AWS Summits 2014

Amazon Redshift パフォーマンス・チューニング

アマゾン データ サービス ジャパン株式会社
八木橋 徹平

2014/07/18

Session TA-08

amazon web services

自己紹介

- 名前
  - 八木橋 徹平（やぎはし てっぺい）
- 所属
  - アマゾンデータサービスジャパン株式会社
    技術統括本部エンタープライズ部
    ソリューションアーキテクト
- 好きなAWS サービス
  - Amazon Redshift、Amazon Kinesis
  - AWS SDK（Java、Node.js）

最新アップデート

- 無料利用枠（2か月間）
  - dw2.largeを730時間*2 に相当する利用量（クラスタも可）
- アジアパシフィックで3年リザーブドインスタンスの25%以上値下げ
- スタートアップ企業からエンタープライズまで幅広くご採用
  - クックパッド様、良品計画様、 すかいらーく様、ガリバー様
  - NTT DOCOMO様 基幹業務システムで採用をご検討

セッションの目的

アーキテクチャの理解を深めていただき、ノード数の追加やインスタンスのスケールアップだけに頼らず（＝コスト増なしに）、Redshiftの効果的なチューニング手法を学んでいただく

アジェンダ

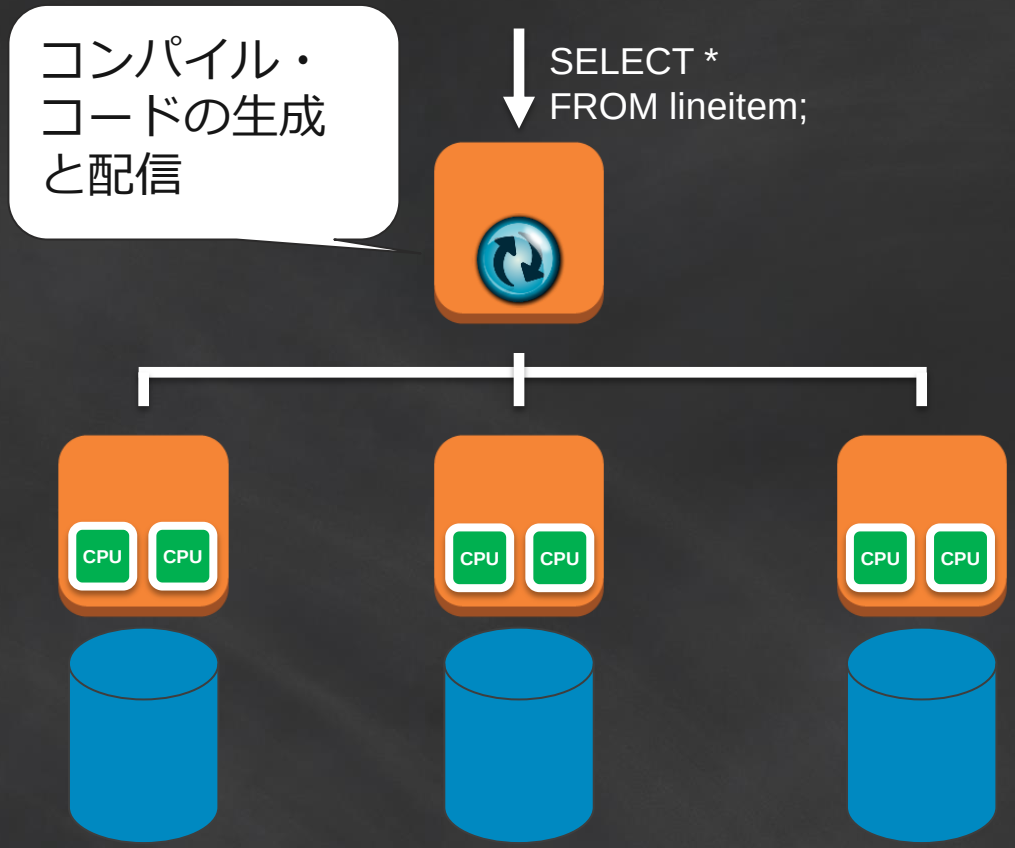
- Redshiftのアーキテクチャ
- テーブル設計
- クエリーの解析
- Workload Management

