\Orchestrating a brighter world NEC

TPC-DSから学ぶPostgreSQLの弱点と今後の展望

NEC ビジネスクリエイション本部
The PG-Strom Project
KaiGai Kohei <kaigai@ak.jp.nec.com>

自己紹介

- 名前： 海外 浩平
- 会社： NEC
- 仕事：
  - PG-Stromプロジェクト リーダー
  - PostgreSQL及び他のOSSプロジェクトへのコントリビューション
  - PG-Stromを軸とした事業立上げ

PG-Stromプロジェクト

- ミッション：

  ヘテロジニアス計算や不揮発メモリなど、新たな半導体技術の進化の成果を全てのユーザの元へ届ける。
- 2012年に海外個人の開発プロジェクトとしてスタート。
  現在はNECがファンドしている。
- 完全なオープンソースプロジェクト（GPL v2）

本日のテーマ

① OLAP系ワークロードを代表する（とされる）
TPC-DSクエリをPostgreSQL 9.5βで実行

② そこから見えてきたPostgreSQLの弱点とは！？

③ これらの課題に対し、開発者コミュニティでは
どのような取り組みが行われているか？
（主に並列分散処理の取り組みについて）

アジェンダ

1. **TPC-DSベンチマークとは？**
2. ベンチマーク結果と分析
3. 改善アプローチ
4. その先の未来

TPC-DSベンチマークとは (1/3)

TPCとは

- TPC: Transaction Processing Performance Council
- ベンダー中立なNPO組織。1988年設立。カリフォルニア州本拠。
- トランザクション処理＆DBベンチマークの定義を目的とする。

定義されているベンチマーク

TPC-C	On-line transaction processing (1992~)
TPC-H	Ad-hoc decision support system (1999~)
TPC-E	Complex on-line transaction processing (2006~)
TPC-DS	Complex decisions support system (2011~)
TPC-DI	Data integration (2013~)

DBの集計性能
情報系システムを想定した
ワークロードを定義

DBの更新性能
業務系システムを想定した
ワークロードを定義

TPC-DSベンチマークとは (2/3)

What is TPC Benchmark™DS

- 意思決定支援ベンチマーク (Decision Support)
- 大量データに対する、実世界のビジネスクエスチョン
  - Ad-hoc, reporting, data mining, ... 等の領域を想定
- プロセッサ (CPU/GPU) とI/Oに対する高負荷を想定
- 様々な種類のクエリを99種類 (103個)定義

