



基幹業務もHadoop(EMR)で!!のその後

BigData-JAWS 勉強会#4

2016/12/12

Future Architect Inc,
Keigo Suda

実際に開発が終わったその後の話

いかに本番利用に堪えられるようになったかの軌跡

- ✓ どういった課題があったか
- ✓ どのように対応したか





須田桂伍 (すだ けいご)

- * 2012年新卒入社(今年5年目 orz)
- * Technology Innovation Group スペシャリスト
- * 最近の専門 -> ビッグデータ領域(インフラ～アプリ)
- * 最近はもっぱらKafkaとストリーム処理エンジンの諸々

宣伝

- 最近やっているIoTためのプラットフォーム構築の話
- Apache Kafka on AWS

12月
14 もう1つのHadoop Summit ~ #HAWQ #Spark
#Kafkaなどの紹介



ハッシュタグ : #futureofdata

募集内容	参加枠 無料	先着順 119/150人
Blogger枠 無料		先着順 0/3人

12月
15 Apache Kafka Meetup Japan #2 @Recruit Technologies

リクルートテクノロジーズさん、会場ありがとうございます！

主催 : Kafka Meetup Japan



ハッシュタグ : #kafkajp

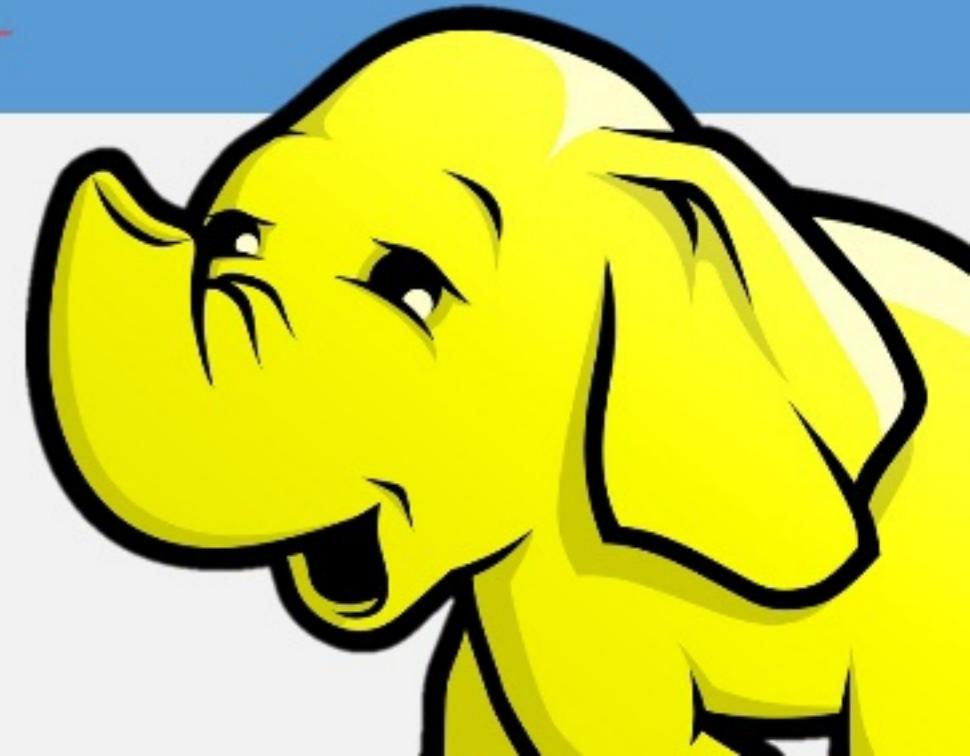
募集内容	抽選 無料	先着順（抽選終了） 96/96人
------	----------	---------------------

もくじ

- EMRとローソン
- 性能テストとチューニング
- まとめ



マチのぼっこステーション
LAWSON





基幹業務もHadoopで!!

ローソンにおける店舗発注業務への
Hadoop + Hive導入とその取り組みについて

設計～開発時の話

Future Architect
Keigo Suda

Tech Blog

Home > Blog > Post

「基幹業務もHadoopで!!」のその後～性能編～

▲ 須田桂伍

2016/10/05

● AWS, EMR, Hadoop, Hive, YARN

Category: Categories: 開発



Categories

- PR (1)
- DB (1)
- 社内制度 (4)
- 開発 (2)
- プロジェクトマネジメント (1)
- 素材 (1)
- カンファレンス (1)
- 告知 (4)
- RP/告知 (2)
- 制度 (2)

Tags

EMRとローソン

店舗発注業務のセンター化

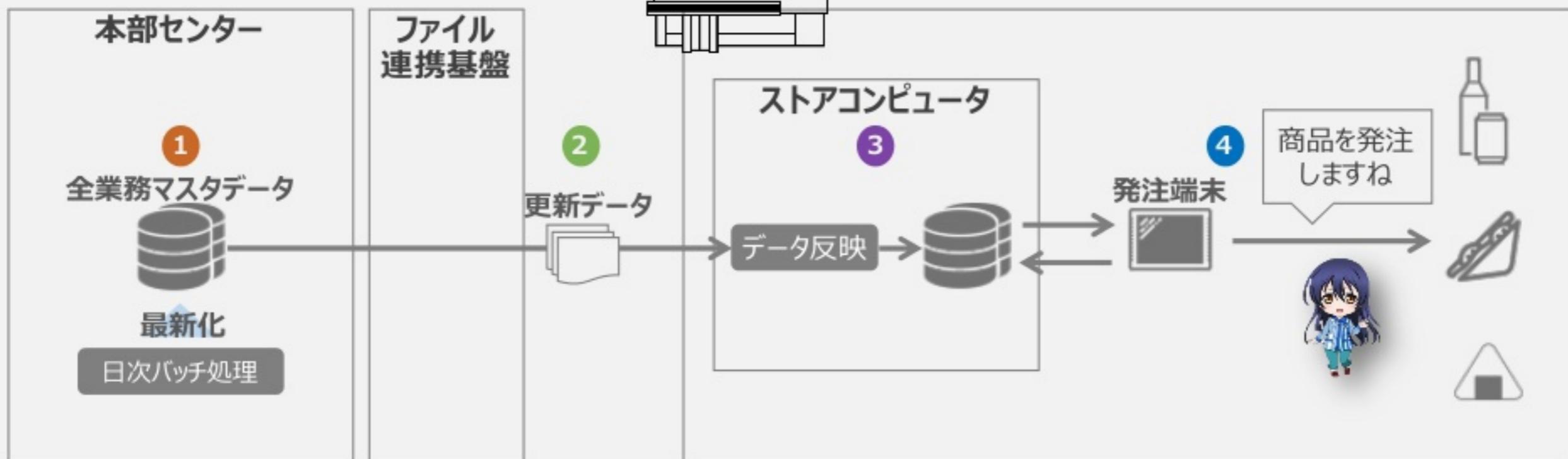
発注時に利用するマスタ作成をセンタ集約

- ✓ 店舗毎に行われていたマスタデータ作成処理を集約
- ✓ 店舗からはAPI経由でマスタデータを参照



これに表示される
各種データを作成

店舗発注業務の裏側

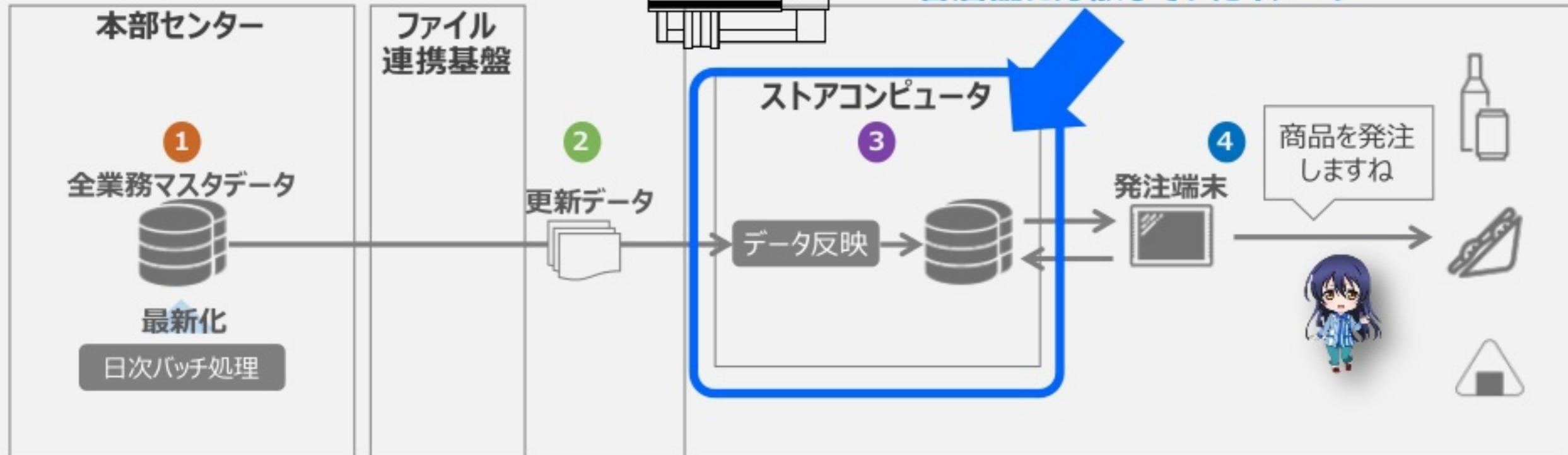


- 1 全業務マスタデータの最新化処理**
ローソン全業務で利用されるマスタデータを
日次バッチで最新化
- 2 店舗へ更新分データのファイル連携**
最新化された全業務マスタデータの更新差分を
各店舗へファイル連携

- 3 更新分データのDB反映処理**
連携されたファイルデータを各店舗にある
ストコン内のDBへ反映する。
- 4 発注時の商品データ参照**
最新化されたマスタデータをもとに発注業務を実施

店舗発注業務の裏側

これまで処理負荷を
各店舗に分散していたイメージ



- 1 全業務マスタデータの最新化処理**
ローソン全業務で利用されるマスタデータを
日次バッチで最新化

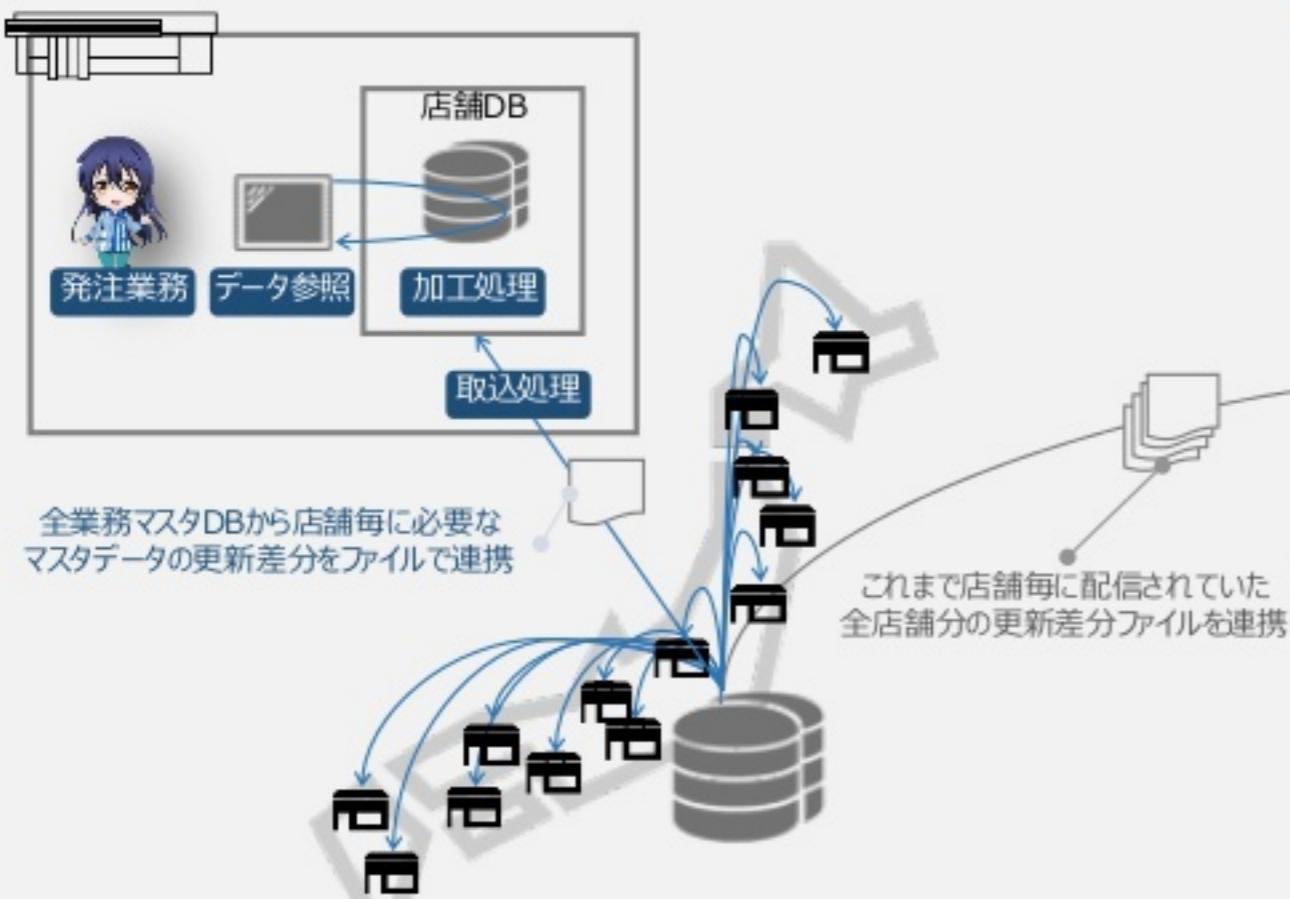
- 2 店舗へ更新分データのファイル連携**
最新化された全業務マスタデータの更新差分を
各店舗へファイル連携