Redshiftのアーキテクチャ

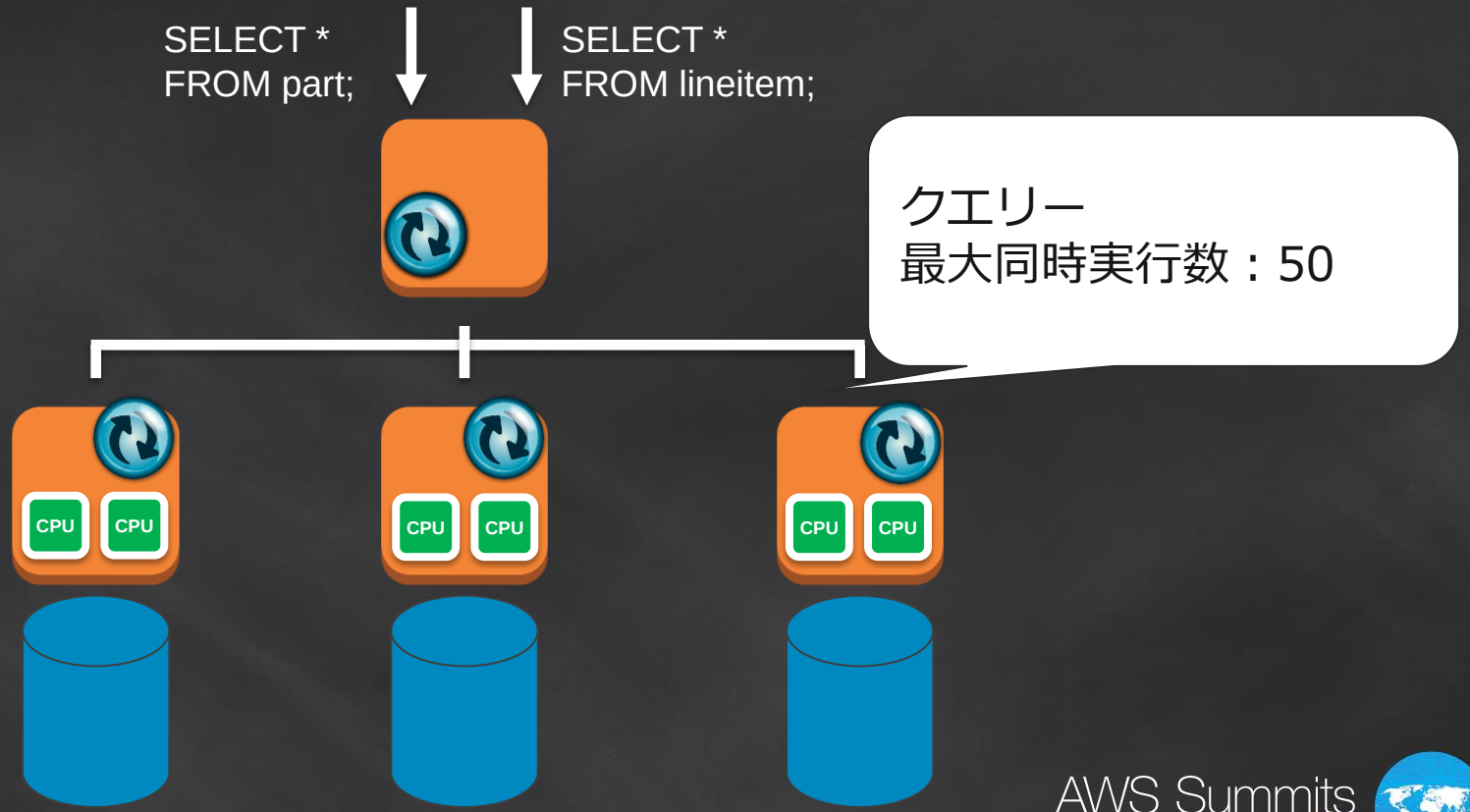
Redshiftのアーキテクチャ

- MPP（超並列演算）
  - CPU、Disk・Network I/Oの並列化
  - 論理的なリソースの括り「ノードスライス」
- データの格納
  - 列指向（カラムナ）
  - 圧縮
- データの通信
  - コンピュート・ノード間の通信
  - 各コンピュート・ノードからリーダー・ノードへの通信
  - 他のAWSサービスとの通信