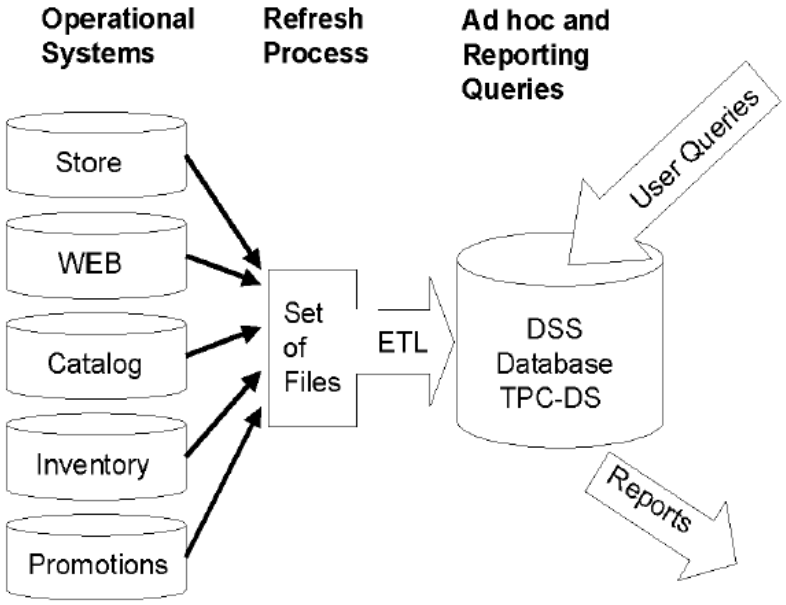


Figure 1-1: TPC-DS benchmark components

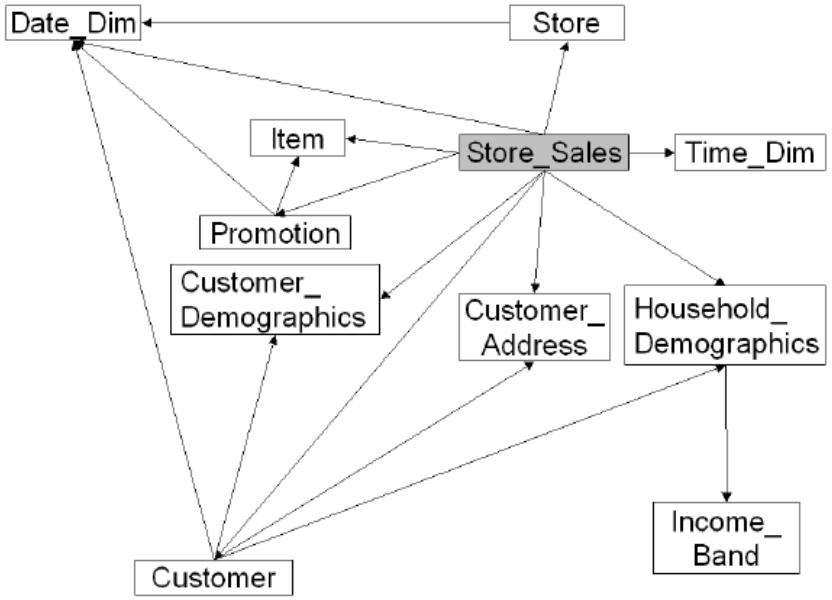
TPC-DSベンチマークとは (3/3)

TPC-DS data/business model

- 全国規模で展開し各地に店舗を持つ小売・流通業。
  店舗・Web・カタログの各販売チャネルを持つ事を想定。
- 業務プロセス
  - 顧客の購買行動・返品履歴をトラッキング
  - 顧客属性によるダイナミックなWebページの生成（CRM）
  - 在庫商品の管理

データ構造

- 販売履歴を中心とする
  スタースキーマ構造



TPC-DSクエリの例 – Query01

```
with customer_total_return as
(select sr_customer_sk as ctr_customer_sk
       ,sr_store_sk as ctr_store_sk
       ,sum(SR_FEE) as ctr_total_return
 from store_returns
     ,date_dim
 where sr_returned_date_sk = d_date_sk
   and d_year = 2000
 group by sr_customer_sk
         ,sr_store_sk)
 select c_customer_id
   from customer_total_return ctr1
       ,store
       ,customer
  where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
                                   from customer_total_return ctr2
                                  where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
    and s_store_sk = ctr1.ctr_store_sk
    and s_state = 'TN'
    and ctr1.ctr_customer_sk = c_customer_sk
  order by c_customer_id
  limit 100;
```

2000年、テネシー州の店舗において、
返品数が店舗平均の20%を越える顧客を
100件検索する

アジェンダ

1. TPC-DSベンチマークとは？
2. **ベンチマーク結果と分析**
   - **フラット化しないサブクエリ**
   - **Nested Loopの見込み違い**
   - **TPC-DSが教えてくれたこと**
3. 改善アプローチ
4. その先の未来

ベンチマーク測定条件

Software

- PostgreSQL v9.5β1
  - work_mem = 96GB
  - shared_buffers = 160GB
  - statement_timeout = 3600000 (1時間で打ち切り)
- Red Hat Enterprise Linux Server 6.6
- NVIDIA CUDA 7.0

