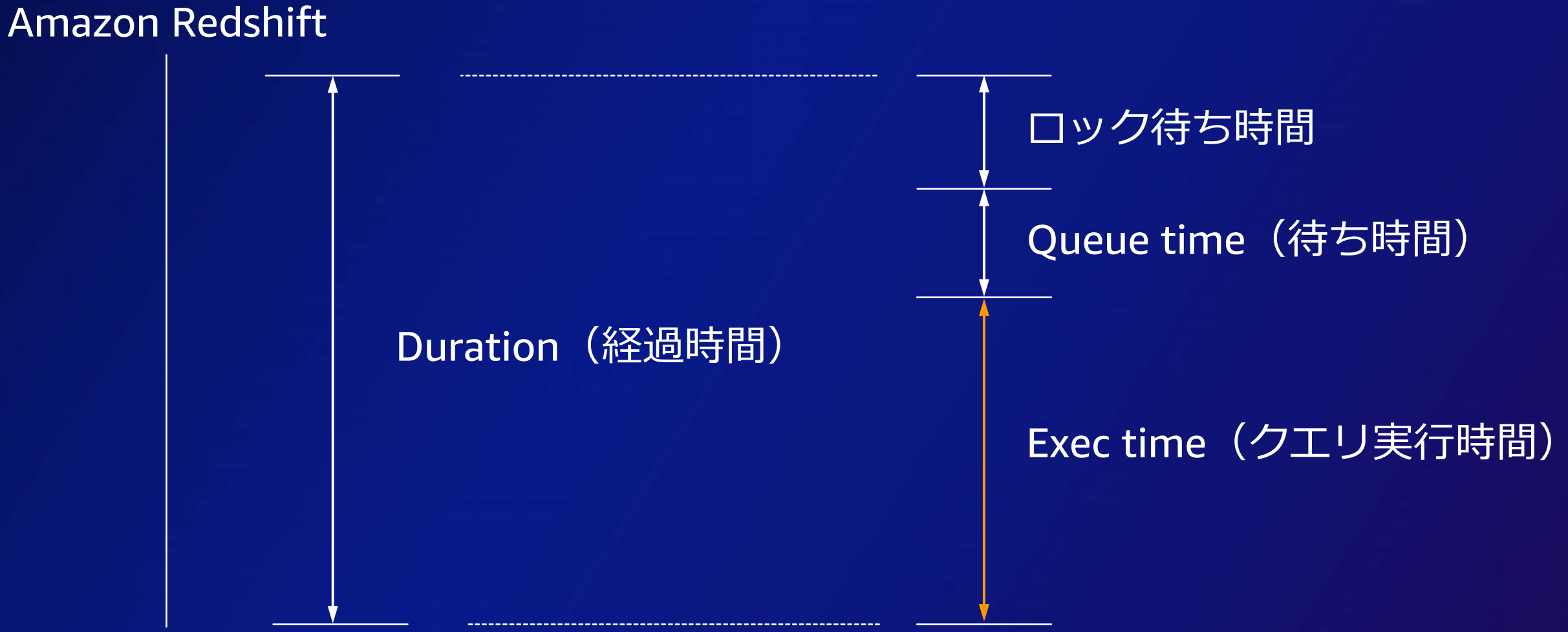
[Query History] – [Query details] – [Query plan]

チューニングアドバイス
✓ 分散キー見直し
✓ ソートキー見直し
など



待たされているのかクエリ自体が遅いのか

待たされているのかクエリ自体が遅いのか



＊ トランザクションにおけるロック待ち時間は Queue time/Exec time のいずれにも含まれない

STL_WLM_QUERY

キュー待ち時間、実行時間、クエリスロット数はいくつだったか、実行されたWLMキュー、クエリの優先度

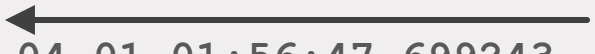
列名	説明
query	クエリID
service_class	クエリが実行されたWLMキュー（WLM サービスクラス ID）
slot_count	WLM クエリスロットの数
total_queue_time	クエリがWLMキュー待ちした時間（マイクロ秒）
total_exec_time	クエリの実行時間（マイクロ秒）
query_priority	クエリの優先度、Lowest/Low/Normal/High/Highest

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_STL_WLM_QUERY.html

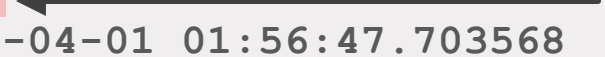
STL_WLM_QUERY

```
select * from stl_wlm_query where query = :pg_last_query_id order by service_class;

-[ RECORD 1 ]-----------+-----------------------------------------------------
userid                  | 1
xid                     | 308727
task                    | 200423
query                   | 200780
service_class           | 1
slot_count              | 1
service_class_start_time | 2023-04-01 01:56:47.699202
queue_start_time        | 2023-04-01 01:56:47.699236
queue_end_time          | 2023-04-01 01:56:47.699236
total_queue_time        | 0
exec_start_time         | 2023-04-01 01:56:47.699243
exec_end_time           | 2023-04-01 01:56:47.703568
total_exec_time         | 4325
service_class_end_time  | 2023-04-01 01:56:47.703568
final_state             | Completed
est_peak_mem            | 0
query_priority          | Normal
service_class_name      | Service class for system user (Health Check)
```

WLMキュー



キュー待ち時間



実行時間

サービスクラス IDの種類・名前との紐付け

- サービスクラスIDと名前の対応は STV_WLM_SERVICE_CLASS_CONFIG で確認
  - https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_STV_WLM_SERVICE_CLASS_CONFIG.html
- サービスクラスに割り当てられている ID の範囲
  - https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/cm-c-wlm-system-tables-and-views.html#wlm-service-class-ids

ID	サービスクラス
1～4	将来の利用のために予約
5	スーパーユーザーキュー
6～13	**手動 WLM キュー（WLM 設定で定義されている）**
14	**ショートクエリアクセラレーション**
15	Amazon Redshift で実行されるメンテナンスアクティビティ用に予約
100～107	**自動 WLM キュー（auto_wlm が true の場合）**

どのセグメントが遅いか、スライスでの偏りは

SVL_QUERY_METRICS（1/2）

クエリ、セグメント、ステップレベルでのメトリクス。中間結果書出しサイズ、スライス間の処理の偏り、セグメント毎の実行時間など

列名	説明
query	クエリID
dimension	query、segment、step のいずれの単位のメトリクスか
segment	セグメント番号
step	ステップ番号
query_cpu_time	クエリ実行に使用されたCPU時間、マルチコアで並列実行されたCPU時間の合計のため、実行時間より長くなることがある。
query_blocks_read	ストレージから読んだブロック数（1ブロック=1MB）、大きいとストレージからの読み込みがボトルネックの可能性あり。
query_temp_blocks_to_disk	ハッシュ・ソート・集計等の中間結果をストレージに書き出したサイズ（MB）。

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_METRICS.html

SVL_QUERY_METRICS（2/2）

列名	説明
cpu_skew	セグメントにおけるスライス単位のCPU使用量の最大値/平均値。値が大きいと特定スライスに処理が偏り、効率よく並列処理できていない。
io_skew	セグメントにおけるスタイス単位のブロック読み込み量の最大値/平均値。値が大きいと特定スライスに処理が偏り、効率よく並列処理できていない。
scan_row_count	ストレージか読んだ行数、削除対象としてマークされた行は、ユーザークエリでのフィルタされる前の行数。
join_row_count	結合された行数、この行数が多いと結合がボトルネックの可能性あり。
nested_loop_join_row_count	Nested Loop で結合された行数、この行数が多いと直積による結合がボトルネックの可能性あり。
return_row_count	クエリがクライアントに返した結果セットの行数。

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_METRICS.html

SVL_QUERY_METRICS_SUMMARY

SVL_QUERY_METRICS のサマリー（1クエリ1行）、ワークロード特性を把握。

```
select * from svl_query_metrics_summary where query = :pg_last_query_id;

-[ RECORD 1 ]--------------+-----------------------------------
...
query_cpu_time             | 2392
query_blocks_read          | 347778
query_execution_time       | 429
query_cpu_usage_percent    | 22.32
query_temp_blocks_to_disk  | 220100
segment_execution_time     | 206
cpu_skew                   | 5.71
io_skew                    | 3.67

...
```

ストレージからの読込量（MB）

ストレージに書出した中間結果のサイズ（MB）

3分半分要しているセグメントがある

スライスでの偏り

SVL_QUERY_METRICS

時間を要しているセグメントとスライス間で偏りがないか確認。

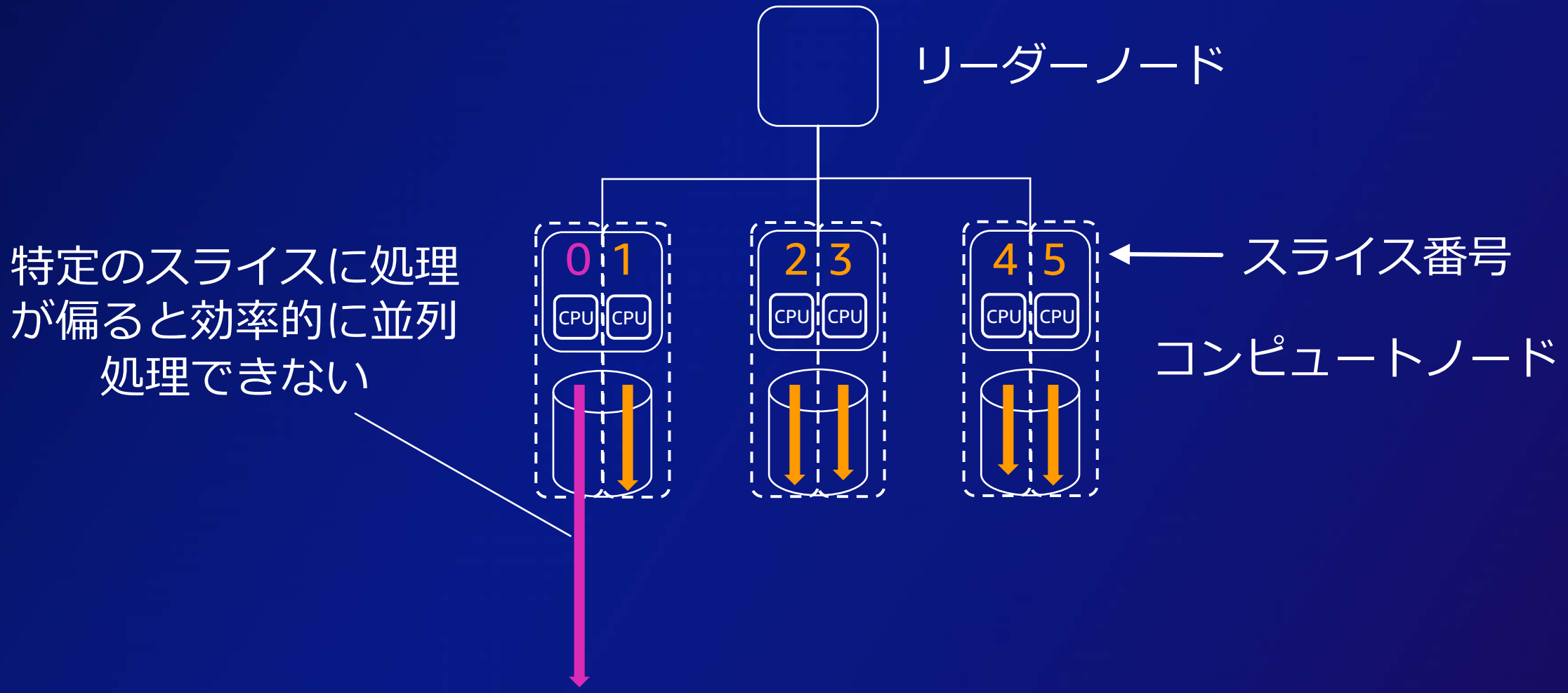
```
select segment,segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;
```

segment	segment_execution_time	cpu_skew	io_skew
2	（単位：秒） 1	5.71	3.67
3	25	1.03	1.02
4	25	4.14	
5	206	1.04	1.00
6	90	1.07	1.00
7	90	1.14	
8	96	1.05	1.00
9	96	1.05	
10	96	1.41	

Segment 5
がボトルネック

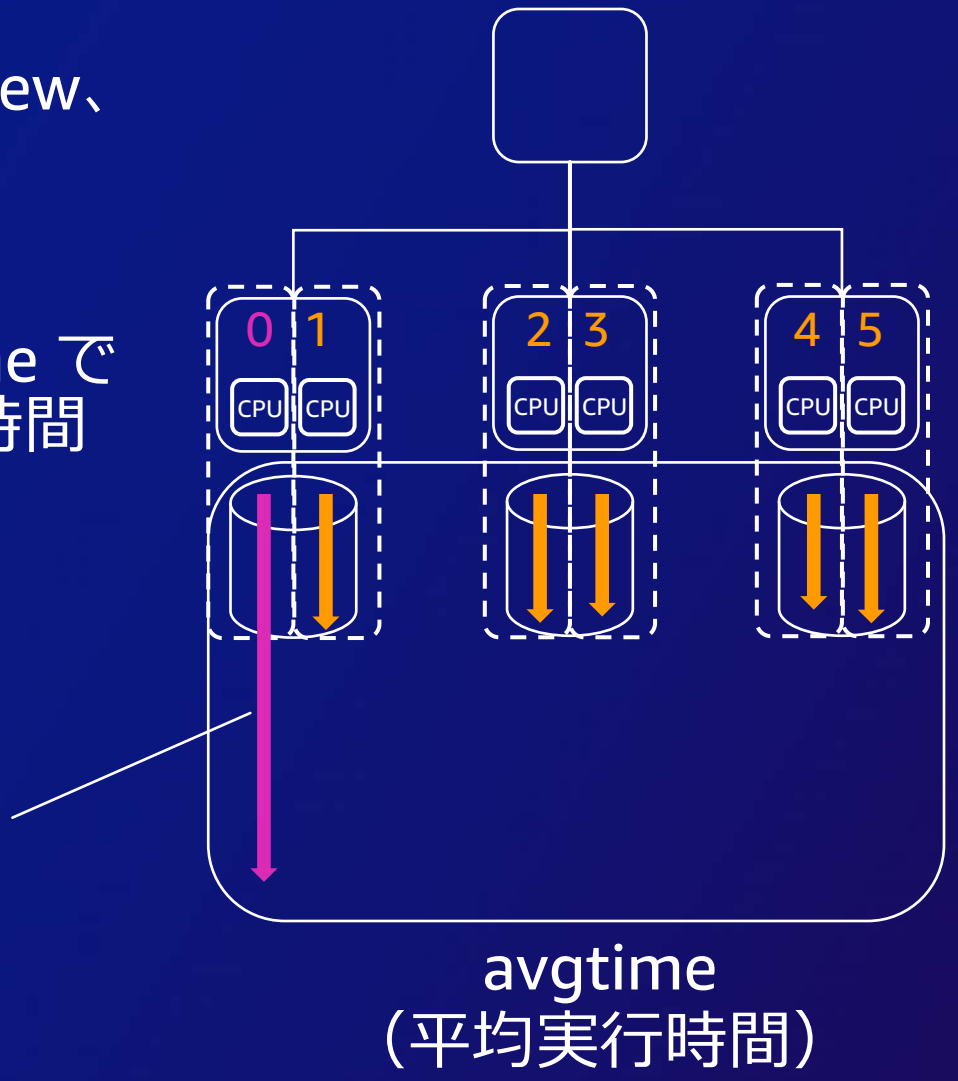
偏りがあるが実行時間が短い
ためボトルネックでない

特定スライスに処理が偏ると非効率



Segment 実行時間の avgtime と maxtime

- SVL_QUERY_METRICS_(SUMMARY) の cpu_skew、io_skew が大きいと偏りがある
- SVL_QUERY_SUMMARY の maxtime、 avgtime で各セグメントにおけるスライスでの最大実行時間と平均実行時間を確認