アーキテクチャ：MPP（超並列演算）

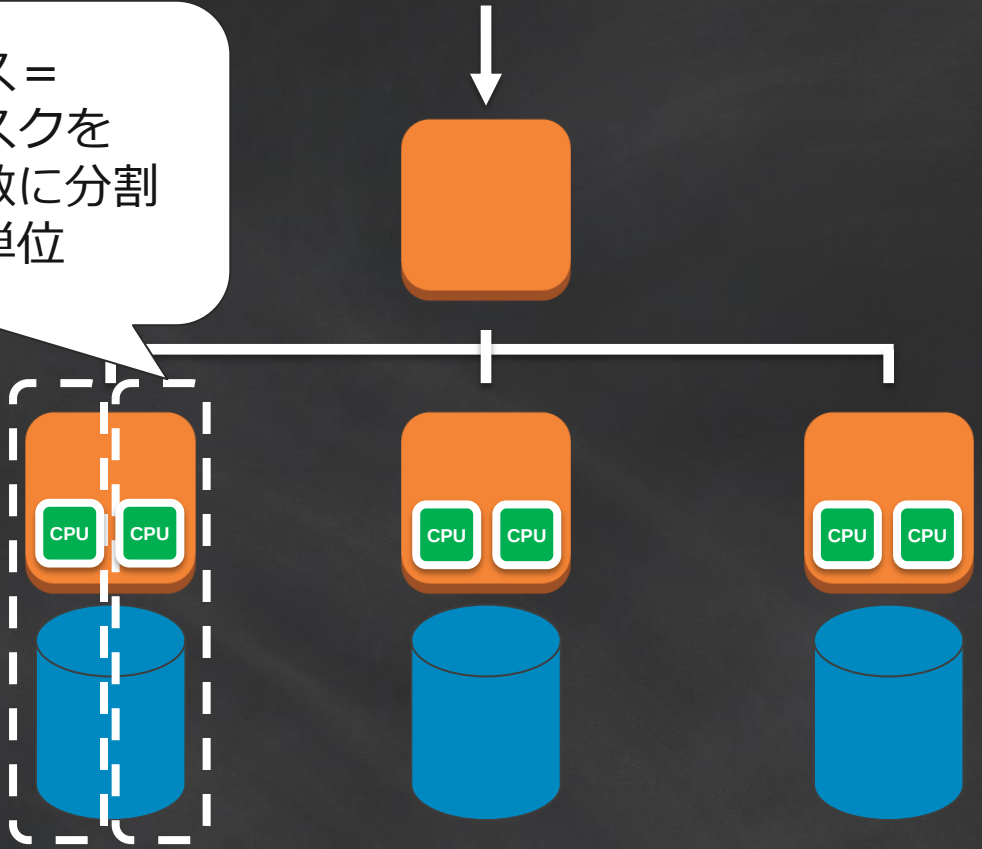


アーキテクチャ：クエリーの並列実行



アーキテクチャ：ノードスライス

ノードスライス＝
メモリとディスクを
CPUコアと同数に分割
した論理的な単位



アーキテクチャ：列指向

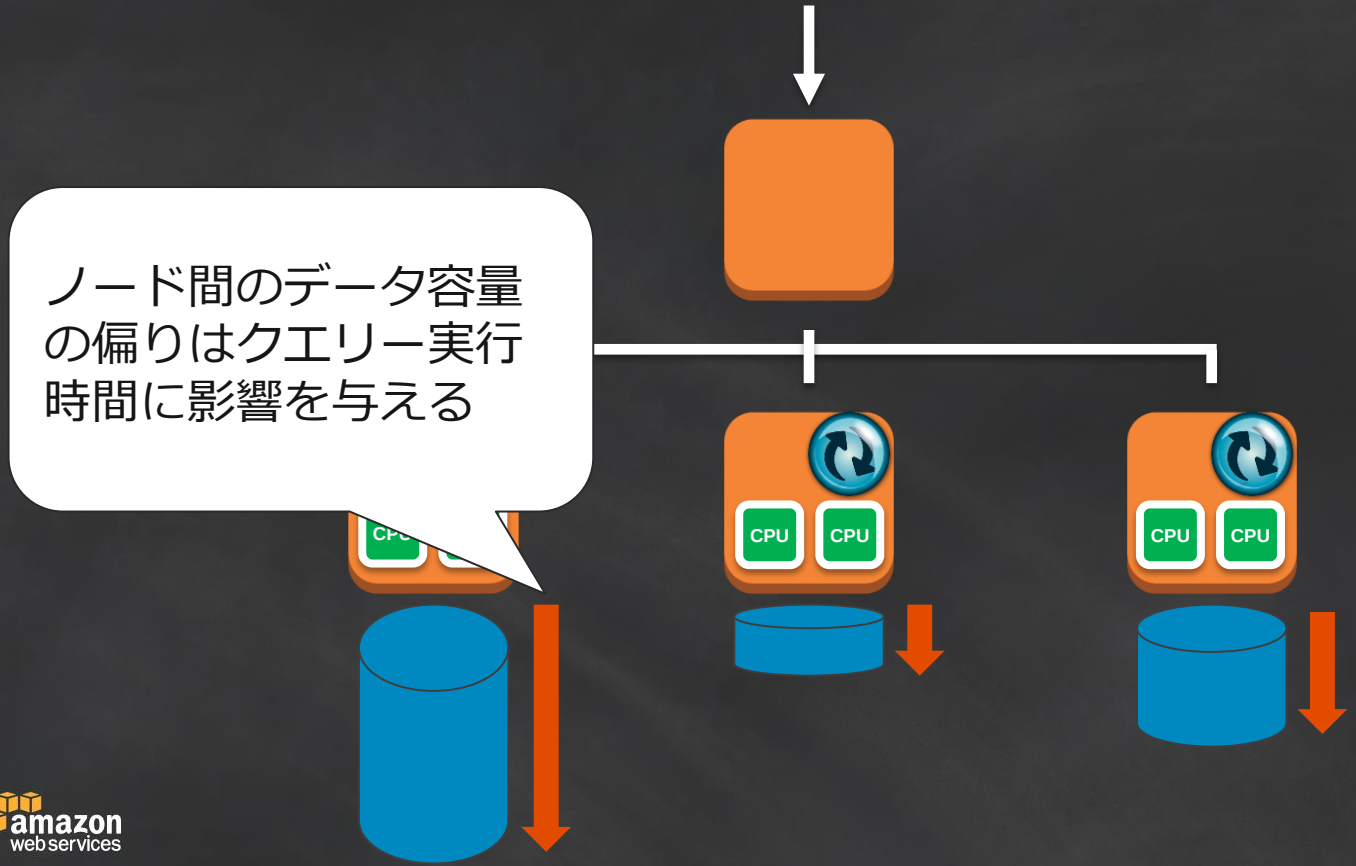
- 行指向（RDBMS）

orderid	name	price
1	Book	100
2	Pen	50
	…	
n	Eraser	70

- 列指向（Redshift）

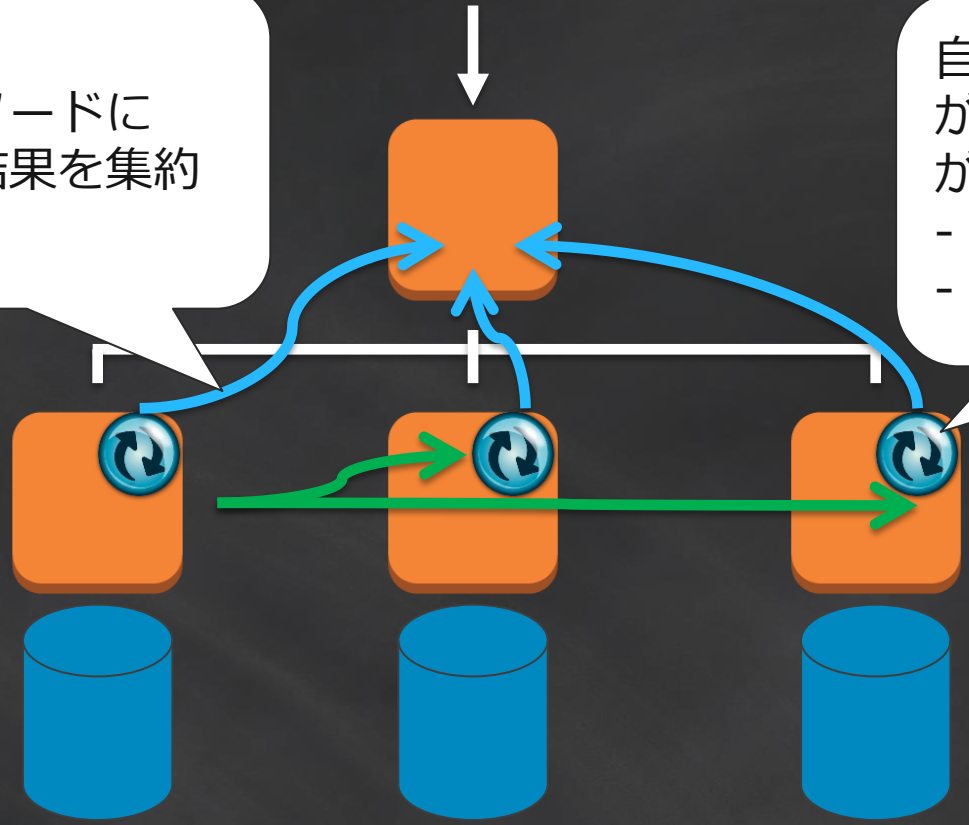
orderid	name	price
1	Book	100
2	Pen	50
	…	
n	Eraser	70