- 3 更新分データのDB反映処理**
連携されたファイルデータを各店舗にある
ストコン内のDBへ反映する。
- 4 発注時の商品データ参照**
最新化されたマスタデータをもとに発注業務を実施

機能のセンター集約

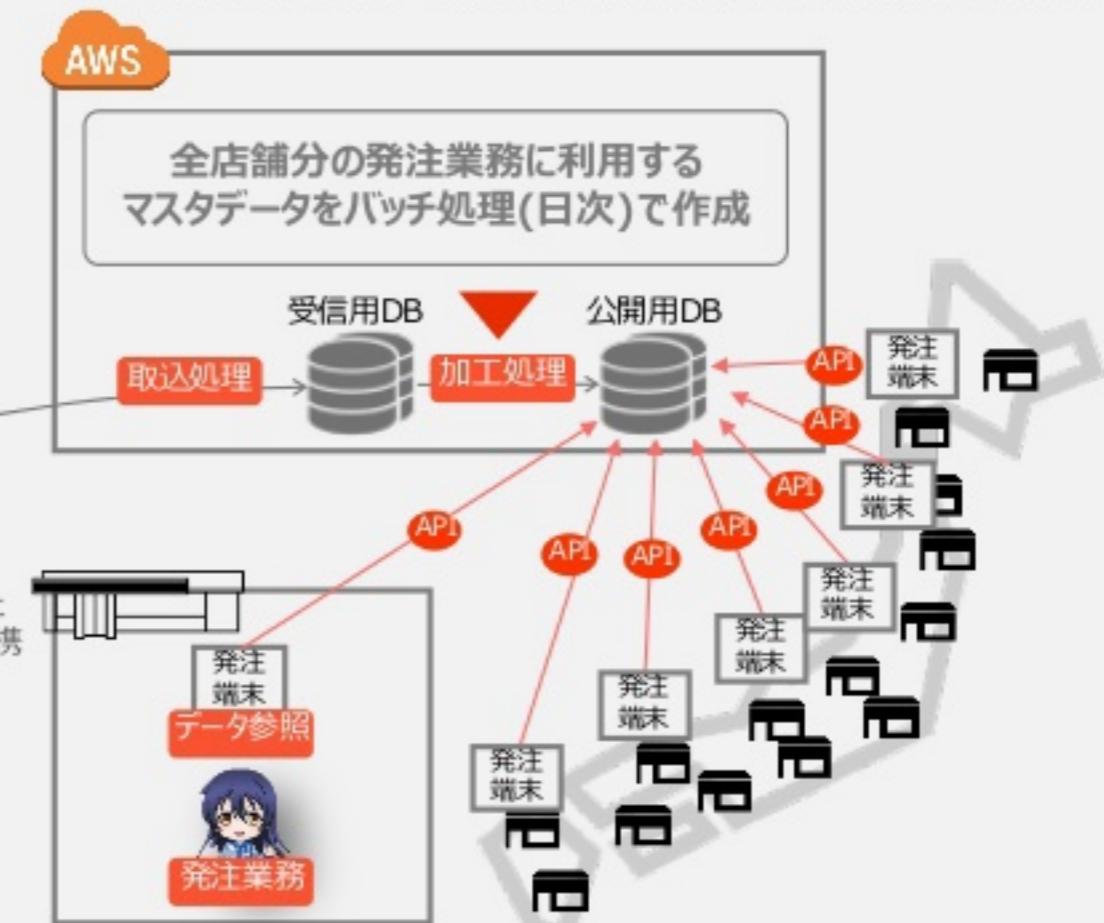
Before

1. 全業務マスタDBから各店舗へ更新差分ファイルを配信
2. 店舗毎にDBへ差分反映後、発注利用マスターを作成
3. 作成されたマスターは発注業務時に発注端末から参照



After

1. 全業務マスタDBから全店舗分の更新差分ファイルを配信
2. 受信用DBへ差分反映後、全店舗分の発注利用マスターを作成
3. 作成されたマスターはREST APIで公開し、発注端末より参照



しかしその壁も高い…



店舗数増加への考慮



膨大なレコード件数



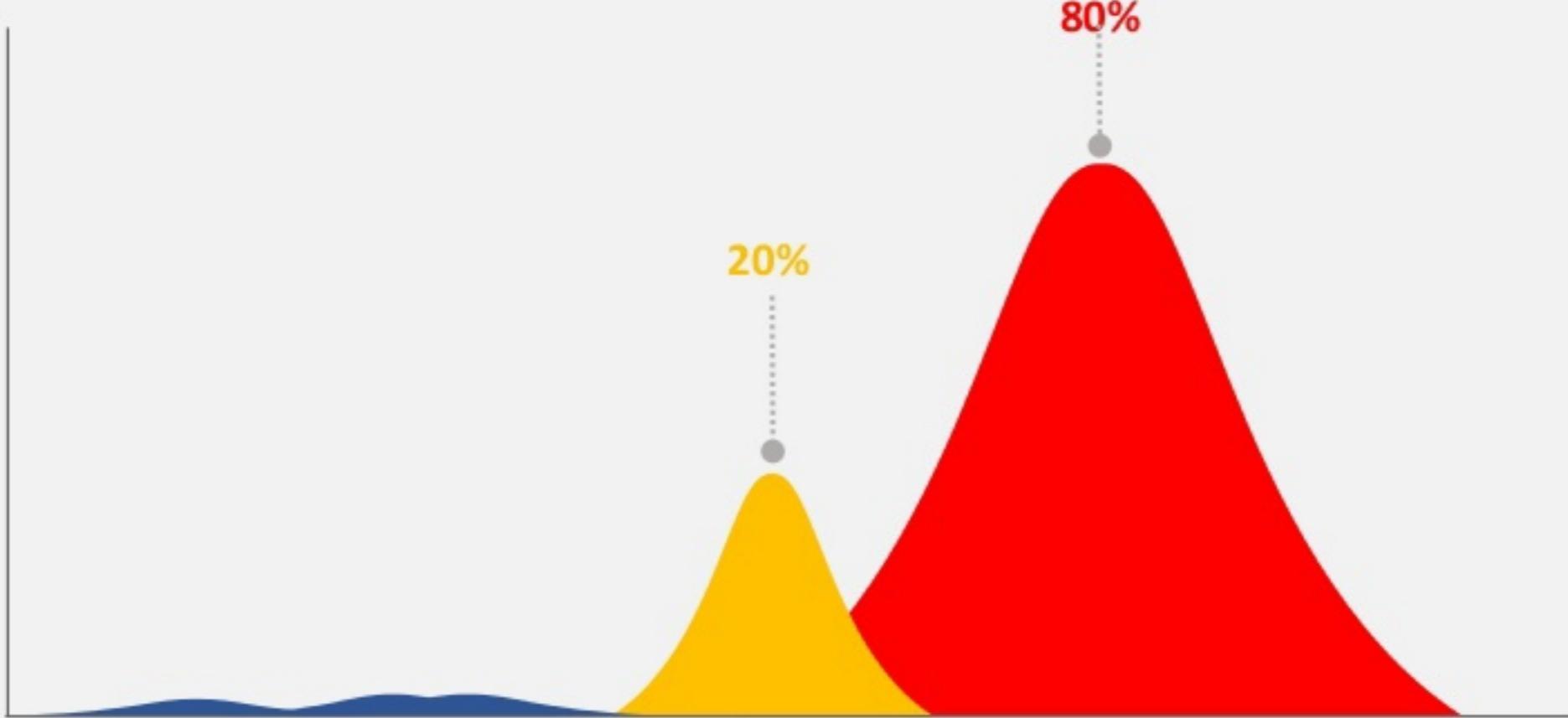
ピーク時の処理多重度



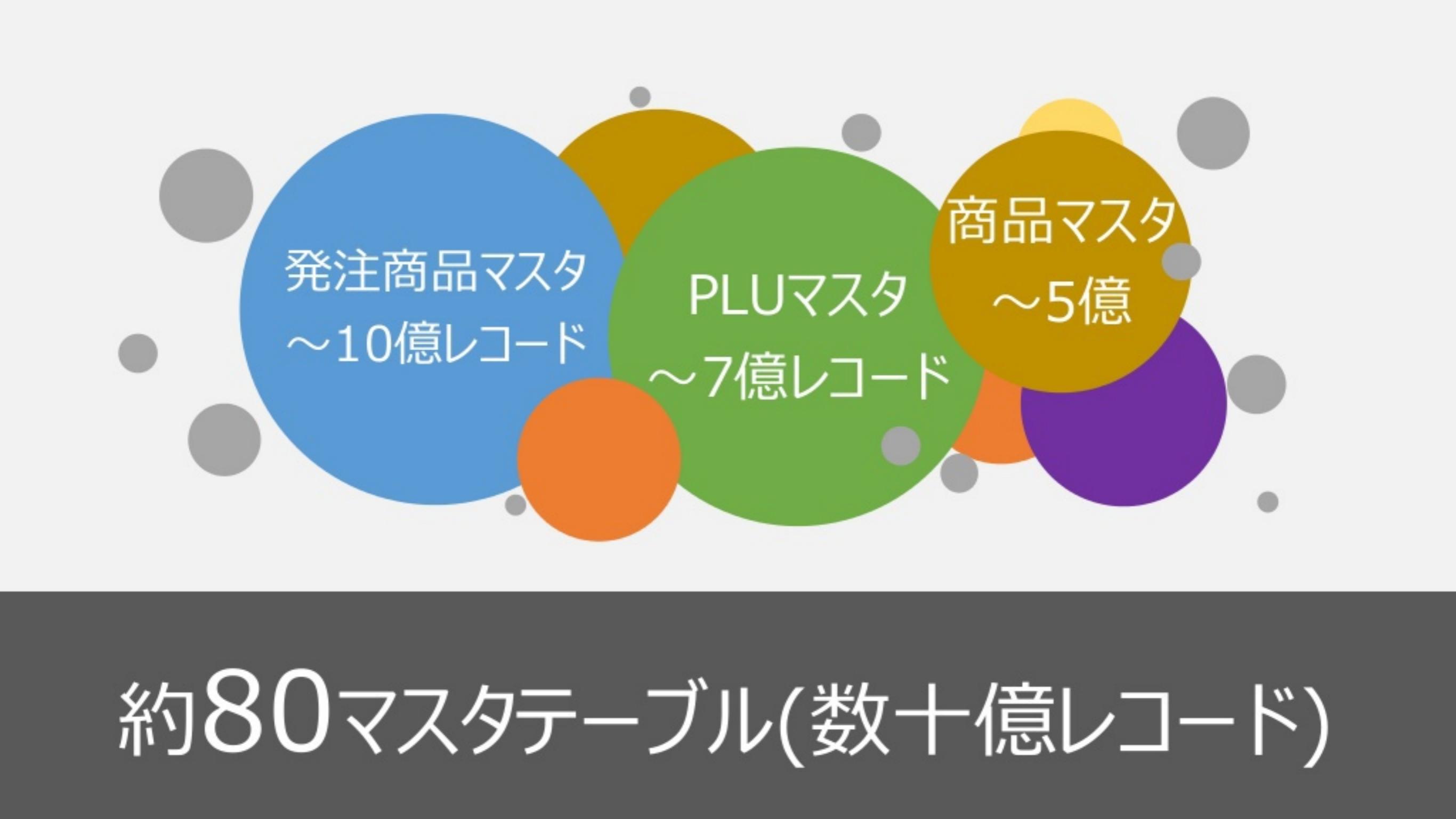
限られたバッチウィンドウ

\$12,000





全店舗分の処理ピークが重なる



発注商品マスター
～10億レコード

PLUマスター
～7億レコード

商品マスター
～5億

約80マスターテーブル(数十億レコード)