SVL_QUERY_SUMMARY（1/2）

1ステップ1行。遅いセグメントの特定、スライスで処理の偏りがないか、中間結果の書出しがないか、不要なデータを読んでないか

列名	説明
query	クエリID
stm	ストリーム
seg	セグメント
step	ステップ
maxtime	セグメントのスライスでの実行時間の最大値（マイクロ秒）
avgtime	セグメントのスライスでの実行時間の平均値（マイクロ秒）
rows	ステップで処理された行数（フィルタ後）
byte	ステップで処理されたバイト数（フィルタ後）

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_SUMMARY.html

SVL_QUERY_SUMMARY（2/2）

列名	説明
label	ステップの操作(例: scan tbl=100448 name =user)、 3 桁のテーブル ID は一時テーブル、tbl=0 は定数値のスキャン。
is_diskbased	ハッシュ・ソート・集計などの中間結果をストレージに書いた場合 true(t)。書き出したサイズは SVL_QUERY_METRICS.query_temp_blocks_to_disk で確認できる。
workmem	ハッシュ・ソート・集計などに使われたワークメモリのサイズ（byte）
is_rrscan	範囲限定スキャン：true(t)。ゾーンマップによる範囲限定スキャンでも true になるため、ソートキーの効果の判断には使えない。
rows_pre_filter	ストレージから読込んだ行数。ステップで処理前の行数、rows は処理後の行数。rows_pre_filter – rows が大きい場合、不要なブロックを読んでおり、ソートキーの設定で削減できる可能性がある。

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_SUMMARY.html

SVL_QUERY_SUMMARY

Stream はシリアル実行

Segment は並行実行されることがある

rows_pre_filter - rows が大きいと不要な読込が多い

```
select stm, seg, step, maxtime, avgtime, rows, label, is_diskbased, workmem, rows_pre_filter from svl_query_summary where query = :pg_last_query_id
order by stm ,seg, step;
```

stm	seg	step	maxtime	avgtime	rows	label	is_diskbased	workmem	rows_pre_filter
0	0	0	424959755	414859823	18000048306	scan tbl=110211 name=lineitem_ok_varchar1	f	0	18000048306
0	0	1	424959755	414859823	18000048306	project	f	0	0
0	0	2	424959755	414859823	18000048306	dist	f	0	0
0	1	0	424974525	424971875	18000048306	scan tbl=1327 name=Internal Worktable	f	0	0
0	1	1	424974525	424971875	18000048306	project	f	0	0
0	1	2	424974525	424971875	18000048306	hash tbl=1045	t	26732396544	0
1	2	0	467994865	456616182	18000048306	scan tbl=110211 name=lineitem_ok_varchar1	f	0	18000048306
1	2	1	467994865	456616182	18000048306	project	f	0	0
1	2	2	467994865	456616182	18000048306	dist	f	0	0
1	3	0	888120062	872225626	18000048306	scan tbl=1576 name=Internal Worktable	f	0	0
1	3	1	888120062	872225626	18000048306	project	f	0	0
1	3	2	888120062	872225626	18595188330	hjoin tbl=1045	f	0	0
1	3	3	888120062	872225626	18595188330	project	f	0	0
1	3	4	888120062	872225626	18595188330	project	f	0	0
1	3	5	888120062	872225626	112	aggr tbl=1056	f	450232320	0
1	3	6	888120062	872225626	112	dist	f	0	0
...									
2	6	0	1407	1407	0	merge	f	0	0
2	6	1	1407	1407	7	project	f	0	0
2	6	2	1407	1407	7	return	f	0	0

maxtime と avgtime の差が大きいとスライスで偏り

メモリで収まらず中間結果をストレージへ書出し

SVL_QUERY_REPORT（1/2）

1スライス1行。SVL_QUERY_SUMMARY より詳細にスライス別の情報を確認できる。

列名	説明
query	クエリID
slice	スライス
segment	セグメント
step	ステップ
start_time	セグメントの実行開始時間
end_time	セグメントの実行終了時間
elapsed_time	セグメントの実行時間（マイクロ秒）
rows	ステップで処理後の行数（スライスあたり）
bytes	ステップで処理後のバイト数（スライスあたり）

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_REPORT.html

SVL_QUERY_REPORT（2/2）

列名	説明
label	ステップの操作(例: scan tbl=100448 name =user)、 3 桁のテーブル ID は一時テーブル、tbl=0 は定数値のスキャン。
is_diskbased	ハッシュ・ソート・集計などの中間結果をストレージに書いた場合 true(t)。書き出したサイズは SVL_QUERY_METRICS.query_temp_blocks_to_disk で確認できる。
workmem	ハッシュ・ソート・集計などに使われたワークメモリのサイズ（byte）
is_rrscan	範囲限定スキャン：true(t)。ゾーンマップによる範囲限定スキャンでも true になるため、ソートキーの効果の判断には使えない。
rows_pre_filter	ストレージから読込んだ行数。ステップで処理前の行数、rows は処理後の行数。rows_pre_filter – rows が大きい場合、不要なブロックを読んでおり、ソートキーの設定で削減できる可能性がある。

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVL_QUERY_REPORT.html

SVL_QUERY_REPORT

start_time、end_time からセグメントが並行実行されているか確認

```
select segment, step, slice, start_time, end_time, elapsed_time, rows, bytes from svl_query_report where query = 2800188 order by
segment, step ,slice;
 segment | step | slice |         start_time         |          end_time          | elapsed_time |    rows    |    bytes
---------+------+-------+----------------------------+----------------------------+--------------+------------+-------------
       0 |    0 |     0 | 2023-04-21 00:04:25.155291 | 2023-04-21 00:05:32.939376 |     67784085 | 1105831627 | 22116632540
       0 |    0 |     1 | 2023-04-21 00:04:25.155403 | 2023-04-21 00:05:38.766747 |     73611344 | 1205274176 | 24105483520
       0 |    0 |     2 | 2023-04-21 00:04:25.155375 | 2023-04-21 00:05:32.911754 |     67756379 | 1122205096 | 22444101920
       0 |    0 |     3 | 2023-04-21 00:04:25.155409 | 2023-04-21 00:05:44.347871 |     79192462 | 1318458166 | 26369163320
       0 |    0 |     4 | 2023-04-21 00:04:25.154956 | 2023-04-21 00:07:48.200542 |    203045586 | 3103552138 | 62071042760
       0 |    0 |     5 | 2023-04-21 00:04:25.154987 | 2023-04-21 00:05:31.099287 |     65944300 | 1107365471 | 22147309420
       0 |    0 |     6 | 2023-04-21 00:04:25.154981 | 2023-04-21 00:05:32.968074 |     67813093 | 1145335102 | 22906702040
       0 |    0 |     7 | 2023-04-21 00:04:25.157083 | 2023-04-21 00:05:25.880848 |     60723765 | 1023442529 | 20468850580
       0 |    0 |     8 | 2023-04-21 00:04:25.155747 | 2023-04-21 00:05:29.795866 |     64640119 | 1090088206 | 21801764120
       0 |    0 |     9 | 2023-04-21 00:04:25.155846 | 2023-04-21 00:05:32.324185 |     67168339 | 1140357965 | 22807159300
       0 |    0 |    10 | 2023-04-21 00:04:25.155854 | 2023-04-21 00:05:32.319552 |     67163698 | 1128260563 | 22565211260
       0 |    0 |    11 | 2023-04-21 00:04:25.156873 | 2023-04-21 00:05:26.992209 |     61835336 | 1044099754 | 20881995080
       0 |    0 |    12 | 2023-04-21 00:04:25.152224 | 2023-04-21 00:05:19.451745 |     54299521 | 1071680574 | 21433611480
       0 |    0 |    13 | 2023-04-21 00:04:25.152327 | 2023-04-21 00:05:17.100144 |     51947817 | 1014295796 | 20285915920
       0 |    0 |    14 | 2023-04-21 00:04:25.152333 | 2023-04-21 00:05:24.189677 |     59037344 | 1183561831 | 23671236620
       0 |    0 |    15 | 2023-04-21 00:04:25.152338 | 2023-04-21 00:05:25.505624 |     60353286 | 1188744980 | 23774899600
       0 |    1 |     0 | 2023-04-21 00:04:25.155291 | 2023-04-21 00:05:32.939376 |     67784085 | 1105831627 |           0
       0 |    1 |     1 | 2023-04-21 00:04:25.155403 | 2023-04-21 00:05:38.766747 |     73611344 | 1205274176 |           0
       0 |    1 |     2 | 2023-04-21 00:04:25.155375 | 2023-04-21 00:05:32.911754 |     67756379 | 1122205096 |           0
       0 |    1 |     3 | 2023-04-21 00:04:25.155409 | 2023-04-21 00:05:44.347871 |     79192462 | 1318458166 |           0
       0 |    1 |     4 | 2023-04-21 00:04:25.154956 | 2023-04-21 00:07:48.200542 |    203045586 | 3103552138 |           0
```

同じ step で rows、bytes に偏りがあるとスライスで処理に偏り

ボトルネックを特定できたら？

遅いステップから実行計画・クエリ文への紐付け

SVL_QUERY_SUMMARY の label → STL_EXPLAIN の plannode の対応

SVL_QUERY_SUMMARY の label	STL_EXPLAIN の plannode
AGGR	Aggregate HashAggregate GroupAggregate
BCAST (broadcast)	DS_BCAST_INNER
DIST (distribute)	DS_DIST_NONE DS_DIST_ALL_NONE DS_DIST_INNER DS_DIST_ALL_INNER DS_DIST_ALL_BOTH
HJOIN (hash join)	Hash Join
MJOIN (merge join)	Merge Join

参考: https://docs.aws.amazon.com/redshift/latest/dg/query-plan-summary-map.html

ステップと実行計画・クエリ文の紐付け

SVL_QUERY_SUMMARY

```
select stm, seg, step, label from svl_que
1400354 order by stm ,seg, step;
 stm | seg | step |                      label
-----+-----+------+-----------------------------------
   0 |   0 |    0 | scan   tbl=110211 name=lineitem
...
   1 |   2 |    0 | scan   tbl=110211 name=lineitem
...
   1 |   3 |    2 | hjoin  tbl=1045
```

STL_EXPLAIN

```
select nodeid,parentid, substring(plannode,1,70) plannode, substring
stl_explain where query = 1400354 order by nodeid, parentid;
 nodeid | parentid |                                        plannode
--------+----------+-----------------------------------------------------------------------
      1 |        0 | XN Merge  (cost=13138947680290.55..13138947680290.57 rows=7 width=20)
      2 |        1 |   ->  XN Network  (cost=13138947680290.55..13138947680290.57 rows=7 wi
      3 |        2 |         ->  XN Sort  (cost=13138947680290.55..13138947680290.57 rows=7
      4 |        3 |               ->  XN HashAggregate  (cost=12138947680290.44..121389476
      5 |        4 |                     ->  XN Hash Join DS_DIST_BOTH  (cost=345149836.16.
      6 |        5 |                           ->  XN Seq Scan on lineitem a  (
      7 |        5 |                           ->  XN Hash  (cost=180000481.28..180000481.2
      8 |        7 |                                 ->  XN Seq Scan on lineitem
```

STL_EXPLAIN

クエリの実行計画、アラートがないか、非効率な操作がないか

列名	説明
userid	ユーザーID
query	クエリID
nodeid	クエリの実行における 1 つ以上のステップに対応するノードの計画ノード識別子。
parentid	親ノードの計画ノード識別子。親ノードには、いくつかの子ノードがあります。例えば、merge join は、結合されたテーブルに対する複数の scan の親です。
plannode	実行計画の操作やコストなど。コンピューティングノードで実行される操作は先頭に XN がつく。
info	Where句のフィルタ条件や結合条件など