データの平準化



データの転送

リーダー・ノードに
各ノードの結果を集約

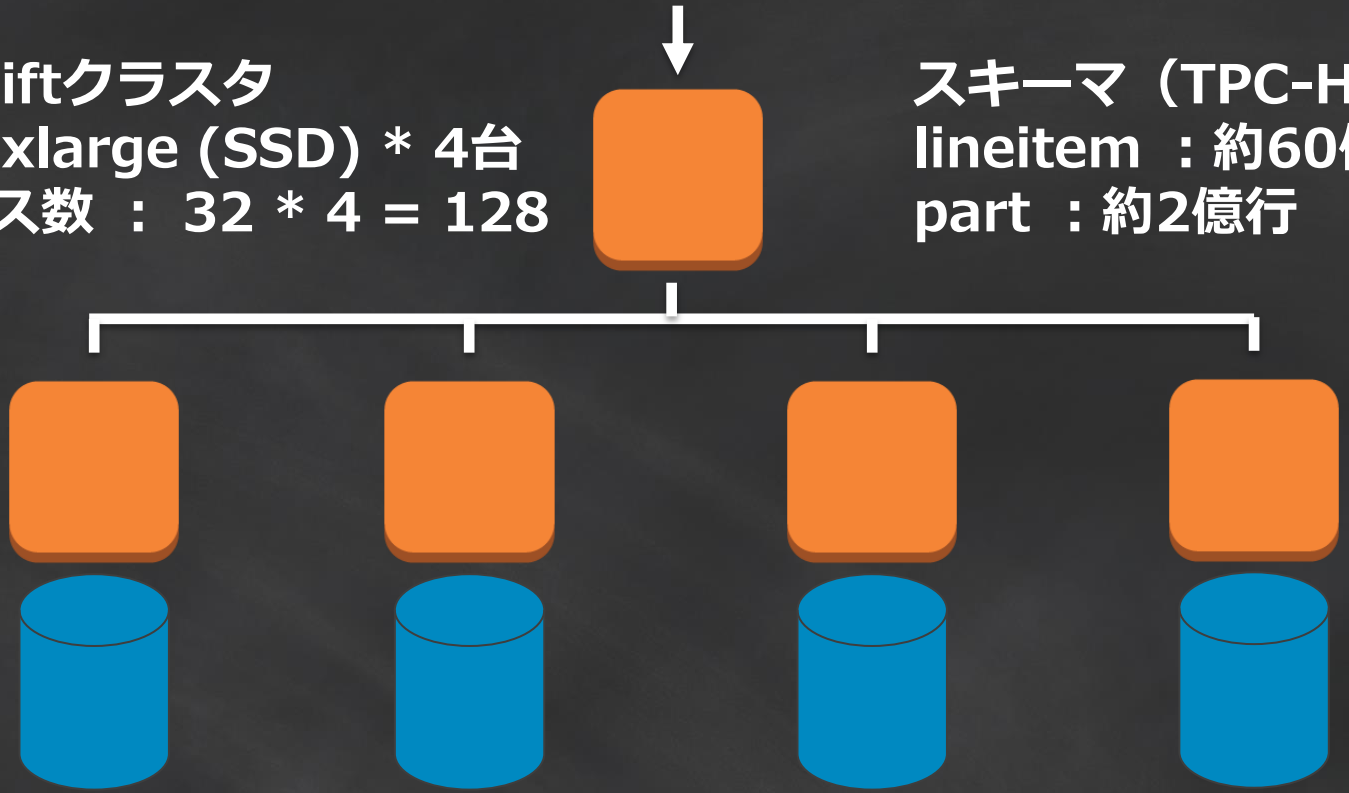


自ノードに必要なデータがない場合、データ転送が発生
- 単一ノード
- ブロードキャスト

検証用環境

検証用環境

Redshiftクラスタ
dw2.8xlarge (SSD) * 4台
スライス数 ： 32 * 4 = 128



スキーマ（TPC-H）
lineitem ：約60億行
part ：約2億行

サンプル・テーブル（TPC-Hから）

```
CREATE TABLE part (
  p_partkey int8 NOT NULL                          ,
  p_name varchar(55) NOT NULL                      ,
  p_mfgr char(25) NOT NULL                         ,
  p_brand char(10) NOT NULL                        ,
  p_type varchar(25) NOT NULL                      ,
  p_size int4 NOT NULL                             ,
  p_container char(10) NOT NULL                    ,
  p_retailprice numeric(12,2) NOT NULL             ,
  p_comment varchar(23) NOT NULL
);
```

サンプル・クエリー

```
select  sum(l_extendedprice* (1 - l_discount)) as revenue
from lineitem, part
where (p_partkey = l_partkey
        and p_brand = 'Brand#12'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 3 and l_quantity <= 3 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON')
    or
    (…)
    or
    (…);
```

テーブル設計

テーブルの分散方式

- EVEN
  - 各レコードがスライスにラウンドロビンで分散されるため均等にデータが蓄積される。
- DISTKEY
  - 明示的に指定したカラムを基準に、各レコードのスライスへの配置が決定される。
  - カラムのカーディナリティによっては、スライス間で大幅な偏りが生じる。
- ALL
  - 全てのレコードが各コンピュート・ノードに配置される。

EVEN vs. DISTKEY（1）

- EVEN

```
select trim(name) tablename, slice,
sum(rows)
from stv_tbl_perm where name='part'
group by name, slice
order
 tablename | slice |   sum
-----------+-------+---------
 part      |     0 | 1600000
 part      |     1 | 1600000
…
 part      |   126 | 1600000
 part      |   127 | 1600000
```

各スライスに均等に分散

- DISTKEY=p_partkey

```
 tablename | slice |   sum
-----------+-------+---------
 part      |     0 | 1596925
 part      |     1 | 1597634
…
 part      |   126 | 1610452
 part      |   127 | 1596154
```

キーのカーディナリティに依存

EVEN vs. DISTKEY（2）

- DISTKEY = p_brand

tablename	slice	sum
part	0	0
part	1	0
part	2	0
part	3	0
part	4	8193350
…		
part	118	8193342
part	119	0
part	120	16384823
part	121	8191943

カーディナリティの低い
カラムでは、データの極端な
偏りが生じる場合がある
= 効率の悪いクエリー

ALL

- 全レコードが各ノードの特定スライスに集約

```
 tablename | slice |   sum
-----------+-------+---------
 part      |     0 |204800000
 part      |     1 |         0
 part      |     2 |         0
 part      |     3 |         0
 part      |     4 |         0
…
 part      |    96 |204800000
 part      |    97 |         0
 part      |    98 |         0
…
```

各ノードの先頭スライスに
全レコードが格納される。

コロケーション（1）

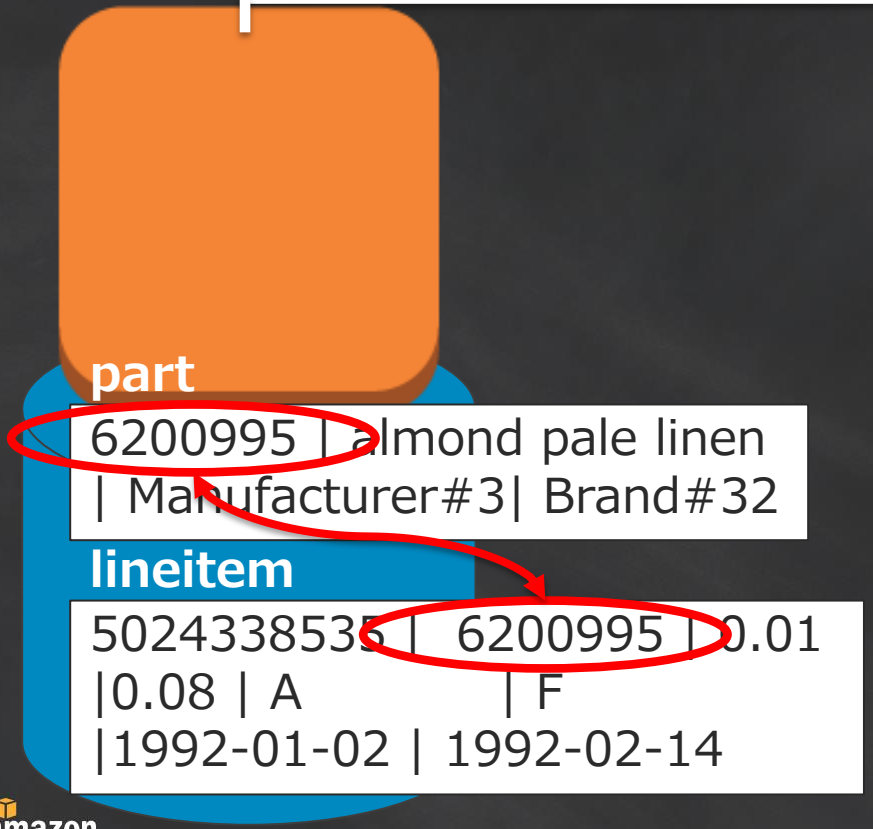
- 関連するレコードのコロケーション
  - ジョイン対象となるレコードを同一ノードに集める。
- コロケーションの方法
  1. ジョインに使用するカラムをDISTKEYとして作成　**または**
  2. 分散方式 ALLでテーブルを作成（マスター・テーブルなど）