EMRでまとめて処理しよう！



スケールアウト&スケールアップの戦略が柔軟

* クラスタ台数、EC2インスタンスタイプ、タスクグループと様々な組み合わせが可能



エコシステム含めた多様な処理オプション

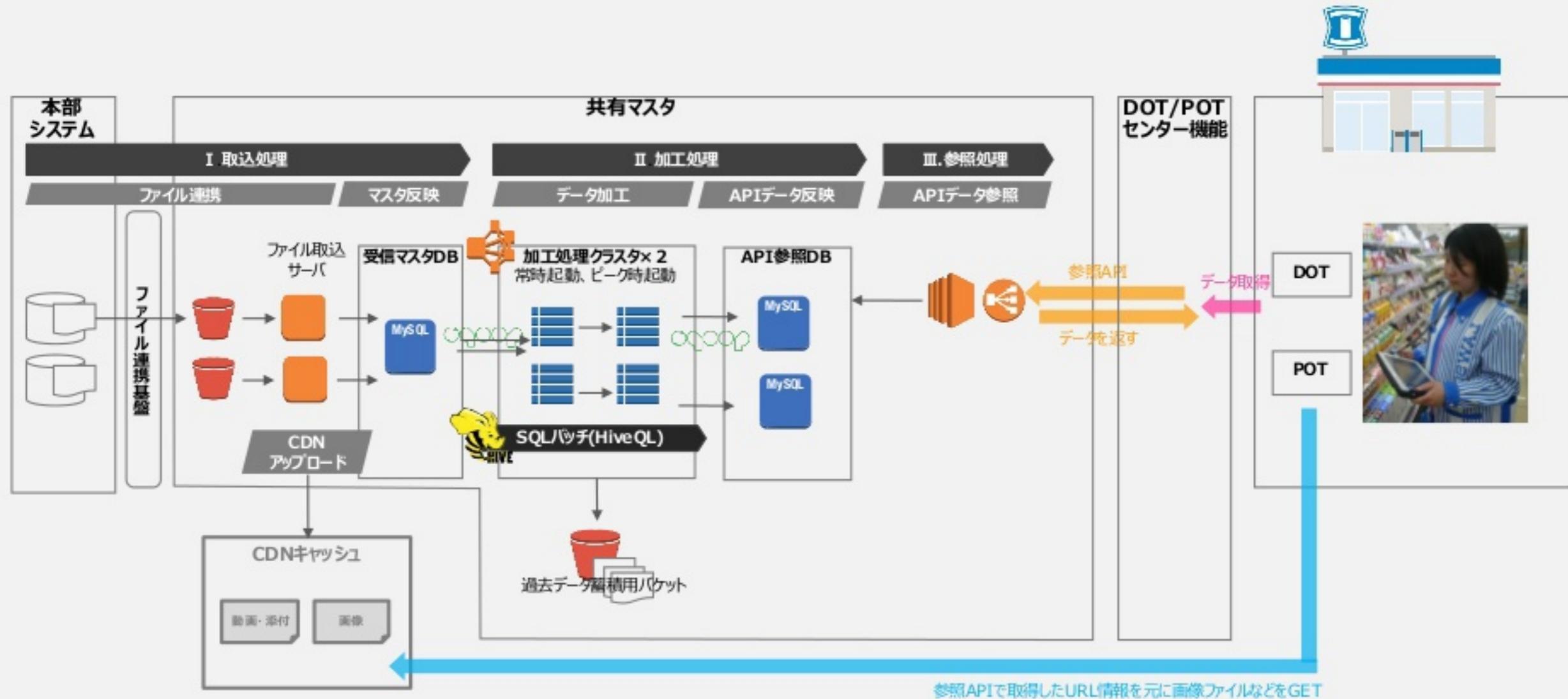
* EMR(というかHadoop)を取り囲むエコシステム群による機能補完



EC2だし、いざとなればなんとかなるでしょ(小並感)