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_STL_EXPLAIN.html

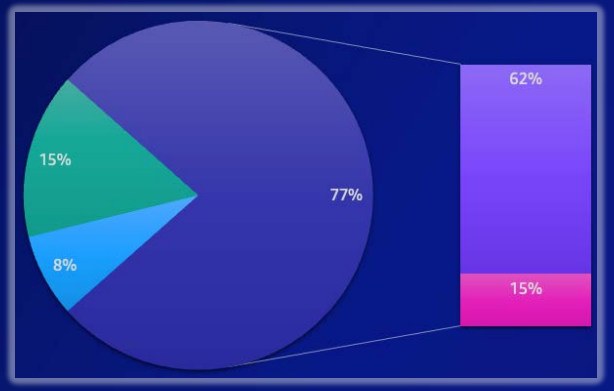
STL_EXPLAIN の plannode の確認ポイント

- Cost
  - クエリオプティマイザが見積もったコスト（予測値のため、あくまで参考値）
  - 下から上に見て、コストが急激に大きくなっている操作はボトルネックの可能性あり
- Join
  - Merge Join：最も効率がよい、結合対象表の分散キー・ソートキーが一致する必要あり
  - Hash Join：Merge Join できないときに選択される
  - Nested Loop：直積、非常に非効率。結合条件が抜けてないか確認する
- Redistribution（ノード間通信）
  - DS_BCAST_INNER：内部表全体を全ノードに転送
  - DS_DIST_ALL_INNER：内部表全体を単一ノードに転送（外部表がALL分散のため）
  - DS_BCAST_BOTH：両方の表を転送

クエリのボトルネック分析方法 サマリー

①時間ベース分析

どこに時間を要しているか

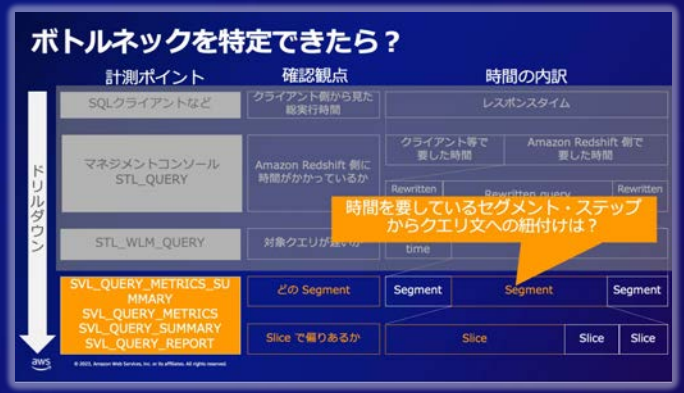


②ボトルネック分析方法

システムテーブル・ビューの見方

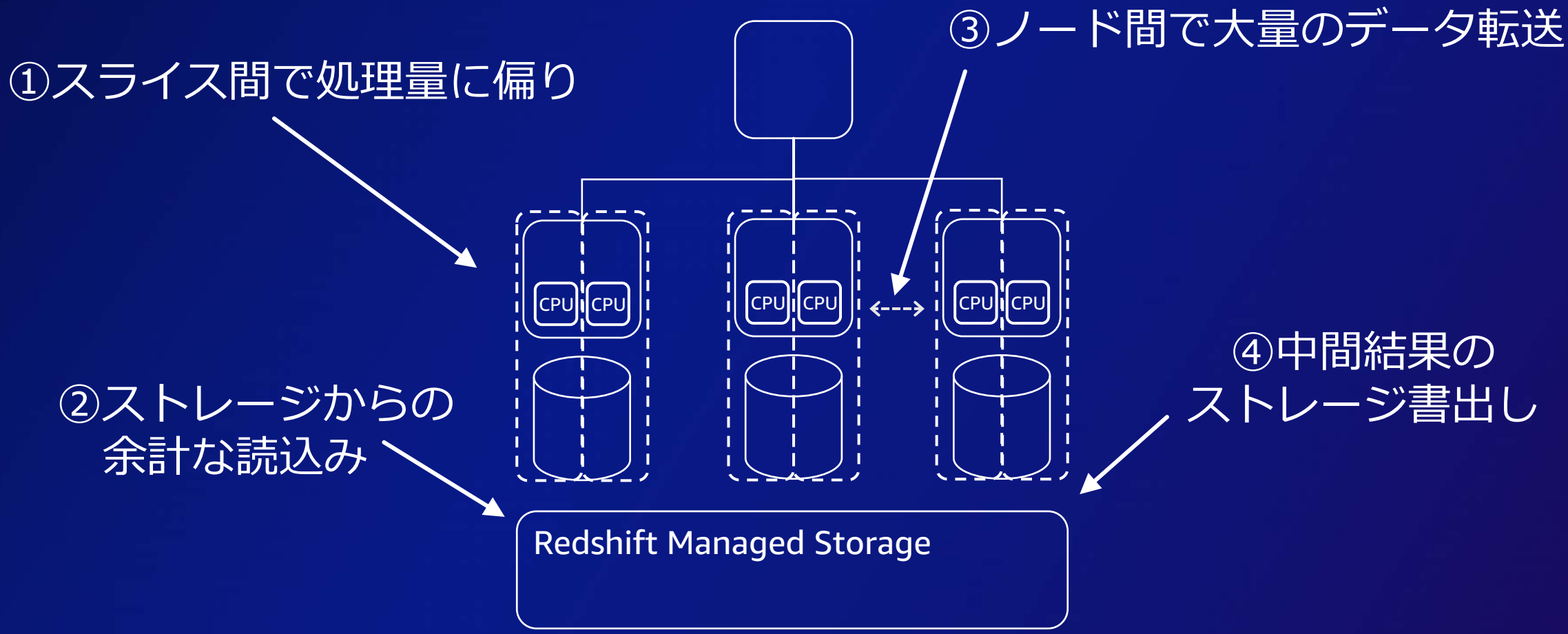
③チューニング箇所特定

クエリ文のチューニング箇所特定



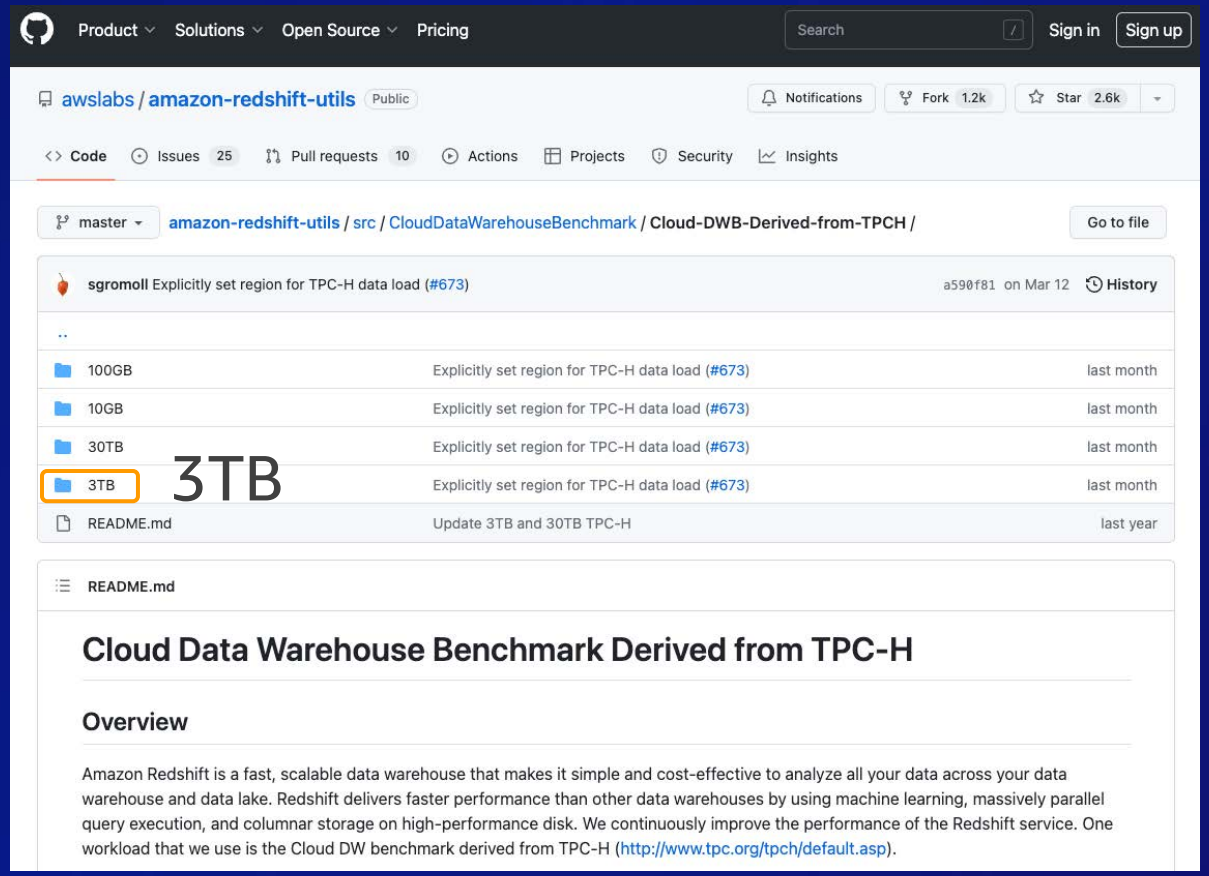
具体的なクエリチューニング例

Amazon Redshift で発生するボトルネック例



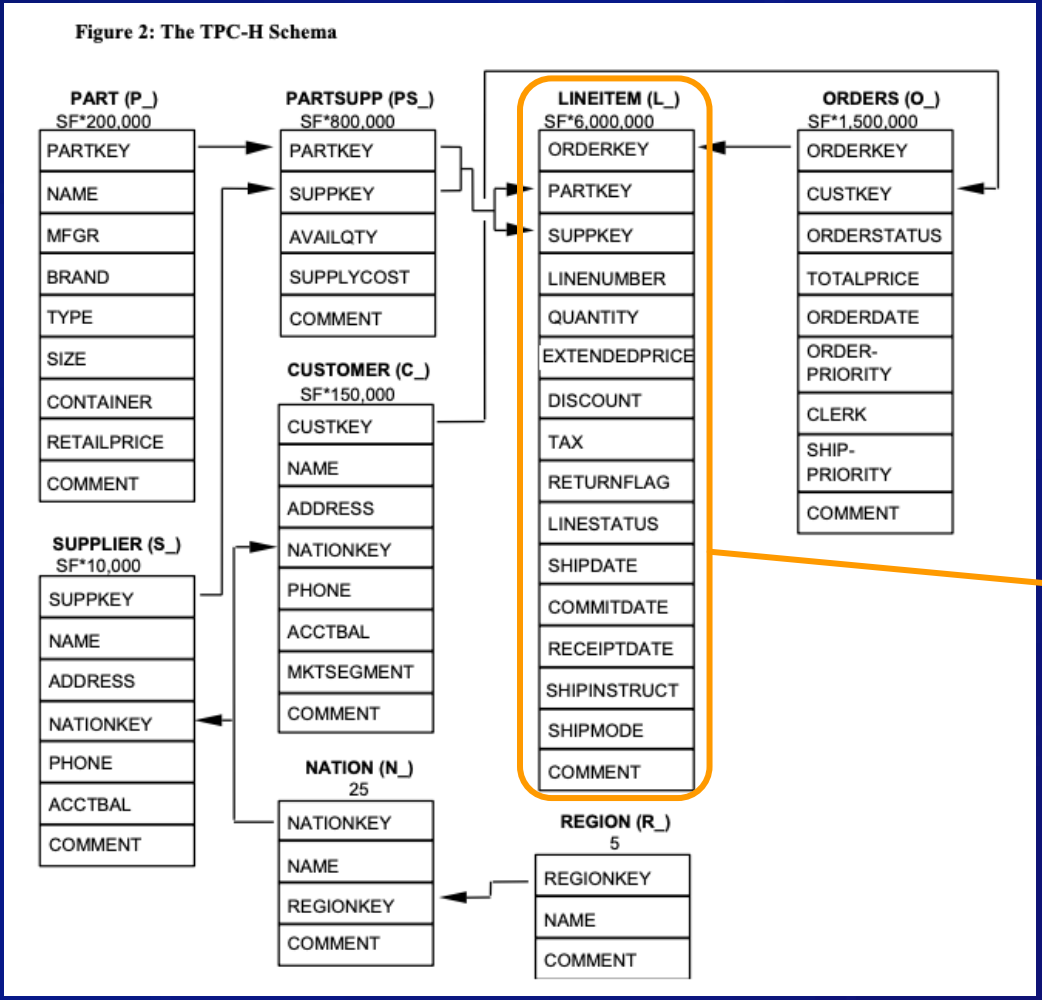
検証に使用したデータ：TPC-H

Cloud DWH Benchmark Derived from TPC-H



https://github.com/awslabs/amazon-redshift-utils/tree/master/src/CloudDataWarehouseBenchmark/Cloud-DWB-Derived-from-TPCH