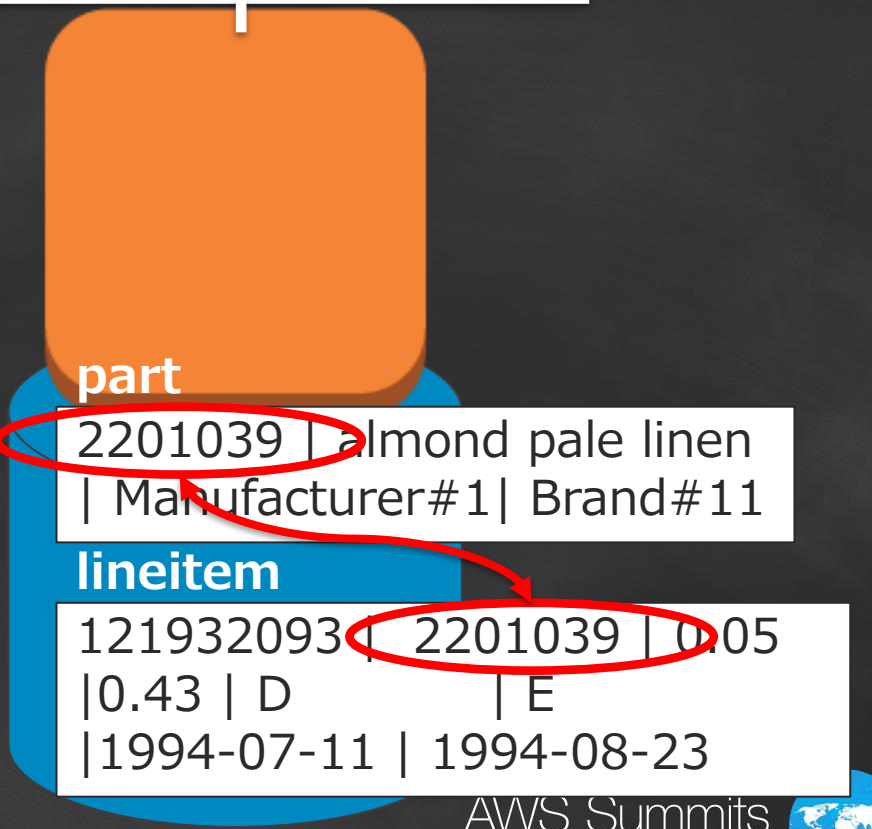
```
select  sum(l_extendedprice* (1 - l_discount)) as revenue
from lineitem, part
Where (p_partkey = l_partkey …
```

または
2. テーブルをALLで作成

1. それぞれをDISTKEYとして作成

コロケーション（2）：DISTKEY





コロケーション（3）：ALL

設計のポイント - 分散方式

サマリー

- 小規模なテーブル（マスター・テーブル）はALLで作成
- ジョインを避けてテーブルを非正規化し、EVENで作成
- 複数テーブルに対して、ジョイン対象のキーをDISTKEYで作成（コロケーション）

データのソート - SORTKEY

- ソートキーに指定した列の値がブロック上にソートされた状態で格納される。
- ソート順序を考慮した上で、最適なクエリー・プランが構築される。
- 各ブロックの最小・最大値をトラッキングしテーブル・スキャン時に、対象外のブロックをスキップ

SORTKEYの例

orderid	…	orderdate
I0001 I0002 I0003 I0004	…	2013/07/17 2013/07/18 2013/07/18 2003/07/19
		2013/08/20 2013/08/21 2013/08/22 2013/08/22

SELECT * FROM orders WHERE orderdate BETWEEN ‘2013-08-01’ AND ‘2013-08-31’;

クエリーに関係のないデータ・ブロックはスキップし、該当するブロックだけを読込む

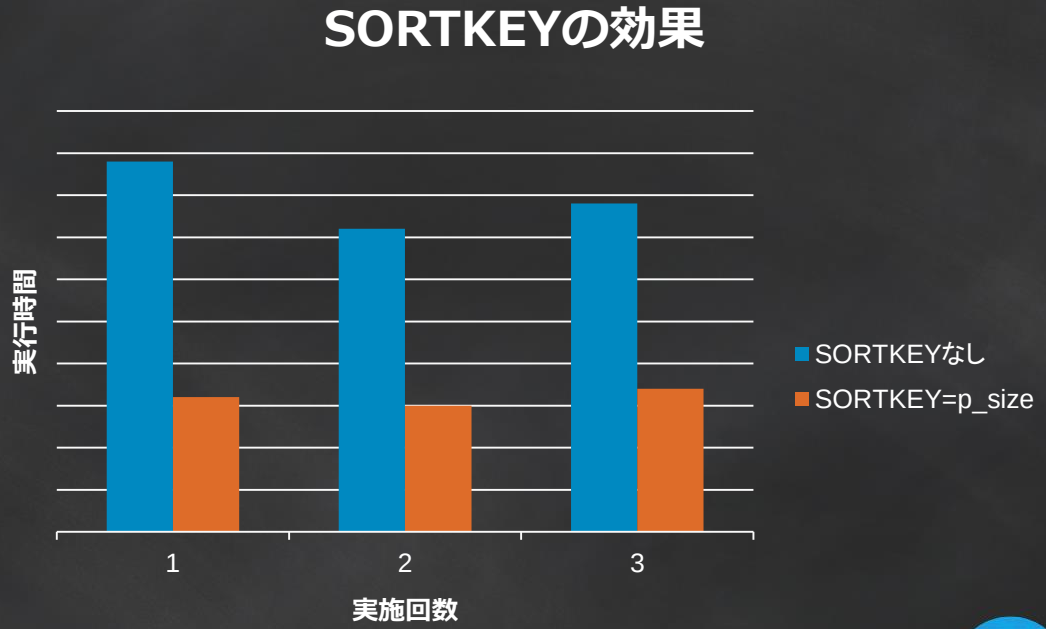
SORTKEYの比較

- p_size列にSORTKEYを指定

実行クエリー

```
select count(*) from part
where p_size between 5 and 8
order by 1;

  count
----------
 16381983
(1 row)
```