* いざとなればSSHで入ってごにょごにょできるし…

アーキテクチャ全体概要

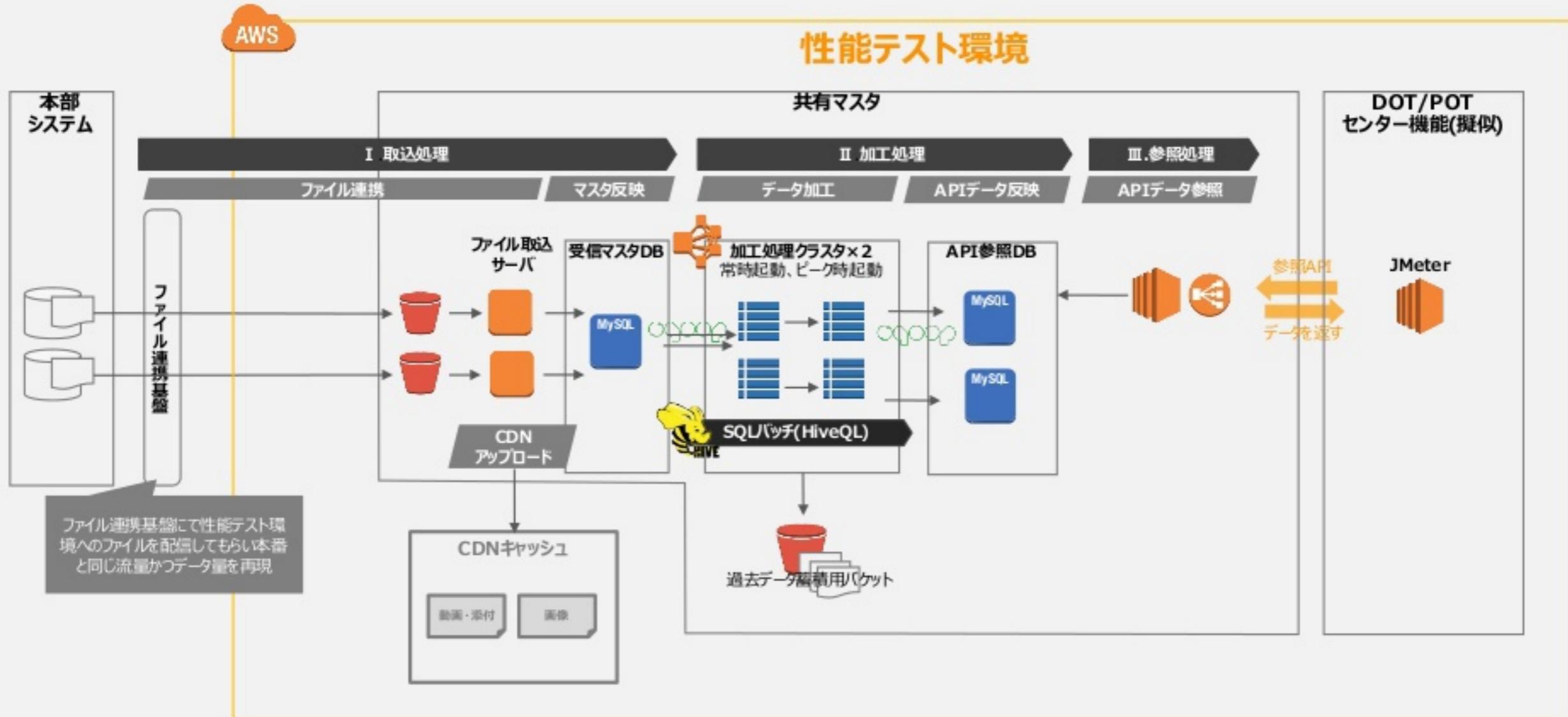


ここまでがカンファレンス時までの話

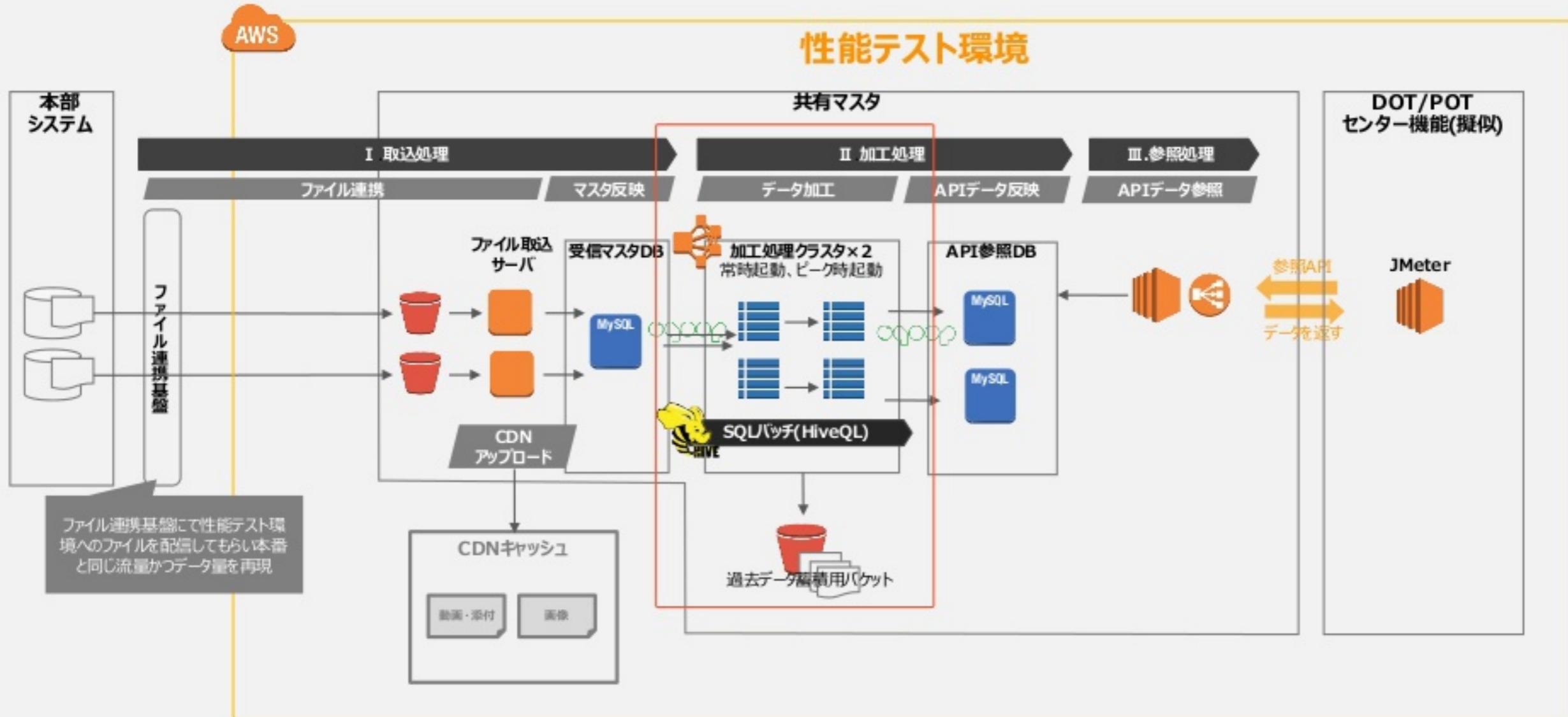


性能テスト/チューニング

性能テスト実施環境



性能テスト実施環境



やっぱり本番負荷は並じやなかつた



ちまたのベストプラクティスがはまらない問題

* ググって出てくるチューニング例は当てはまらないことが多かった orz



思っていた以上にジョブ投入多密度が高かつた問題

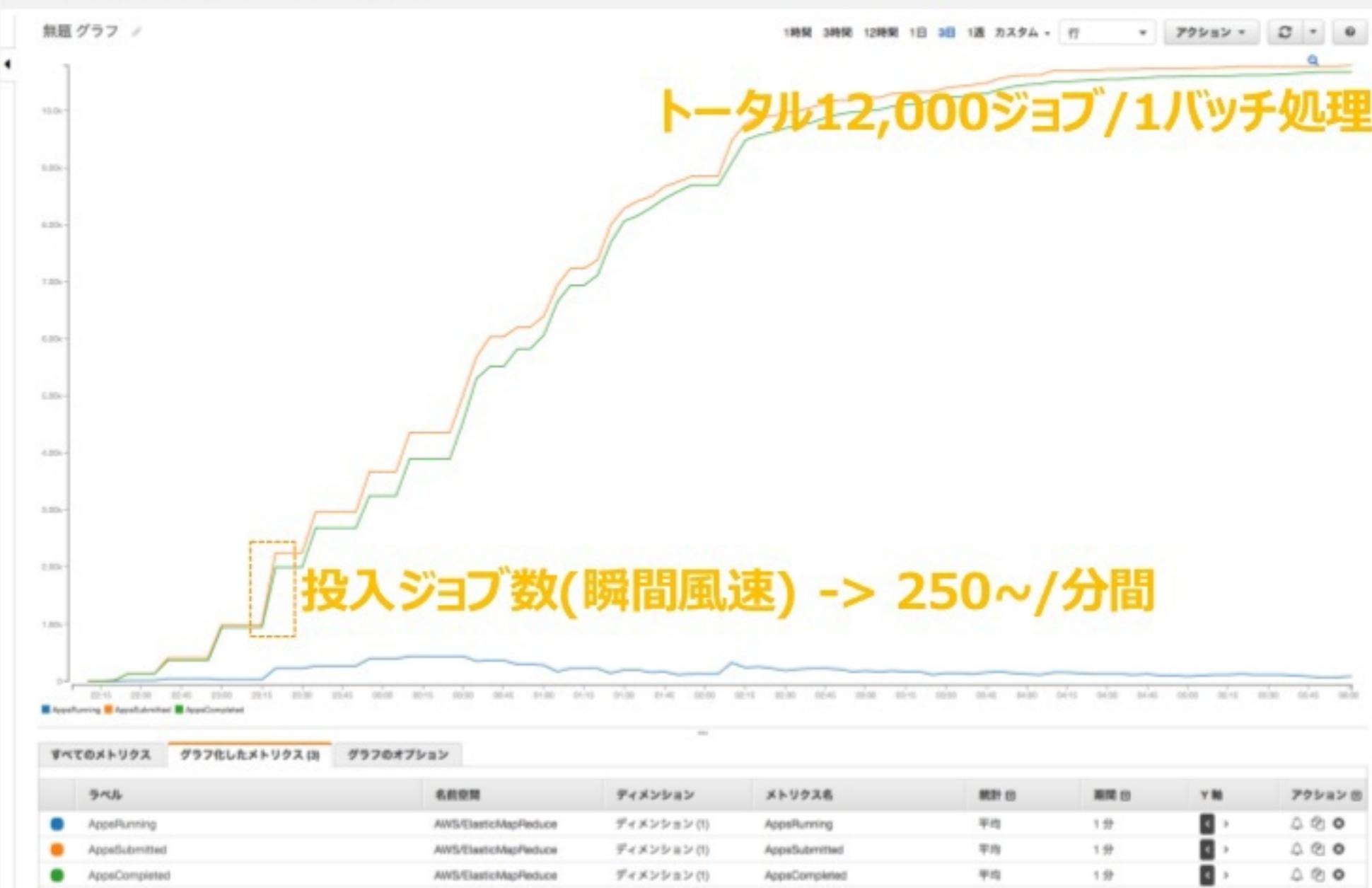
* 実環境では当初想定よりもかなりの処理が多重で実行される結果に



RDSがふん詰まる問題

* スループットが思った以上に出ない&EMRからボトルネックが移動してきた

数値でみる処理状況



数値でみる処理状況



特徴

- とにかくジョブの同時投入数、稼働数が多い
 - 1マスタ作成処理につき平均20ワーク作成ほど * 80マスタテーブル * 店舗数分
- 一つ一つのクエリは結構重たい
 - 非正規化処理が中心のため大量読み取り&大量書き出し

処理時間の推移でみるチューニング過程

まずは1/80マスタを全店舗分

90分 → 25分



80マスタを全店舗分

∞ → 90分

チューニング対応一覧

- 以下は主にEMR関連で、細々としたアプリケーションのチューニングなども別途対応

対象	対応内容	対応詳細
Hive	クエリチューニング	<ul style="list-style-type: none">パーティショニング生成ワーク数の削減
Hive	ファイルフォーマットの変更	-
Hive	データ圧縮	<ul style="list-style-type: none">圧縮アルゴリズムの選択中間データの圧縮転送データの圧縮
Hive	アクセスプランの効率化	<ul style="list-style-type: none">Vectorizationの有効化CBO有効化
Hive	処理プロセス(スレッド数)の効率化	<ul style="list-style-type: none">Reducer数の調整Shuffleコネクション数の調整
Hive	結合処理の最適化	<ul style="list-style-type: none">パケットの利用MapJoinの利用
Hive	リソース利用の最適化	<ul style="list-style-type: none">JVMヒープサイズの変更Tezコンテナの再利用
YARN	コンテナ配布の最適化	<ul style="list-style-type: none">割当コンテナサイズの範囲調整コンテナサイズの変更
YARN	スケジューラ調整	<ul style="list-style-type: none">スケジューラの変更処理用キューの設定
DFS	NameNode閑速	<ul style="list-style-type: none">NameNode処理スレッド数調整
その他	システム構成変更	タスクインスタンスグループの追加

処理時間の推移でみるチューニング過程

主にHiveのクエリやパラメータチューニング

まずは1/80マスタを全店舗分

90分 → 25分



80マスタを全店舗分

∞ → 90分

クエリチューニング

- パーティション利用の廃止

- パーティション後のファイルサイズが小さくなりがちで、結果IO効率が悪くなっていた
- パーティション作成処理のオーバーヘッドの積み重ねが処理時間のウェイトを占めるようになっていた

- ワークテーブル数の削減(可読性をさげない程度に)

- 複数ワークテーブルを集約して一つのクエリにする
- 1クエリでさばく処理量を増やすことでIO効率をあげていく(UNION ALLなど)

クエリをまとめていく例

```
1 //これまで規約に則っていたので、以下のようにクエリが分割されてしまっているものもあった
2
3 -- *****
4 -- 处理名 XXXXXX
5 -- 处理概要 XXXXXX
6 --
7 INSERT INTO TABLE
8   WORK_TABLE
9 SELECT
10   COL1
11 , COL2
12 , COL3
13 FROM
14   TABLE02
15 GROUP BY
16   COL1
17 , COL2
18 HAVING
19   MAX(COL3) != 'some_value'
20 ;
21
22 -- *****
23 -- 处理名 XXXXXX
24 -- 处理概要 XXXXXX
25 --
26 INSERT INTO TABLE
27   WORK_TABLE
28 SELECT
29   COL1
30 , COL2
31 , COL3
32 FROM TABLE03
33 INNER JOIN
34   TABLE04
35 ON
36   TABLE03.COL1 = TABLE04.COL1
37 WHERE
38   TABLE03.COL2 = 'some_value'
39 ;
```



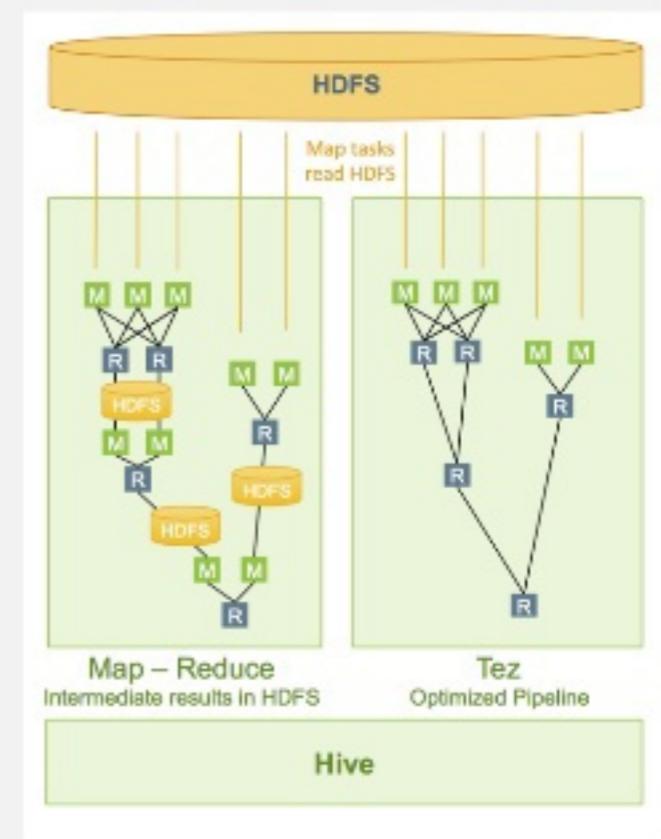
```
1 //同一ワークテーブルへの追記処理はまとめあげる
2
3 INSERT INTO TABLE
4   WORK_TABLE
5 SELECT
6   COL1
7 , COL2
8 , COL3
9 FROM
10  TABLE02
11 GROUP BY
12   COL1
13 , COL2
14 HAVING
15   MAX(COL3) != 'some_value'
16 UNION ALL
17 -- *****
18 -- 处理名 XXXXXX
19 -- 处理概要 XXXXXX
20 --
21 SELECT
22   COL1
23 , COL2
24 , COL3
25 FROM TABLE03
26 INNER JOIN
27   TABLE04
28 ON
29   TABLE03.COL1 = TABLE04.COL1
30 WHERE
31   TABLE03.COL2 = 'some_value'
32 UNION ALL
33 ....
```