検証に使用したデータ：TPC-H の ER 図



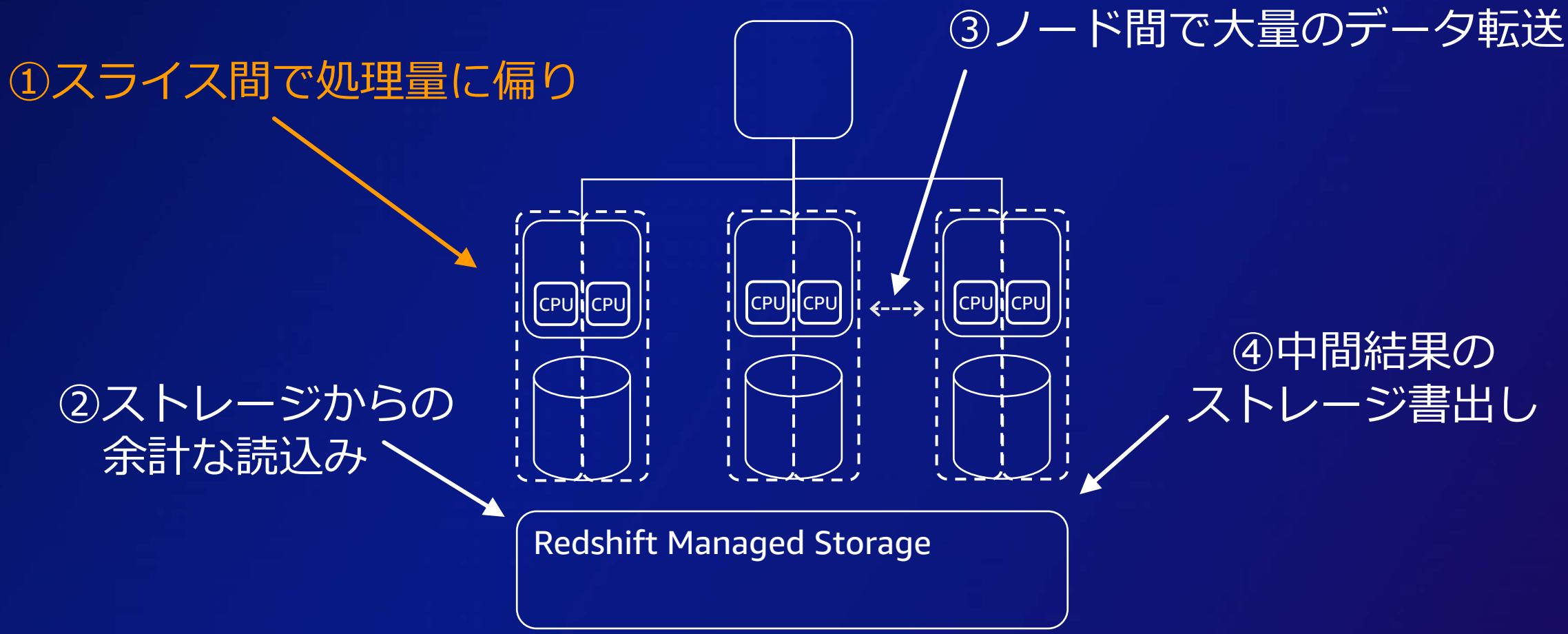
検証シナリオによって lineitem テーブルの

- データ増幅
- カラム型変更

https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf

検証環境：ra3.4xlarge x 4 node

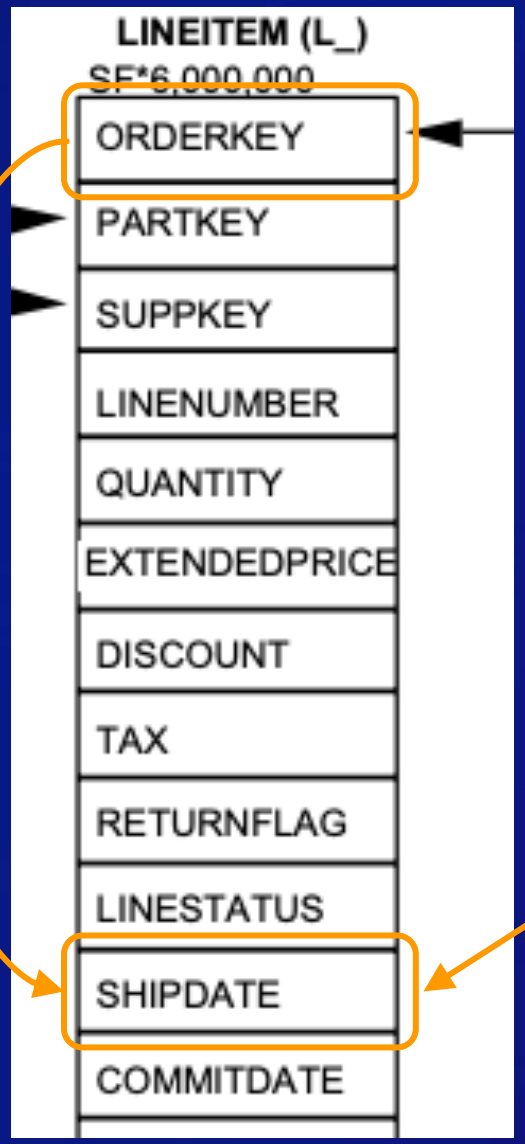
①スライス間で処理量に偏り



①スライス間で処理量に偏り

検証に使用したデータ

①分散キー変更
orderkey → shipdate



②特定の値を増幅
偏りを作る

①スライス間で処理量に偏り

検証に使用したデータ

- 分散キーを l_shipdate に

```
create table lineitem (
…
) distkey(l_shipdate) sortkey(l_shipdate,l_orderkey);
```

- TPC-H(3TB) のデータをロード

```
copy lineitem from 's3://redshift-downloads/TPC-
H/2.18/3TB/orders/' …
```

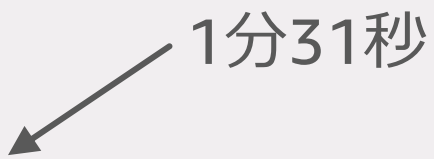
- l_shipdate の特定の値に偏りを作る（データ増幅）

```
insert into lineitem(select … '1997-07-03'… from lineitem
limit 1992505668);
```

①スライス間で処理量に偏り

チューニング前

```
select a.l_shipdate, count(a.l_shipdate)
from lineitem a, lineitem b
where a.l_orderkey = b.l_orderkey
    and a.l_shipdate = b.l_shipdate
group by a.l_shipdate
order by count(a.l_shipdate) desc
limit 3;

 l_shipdate |   count
------------+------------
 1997-07-03 | 5756528960
 1996-01-29 |    7741401
 1992-07-14 |    7740931
(3 rows)

Time: 91035.609 ms (01:31.036)
```



①スライス間で処理量に偏り

ボトルネック分析

SVL_QUERY_METRICS

```
select dimension, segment, segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;

 dimension | segment | segment_execution_time | cpu_skew | io_skew
-----------+---------+------------------------+----------+---------
 segment   |       0 |                     91 |     3.26 |    3.47
```

Segment 0 で1分31秒

スライス間で偏り

①スライス間で処理量に偏り

ボトルネック分析

SVL_QUERY_SUMMARY

```
select stm, seg, maxtime, avgtime, label from svl_query_summary where
query = :pg_last_query_id order by stm, seg;
```

最大91秒

平均28秒

stm	seg	maxtime	avgtime	label
0	0	91234057	28525276	save tbl=1068
0	0	91234057	28525276	project
0	0	91234057	28525276	scan tbl=111347 name=lineitem
0	0	91234057	28525276	sort tbl=1067
0	0	91234057	28525276	project
0	0	91234057	28525276	project
0	0	91234057	28525276	mjoin tbl=1057
0	0	91234057	28525276	scan tbl=111347 name=lineitem
0	0	91234057	28525276	project

①スライス間で処理量に偏り

ボトルネック分析

SVL_QUERY_REPORT

```
select segment, slice, elapsed_time, rows, rows_pre_filter from
svl_query_report where query = :pg_last_query_id order by segment, step,
slice;
 segment | slice | elapsed_time |    rows     | rows_pre_filter
---------+-------+--------------+-------------+-----------------
       0 |     0 |     24525493 | 1105831627  |      1105831627
       0 |     1 |     26877039 | 1205274176  |      1205274176
       0 |     2 |     25144271 | 1122205096  |      1122205096
       0 |     3 |     29230330 | 1318458166  |      1318458166
       0 |     4 |     91234057 | 3103552138  |      3103552138
       0 |     5 |     23518775 | 1107365471  |      1107365471
       0 |     6 |     24178578 | 1145335102  |      1145335102
```

スライス4は行数、処理時間が約3倍

①スライス間で処理量に偏り

ボトルネック分析

SVV_TABLE_INFO

```
select "table", diststyle, sortkey1, skew_rows from svv_table_info where
"table" = 'lineitem’;

        table          |    diststyle     |  sortkey1   | skew_rows
-----------------------+------------------+-------------+-----------
 lineitem              | KEY(l_shipdate) | l_shipdate |      3.06
```

行の分散に偏り

①スライス間で処理量に偏り

ボトルネック分析

STV_TBL_PERM

```
select slice, rows, block_count from stv_tbl_perm where name
= 'lineitem' order by rows desc limit 5;
 slice |    rows     | block_count
-------+-------------+-------------
     4 | 3103552138 |      158299
     3 | 1318458166 |       62623
     1 | 1205274176 |       57244
    15 | 1188744980 |       56418
    14 | 1183561831 |       56175
(5 rows)
```

←── スライス4 は他のスライスより、行数、ブロック数が約3倍

①スライス間で処理量に偏り

チューニング案検討

- クエリの結合キーは l_shipdate、l_orderkey
- 分散キーは値の種類が多く、各値に偏りがない列が適している

	l_shipdate		l_orderkey	
値の種類： select count(distinct 列)	2526	少ない	4500000000	
値の偏り： select 列, count(*) … group by 列 order by count(*) desc;	l_shipdate	count	l_orderkey	count
	1997-07-03	5756528960	5037202980	23
	1996-01-29	7494223	9775295239	23
	1996-11-11	7492092	1334586209	23

偏りがある

①スライス間で処理量に偏り

チューニング実施

分散キーを l_orderkey に変更

```
alter table lineitem alter distkey l_orderkey;
```

* 分散スタイルや分散キーの変更時は VACUUM は不要。

SVV_TABLE_INFO

```
select "table", diststyle, sortkey1, skew_rows from svv_table_info where
"table" = 'lineitem’;

  table    |     diststyle     |  sortkey1   | skew_rows
-----------+-------------------+-------------+-----------
 lineitem | KEY(l_orderkey) | l_shipdate |      1.00
```

←── 偏りが解消

①スライス間で処理量に偏り

チューニング結果

```
select a.l_shipdate, count(a.l_shipdate)
from lineitem a, lineitem b
where a.l_orderkey = b.l_orderkey
    and a.l_shipdate = b.l_shipdate
group by a.l_shipdate
order by count(a.l_shipdate) desc
limit 3;

 l_shipdate |   count
------------+------------
 1997-07-03 | 5756528960
 1996-01-29 |    7741401
 1992-07-14 |    7740931
(3 rows)

Time: 33801.996 ms (00:33.802)
```

1/3に短縮（1分31秒→33秒）

①スライス間で処理量に偏り

チューニング結果

SVL_QUERY_METRICS

```
select dimension, segment, segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;

 dimension | segment | segment_execution_time | cpu_skew | io_skew
-----------+---------+------------------------+----------+---------
 segment   |       0 |                     34 |     1.02 |    1.02
 segment   |       1 |                     34 |     2.91 |
```

スライス間の偏りが改善

①スライス間で処理量に偏り

チューニング結果

SVL_QUERY_SUMMARY

```
select stm, seg, maxtime, avgtime, label from svl_query_summary where
query = :pg_last_query_id order by stm, seg;

 stm | seg | maxtime   | avgtime   |                          label
-----+-----+-----------+-----------+-------------------------------------
   0 |   0 | 33504370 | 32864749 | scan   tbl=120466 name=lineitem
   0 |   0 | 33504370 | 32864749 | project
...
   0 |   0 | 33504370 | 32864749 | project
   0 |   1 | 33506991 | 33506866 | save   tbl=1071
   0 |   1 | 33506991 | 33506866 | project
   0 |   1 | 33506991 | 33506866 | scan   tbl=40408 name=Internal Worktabl
...
   0 |   1 | 33506991 | 33506866 | merge
```

maxtime と avgtime が同程度に

①スライス間で処理量に偏り

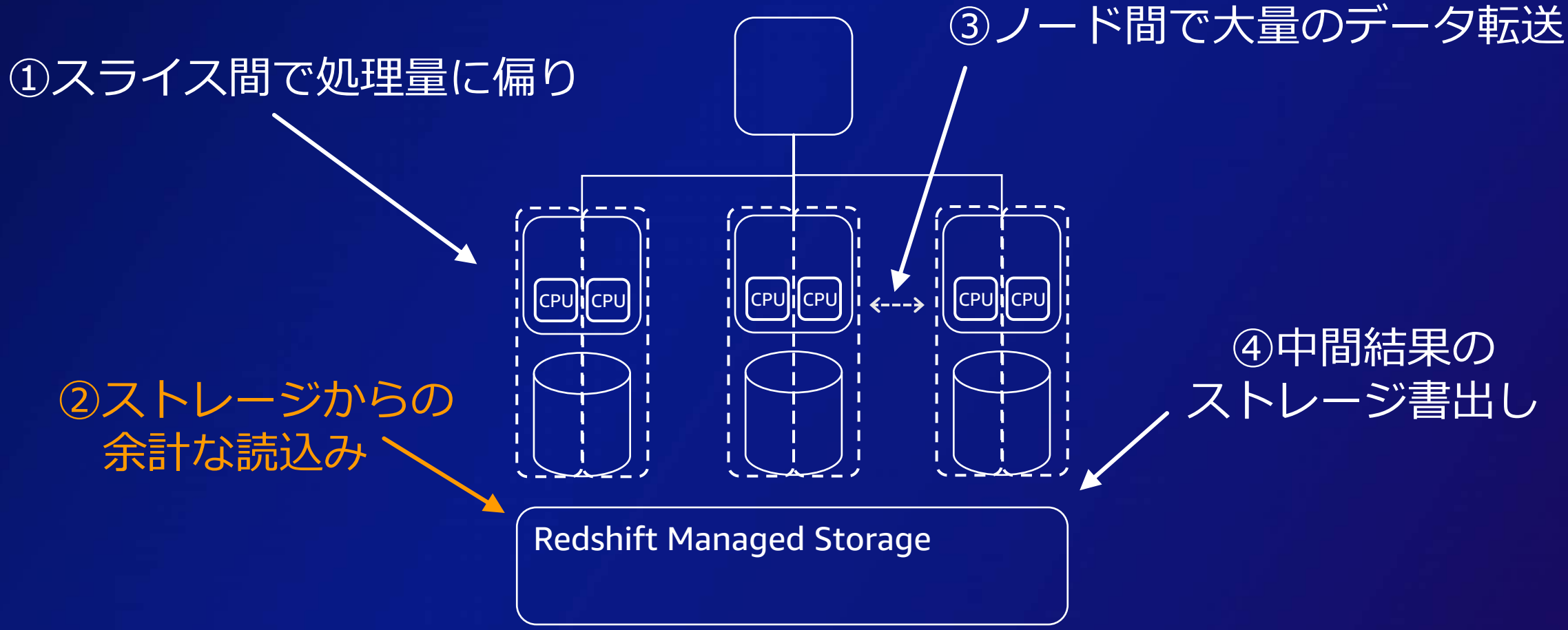
チューニング結果サマリー

- 遅いセグメントを特定、スライス間で偏りがあることを確認
- 2つの結合列のうちカーディナリティが高く・偏りのない列を分散キーに
- スライス別実行時間が均等、2.3倍高速

```
select a.l_shipdate, count(a.l_shipdate)
from lineitem a, lineitem b
where a.l_orderkey = b.l_orderkey
    and a.l_shipdate = b.l_shipdate
group by a.l_shipdate
order by count(a.l_shipdate) desc
limit 3;
```