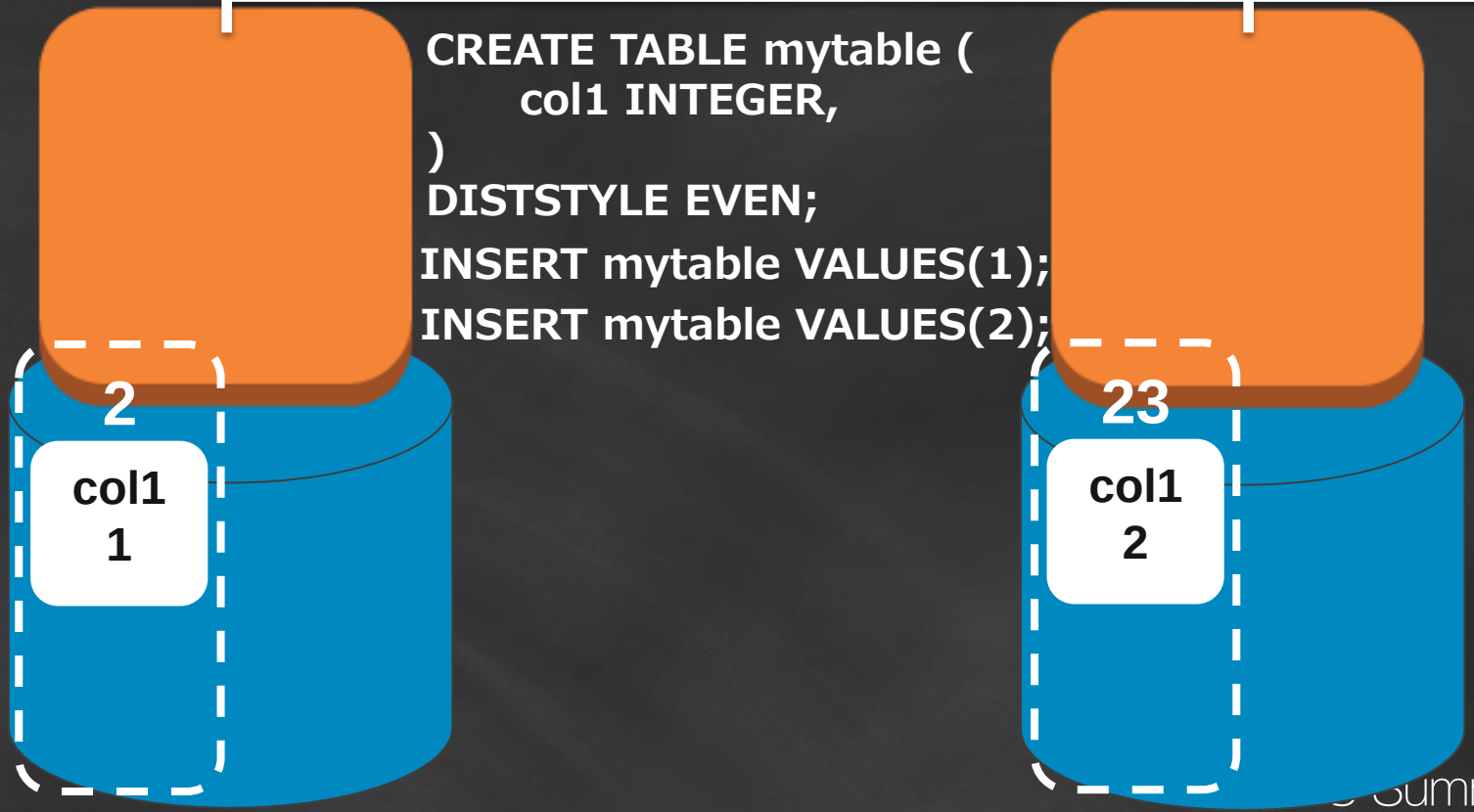
設計のポイント（SORTKEY）

サマリー

- クエリー対象外のブロックをスキップすることにより、ディスクI/Oを効率化
- 日付（DATEまたはTIMESTAMP）等の増加する型が一般的に用いられる。
- ジョインには、ハッシュジョインより効率的なマージジョインが適用
  -> カラムがDISTKEY且つSORTKEYである場合

補足：データの蓄積（１）

```
CREATE TABLE mytable (
      col1 INTEGER,
)
DISTSTYLE EVEN;
INSERT mytable VALUES(1);
INSERT mytable VALUES(2);
```



補足：データの蓄積（2）

```
INSERT INTO mytable VALUES(1);

select tbl, col, slice, blocknum from stv_blocklist where tbl = (select
oid from pg_class where relname = 'mytable') order by slice, col,
blocknum;
  tbl   | col | slice | blocknum
--------+-----+-------+----------
 108247 |   0 |     2 |        0
 108247 |   1 |     2 |        0
 108247 |   2 |     2 |        0
 108247 |   3 |     2 |        0
(4 rows)
```

COL1

ROWID

トランザクション管理用

補足：データの蓄積（3）

```
INSERT INTO mytable VALUES(2);

  tbl   | col | slice | blocknum
--------+-----+-------+----------
 108247 |   0 |     2 |
 108247 |   1 |     2 |
 108247 |   2 |     2 |
 108247 |   3 |     2 |
 108247 |   0 |    23 |
 108247 |   1 |    23 |
 108247 |   2 |    23 |        0
 108247 |   3 |    23 |        0
(8 rows)
```

DISTSTYLEがEVENであるため、別スライスに新たなブロックが作成される

クエリーの解析

クエリー・プランの読み方（１）

```
explain select  sum(l_extendedprice * (1 - l_discount)) as revenue …


XN Aggregate  (cost=98294752948050.60..98294752948050.66 rows=1 width=22)
  ->  XN Hash Join DS_BCAST_INNER \
      (cost=3071948.80..98294752947793.28 rows=102946 width=22)
                                  neitem (cost=0.00..107520152.32 \
                                   rows=222736043 width=40)
                                60000.00..2560000.00 rows=204779520 width=40)
                                 on part  (cost=0.00..2560000.00 \
                                  rows=204779520 width=40)
```

全体のコストと各ステップにかかる相対的なコストが記載
-> どのステップが高コストかを判断

クエリー・プランの読み方（２）

```
explain select  sum(l_extendedprice * (1 - l_discount)) as revenue …


XN Aggregate  (cost=98294752948050.66..98294752948050.66 rows=1 width=22)
  ->  XN Hash Join DS_BCAST_INNER \
      (cost=3071948.80..98294752947793.
        ->  XN Seq Scan on lineitem  (co
                                     ro
        ->  XN Hash  (cost=2560000.00..2                    width=40)
              ->  XN Seq Scan on part  (cost=0.00..2560000.00 \
                                        rows=204779520 width=40)
```

ジョイン時のインナーテーブルの取り扱い

データ転送の極小化（１）

- ジョインやグループ関数を伴うクエリーは、多量のデータ転送が発生する可能性がある。
- データ転送のオプション
  - DS_DIST_NONE、DS_DIST_ALL_NONE：Good
  - DS_DIST_INNER、DS_DIST_ALL_INNER：インナーテーブルの転送
  - **DS_BCAST_INNER**：インナーテーブルの全ノードへの転送
  - **DS_DIST_BOTH**：インナー、アウターテーブルの転送

データ転送の極小化（２）

- 特にDS_BCAST_INNERとDS_DIST_BOTHは排除すべき
- テーブルのコロケーションが対処法となる
  1. ジョインに使用するカラムをDISTKEYとして作成　**または**
  2. 分散方式 ALLでテーブルを作成（マスター・テーブルなど）

中間結果の出力

- Redshift は中間結果をディスクに書込む
  - メモリの枯渇を防ぎ、確実にクエリーを実行
  - 更なるディスクI/Oは、パフォーマンスの劣化を生じる

```
// どのステップでI/Oが発生しているかを特定
select query, step, rows, workmem, label, is_diskbased
from svl_query_summary
where query = 1025 order by workmem desc;

query| step|  rows   |  workmem  |     label      | is_diskbased
-----+-----+--------+-----------+----------------+--------------
1025 |  0  |16000000| 141557760 |scan tbl=9      | f
1025 |  2  |16000000| 135266304 |hash tbl=142    | t
```