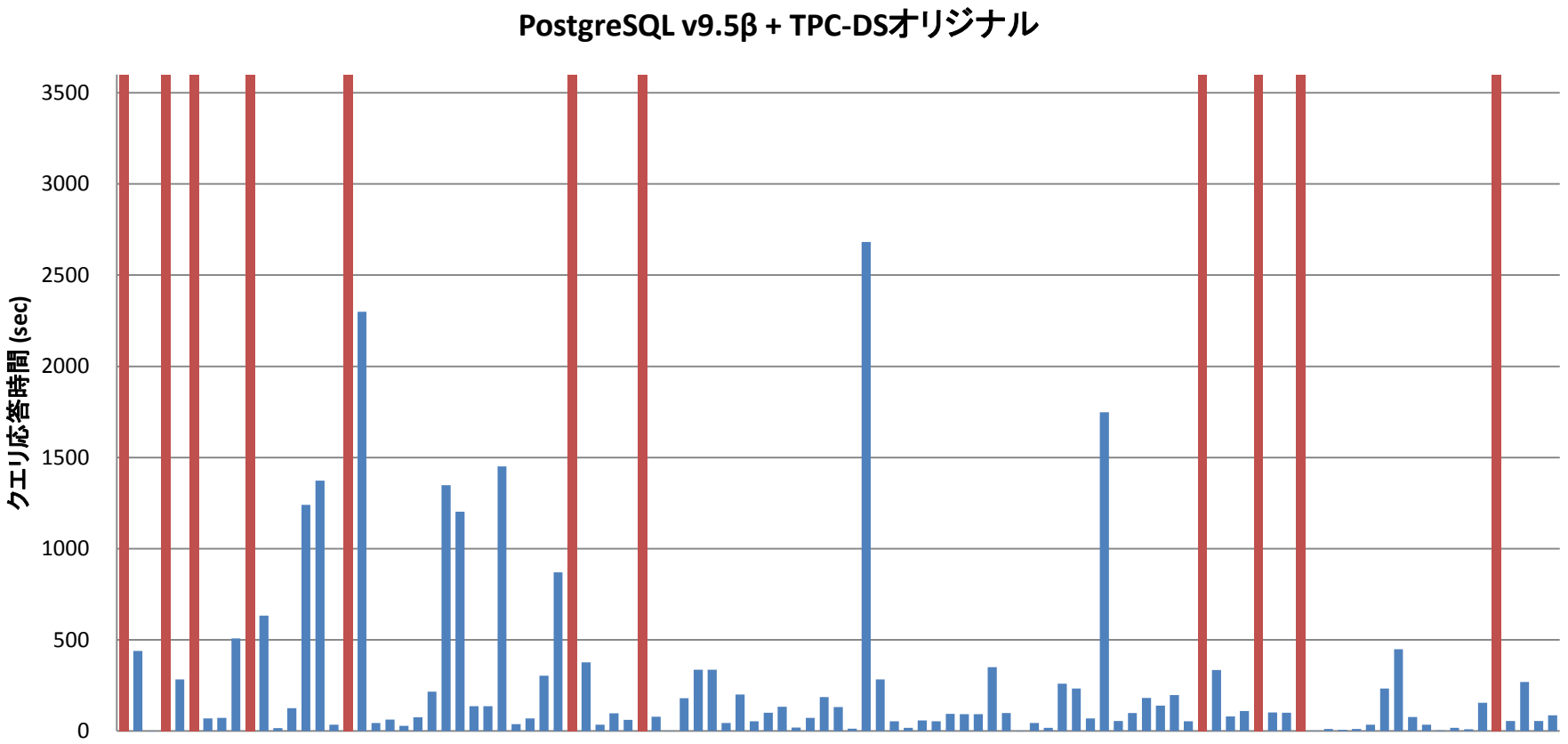
Hardware

- Dell PowerEdge T630
  - CPU: Intel Xeon E5-2670v3 (2.3GHz, 12C) x 2
  - GPU: NVIDIA Tesla K20c (706MHz, 2496C) x 1
  - RAM: 384GB (16GB RDIMM 2133MT/s) x 24
  - HDD: 300GB (SAS, 10krpm) x 8; RAID5

ベンチマーク前提

- Scaling Factor = 100
- pg_prewarm により全データを事前にRAMへロード

ベンチマーク結果① – PostgreSQL v9.5β + TPC-DSオリジナル



- 103中11本のクエリが、1時間(=3600秒)経過しても終了せず。

ベンチマーク結果の分析 (1/5) – Query01を題材に

```
with customer_total_return as
(select sr_customer_sk as ctr_customer_sk
       ,sr_store_sk as ctr_store_sk
       ,sum(SR_FEE) as ctr_total_return
 from store_returns
     ,date_dim
 where sr_returned_date_sk = d_date_sk
   and d_year =2000
 group by sr_customer_sk
         ,sr_store_sk)
 select c_customer_id
   from customer_total_return ctr1
       ,store
       ,customer
  where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
                                   from customer_total_return ctr2
                                  where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
    and s_store_sk = ctr1.ctr_store_sk
    and s_state = 'TN'
    and ctr1.ctr_customer_sk = c_customer_sk
  order by c_customer_id
  limit 100;
```

ベンチマーク結果の分析 (2/5) – Query01の実行計画

```
Limit (cost=433929567.15..433929567.40 rows=100 width=17)
 CTE customer_total_return (cost=0.00..2773.22 rows=138661 width=48)
    ...(CTE省略)...
 -> Sort (cost=432901698.79..432901711.73 rows=5174 width=17)
    Sort Key: customer.c_customer_id
    -> Nested Loop (cost=0.43..432901501.05 rows=5174 width=17)
       -> Nested Loop (cost=0.00..432881293.33 rows=5174 width=8)
          Join Filter: (ctr1.ctr_store_sk = store.s_store_sk)
          -> CTE Scan on customer_total_return ctr1
                          (cost=0.00..432850070.69 rows=46220 width=16)
             Filter: (ctr_total_return > (SubPlan 2))
             SubPlan 2
              -> Aggregate (cost=3121.61..3121.62 rows=1 width=32)
                 -> CTE Scan on customer_total_return ctr2
                                (cost=0.00..3119.87 rows=693 width=32)
                    Filter: (ctr1.ctr_store_sk = ctr_store_sk)
          -> Materialize (cost=0.00..24.25 rows=45 width=8)
             -> Seq Scan on store (cost=0.00..24.02 rows=45 width=8)
                Filter: (s_state = 'TN'::bpchar)
       -> Index Scan using customer_pkey on customer
                           (cost=0.43..3.90 rows=1 width=25)
          Index Cond: (c_customer_sk = ctr1.ctr_customer_sk)
(26 rows)
```