分散キー	SVV_TABLE_INFO skew_rows	SVL_QUERY_SUMMARY maxtime : avgtime	実行時間
l_shipdate	3.06	91秒：28秒	91秒
l_orderkey	1	33秒：32秒	33秒
削減／高速化効果	偏り解消	均等分散	2.75倍高速

②ストレージからの余計な読込み



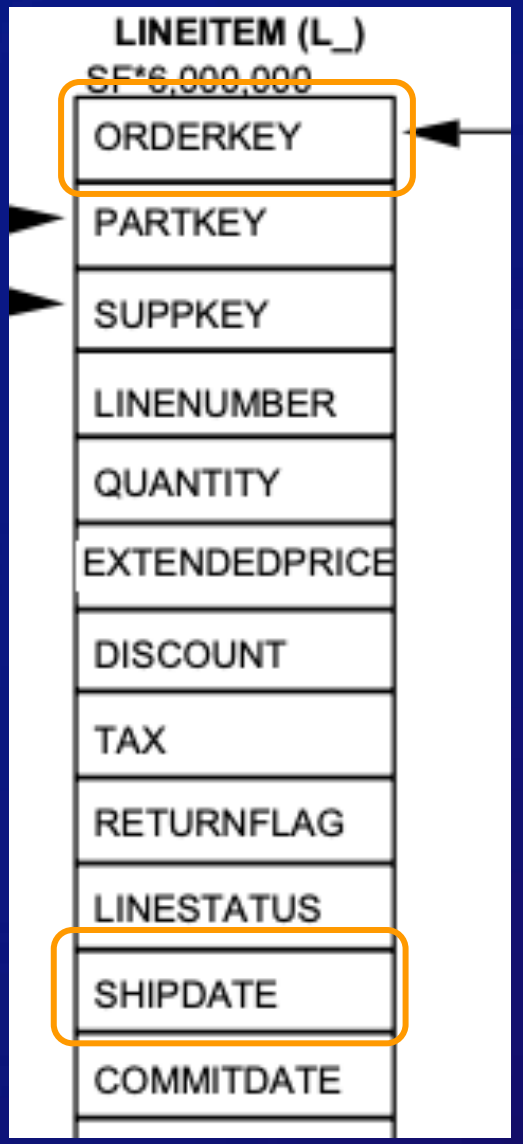
②ストレージからの余計な読込み

検証に使用したテーブル

ソートキーの順番変更

1. shipdate
2. orderkey

1. orderkey
2. shipdate



②ストレージからの余計な読込み

チューニング前

```
select min(l_shipdate), max(l_shipdate)
from lineitem
where l_shipdate between '1996-01-01' and '1997-12-31';
    min      |    max
-------------+-------------
 1996-01-01 | 1997-12-31
(1 row)

Time: 2277.182 ms (00:02.277)
```

フィルタ条件

2.2秒

②ストレージからの余計な読込み

ボトルネック分析

SVL_QUERY_METRICS

```
elect dimension, segment, segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;

 dimension | segment | segment_execution_time | cpu_skew | io_skew
-----------+---------+------------------------+----------+---------
 segment   |       0 |                      2 |     1.22 |    1.22
```

Segment 0 がネック

②ストレージからの余計な読込み

ボトルネック分析

SVL_QUERY_SUMMARY

```
select seg, substring(label,1,30), rows, rows_pre_filter, is_rrscan from
svl_query_summary where query = :pg_last_query_id order by stm, seg, step;
 seg |           substring            |    rows    | rows_pre_filter | is_rrscan
-----+--------------------------------+------------+-----------------+---
   0 | scan   tbl=108285 name=lineite | 5468882659 |     18000048306 | t
   0 | project                        | 5468882659 |               0 | f
   0 | project                        | 5468882659 |               0 | f
   0 | aggr   tbl=1057                |         16 |               0 | f
   1 | scan   tbl=1057 name=Internal  |         16 |               0 | f
   1 | return                         |         16 |               0 | f
   2 | scan   tbl=43826 name=Internal |         16 |               0 | f
   2 | aggr   tbl=1062                |          1 |               0 | f
```

Segment 0 がネック

180億件読み込み、54億件にフィルタ
約70%が不要な読取り

ソートキーの効果
は判断できない

②ストレージからの余計な読込み

チューニング案検討

- クエリのフィルタ条件は l_shipdate
- 複数列指定する場合、条件に指定する頻度の高い順、カーディナリティーの低い（値の種類が少ない）順

	l_shipdate	l_orderkey
値の種類： select count(distinct 列)	2526	4500000000

②ストレージからの余計な読込み

チューニング実施

ソートキーの順番を l_shipdate、l_orderkey に変更

```
alter table lineitem alter compound
sortkey(l_shipdate, l_orderkey);

vacuum full lineitem boost;
```

②ストレージからの余計な読込み

チューニング結果

```
select min(l_shipdate), max(l_shipdate)
from lineitem
where l_shipdate between '1996-01-01' and '1997-12-31';

    min       |     max
--------------+-------------
 1996-01-01 | 1997-12-31
(1 row)

Time: 720.650 ms
```

フィルタ条件

1/3に短縮（2.2秒→720ミリ秒）

②ストレージからの余計な読込み

チューニング結果

SVL_QUERY_SUMMARY

```
select label, rows, rows_pre_filter from svl_query_summary where query
= :pg_last_query_id order by stm, seg, step;
                  label                        |    rows     | rows_pre_filter
-----------------------------------------------+-------------+-----------------
 scan   tbl=107017 name=lineitem               | 5468882659  |      5473383140
 project                                       | 5468882659  |                0
 project                                       | 5468882659  |                0

...
```

54.7億件読み込み、54.6億件にフィルタ

②ストレージからの余計な読込み

チューニング結果サマリー

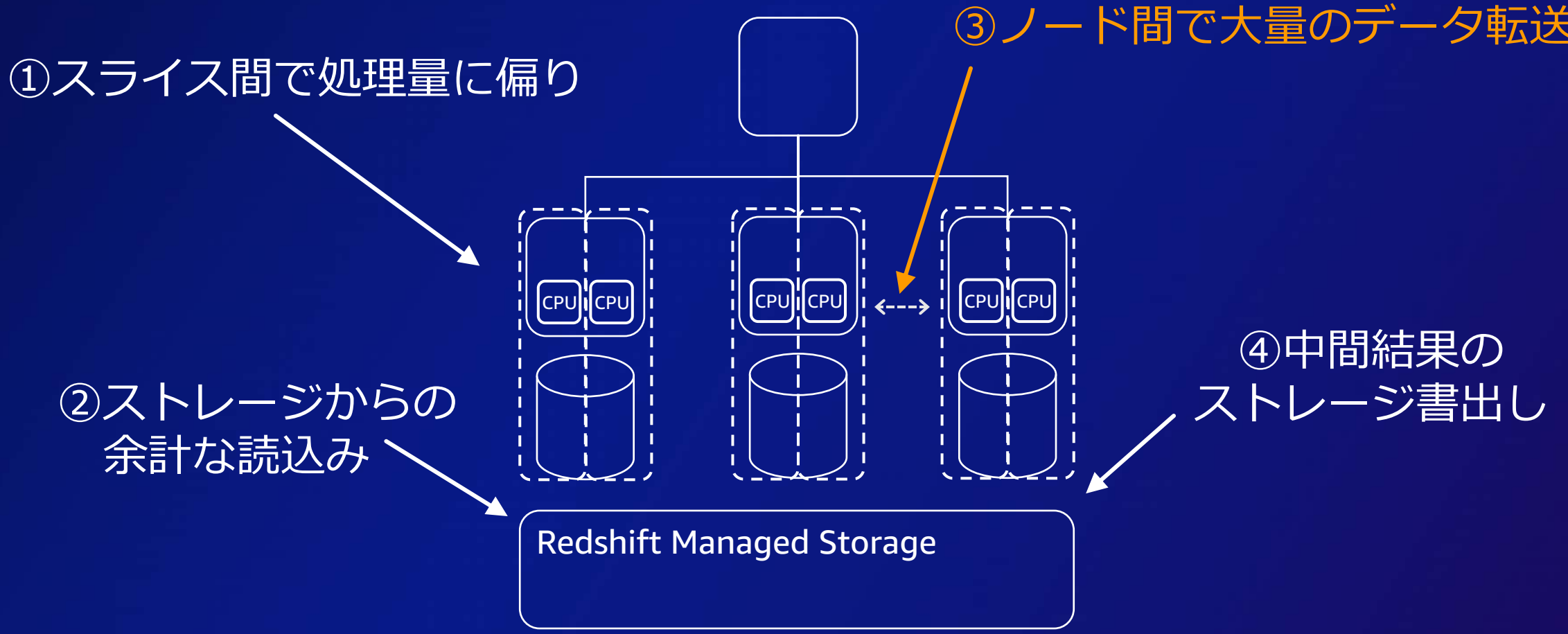
- svl_query_summary.rows_pre_filter/rows から不要な読込みが多いことを確認
- ソートキー順でフィルタ条件に使われ、カーディナリティ低い列を先に

```
select min(l_shipdate),
max(l_shipdate)
from lineitem
where l_shipdate between '1996-01-
01' and '1997-12-31';
```

ソートキー	読込行数	実行時間
l_orderkey, l_shipdate	約180億行	2.2秒
l_shipdate, l_orderkey	約54億行	708ミリ秒
削減／高速化効果	1/3 に削減	3倍高速

*ソートキーの指定順を変更する場合は同じテーブルにアクセスしている他のクエリの性能にも影響があるため、他のクエリを考慮し、最大公約数になるソートキー順を選定する。

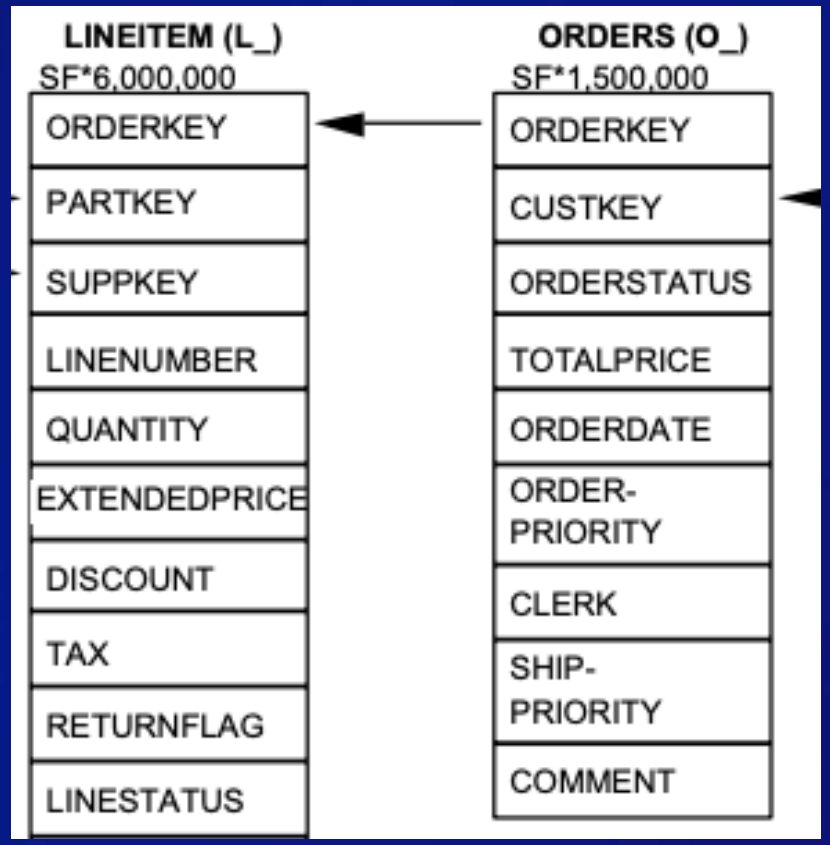
③ノード間で大量のデータ転送



③ノード間で大量のデータ転送

検証に使用したテーブル

lineitem、orders の分散スタイルを EVEN に変更



③ノード間で大量のデータ転送

チューニング前

```
select /* TPC-H Q4 */ o_orderpriority, count(*) as order_count
from  orders
where o_orderdate >= date '1994-04-01'
  and o_orderdate < dateadd(month, 3, cast('1994-04-01' as date))
  and exists (
    select * from lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by o_orderpriority
order by o_orderpriority;

Time: 240539.269 ms (04:00.539)
```

③ノード間で大量のデータ転送

ボトルネック分析

SVL_QUERY_METRICS

```
select dimension, segment, segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;
```

dimension	segment	segment_execution_time	cpu_skew	io_skew
segment	0	2	1.61	1.57
segment	1	2	1.36	
segment	2	72	1.12	1.03
segment	3	238	1.06	
segment	4	239	1.05	
segment	5	239	1.23	

セグメント3-5で約4分

③ノード間で大量のデータ転送

ボトルネック分析

SVL_QUERY_SUMMARY

```
select stm, seg, step, maxtime, avgtime, label from svl_query_summary where query
= :pg_last_query_id and seg between 3 and 4 order by stm, seg, step;
```

stm	seg	step	maxtime	avgtime	label
1	3	0	238663629	203734090	scan tbl=54480 name=Internal Worktable
1	3	1	238663629	203734090	aggr tbl=1065
1	3	2	238663629	203734090	project
1	3	3	238663629	203734090	dist
1	3	4	238663629	203734090	merge
1	3	5	238663629	203734090	aggr tbl=1085
...					
1	4	2	238712466	238712156	hjoin tbl=1071
1	4	3	238712466	238712156	project
1	4	4	238712466	238712156	project
1	4	5	238712466	238712156	aggr tbl=1076
1	4	6	238712466	238712156	dist