クエリー解析のポイント

サマリー

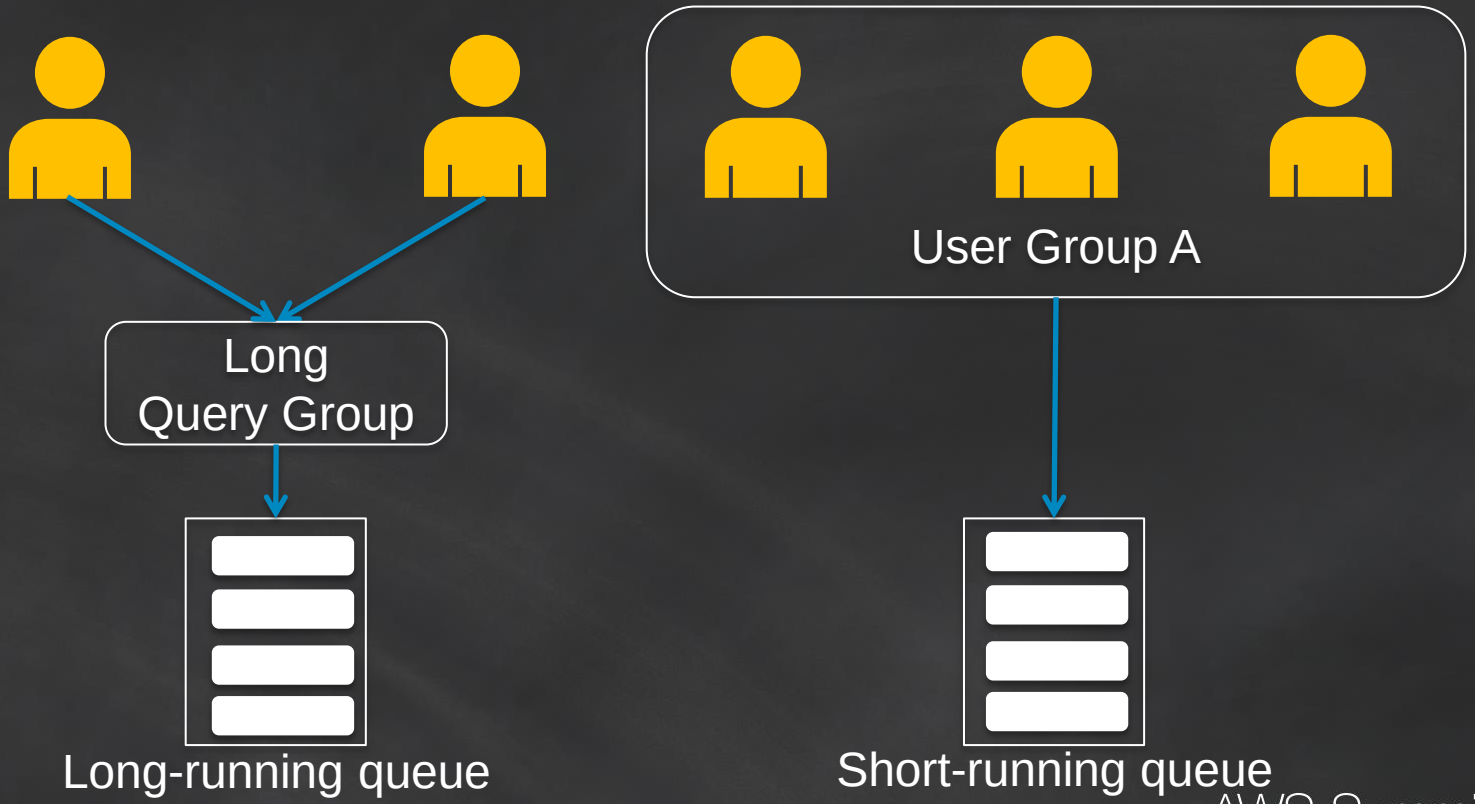
- クエリー・プランから、相対的に高コストなステップを割り出す。
- 中間結果の出力が高コストの起因となっていないか。
- クエリーの書換えやコロケーションを含むテーブル定義の変更を検討

Workload Management

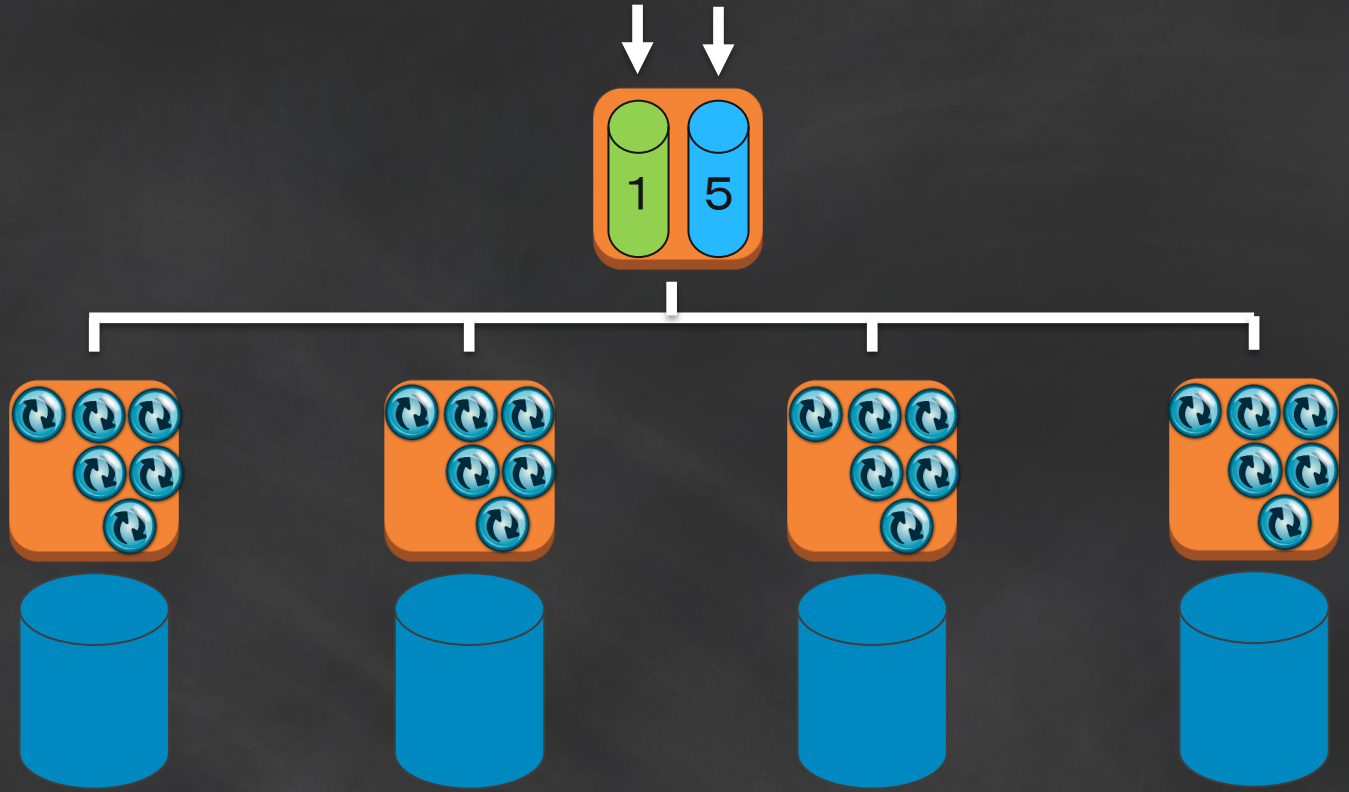
Workload Managementの概要

- 長期実行クエリーは、クラスタ全体のボトルネックとなり、短期実行クエリーを待たせる可能性がある。
- クエリー並列度の上限を設けた複数のキューを定義
- クエリーを指定されたキューにルーティング