パラメータチューニング

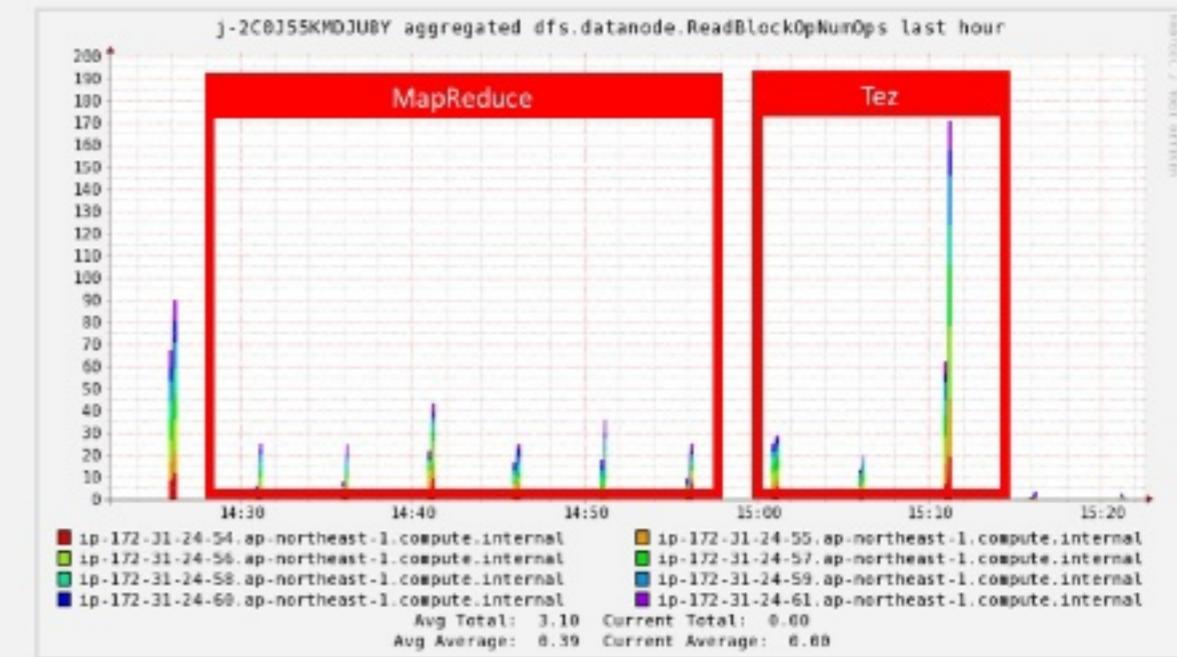
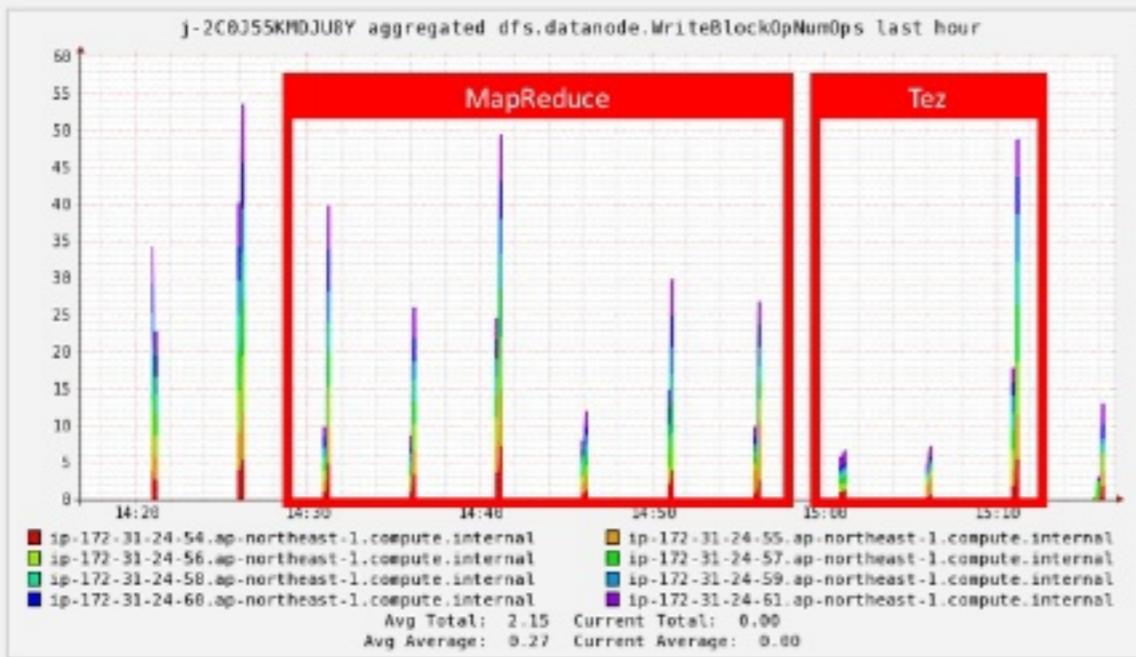
- **処理エンジンの変更**
 - 処理エンジンをTezに変更することでオンメモリで処理を行い、ディスクIOを減らす
- **Reducer数の変更**
 - 最後のファイル書き込みがボトルネックとなる処理が多かったため、起動Reducer数を増やして対応
- **MapJoinの積極的活用**
 - MapJoinをどんどん誘導していく
- **ファイルフォーマットの選択と圧縮方式の選択**
 - 処理に適したファイルフォーマット選択とディスクIO負荷を軽減するための適切な圧縮方式選択

処理エンジンの変更

- Tezに変更することで処理をオンメモリに切り替え
 - MRは多少乱暴に書いても動き切ってくれる安心感はあったけど…



処理エンジンの変更



処理エンジンの変更

- オンメモリ処理への切り替えに伴いコアノードのインスタンスタイプ[†]を変更
 - d2.4xlarge -> r3.8xlarge
- 同時ジョブ数の増加で最後のHDFSへの書き出しの遅さが目立つようにな
● 起動 Reducer数を増やすことで対応(愚直に計測…)

Reducer数の変更/コンテナサイズの調整

- 以下の可変要素を調整しながらベターなパラメータをひたすら探る
 - コンテナサイズ * 起動Reducer数 * 同時投入ジョブ数

No	Container size [MB]	java.opts	bytes.per.red	reducer.para	auto.conv.join	API接続対象	バラ	Application ID	実行時間	完了時間	処理時間	コンテナ数	Reducer数	備考	
1	4096	3200	64000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461633462805_0071	Tue Apr 26 14:37:54	Tue Apr 26 14:53:14	0:15:20	21	26	
2	4096	3200	16000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461633462805_0072	Tue Apr 26 15:04:33	Tue Apr 26 15:19:15	0:14:42	21	100	
3	4096		32000000	TRUE	TRUE	商品マスター	1	FIFO				#VALUE!			
4	2048	1600	64000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461652572652_0001	Tue Apr 26 15:45:54	Tue Apr 26 16:02:53	0:16:59	48	36	
5	2048	1600	16000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461652572652_0002	Tue Apr 26 16:05:43	Tue Apr 26 16:21:42	0:15:59	68	~90	
6	2048	2000	16000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461652572652_0003	Tue Apr 26 16:40:22	実行失敗	#VALUE!	31		
7	2048	2400	16000000	FALSE	TRUE	商品マスター	1	FIFO	application_1461652572652_0004	Tue Apr 26 17:24:27	Tue Apr 26 17:40:37	0:16:10			
8	3072	2400	16000000	FALSE	TRUE	商品マスター	1	FIFO	application_1461670514327_0001	Tue Apr 26 20:35:39	Tue Apr 26 20:50:51	0:15:12			
9	8192	6400	16000000	FALSE	TRUE	商品マスター	1	FIFO	application_1461671786132_0001	Tue Apr 26 20:56:45	Tue Apr 26 21:14:43	0:17:58			
10	4096	3200	64000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461719632379_0043	Wed Apr 27 12:45:25	Wed Apr 27 13:00:47	0:15:22			再現性確認
11	4096	3200	128000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461719632379_0044	Wed Apr 27 13:08:13	Wed Apr 27 13:25:55	0:17:42			
12	4096	3200	256000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461719632379_0045	Wed Apr 27 13:31:41	Wed Apr 27 13:50:00	0:18:19			
13	4096	2400	200000000	TRUE	TRUE	商品マスター	1	FIFO				#VALUE!			out of memory エラー発生
14	4096	3200	160000000	TRUE	TRUE	商品マスター	1	FIFO				#VALUE!			out of memory エラー発生
15	4096	3200	320000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461719632379_0046	Wed Apr 27 15:23:04	Wed Apr 27 15:38:22	0:15:18			
16	4096	1600	160000000	TRUE	TRUE	商品マスター	1	FIFO	application_1461719632379_0049	Wed Apr 27 15:40:43	Wed Apr 27 15:56:00	0:15:17			
17	4096	3200	64000000	TRUE	TRUE	発注商品	1					#VALUE!			out of memory エラー発生
18	4096	3200	64000000	TRUE	FALSE	発注商品	1	FIFO	application_1461742142552_0030	Wed Apr 27 17:49:21	Wed Apr 27 18:29:05	0:39:44			
19	4096	3200	128000000	TRUE	FALSE	発注商品	1	FIFO	application_1461742142552_0031	Wed Apr 27 18:39:03	Wed Apr 27 19:31:25	0:52:22			
20	4096	3200	32000000	TRUE	FALSE	発注商品	1	FIFO	application_1461742142552_0032	Wed Apr 27 19:39:48	Wed Apr 27 20:11:34	0:31:46			
21	4096	3200	64000000	TRUE	FALSE	発注商品	18	FIFO	application_1461811742871_0265	Thu Apr 28 14:32:56	Thu Apr 28 16:39:38	2:06:42			out of memory エラー発生
22	4096	3200	64000000	TRUE	FALSE	発注商品	18	FIFO	application_1461830090113_0001	Thu Apr 28 16:55:58	Thu Apr 28 18:48:00	1:52:02			
23	4096	3200	128000000	TRUE	FALSE	発注商品	18	FAIR	application_1461830090113_0019	Thu Apr 28 19:23:44	Thu Apr 28 21:30:52	2:07:08			
24	4096	3200	160000000	TRUE	FALSE	発注商品						#VALUE!			

Reducer数の変更/コンテナサイズの調整

- 調整パラメーター一覧

```
-- Reducer関連
```

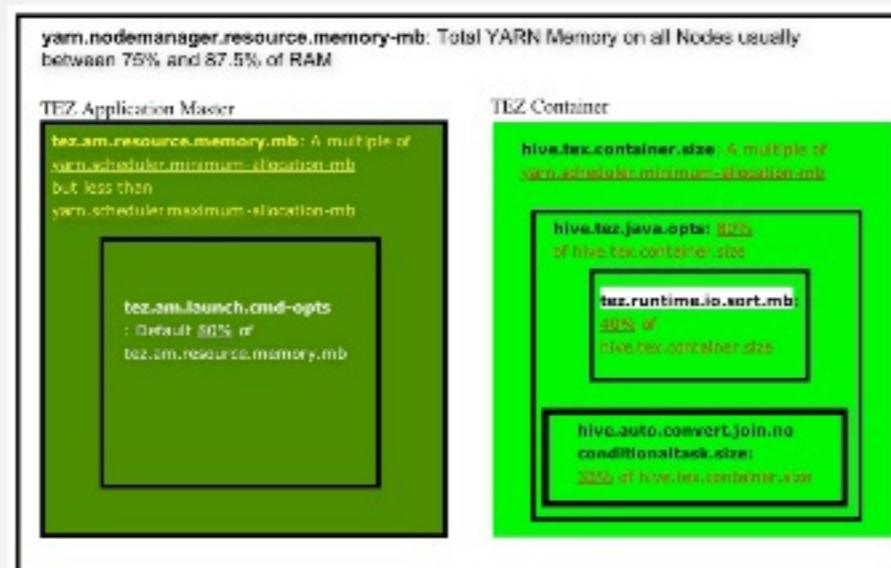
```
SET hive.tez.auto.reducer.parallelism=true;  
SET hive.exec.reducers.bytes.per.reducer=64000000;
```

```
-- YARN Container関連
```

```
SET hive.tez.container.size=4096;  
SET hive.tez.java.opts=-Xmx3200m;  
SET hive.tez.cpu.vcores=1;  
SET hive.prewarm.enabled=true;  
SET hive.prewarm.numcontainers=30;
```

(参考)コンテナにおけるメモリ利用内訳

- コンテナのメモリ利用内訳はベストプラクティスに従って設定
 - <https://community.hortonworks.com/articles/14309/demystify-tez-tuning-step-by-step.html>
- ヒープサイズはやや大きめにとっている



MapJoinへの積極的誘導

- 最初は各クエリの各ワークをレビューしながらヒント句で固定
 - 発狂 ハヤヒヤヒヤ(°▽°ミ°▽°)ヒヤヒヤヒヤ
 - そもそもクエリ内のロジックやら処理データ量やら変わったらどうするのこれ…
- もうAutoにまかせちゃう
 - MapJoinだとまずいものだけをMap Joinさせない
 - がむしゃらにMapJoinさせようとするとHashテーブルがメモリに乗り切らずOOM