ベンチマーク結果の分析 (3/5) – 犯人は誰ぞ？

```
with customer_total_return as
(select sr_customer_sk as ctr_customer_sk
       ,sr_store_sk as ctr_store_sk
       ,sum(SR_FEE) as ctr_total_return
 from store_returns
     ,date_dim
 where sr_returned_date_sk = d_date_sk
   and d_year =2000
 group by sr_customer_sk
         ,sr_store_sk)
 select c_customer_id
   from customer_total_return ctr1
       ,store
       ,customer
  where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
                                   from customer_total_return ctr2
                                  where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
    and s_store_sk = ctr1.ctr_store_sk
    and s_state = 'TN'
    and ctr1.ctr_customer_sk = c_customer_sk
  order by c_customer_id
  limit 100;
```

WHERE条件句がctr1テーブルから
読み出したレコードの内容に依存
（parametalized）している。
⇒ ctr1テーブルのレコード数と同じ
回数だけサブクエリを繰り返す！

ベンチマーク結果の分析 (4/5) – 繰り返しをJoinで置き換え

```
with customer_total_return as
(...CTE省略...)
select c_customer_id
   from customer_total_return ctr1
       ,store
       ,customer
      ,(select ctr_store_sk
              ,avg(ctr_total_return)::numeric(7,2) avg_total_return
          from customer_total_return
         group by ctr_store_sk) ctr2
  where ctr1.ctr_store_sk = ctr2.ctr_store_sk
    and ctr1.ctr_total_return > avg_total_return*1.2
    and s_store_sk = ctr1.ctr_store_sk
    and s_state = 'TN'
    and ctr1.ctr_customer_sk = c_customer_sk
  order by c_customer_id
  limit 100;
```

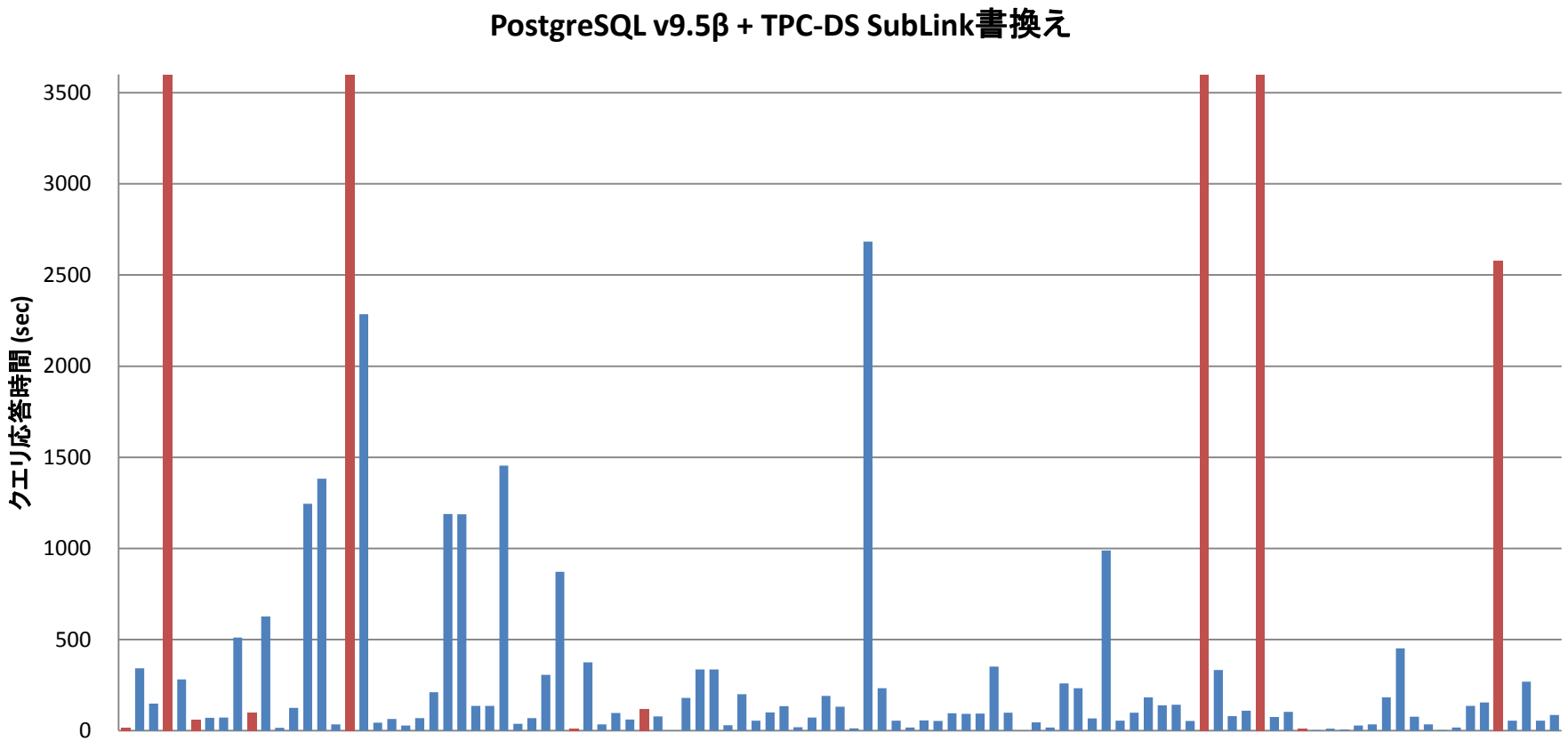
先に ctr_store_sk 毎の平均値を一回だけ計算し、次にcustomer_total_returnの結果とINNER JOINを行う。