WLMの実装（１）


User Group A
Long
Query Group
Long-running queue
Short-running queue

WLMの実装（2）



WLMの効果

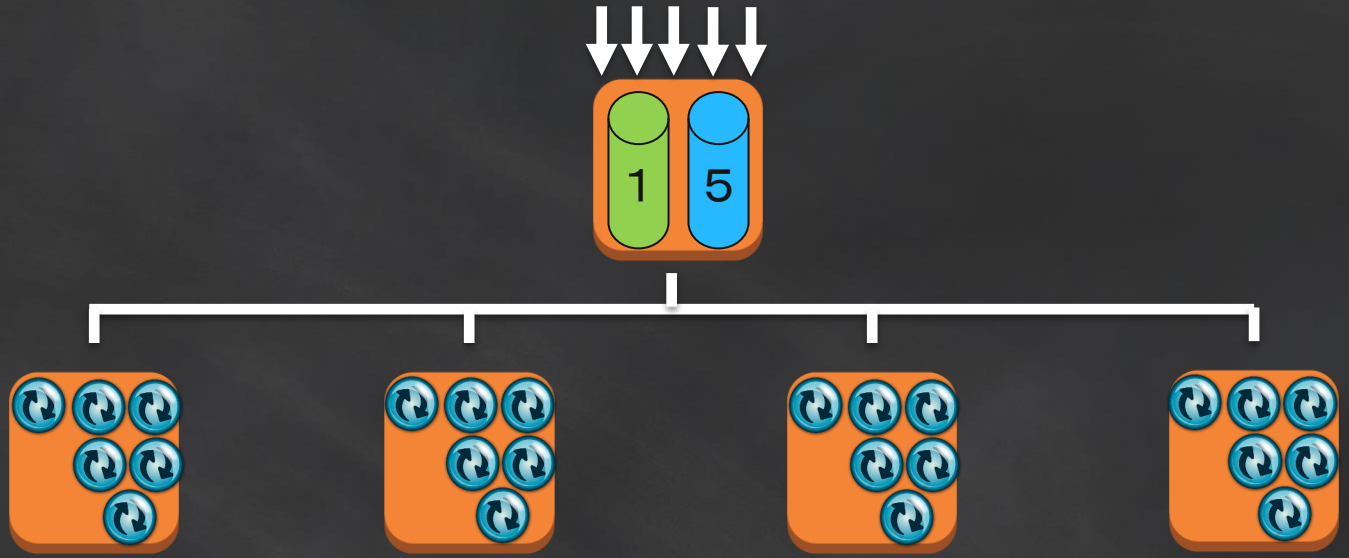
- キュー単位でクエリー並列度を保障
  - メモリのアロケーションも指定可能
- 特定ユーザ（群）によるクラスタ占有を回避
- 並列度の増加は、性能の向上にはつながらない
  -> リソース競合の可能性

WLMの制限

- 複数キュー間のリソース共有や重みづけ
- キューの動的な設定変更 -> クラスタの再起動が必要

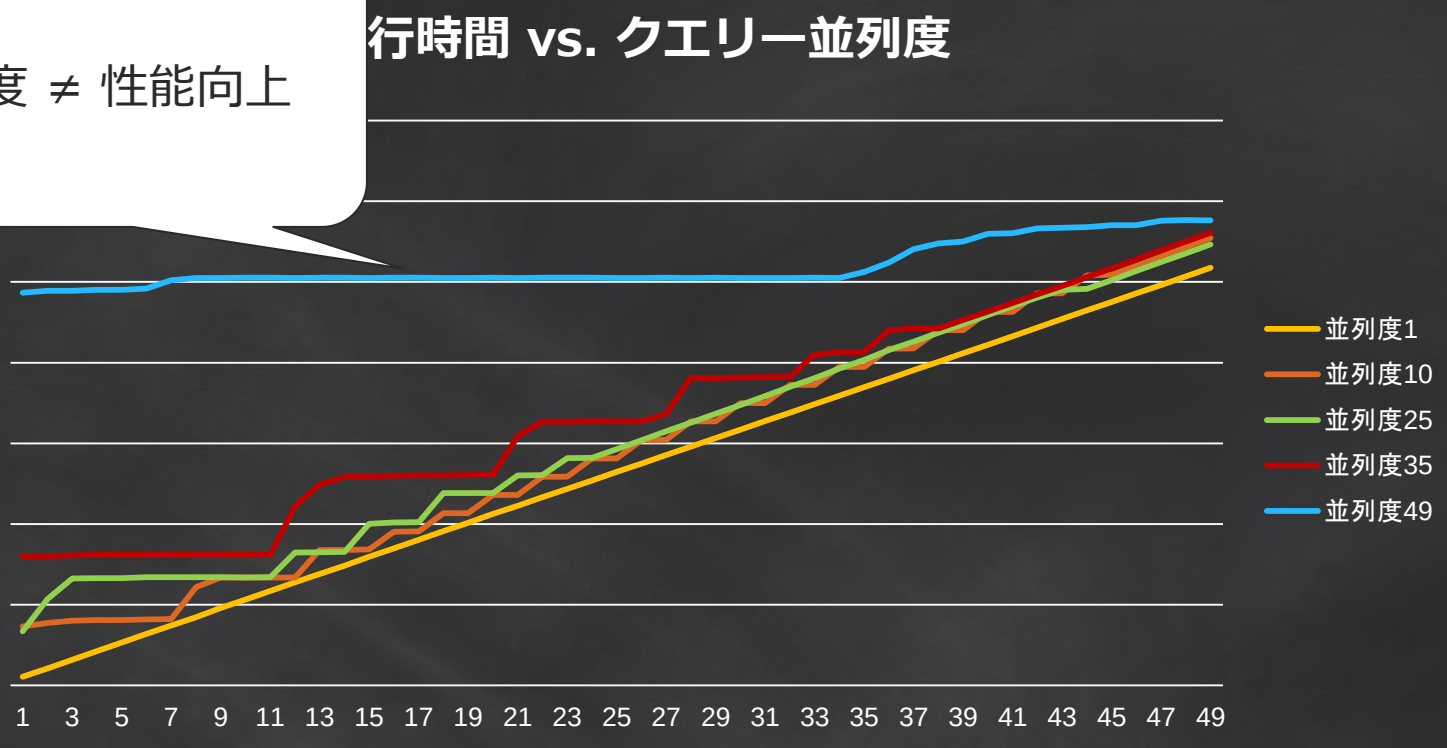
性能比較（１）

クライアント・アプリ：49スレッド並列
WLMキューの並列数：1,10,25,35,49



性能比較（2）

高並列度 ≠ 性能向上



まとめ

- テーブル設計の段階で、分散方式を検討
- データ転送を極小化するため、コロケーションを活用
- WLMにより適切なリソース配分を行う

2014.09.09 SAVE THE DATE

http://csd.awseventsjapan.com/

Cloud Storage & DB Day　検索

amazon web services

AWS Summits 2014

ご清聴ありがとうございました。