-- Map Join関連

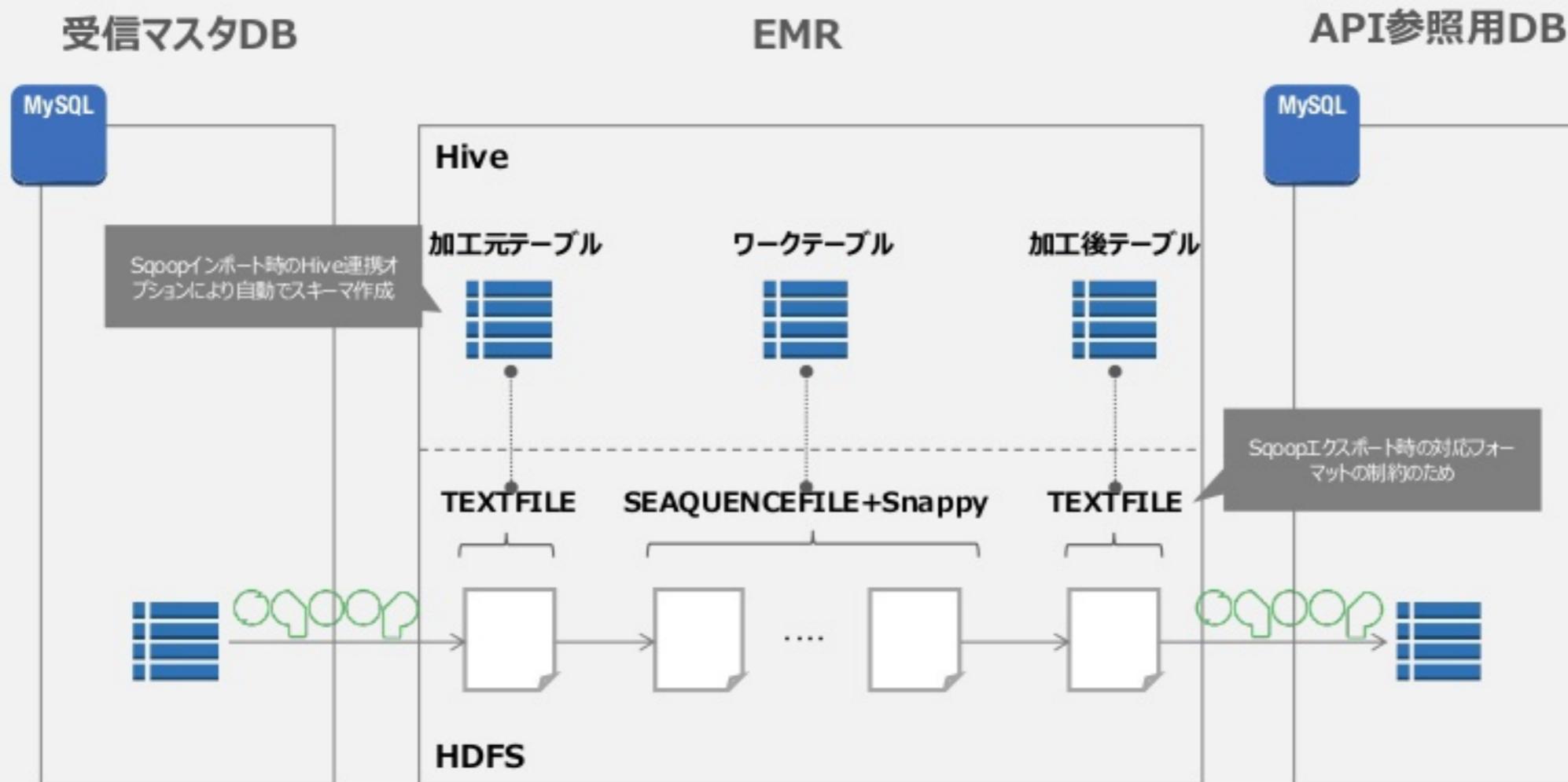
```
SET hive.auto.convert.join=true;  
SET hive.auto.convert.join.noconditionaltask.size=1300000000;
```