ベンチマーク結果の分析 (5/5) – 修正クエリの実行結果

```
Limit (cost=1059793.62..1059793.87 rows=100 width=17)
   .....(省略).....
   -> Hash Join (cost=3506.15..7201.47 rows=5174 width=8)
      Hash Cond: (ctr1.ctr_store_sk = customer_total_return.ctr_store_sk)
      Join Filter: (ctr1.ctr_total_return >
                    (avg(customer_total_return.ctr_total_return)::numeric(7,2)) * 1.2)
      Rows Removed by Join Filter: 407779
      -> CTE Scan on customer_total_return ctr1 (cost=0.00..2773.22 rows=138661 width=48)
                                        (actual time=10421.995..10936.477 rows=5435529 loops=1)
      -> Hash (cost=3505.87..3505.87 rows=22 width=30)
         Buckets: 1024 Batches: 1 Memory Usage: 10kB
         -> Merge Join (cost=3504.43..3505.87 rows=22 width=30)
            Merge Cond: (store.s_store_sk = customer_total_return.ctr_store_sk)
            -> Sort (cost=25.26..25.37 rows=45 width=8) (actual time=0.165..0.168 rows=45 loops=1)
               Sort Key: store.s_store_sk
               -> Seq Scan on store (cost=0.00..24.02 rows=45 width=8)
                                   (actual time=0.012..0.140 rows
                  Filter: (s_state = 'TN'::bpchar)
                  Rows Removed by Filter: 357
            -> Sort (cost=3479.17..3479.67 rows=200 width=22)
                  (actual time=5266.754..5266.765 rows=199 loops=1)
               Sort Key: customer_total_return.ctr_store_sk
               -> HashAggregate (cost=3466.53..3469.53 rows=200 width=40)
                                (actual time=5266.589..5266.711 rows=202 loops=1)
                  Group Key: customer_total_return.ctr_store_sk
                  -> CTE Scan on customer_total_return (cost=0.00..2773.22 rows=138661 width=40)
                                                  (actual time=0.001..3833.790 rows=5435529 loops=1)
   .....(省略).....
Planning time: 10.193 ms
Execution time: 17775.038 ms
```

極端に実行回数の多かった
サブクエリ内の処理

ベンチマーク結果② – PostgreSQL v9.5β + TPC-DS修正



- クエリ01の実行時間：？？ ➔ 17.78sec
- クエリ01, 06, 10, 30, 35, 81, 95の7本でSunLink書換え

➔ 時間内に終了しないクエリ 11本 ➔ 4本

反省会：なぜサブクエリの実行が非効率になってしまったか？

- TPC-DSのケースは機械的にサブクエリ➔JOINへと書き換え可能。
- PostgreSQLもサブクエリをJOINへと書き換える機構は持っている。
  - が、書き換えられるパターンが限定的であるため。
- 某シェアNo.1商用DBなどはｺﾞﾆｮｺﾞﾆｮｺﾞﾆｮ....

```
subquery_planner(....)
 -> pull_up_sublinks(....)
     -> pull_up_sublinks_jointree_recurse(....)
         -> pull_up_sublinks_qual_recurse(....)
             -> convert_ANY_sublink_to_join(....)
                  :
             /*
              * The sub-select must not refer to any Vars of the parent
              * query. (Vars of higher levels should be okay, though.)
              */
             if (contain_vars_of_level((Node *) subselect, 1))
                 return NULL;
                  :
```

サブクエリ外の値を
参照していたら諦める

更なる分析 (1/4) – Query16を題材に

```
select count(distinct cs_order_number) as "order count"
      ,sum(cs_ext_ship_cost) as "total shipping cost"
      ,sum(cs_net_profit) as "total net profit"
  from catalog_sales cs1
      ,date_dim
      ,customer_address
      ,call_center
 where d_date between '1999-2-01' and
                (cast('1999-2-01' as date) + '60 days'::interval)
   and cs1.cs_ship_date_sk = d_date_sk
   and cs1.cs_ship_addr_sk = ca_address_sk
   and ca_state = 'IL'
   and cs1.cs_call_center_sk = cc_call_center_sk
   and cc_county in ('Williamson County','Williamson County',
                     'Williamson County','Williamson County', 'Williamson County')
   and exists (select *
               from catalog_sales cs2
               where cs1.cs_order_number = cs2.cs_order_number
                 and cs1.cs_warehouse_sk <> cs2.cs_warehouse_sk)
   and not exists(select *
                  from catalog_returns cr1
                  where cs1.cs_order_number = cr1.cr_order_number)
 order by count(distinct cs_order_number)
 limit 100;
```