DIST（再分散）によるノード間通信

③ノード間で大量のデータ転送

ボトルネック分析

STL_EXPLAIN

```
select substring(plannode, 1, 50) plannode, substring(info,1,60) info from stl_explain where query
= :pg_last_query_id order by nodeid, parentid;

                    plannode                                    |                              info
----------------------------------------------------------------+-----------------------------------------------------
 XN Merge  (cost=118869515118573.33..11886951511857 | Merge Key: orders.o_orderpriority
   ->  XN Network  (cost=118869515118573.33..118869 |   Send to leader
         ->  XN Sort  (cost=118869515118573.33..118 | Sort Key: orders.o_orderpriority
               ->  XN HashAggregate  (cost=11786951 |
                     ->  XN Hash Join DS_DIST_BOTH  | Outer Dist Key: l_orderkey Inner Dist Key:
orders.o_orderkey
                           ->  XN HashAggregate  (c |
                                 ->  XN Seq Scan on | Filter: (l_commitdate < l_receiptdate)
                           ->  XN Hash  (cost=25387 |
                                 ->  XN Seq Scan on | Filter: ((o_orderdate < '1994-07-01'::date) AND
(o_orderdate
```

orderkey列での結合でDS_DIST_BOTH（再分散）でノード間通信

③ノード間で大量のデータ転送

ボトルネック分析

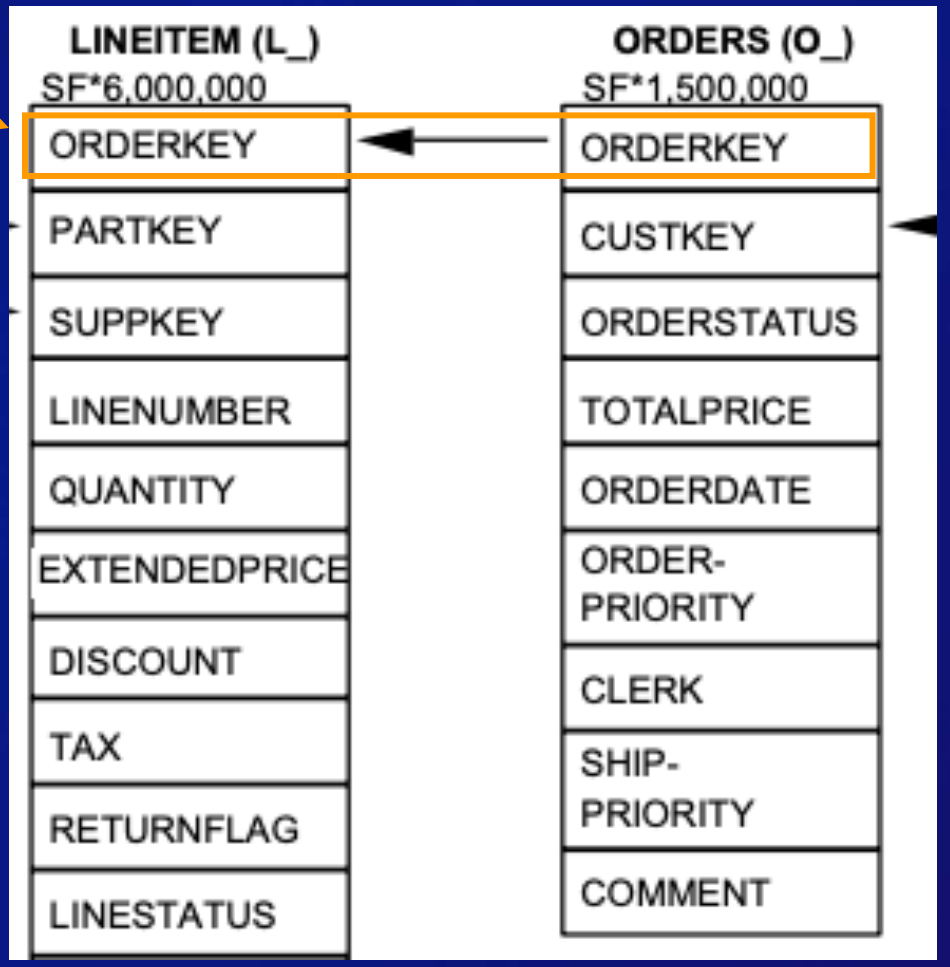
```
select /* TPC-H Q4 */ o_orderpriority, count(*) as order_count
from  orders
where o_orderdate >= date '1994-04-01'
  and o_orderdate < dateadd(month, 3, cast('1994-04-01' as date))
  and exists (
    select * from lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by o_orderpriority
order by o_orderpriority;
```

lineitem と orders を orderkey で結合

③ノード間で大量のデータ転送

ボトルネック分析

lineitem と orders の結合キー orderkey でKEY分散すると、再分散を抑制できるのでは



参考: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf

③ノード間で大量のデータ転送

チューニング実施

分散スタイルを KEY(orderkey) に変更

```
alter table lineitem alter diststyle key distkey l_orderkey;
alter table orders alter diststyle key distkey o_orderkey;
```

* 分散スタイルや分散キーの変更時は VACUUM は不要。

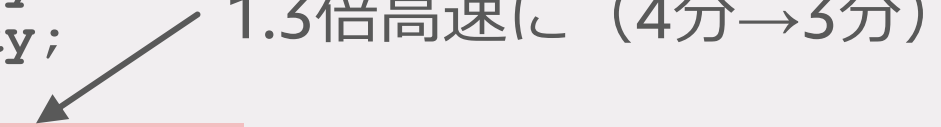
SVV_TABLE_INFO で確認

```
select "table", diststyle, sortkey1  tbl_rows from svv_table_info where
"table" in ('lineitem', 'orders');

  table    |     diststyle     |  tbl_rows
-----------+-------------------+-------------
 lineitem | KEY(l_orderkey) | l_shipdate
 orders   | KEY(o_orderkey) | o_orderdate
```

③ノード間で大量のデータ転送

チューニング結果

```
select /* TPC-H Q4 */ o_orderpriority, count(*) as order_count
from  orders
where o_orderdate >= date '1994-04-01'
  and o_orderdate < dateadd(month, 3, cast('1994-04-01' as date))
  and exists (
    select * from lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by o_orderpriority
order by o_orderpriority;

Time: 180851.958 ms (03:00.852)
```



③ノード間で大量のデータ転送

チューニング結果

SVL_QUERY_METRICS

```
select dimension, segment, segment_execution_time, cpu_skew, io_skew from
svl_query_metrics where query = :pg_last_query_id and dimension =
'segment' order by dimension, segment;

 dimension | segment | segment_execution_time | cpu_skew | io_skew
-----------+---------+------------------------+----------+---------
 segment   |       0 |                      2 |     1.90 |    1.85
 segment   |       1 |                    179 |     1.11 |    1.00
 segment   |       2 |                    179 |     1.25 |
```

最遅のセグメントで約3分

③ノード間で大量のデータ転送

チューニング結果

SVL_QUERY_METRICS

```
select stm, seg, step, maxtime, avgtime, label from svl_query_summary
where query = :pg_last_query_id and seg between 1 and 2 order by stm, seg,
step;

 stm | seg | step |  maxtime  |  avgtime  |                    label
-----+-----+------+-----------+-----------+-----------------------------
   1 |   1 |    0 | 179109498 | 161686456 | scan   tbl=107017 name=lineit
   1 |   1 |    1 | 179109498 | 161686456 | project
...
   1 |   1 |    6 | 179109498 | 161686456 | hjoin  tbl=1065
   1 |   1 |    9 | 179109498 | 161686456 | aggr   tbl=1059
   1 |   1 |   10 | 179109498 | 161686456 | dist
...
   1 |   2 |    0 | 179149108 | 179149023 | scan   tbl=54279 name=Intern
   1 |   2 |    1 | 179149108 | 179149023 | aggr   tbl=1070
```

DISTがまだある？

③ノード間で大量のデータ転送

チューニング結果

STL_EXPLAIN

```
select substring(plannode, 1, 50) plannode, substring(info,1,60) info from stl_explain where query
= :pg_last_query_id order by nodeid, parentid;

 plannode                          |                                  info
-----------------------------------------------------+-------------------------------------------------
---------
 XN Merge  (cost=118633461508816.00..11863346150881 | Merge Key: orders.o_orderpriority
   ->  XN Network  (cost=118633461508816.00..118633 |   Send to leader
         ->  XN Sort  (cost=118633461508816.00..118 | Sort Key: orders.o_orderpriority
               ->  XN HashAggregate  (cost=11763346 |
                     ->  XN Hash Join DS_DIST_NONE  | Hash Cond: ("outer".l_orderkey = "inner".o_orderkey)
                           ->  XN HashAggregate (c |
                                 ->  XN Seq Scan on | Filter: (l_commitdate < l_receiptdate)
                           ->  XN Hash  (cost=25343 |
                                 ->  XN Seq Scan on | Filter: ((o_orderdate < '1994-07-01'::date) AND
(o_orderdate
```

DS_DIST_NONE のため再分散は発生していない

③ノード間で大量のデータ転送

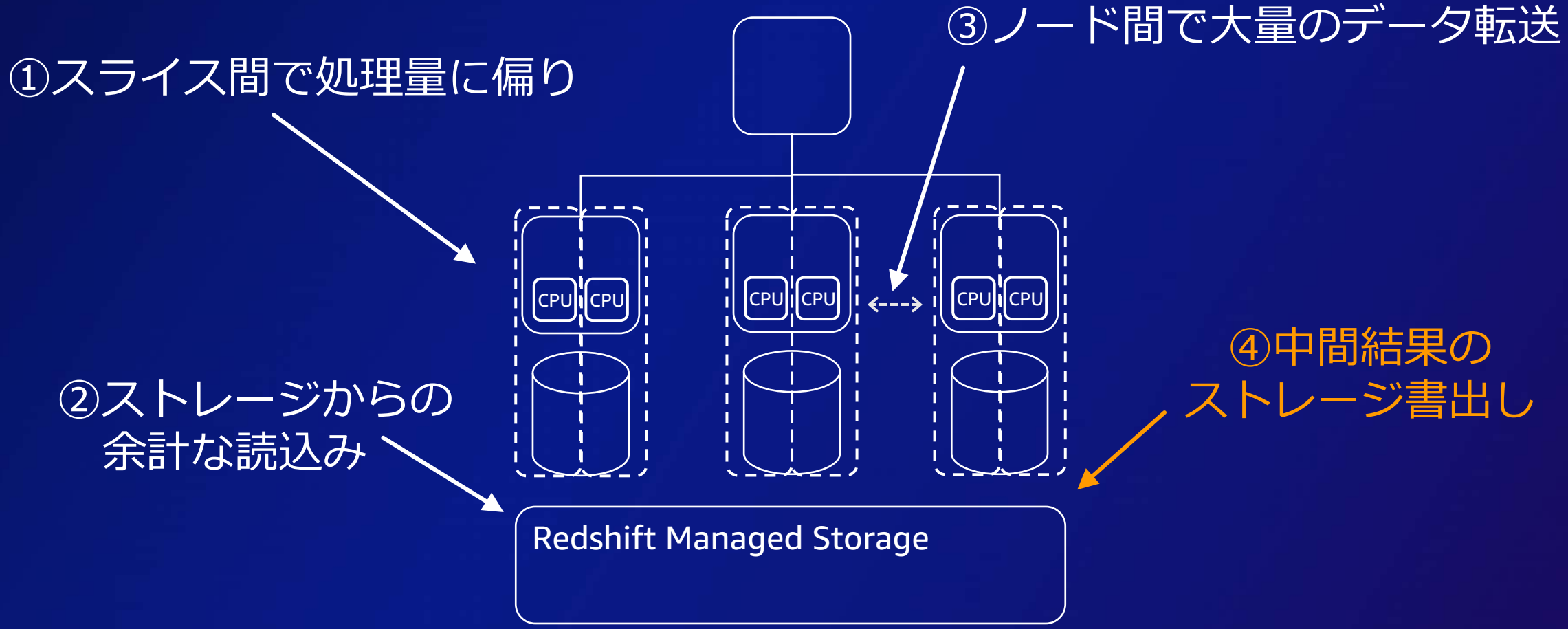
チューニング結果サマリー

- 遅いセグメントを特定、実行計画から再分散（ノード間通信）の発生を確認
- 結合キーを分散キーに指定
  - EVEN分散→KEY分散（orderkey）

```
select /* TPC-H Q4 */ o_orderpriority, count(*) as
order_count
from  orders where o_orderdate >= date '1994-04-01'
  and o_orderdate < dateadd(month, 3, cast('1994-04-
01' as date))
  and exists (
    select * from lineitem where
      l_orderkey = o_orderkey and l_commitdate <
l_receiptdate)
group by o_orderpriority
order by o_orderpriority;
```

分散スタイル	再分散	実行時間
EVEN	DS_DIST_BOTH	4分
KEY(orderkey)	DS_DIST_NONE	3分
削減／高速化効果	再分散なしに	1.3倍高速

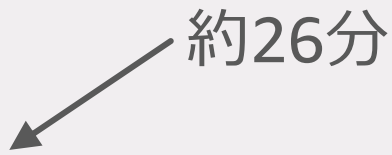
④中間結果のストレージ書き出し



④中間結果のストレージ書き出し

チューニング前

```
select a.l_linenumber, avg(a.l_quantity)
from lineitem a, lineitem b
where
    collate(a.l_orderkey, 'case_insensitive') =
collate(b.l_orderkey, 'case_insensitive')
    and a.l_shipdate = b.l_shipdate
group by a.l_linenumber
order by a.l_linenumber;

...

Time: 1547123.222 ms (25:47.123)
```



④中間結果のストレージ書き出し

ボトルネック分析

SVL_QUERY_METRICS

```
select dimension, query_temp_blocks_to_disk, segment,
segment_execution_time from svl_query_metrics where query
= :pg_last_query_id and dimension in ('query', 'segment') order by
dimension, segment;
```

dimension	query_temp_blocks_to_disk	segment	segment_execution_time
query	2028253		
segment		0	538
segment		1	539
segment		2	552
segment		3	1008
segment		4	1008

2TBの中間結果を
ストレージに書き出し

Segment 3、4で17分

④中間結果のストレージ書き出し

ボトルネック分析

SVL_QUERY_SUMMARY

```
select stm, seg, step, maxtime, label, is_diskbased, workmem from svl_query_summary where query
= :pg_last_query_id order by stm, seg, step;
```