ファイルフォーマットの選択/圧縮方式の選択

- (中間テーブルの)ファイルフォーマット選択にあたっての要件としては以下
 - どうせ終わったら消すだけのワークなので早く終わればそれでよし！
 - スキーマ情報はクエリの中でDDL発行していたのでファイルフォーマットでカバーする必要なし
- カラムナ型のファイルフォーマットも試したもののは処理特性もありぱっとせず
 - 非正規化に近い処理をひたすら繰り返すため、カラムナフォーマットの利点をいかしきれず

```
-- ファイルフォーマット関連&圧縮関連
SET hive.default.fileformat=sequencefile;
SET mapred.output.compression.type=BLOCK;
SET hive.exec.orc.default.compress=SNAPPY;
SET hive.exec.compress.intermediate=true;
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

ファイルフォーマットの選択/圧縮方式の選択



(参考)ファイルフォーマットの選択/圧縮方式の選択

ファイルフォーマット	圧縮アルゴリズム	圧縮タイプ	処理時間
TEXT	SNAPPY	-	24:41:00
SEQUENCE	SNAPPY	BLOCK	24:10:00
	SNAPPY	RECORD	27:22:00
ORC	SNAPPY	ファイルレベル	27:17:00
		ビルトイン	27:00:00

ここまでで単位のマスタ作成は目標時間をきる

まずは1/80マスタを全店舗分

90分 → 25分



80マスタを全店舗分

∞ → 90分

次は本番同等の多重度でやってみて

主にYARNにおけるチューニング

まずは1/80マスタを全店舗分

90分 → 25分



80マスタを全店舗分

∞ → 90分

やっぱり本番負荷は並じやなかつた



ちまたのベストプラクティスがはまらない問題

* ググって出てくるチューニング例は当てはまらないことが多かった orz



思っていた以上にジョブ投入多密度が高かった問題

* 実環境では当初想定よりもかなりの処理が多重で実行される結果に



RDSのご機嫌問題

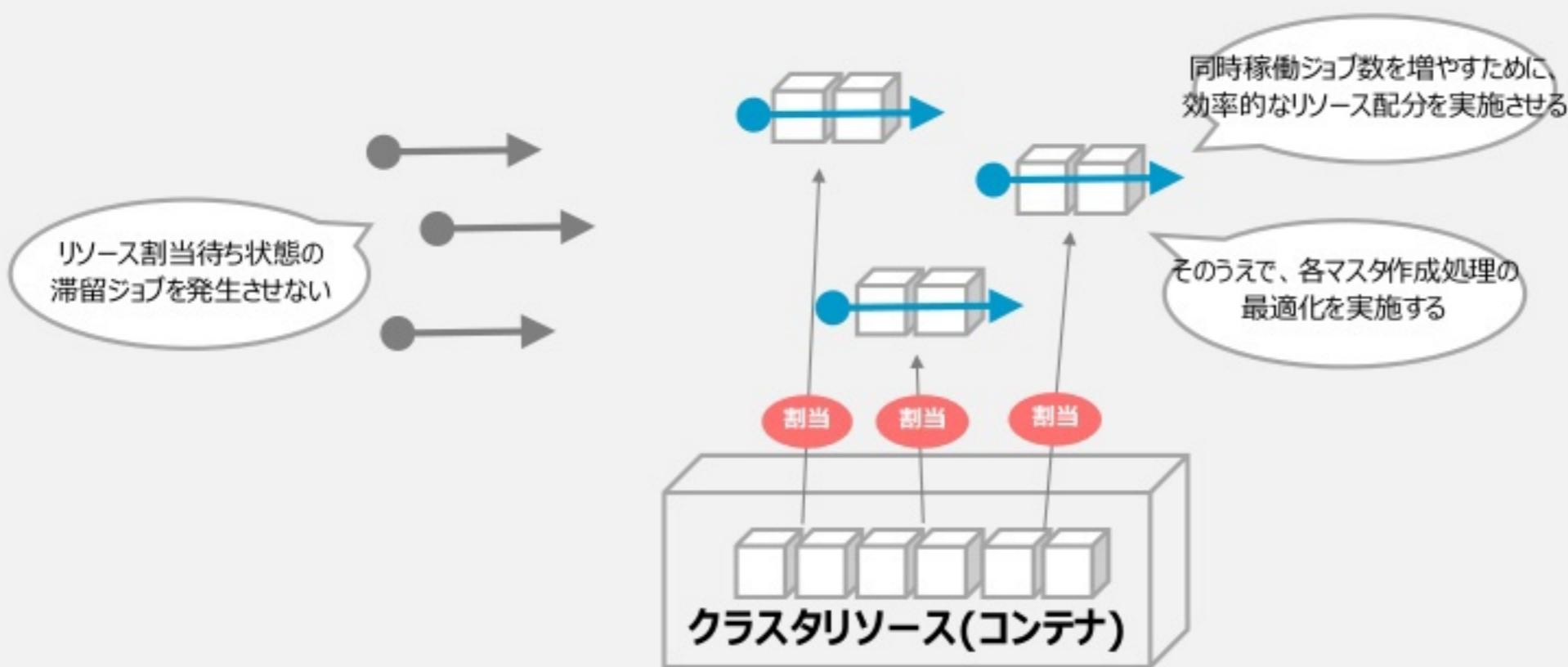
* スループットが思った以上に出ない&EMRからボトルネックが移動してきた

本当に大変だったのはリソースコントロール

- なだれ込むジョブにそもそもクラスタが耐えられず orz
 - ジョブがペンディングの嵐
 - カーネルのプロセス数上限に達してそもそもジョブ動かない

リソースコントロールのための対応

- コンテナ配布の最適化
- スケジューラの調整



コンテナ配布の最適化

- 利用できるコンテナ数をとにかく増やす
 - YARNアプリケーションごとにコンテナ数を調整(節約)
 - 前述したMapJoinなども考慮して、全体の性能が落ちずかつ複数処理がまわるようなコンテナサイズをさぐる
- Sqoopによるインポート/エクスポート処理は起動Map数が指定できるため、起動に必要なサイズにまでぎりぎり減らす

生々しい軌跡

- コンテナのメモリサイズを調整しながらギリギリのラインをさぐる
 - 結果的に512MB/コンテナあれば同じ処理時間を保ちながら処理ができた
 - これより小さくなると、そもそもOOMで動かなくなる

No.	実行順序	実行タスク	table_size	DB (table)	map#	container(MB)	ava.opts (MB)	システムカラム利用	並列度	dm_resource	a_impreducsum.commandd-opt	実行時間	実行時間(実効アラム)	並列度	並列度/table	備考	
1	なし	実行	10GB	9	2	4096	2918	否	4L	9L	7295	5837	Tue May 24 10:58:01	Tue May 24 11:22:34	0:24:34	0:02:44	
2	なし	実行	10GB	9	6	4096	2918	否	4L	9L	7295	5837	Tue May 24 11:24:21	Tue May 24 11:49:24	0:24:55	0:02:46	
3	なし	実行	10GB	9	2	32288	9830	否	4L	9L	7295	5837	Tue May 24 12:06:21	Tue May 24 12:31:04	0:24:40	0:02:44	
4	なし	実行	10GB	9	2	512	410	否	4L	9L	7295	5837	Tue May 24 12:40:51	Tue May 24 12:56:46	0:24:41	0:02:45	container size 512MB での処理速度と実効アラムの確認 => OK
5	なし	実行	10GB	9	2	512	410	是	4L	9L	7295	5837	Tue May 24 13:50:41	Tue May 24 14:12:44	0:22:03	0:02:37	
6	なし	実行	10GB	9	2	512	410	是	4L	9L	7295	5837	Tue May 24 14:15:01	中止	--	--	1時間以上かかったため中断した。
7	なし	実行	10GB	9	2	512	410	是	4L	9L	7295	5837	Tue May 24 14:29:51	中止	--	--	1時間以上かかったため中断した。
8	なし	実行	10GB	9	2	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 16:28:01	Tue May 24 16:40:36	0:12:35	0:01:23	
9	なし	実行	10GB	9	2	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 16:45:21	Tue May 24 17:16:12	0:16:50	0:01:52	
10	なし	実行	10GB	9	2	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 17:23:56	Tue May 24 17:44:29	0:12:31	0:01:34	
11	なし	実行	10GB	9	2	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 17:41:51	Tue May 24 17:52:23	0:10:27	0:01:45	
12	なし	実行	10GB	9	1	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 17:54:31	Tue May 24 18:07:14	0:12:35	0:01:24	
13	なし	実行	10GB	9	2	512	410	是	4L	order PK(全T)	7295	5837	Tue May 24 18:10:20	中止	--	--	10分以上かかったため中断した。手動
14	なし	実行	10GB	9	3	512	410	是	4L	order PK(TEMPt0w)	7295	5837	Tue May 24 19:11:01	中止	--	--	20分以上かかってしまったため、画面を一手で見て監視する必要があると判断
15	なし	実行	10GB	9	3	512	410	是	4L	order PK(全T)	5837	5837	Tue May 24 19:20:11	Tue May 24 19:33:01	0:13:50	0:00:57	実行時間で止まらなかったときのアンダーフォード
16	なし	実行	10GB	18	3	512	410	是	4kb	order PK(全T)	7295	5837	Tue May 24 22:11:51	Tue May 24 22:43:56	0:32:05	0:01:47	通常なら7.7KBを確認
17	なし	実行	10GB	36	1	512	410	是	4L	order PK(全T)	7295	5837	Wed May 25 11:08:31	Wed May 25 12:27:21	1:18:46	0:02:11	appendending 33draining 時間は別シート
18	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	7295	5837	Wed May 25 12:35:35	Wed May 25 13:07:21	0:31:48	0:01:48	
19	なし	実行	10GB	36	3	512	410	是	4kb	order PK(全T)	7295	5837	Wed May 25 13:24:51	Wed May 25 13:56:51	0:31:59	0:01:47	通常8秒ロード
20	なし	実行	10GB	36	1	512	410	是	4L	order PK(全T)	7295	5837	Wed May 25 14:05:01	Wed May 25 14:17:41	0:12:29	0:01:23	データ一括読み込みを確認 => OK
21	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	7295	5837	Wed May 25 14:23:11	Wed May 25 14:59:03	0:36:31	0:01:58	
22	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	3648	2918	Wed May 25 15:16:06	中止	--	--	負荷過度になるか確認 => vcoreに制限をかけて30Lを走らせて中止
23	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	3648	2918	Wed May 25 16:35:44	Wed May 25 17:58:01	1:20:21	0:02:14	正常は出来ないと判断し中止
24	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	3648	2918	Wed May 25 18:12:21	中止	--	--	20分以内確認するため中止 36GBで爆発で止った場合と同じであると予想
25	なし	実行	10GB	36	3	512	410	是	4L	order PK(全T)	3648	2918	Wed May 25 18:50:11	中止	--	--	20分以内確認するため中止 36GBで爆発で止った場合と同じであると予想
26	なし	実行	10GB	36	27	512	410	是	4L	order PK(全T)	3648	2918	Wed May 25 19:16:21	Wed May 25 20:11:24	0:55:04	0:02:03	
AZ 対応確認																	
27	あり	実行	10GB	9	3	4096	3277	是	4L	order PK(全T)	3648	2918	Thu May 26 12:07:21	Thu May 26 12:21:23	0:13:56	0:01:33	AZ 対応用他の性能確認、containerサイズによる違い
28	あり	実行	10GB	18	1	512	410	是	4L	order PK(全T)	3648	2918	Thu May 26 12:31:01	Thu May 26 13:15:54	0:44:49	0:02:29	
29	あり	実行	10GB	9	1	512	410	是	4L	order PK(全T)	7295	5837	Thu May 26 13:25:11	Thu May 26 13:41:06	0:15:52	0:01:46	
30	あり	実行	10GB	9	3	512	410	是	4L	order PK(全T)	3648	2918	Thu May 26 13:57:31	Thu May 26 14:14:03	0:16:29	0:01:50	
31	あり	実行	10GB	9	3	512	410	是	4L	order PK(全T)	3648	2918	Thu May 26 14:17:21	Thu May 26 14:45:03	0:27:37	0:03:04	WRITE IOが走っていないPELがない、再実行 => WRITE IOが走るPEL
32	あり	実行	10GB	9	3	512	410	是	4L	order PK(全T)	3648	2918	Thu May 26 14:53:31	Thu May 26 15:16:11	0:22:39	0:02:31	256GBストレージで20%削減して再実行 =>
33	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	3648	2918	Thu May 26 15:25:41	Thu May 26 15:50:56	0:25:07	0:02:47	
34	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	3648	2918	Thu May 26 16:19:51	Thu May 26 16:49:51	0:30:01	0:03:26	
35	あり	実行	10GB	9	3	512	410	是	4L	order PK(全T)	3648	2918	Thu May 26 17:05:41	Thu May 26 17:19:44	0:14:04	0:01:34	
36	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	3648	2918	Thu May 26 17:24:11	Thu May 26 17:39:16	0:15:01	0:01:40	
37	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	3648	2918	Thu May 26 17:47:31	Thu May 26 18:17:29	0:29:58	0:03:26	
38	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	3648	2918	Thu May 26 18:24:41	中止	--	--	#VAL(81) #VAL(81) 89.48% = 見込A 22分 100%
39	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	1024	819	Thu May 26 18:36:31	Thu May 26 18:52:34	0:15:58	0:01:48	
40	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	512	410	Thu May 26 18:54:31	Thu May 26 19:13:05	0:16:26	0:01:50	
41	あり	実行	10GB	9	3	512	410	是	4L	sort (tempa_id)	512	410	Thu May 26 19:18:51	Thu May 26 19:34:24	0:15:26	0:01:43	

スケジューラの調整

- スケジューラをFair Schedulerに変更
 - とにかく終わったものからどんどんRDSに流してしまったかったため、ペンドイングをなくしたかった
 - とはいえ、ジョブに応じてコンテナの配分は調整したかったので、そこはキュー設計で頑張る

キュー設定によるリソース割り当て優先度の調整

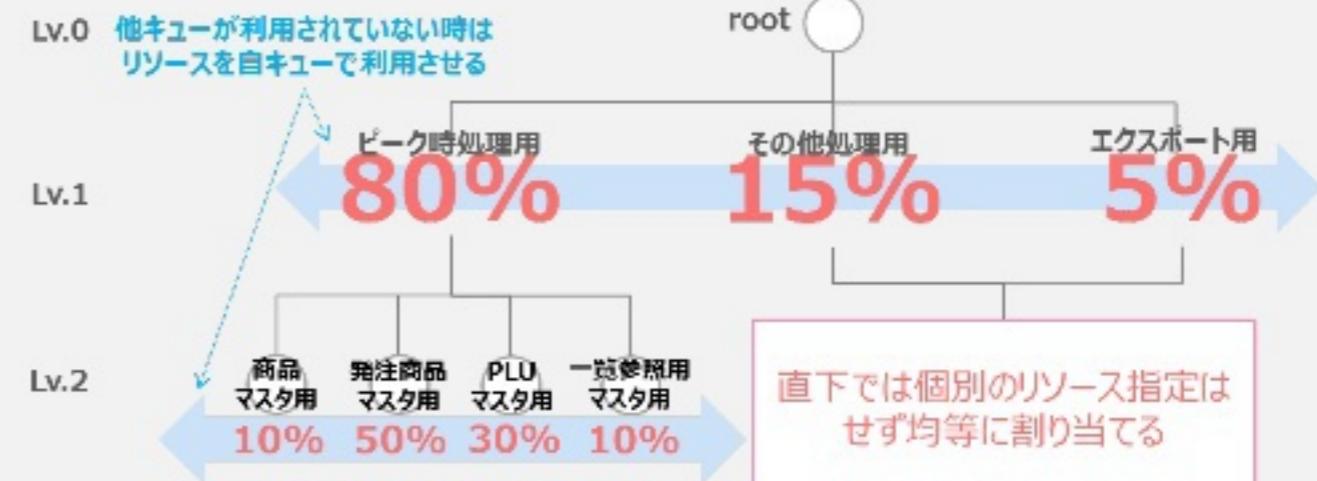
前 提

- YARNスケジューラとして「Fair Scheduler」を利用する。
- Fair Schedulerにより、マスタ作成処理に重みづけを行い、クラスタのリソースを綿密にコントロールすることを目的とする。
※Capacity Schedulerはキュー内ではリソースを均等分配できずペンディング状態の処理が発生してしまう可能性があるため。

キュー設定

root

- peak(業務優先度高&リソース要求高)
 - shohin
 - hatchu_shohin
 - ichiran_sansho_yo_shohin
 - plu
- others(業務優先度低&リソース要求低)
 - sqoop(Sqoop処理用)
 - import
 - export



キュー設定によるリソース割り当て優先度の調整

業務優先度高 & リソース要求大

```
<?xml version="1.0"?>
<allocations>
  <!--ピーク時処理用キュー-->
  <queue name="peak">
    <weight>80.0</weight> さらにその中で配分
    <schedulingPolicy>fair</schedulingPolicy>
  <!--商品マスター作成用キュー-->
  <queue name="shohin">
    <weight>10.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
  </queue>
  <!--発注商品作成用キュー-->
  <queue name="hatchu_shohin">
    <weight>50.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
  </queue>
  <!--PLUマスター作成用キュー-->
  <queue name="plu">
    <weight>30.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
  </queue>
  <!--一覧参照用商品マスター作成キュー-->
  <queue name="ichiran_sansho_yo_shohin">
    <weight>10.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
  </queue>
</queue>
```

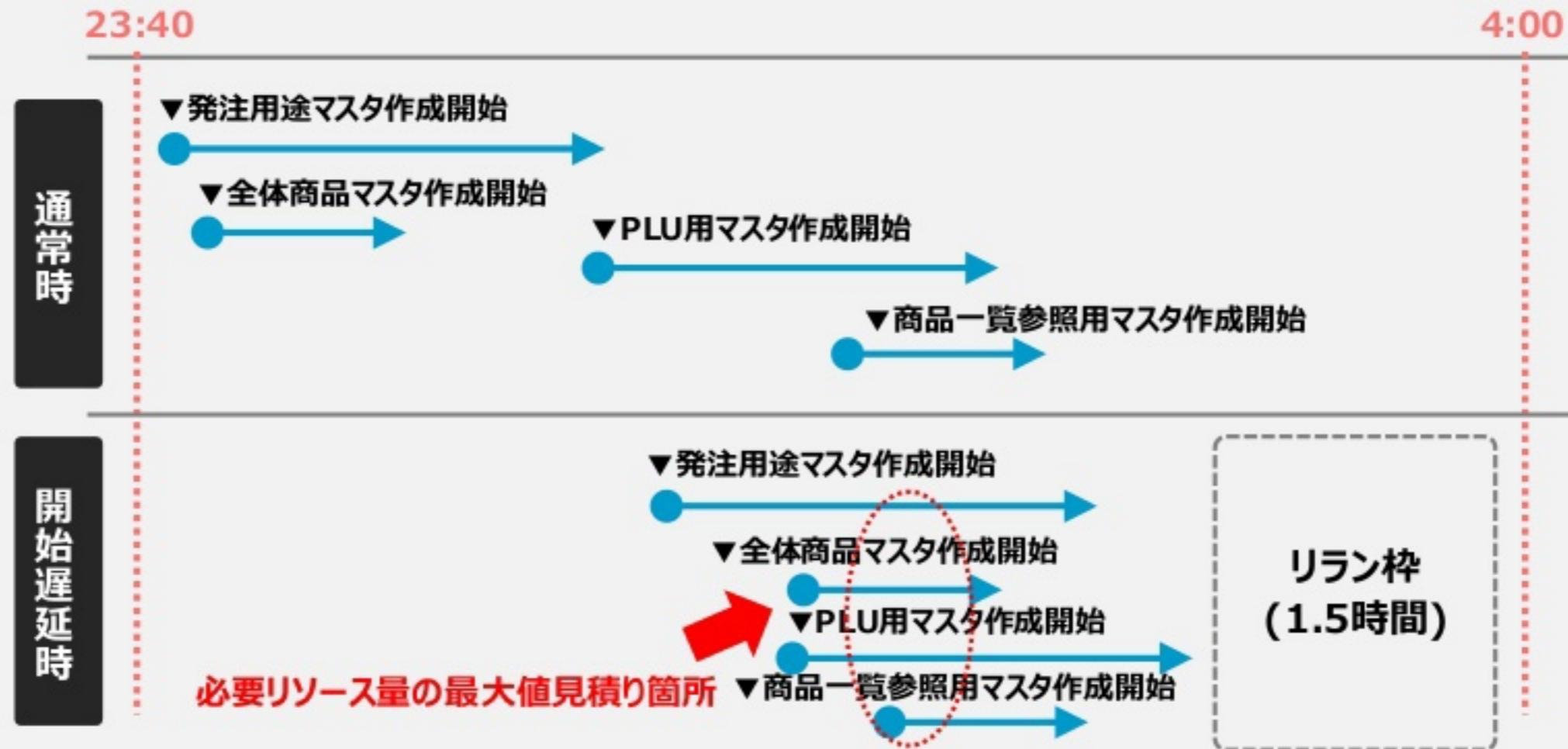
業務優先度中低 & リソース要求小