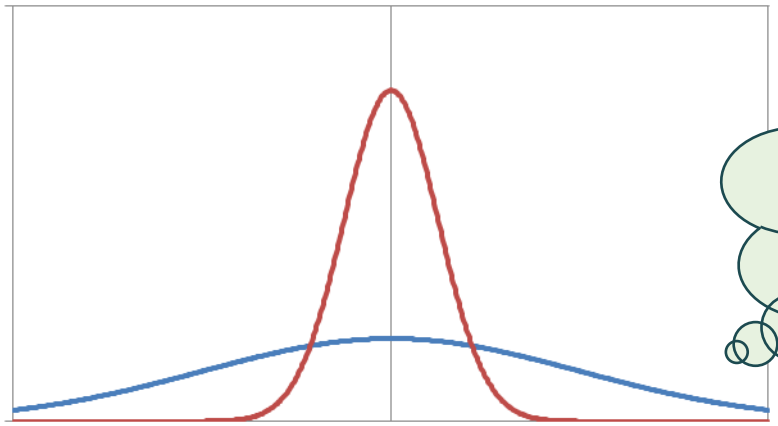
更なる分析 (2/4) – Query16の実行計画 (SF=1で実行...)

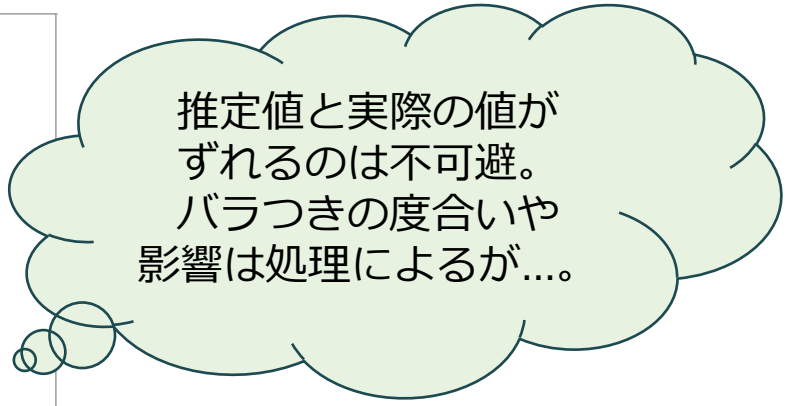
```
....(省略)....
-> Nested Loop Anti Join (cost=83650.17..162632.42 rows=1 width=20)
                         (actual time=975.907..25308.315 rows=495 loops=1)
   Join Filter: (cs1.cs_order_number = cr1.cr_order_number)
   Rows Removed by Join Filter: 97309298
   -> Nested Loop (cost=83650.17..155743.64 rows=1 width=20)
                  (actual time=926.407..3219.101 rows=1462 loops=1)
      -> Nested Loop (cost=83649.88..155741.95 rows=4 width=28)
                     (actual time=920.546..3133.191 rows=49929 loops=1)
         Join Filter: (cs1.cs_call_center_sk = call_center.cc_call_center_sk)
         Rows Removed by Join Filter: 250035
               :
         ....(省略; これ以下は合計で3.13secしか要していない)....
               :
   -> Seq Scan on catalog_returns cr1 (cost=0.00..5599.67 rows=144067 width=8)
                                      (actual time=0.001..6.348 rows=66560 loops=1462)
 Planning time: 1.644 ms
 Execution time: 25310.373 ms
```

- 問題サイズを1/100に縮減して実行時情報を採取した結果...。
- NestLoopの処理コストは $O(N_{outer} \times N_{inner})$
  - ➔ 推定誤差が容易に計算量の爆発に繋がる
- 144千行のつもりが、1,462x66,560≒97百万行に膨れ上がってしまった(!)