26GBをメモリでソート、ストレージに書出し

stm	seg	step	maxtime	label	is_diskbased	workmem
0	0	0	538841109	scan tbl=110211 name=lineitem	f	0
0	0	1	538841109	project	f	0
0	0	2	538841109	dist	f	0
0	1	0	538856386	scan tbl=3399 name=Internal Worktable	f	0
0	1	1	538856386	project	f	0
0	1	2	538856386	hash tbl=1045	t	26732396544
1	2	0	552474705	scan tbl=110211 name=lineitem	f	0
1	2	1	552474705	project	f	0
1	2	2	552474705	dist	f	0
1	3	0	1007056682	scan tbl=3727 name=Internal Worktable	f	0
1	3	1	1007056682	project	f	0
1	3	2	1007056682	hjoin tbl=1045	f	0
...						
1	4	0	1007575421	scan tbl=3728 name=Internal Worktable	f	0
1	4	1	1007575421	aggr tbl=1059		

ハッシュ結合

④中間結果のストレージ書き出し

ボトルネック分析

SVL_QUERY_REPORT

```
select segment, slice, min(start_time) start_time_min, max(end_time) end_time_max
from svl_query_report
where query = :pg_last_query_id and segment in (3, 4) and slice = 0
group by segment, slice
order by segment, slice;
```

Segment 3、4は並行実行

segment	slice	start_time_min	end_time_max
3	0	2023-04-14 05:59:12.945673	2023-04-14 06:15:26.334688
4	0	2023-04-14 05:59:12.941083	2023-04-14 06:16:00.515626

* Segment 3 と 4 の開始・終了時間が同時刻帯のため、並行実行されている。Segment 3 と 4 と合わせて17分（1008秒）要している。

④中間結果のストレージ書き出し

ボトルネック分析

```
select a.l_linenumber, avg(a.l_quantity)
from lineitem a, lineitem b
where
    collate(a.l_orderkey, 'case_insensitive') =
collate(b.l_orderkey, 'case_insensitive')
    and a.l_shipdate = b.l_shipdate
group by a.l_linenumber
order by a.l_linenumber;
```

結合列に collate 関数を使用

結合キーが分散キーかつソートキーであるが、マージ結合になっていない

④中間結果のストレージ書き出し

チューニング実施

```
select a.l_linenumber, avg(a.l_quantity)
from lineitem a, lineitem b
where
    a.l_orderkey = b.l_orderkey
    and a.l_shipdate = b.l_shipdate
group by a.l_linenumber
order by a.l_linenumber;
```

collate 関数を削除

④中間結果のストレージ書き出し

チューニング実施

collate 関数なしで大文字・小文字を区別しないよう、データベースか、テーブルで case_insensitive を指定。

- データベースレベル
  - データが格納されると ALTER できない。別DBを作成してデータコピー

```
create database tpch_3tb collate case_insensitive;
```

- テーブルレベル
  - ALTER できない、別テーブルを CREATE してデータコピー

```
create table lineitem (
 l_orderkey varchar(11) collate case_insensitive encode raw
);
```

* 移行や PoC ではデータ投入前に設定するよう注意

④中間結果のストレージ書き出し

チューニング結果

```
select a.l_linenumber, avg(a.l_quantity)
from lineitem a, lineitem b
where
    a.l_orderkey = b.l_orderkey
    and a.l_shipdate = b.l_shipdate
group by a.l_linenumber
order by a.l_linenumber;

...

Time: 109130.947 ms (01:49.131)
```

1/13に短縮（26分→2分）

④中間結果のストレージ書き出し

チューニング結果

SVL_QUERY_METRICS

```
select dimension, query_temp_blocks_to_disk, segment,
segment_execution_time from svl_query_metrics where query
= :pg_last_query_id and dimension in ('query', 'segment') order by
dimension, segment;
;
 dimension | query_temp_blocks_to_disk | segment | segment_execution_time
-----------+---------------------------+---------+------------------------
 query     |                           |         |
 segment   |                           |       0 |                    109
 segment   |                           |       1 |                    109
```

中間結果の書出しはゼロ

④中間結果のストレージ書き出し

チューニング結果

SVL_QUERY_SUMMARY

```
select stm, seg, step, maxtime, label, is_diskbased, workmem from svl_query_summary where query
= :pg_last_query_id order by stm, seg, step;
```

stm	seg	step	maxtime	label	is_diskbased	workmem
0	0	0	108904062	scan tbl=110211 name=lineitem	f	0
0	0	1	108904062	project	f	0
0	0	2	108904062	project	f	0
0	0	3	108904062	project	f	0
0	0	4	108904062	mjoin tbl=1043	f	0
0	0	5	108904062	project	f	0
0	0	6	108904062	project	f	0
0	0	7	108904062	aggr tbl=1051	f	1670774784
0	0	8	108904062	dist	f	0
0	0	9	108904062	scan tbl=110211 name=lineitem	f	0
0	0	10	108904062	project	f	0
0	0	11	108904062	project	f	0
0	1	0	108911239	scan tbl=3098 name=Internal Worktable	f	0

マージ結合

④中間結果のストレージ書き出し

チューニング結果サマリー

- 中間結果のストレージ書出しが多く、マージ結合の要件を満たしているのにハッシュ結合に
- クエリの結合キーの collate 関数削除
- マージ結合になり、13倍高速

```
select a.l_linenumber, avg(a.l_quantity)
from lineitem a, lineitem b
where
    collate(a.l_orderkey, 'case_insensitive') =
collate(b.l_orderkey, 'case_insensitive')
    and a.l_shipdate = b.l_shipdate
group by a.l_linenumber
order by a.l_linenumber;
```

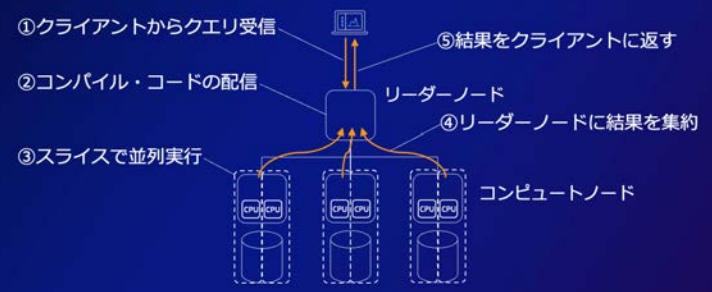
クエリ書換え	結合方式	中間結果書出しサイズ	実行時間
collate関数あり	ハッシュ結合	1.9TB	26分
collate関数なし	マージ結合	0	2分
削減／高速化効果		100%削減	13倍高速

まとめ

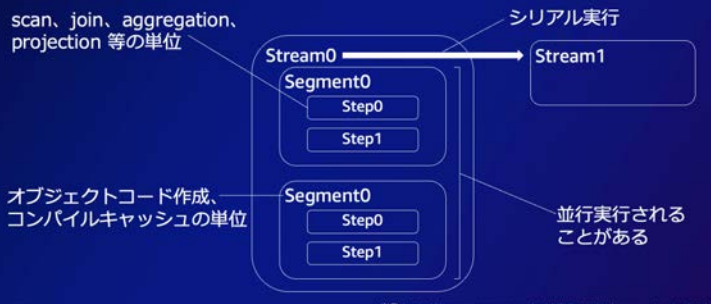
まとめ

アーキテクチャなど前提知識

クエリ実行の流れ

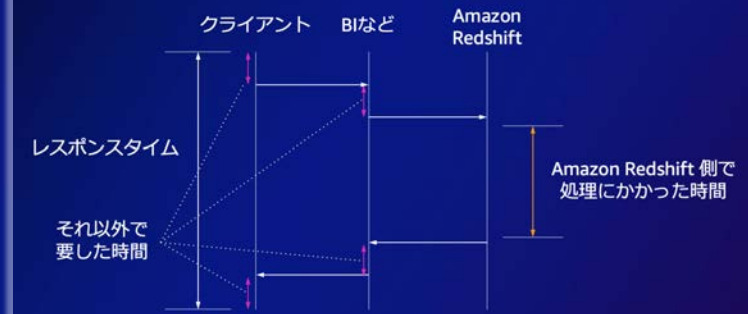


ステップ → セグメント → ストリーム



ボトルネック分析方法

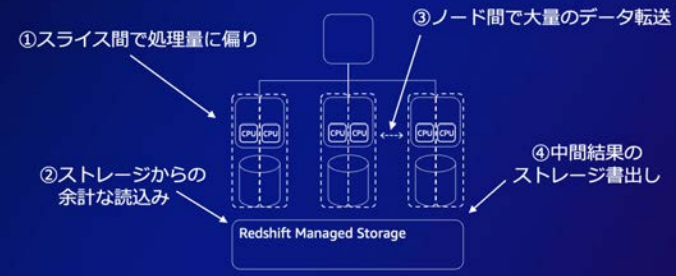
Amazon Redshift 側で時間がかかっているか



どのセグメントが遅いか、スライスでの偏りは

チューニング例

Amazon Redshift で発生するボトルネック例



④中間結果のストレージ書き出し

ボトルネック分析

SVL_QUERY_SUMMARY

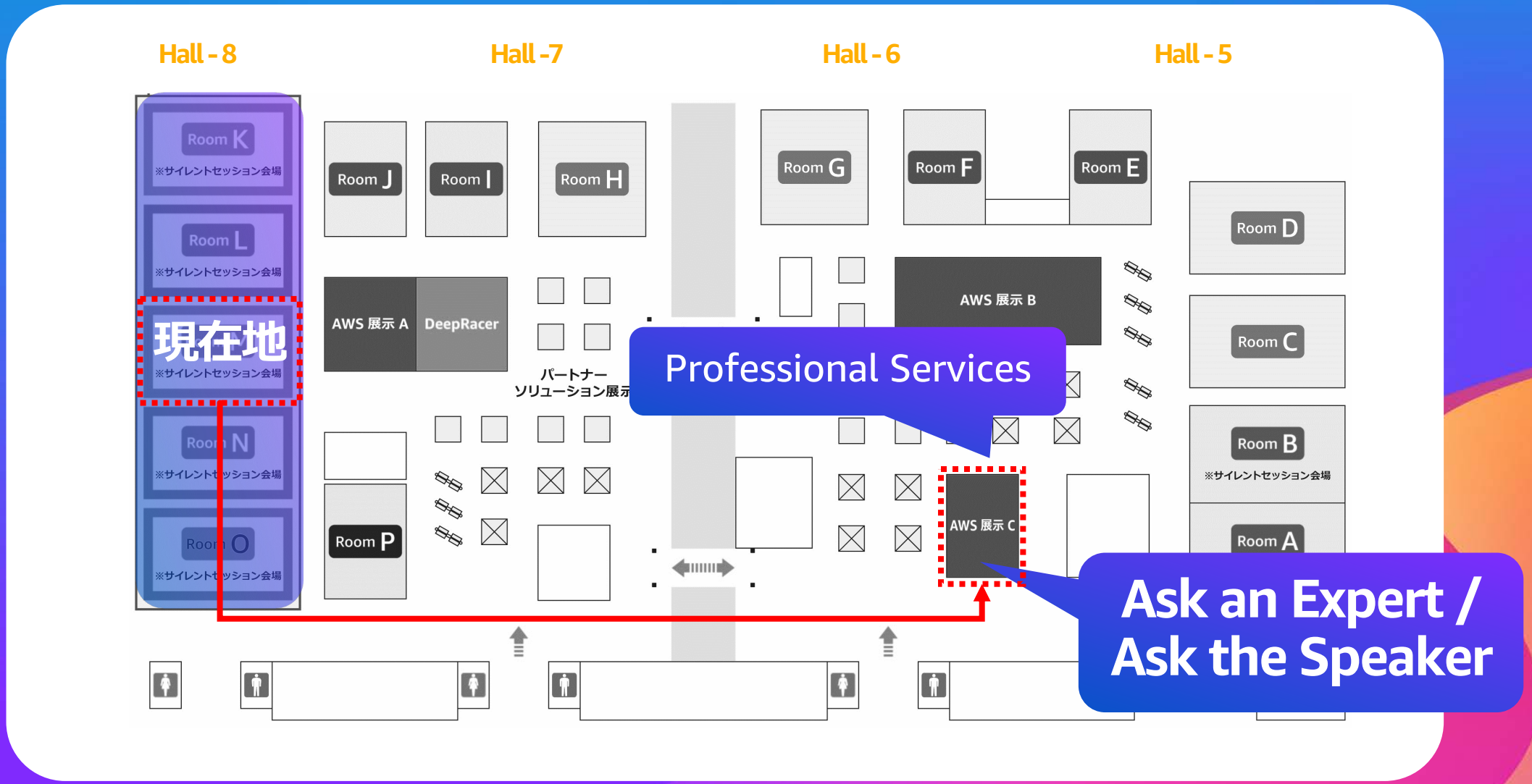
```
select stm, seg, step, maxtime, label, is_diskbased, workmem from svl_query_summary where query
= :pg_last_query_id order by stm, seg, step;
```

26GBをメモリでソート、ストレージに書出し

stm	seg	step	maxtime	label	is_diskbased	workmem
0	0	0	538841109	scan tbl=110211 name=lineitem	f	0
0	0	1	538841109	project	f	0
0	0	2	538841109	dist	f	0
0	1	0	538856386	scan tbl=3399 name=Internal Worktable	f	0
0	1	1	538856386	project	f	0
0	1	2	538856386	hash tbl=1045	t	26732396544
1	2	0	552474705	scan tbl=110211 name=lineitem	f	0
1	2	1	552474705	project	f	0
1	2	2	552474705	dist	f	0
1	3	0	1007056682	scan tbl=3727 name=Internal Worktable	f	0
1	3	1	1007056682	project	f	0
1	3	2	1007056682	hjoin tbl=1045	f	0
...						
1	4	0	1007575421	scan tbl=3728 name=Internal Worktable	f	0
1	4	1	1007575421	aggr tbl=1059		