```
<!--店舗開発&他マスター作成用キュー-->
<queue name="others">
  <weight>10.0</weight>
  <schedulingPolicy>fair</schedulingPolicy>
```

Sqoop処理用途(必要最低限のみ)

```
<!--Sqoop処理用キュー-->
<queue name="sqoop">
  <weight>10.0</weight>
  <schedulingPolicy>fair</schedulingPolicy>
<!--Sqoop Import処理キュー-->
<queue name="import">
  <schedulingPolicy>fair</schedulingPolicy>
</queue>
<!--Sqoop Export処理キュー-->
<queue name="export">
  <schedulingPolicy>fair</schedulingPolicy>
```

リソース配分の決定までの道のり



そして90分にまでようやっと短縮!!

まずは1/80マスタを全店舗分

90分 → 25分



80マスタを全店舗分

∞ → 90分

おまけ

やっぱり本番負荷は並じやなかつた



ちまたのベストプラクティスがはまらない問題

* ググって出てくるチューニング例は当てはまらないことが多かったorz



思っていた以上にジョブ“投入多密度が”高かった問題

* 実環境では当初想定よりもかなりの処理が多重で実行される結果に



RDSのご機嫌問題

* スループットが思った以上に出ない&EMRからボトルネックが移動してきた

EMRと同じぐらい大変だったRDSチューニング

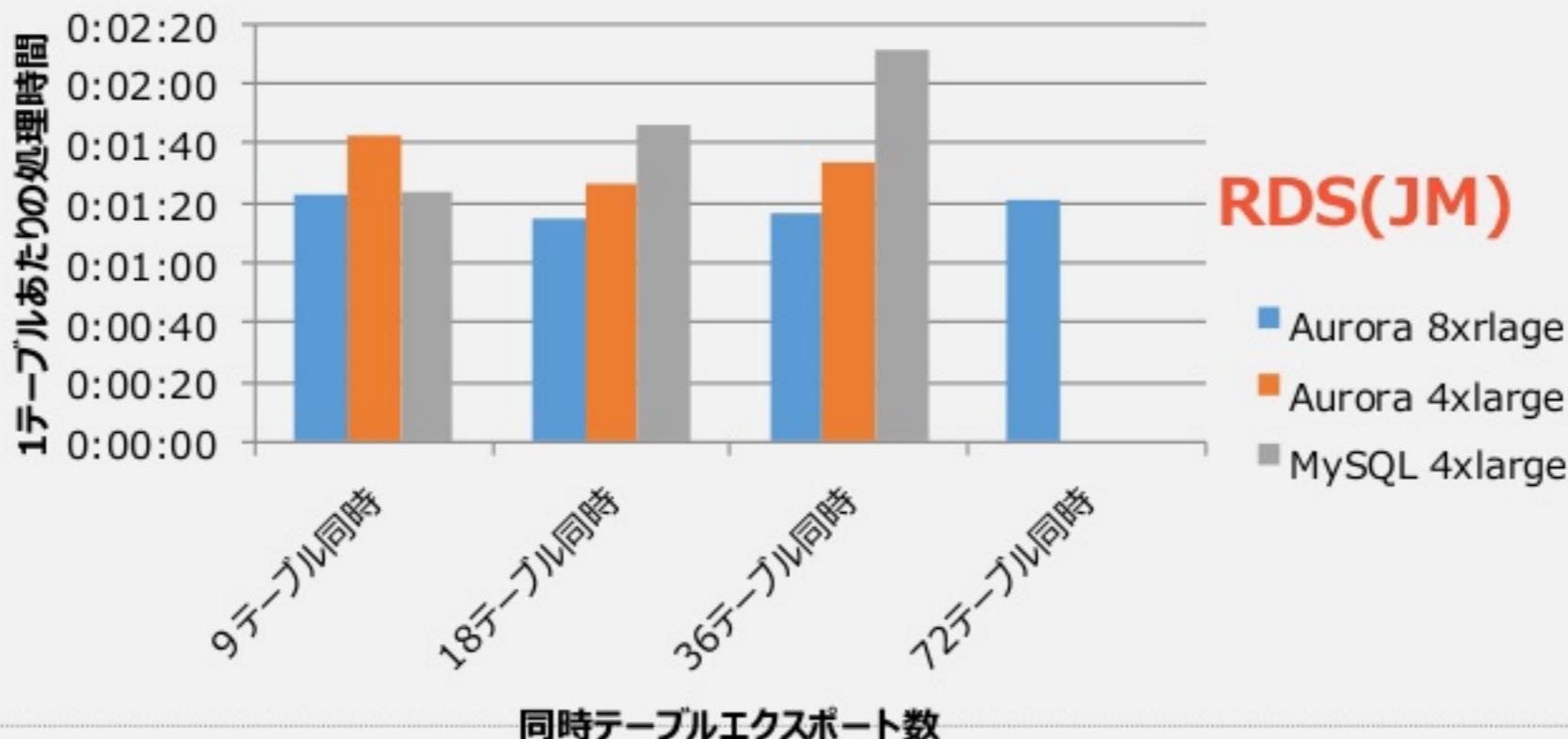
- 最初は最初はほんとにSqoopによるエクスポートが終わらなかつた…
 - Sqoopエクスポートの多重度があがるとRDS(MySQL)のIOがつまる…
 - しまいにはセッションきられる
- 以下を中心にチューニングし、なんとか流し切れるまでに
 - エクスポート対象のテーブルを事前にPKでソート
 - なるべくディスクへの書き込みによるIOを遅延させる
 - innodb_buffer_pool_size
 - innodb_max_dirty_pages_pct
 - 書き込み周りのスレッド数を微調整
 - innodb_write_io_threads
 - innodb_thread_concurrency

試しにAurora(MySQL)に変更したみたら

- あんなに頑張った結果がノンチューニングで抜かれる(しかも台数も半分で…)
- 多重度が上がるほど性能が安定&エクスポートするテーブルサイズによらず安定
- マルチAZにしても非同期だから性能劣化なし
- 大量データのエクスポート先としても結構有能

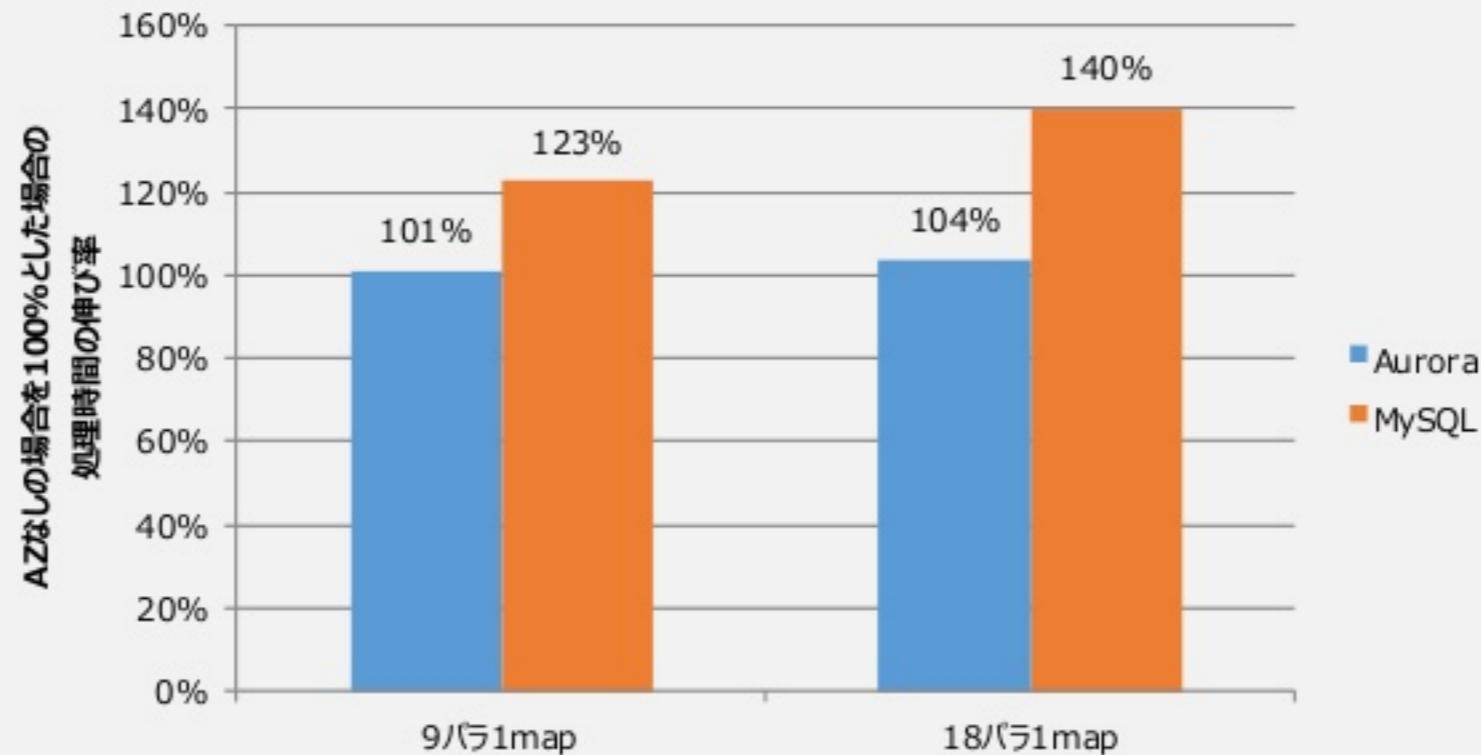
(参考)エクスポート並列数による処理時間結果

No.	RDS	AZ構成	バラード	map数	処理時間	処理時間/table	参考処理時間 (MySQL)	参考処理時間/ table(MySQL)
7	r3.4xlarge	なし	9	1	0:15:26	0:01:43	0:12:35	0:01:24
8	r3.4xlarge	なし	18	1	0:25:50	0:01:26	0:32:00	0:01:47
13	r3.4xlarge	あり	36	1	0:56:18	0:01:34	1:18:46	0:02:11
16	r3.8xlarge	なし	9	1	0:12:30	0:01:23		
17	r3.8xlarge	なし	18	1	0:22:22	0:01:15		
18	r3.8xlarge	なし	36	1	0:46:10	0:01:17		
21	r3.8xlarge	なし	72	1	1:37:21	0:01:21		



(参考)Multi AZによる処理時間

No.	RDS	AZ構成	パラ	map数	処理時間	処理時間/table	参考処理時間 (MySQL)	参考処理時間/ table
7	r3.4xlarge	なし	9	1	0:15:26	0:01:43	0:12:35	0:01:24
8	r3.4xlarge	なし	18	1	0:25:50	0:01:26	0:32:00	0:01:47
11	r3.4xlarge	あり	9	1	0:15:36	0:01:44	0:15:26	0:01:43
12	r3.4xlarge	あり	18	1	0:26:45	0:01:29	0:44:49	0:02:29



まとめ

まとめ：教訓

- とにかくワークは小さく保つ!!(迫真)
 - プランの可読性が全然違う
- ちまたのチューニング情報はとつかかりとして有効
 - ワークロードが異なれば傾向も変わる
- リソース配分部分は結構盲点
 - 情報も少なく一番業務要件できる部分なのでちゃんと特性を知った上で設計する
- Auroraって万能ですね！
 - 重たいデータのオフロードもいけるじゃん



ありがとうございました!!