更なる分析 (3/4) – 海外の愚痴

実行計画作成時に、推定行数のリスク（変動）を考慮していない





推定値と推定誤差

- 推定値が実測値と異なるのは不可避
- 推定誤差による影響は処理タイプによって異なる。
  - HashJoin　：O(⊿*N* +⊿*M*)
  - MergeJoin　：O(⊿*N* log(⊿*N*) +⊿*M* log(⊿*M*))
  - NestLoop　：O(⊿*N* ×⊿*M*)

➔仕方ないので、*O(NM)* 処理であるNested Loopを無効化する。

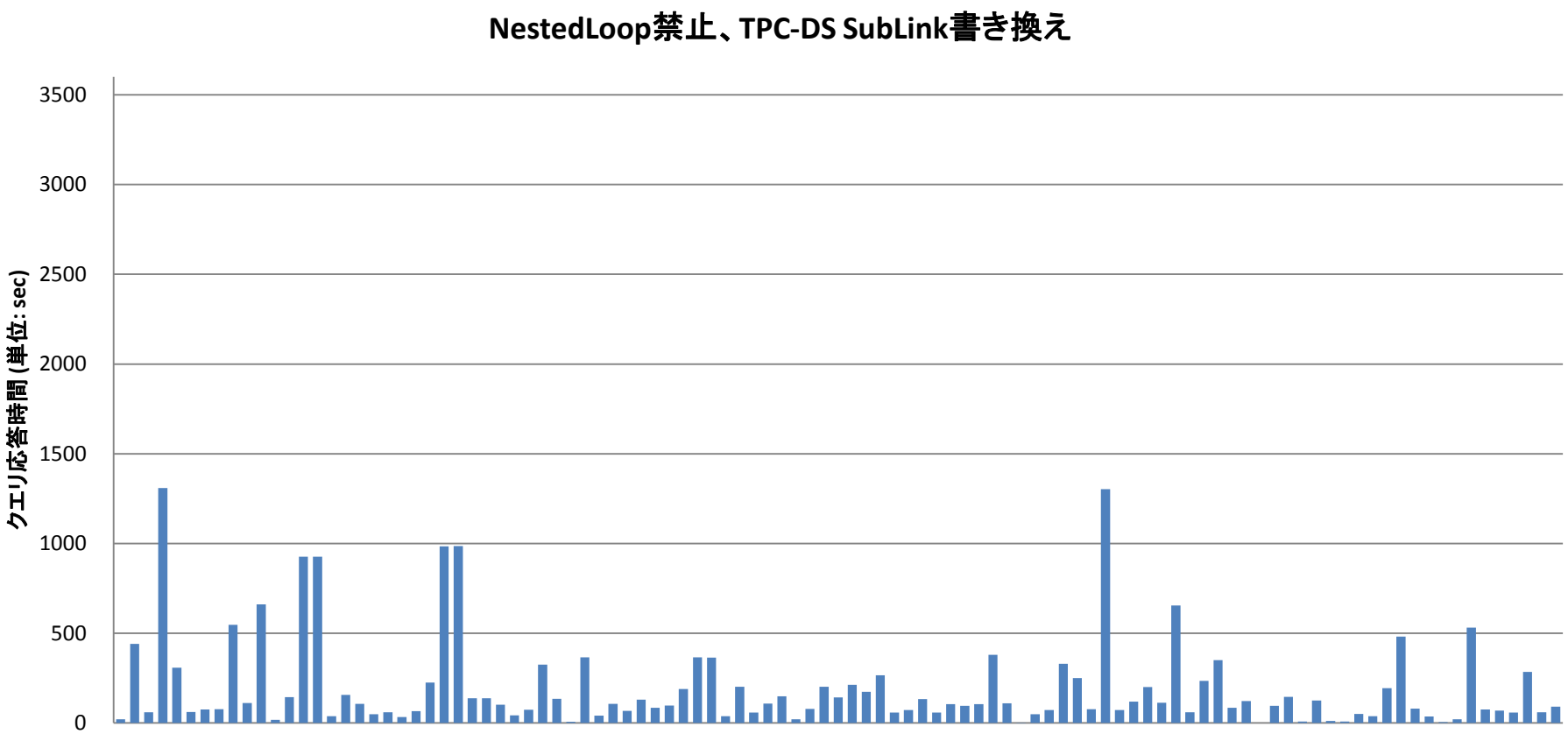
```
SET enable_nestloop = off;
```

更なる分析 (4/4) – Query16の実行計画 (SF=1, NestLoop禁止)

```
SET enabled_nestloop = off;
....(省略)....
-> Hash Anti Join (cost=95880.57..165398.59 rows=1 width=20)
                  (actual time=804.945..1600.695 rows=495 loops=1)
   Hash Cond: (cs1.cs_order_number = cr1.cr_order_number)
   -> Hash Join (cost=88480.07..157998.06 rows=1 width=20)
                (actual time=746.820..1542.702 rows=1462 loops=1)
      Hash Cond: (cs1.cs_call_center_sk = call_center.cc_call_center_sk)
                 :
         ....(省略)....
                 :
   -> Hash (cost=5599.67..5599.67 rows=144067 width=8)
           (actual time=56.534..56.534 rows=144067 loops=1)
      Buckets: 262144 Batches: 1 Memory Usage: 7676kB
      -> Seq Scan on catalog_returns cr1 (cost=0.00..5599.67 rows=144067 width=8)
                                         (actual time=0.006..35.087 rows=144067 loops=1)
 Planning time: 1.198 ms
 Execution time: 1601.838 ms
```