ハッシュ結合

この後 30 分程度、ご質問にお答えします
Ask an Expert / Ask the speaker ブースまでお越しください



Appendix

Appendix

- システムテーブル・ビュー一覧
- クエリ単体性能計測時のポイント
- 参考情報

システムテーブル・ビュー一覧（1/2）

クエリの性能分析に使う主なシステムテーブル・ビュー

システムテーブル・ビュー名	説明
STL_QUERY	クエリの開始・終了時刻やクエリ文字列、Concurrency Scaling実行状況など
STL_WLM_QUERY	WLMキュー待ちがなかったか、どのキューで実行されたか、クエリスロット数はいくつだったか
STL_QUERY_METRICS_SUMMARY	STL_QUERY_METRICS のサマリー
STL_QUERY_METRICS	中間結果書出しサイズ、スライス間の処理の偏り、セグメント毎の実行時間
SVL_QUERY_SUMMARY	step単位の情報・遅い segment の特定、スライスで処理の偏りがないか、中間結果の書出しがないか、不要なデータを読んでないか
SVL_QUERY_REPORT	SVL_QUERY_SUMMARYよりさらに詳細な slice単位の情報
STL_EXPLAIN	クエリの実行計画、アラートがないか、非効率な操作がないか

出典: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/c_types-of-system-tables-and-views.html

システムテーブル・ビュー一覧（2/2）

クエリの性能分析に使う主なシステムテーブル・ビュー

システムテーブル・ビュー名	説明
SVL_QLOG	結果キャッシュが使われたかどうか
SVL_STATEMENTTEXT	TRUNCATE や DDL 処理を含むすべてのコマンドの結果
STL_ALERT_EVENT_LOG	クエリ性能改善のための解決策のアドバイス
SVL_QUERY_HISTORY	クエリの性能情報のサマリー

出典: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/c_types-of-system-tables-and-views.html

SYS_QUERY_HISTORY

```
select * from sys_query_history;

user_id           | 100
query_id          | 2403286
query_label       | default
transaction_id    | 1546116
session_id        | 1073938965
database_name     | tpch_3tb
query_type        | SELECT
status            | failed
result_cache_hit  | f
start_time        | 2023-04-18 13:12:32.992071
end_time          | 2023-04-18 13:12:45.489761
elapsed_time      | 12497690
queue_time        | 0
execution_time    | 12469230
error_message     | Query (2403288) cancelled on user's request
returned_rows     | 0
returned_bytes    | 0
query_text        | select /* TPC-H Q14 */ 100.00 * sum(case when p_type like 'PROMO%' then l_extendedprice * (1 -
l_discount) else 0 end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue from lineitem, part where l_partkey =
p_partkey and l_shipdate >= date '1995-09-01' and l_shipdate < dateadd(month, 1, cast('1995-09-01' as date)) ;
redshift_version | 1.0.49087
usage_limit       |
compute_type      | primary
compile_time      | 8192146
planning_time     | 16556
lock_wait_time    | 24
```


参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/SYS_QUERY_HISTORY.html

クエリ単体性能計測時のポイント

- 計測前
  - アクセステーブルの VACUUM、ANALYZE を行う
- 計測時
  - 無風状態で計測する
  - リザルトキャッシュの無効化
  - AutoMV の無効化
  - コンパイル時間を考慮し、複数回実行する
  - クエリの性能情報の取得
  - バージョン情報構成情報を取得する

性能計測前に VACUUM、ANALYZE を行う

- 全テーブルの VACUUM、ANALYZE を行う

```
vacuum;
analyze;
```

- テーブル単位で VACUUM、ANALYZE を行う

```
vacuum full schema1.table1 boost;
analyze schema1.table1;
```

← 他のクエリの性能に影響がでる可能性
追加リソースを使っても速く終わらせたい場合

＊ VACUUM はデフォルトで95% がソート済みのテーブルのソートフェーズはスキップされる
＊ ANALYZE はデフォルトで前回実行後データ変更が10%未満の場合はスキップされる

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_VACUUM_command.html
https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_ANALYZE.html

無風状態で計測する

- PoC 環境などでクエリの単体性能を計測する場合は他のワークロードが流れていない無風状態で計測。

- 他のワークロードの影響を受けると、同じクエリでも実行タイミングによって実行時間にバラツキが発生。

リザルトキャッシュの無効化

リザルトキャッシュが有効な場合、同じクエリを複数回実行するとキャッシュされた結果セットが返されるため、セッション／ユーザーレベルのいずれかで無効化する。

- セッションレベル

```
set enable_result_cache_for_session = off;
```

- ユーザーレベル

```
alter user <ユーザー名> set enable_result_cache_for_session = off;
```

- 設定確認方法（結果は "on" or "off"）

```
show enable_result_cache_for_session;
```

参考: https://docs.aws.amazon.com/redshift/latest/dg/r_enable_result_cache_for_session.html

AutoMV の無効化

繰り返し実行される結合・集約クエリを学習し、自動的にマテリアライズドビューが作成される。 PoC 実施時のみ無効化し、実運用時は有効化を推奨。

- セッションレベル

```
set mv_enable_aqmv_for_session=off;
```

- ユーザーレベル

```
alter user <ユーザー名> set mv_enable_aqmv_for_session=off;
```

- 設定確認方法（結果は "on" or "off"）

```
show mv_enable_aqmv_for_session;
```

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/materialized-view-auto-mv.html

クエリのコンパイル時間の考慮

1回目はコンパイル時間を含むため、複数回実行、2回目以降を採用。

計測

回目	実行時間
1回目	20秒
2回目	12秒
3回目	10秒

採用する実行時間

コンパイル時間	実行時間
含む	20秒
含まない	平均11秒

← 基本こちらを採用

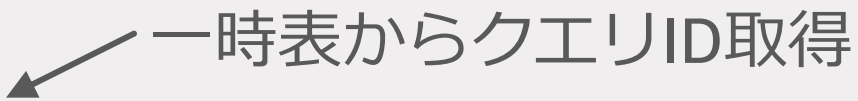
＊ クエリ単位ではなくセグメント単位でキャッシュされ、クラスターの再起動後も保持される。
＊ コンパイルキャッシュは Amazon Redshift のバージョン間で互換性がないため、アップグレード後にクエリが実行されると、再コンパイルされる。

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/c-query-performance.html

クエリの性能情報の取得

クエリIDを一時表に保存、システムテーブル・ビューから情報取得

```
-- 計測対象のクエリ実行
select … from … ;

-- クエリIDを取得、一時表に保存
select pg_last_query_id() as qid into temp table qid_table;

-- クエリIDから性能情報取得
select * from svl_query_summary where query=(select qid from qid_table);
```



* クライアントが psql の場合は “select pg_last_query_id()” → “¥gset” で変数にセット、”: pg_last_query_id” で参照する方法も

クエリの性能情報の取得

計測対象が CTAS の場合

```
-- クエリに識別用コメントを付与
CREATE TABLE temp_table1 /* cats_tmp1 */ …

-- クエリID取得
select query as qid into temp table qid_table from (select query from
STL_QUERY where querytxt like '% cats_tmp1 %' order by starttime desc)
limit 1;

-- クエリIDから性能情報取得
select * from svl_query_summary where query=(select qid from qid_table);
```

* CREATE TABLE 実行直後に自動的に ANALYZE が実行されるため、pg_last_query_id() で クエリIDを取得できない

（参考）単体性能計測SQLスクリプト

```
--current timestamp (utc)
select getdate();

--timing on
¥timing on

--pager off
¥pset pager off

-- result cache off
set enable_result_cache_for_session=off;

-- execute target query
¥i lineorder_count.sql

-- query id
select pg_last_query_id();
¥gset

-- execution time
select userid,
        trim(database) "database",
        trim(label) as label,
        query,
        xid,
        pid,
        datediff(milliseconds, starttime, endtime) as
"exec_time(ms)",
        starttime,
        endtime,
        aborted,
        insert_pristine,
        concurrency_scaling_status,
        trim(querytxt) as query_text
        from stl_query where query = :pg_last_query_id;
```

```
-- show execution plan
select query,
        stm,
        seg,
        step,
        maxtime,
        avgtime,
        rows,
        bytes,
        lpad(' ',stm+seg+step) || label as label
from svl_query_summary
where query = :pg_last_query_id
order by stm, seg, step;

-- svl_query_metrics
select *
from svl_query_metrics
where query = :pg_last_query_id;

-- stl_explain
select query,
        nodeid,
        parentid,
        plannode,
        info
from stl_explain
where query = :pg_last_query_id
order by 1,2;

-- stl_wlm_query
select * from stl_wlm_query
where query = :pg_last_query_id
order by service_class;

-- svl_query_report
select * from svl_query_report
where query = :pg_last_query_id
order by segment, step, slice;
```


＊ psql での実行前提

バージョン情報などの取得

クエリ実行時のバージョン情報などを取得

```
-- Amazon Redshift のバージョン情報
select version();

-- テーブル情報取得（分散キー、ソートキー、未ソート率、統計情報の状態）
select * from svv_table_info;

-- 分散キー・ソートキーの定義の取得
set search_path to '$user', schema1;
select * from pg_table_def where (distkey = true or sortkey <> 0) and
schemaname not in ('pg_catalog') order by schemaname, tablename, sortkey;
```

SVV_TABLE_INFO の見方

```
select "table", diststyle, sortkey1_enc, sortkey_num, size, unsorted, stats_off, tbl_rows, skew_rows, vacuum_sort_benefit from svv_table_info;

  table   |    diststyle     |           sortkey1_enc           | sortkey_num |  size  | unsorted | stats_off |  tbl_rows   | skew_rows | vacuum_sort_benefit
----------+------------------+----------------------------------+-------------+--------+----------+-----------+-------------+-----------+--------------------
 region   | KEY(r_regionkey) | none                             |           1 |     60 |     0.00 |      0.00 |           5 |    100.00 |                0.00
 nation   | KEY(n_nationkey) | none                             |           1 |    196 |     0.00 |      0.00 |          25 |    100.00 |                0.00
 customer | KEY(c_custkey)   | none                             |           1 |  41259 |     0.00 |      0.00 |   450000000 |      1.00 |                0.00
 supplier | KEY(s_suppkey)   | none                             |           1 |   2783 |     0.00 |      0.00 |    30000000 |      1.00 |                0.00
 lineitem | KEY(l_orderkey)  | none                             |           2 | 863171 |     0.00 |      0.00 | 18000048306 |      1.00 |                0.00
 orders   | KEY(o_orderkey)  | none                             |           2 | 225308 |     0.00 |      0.00 |  4500000000 |      1.00 |                0.00
 partsupp | KEY(ps_partkey)  | none                             |           1 | 161129 |     0.00 |      0.00 |  2400000000 |      1.00 |                0.00
 part     | KEY(p_partkey)   | none                             |           1 |  26960 |     0.00 |      0.00 |   600000000 |      1.00 |                0.00
```

0 でない場合は VACUUM　0 でない場合は ANALYZE

* レコードが0件のテーブルは SVV_TABLE_INFO に表示されない
* ソートキーが設定されてないテーブルは unsorted、vacuum_sort_benefit が null になる

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_SVV_TABLE_INFO.html

PG_TABLE_DEF の見方

```
select * from pg_table_def where distkey = true or sortkey <> 0 order by tablename, sortkey;

 schemaname | tablename |   column    |  type   | encoding | distkey | sortkey | notnull
------------+-----------+-------------+---------+----------+---------+---------+---------
 public     | customer  | c_custkey   | bigint  | none     | t       |       1 | t
 public     | lineitem  | l_shipdate  | date    | none     | f       |       1 | t
 public     | lineitem  | l_orderkey  | bigint  | az64     | t       |       2 | t
 public     | nation    | n_nationkey | integer | none     | t       |       1 | t
 public     | orders    | o_orderdate | date    | none     | f       |       1 | t
 public     | orders    | o_orderkey  | bigint  | az64     | t       |       2 | t
 public     | part      | p_partkey   | bigint  | none     | t       |       1 | t
 public     | partsupp  | ps_partkey  | bigint  | none     | t       |       1 | t
 public     | region    | r_regionkey | integer | none     | t       |       1 | t
 public     | supplier  | s_suppkey   | integer | none     | t       |       1 | t
```

ソートキーと指定順
第2ソートキー以降を確認

参考: https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/r_PG_TABLE_DEF.html

参考情報

- Amazon Redshift: Ten years of continuous reinvention
  - https://www.amazon.science/latest-news/amazon-redshift-ten-years-of-continuous-reinvention
- Amazon Redshift re-invented
  - https://www.amazon.science/publications/amazon-redshift-re-invented
- Amazon Redshift re-invented @SIGMOD'22
  - https://www.youtube.com/watch?v=ItFCuy0S7wk
- Reinventing Amazon Redshift (Ippokratis Pandis)
  - https://www.youtube.com/watch?v=ir6V7DkBe-w
- AWS re:Invent 2022 - [NEW] Amazon Redshift: 10 yrs of integration & data sharing innovation (ANT345)
  - https://www.youtube.com/watch?v=eIZcoXg39tg
- Cloud Data Warehouse Benchmark Derived from TPC-H
  - https://github.com/awslabs/amazon-redshift-utils/tree/master/src/CloudDataWarehouseBenchmark/Cloud-DWB-Derived-from-TPCH
- Amazon Redshift Utilities
  - https://github.com/awslabs/amazon-redshift-utils

参考情報

- Amazon Redshift パフォーマンスチューニングテクニックと最新アップデート
  - https://www.slideshare.net/AmazonWebServicesJapan/amazon-redshift-118303349
- Amazon Redshift テーブル設計詳細ガイド 分散スタイルとソートキーの決定方法
  - https://d1.awsstatic.com/events/jp/2017/summit/slide/D3T1-5.pdf
- Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys
  - https://aws.amazon.com/jp/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/
- Top 10 performance tuning techniques for Amazon Redshift
  - https://aws.amazon.com/jp/blogs/big-data/top-10-performance-tuning-techniques-for-amazon-redshift/
- Diagnostic queries for query tuning
  - https://docs.aws.amazon.com/redshift/latest/dg/diagnostic-queries-for-query-tuning.html
- Fast and effective distribution-key recommendation for Amazon Redshift
  - https://www.amazon.science/publications/fast-and-effective-distribution-key-recommendation-for-amazon-redshift
- Monitor and optimize queries on the new Amazon Redshift console
  - https://aws.amazon.com/jp/blogs/big-data/monitor-and-optimize-queries-on-the-new-amazon-redshift-console/

Thank you!

畔勝 洋平
アマゾン ウェブ サービス ジャパン合同会社
プロフェッショナルサービス本部 シニアビッグデータコンサルタント