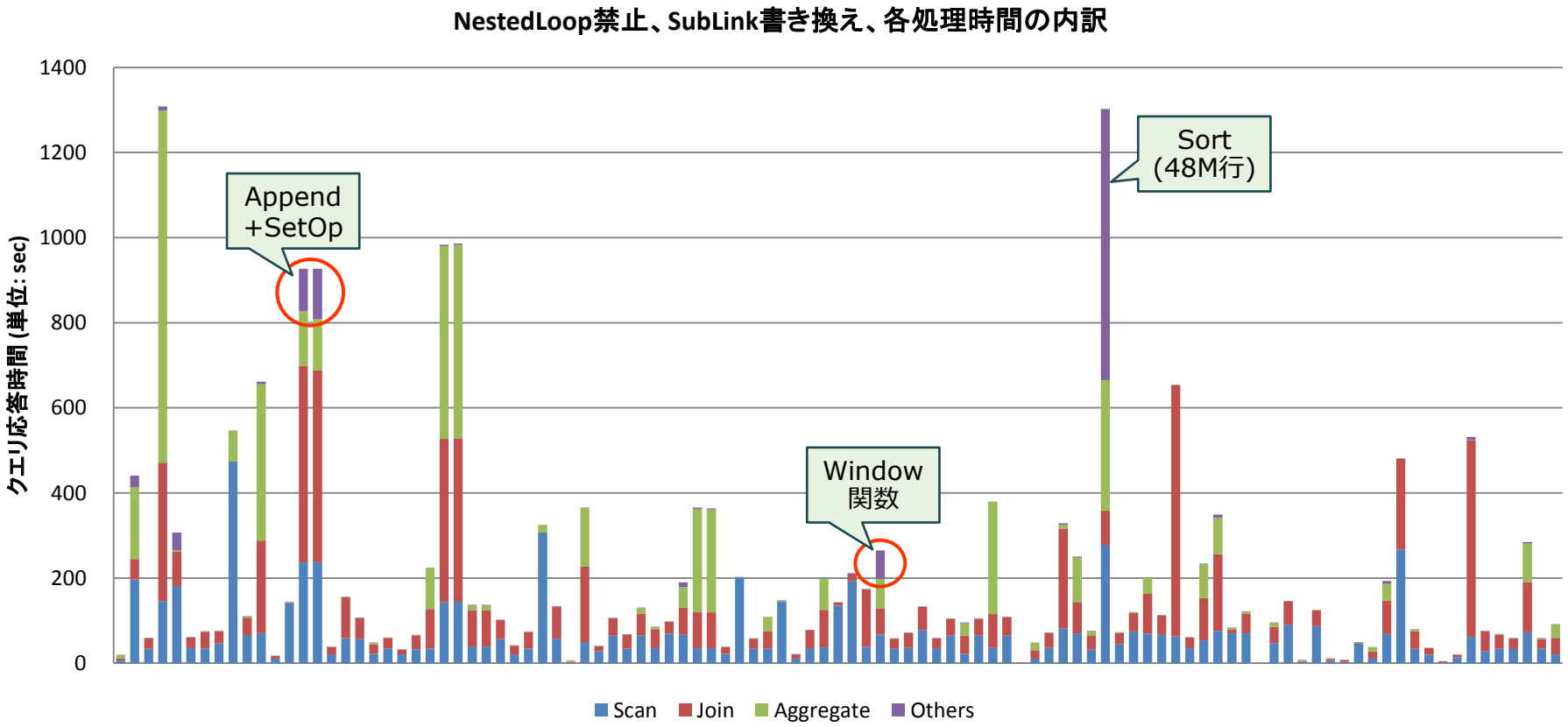
- NestLoopの代わりにHashJoinを使用した結果
  - N_{outer} の推定誤差は変わらない 1 ➔ 1462
  - その場合でも、処理すべき行数が膨れ上がるという事はない
- 結果、25.3sec ➔ 1.60sec へスピードアップ

ベンチマーク結果③ – NestedLoop禁止 + TPC-DS修正



- HashJoinを強制した事で、1時間以上応答を返さないクエリは消滅
- downside:
  - 巨大テーブル間のJOINではメモリを馬鹿食いする。
  - X exists in (SELECT ...) では立ち上がりが遅い。

ベンチマーク結果③‘ – 処理時間の内訳



- Scan(34.4%)、Join(36.9%)、Aggregate(23.3%)で総処理時間の95%
  - ✓ 但し、Scanはon-memoryのデータ転送である事に留意。
- その他 (5.3%) の中で目立つのは…。
  - Sort、SetOp、Window関数

TPC-DSの結果から得られた知見

前提：TPC-DSは世間一般のBIワークロードを反映している

- 裏ボス
  - Planner
- ラスボス
  - Scan
  - Join
  - Aggregation
- 中ボス
  - Sort
  - SetOp
  - Window関数

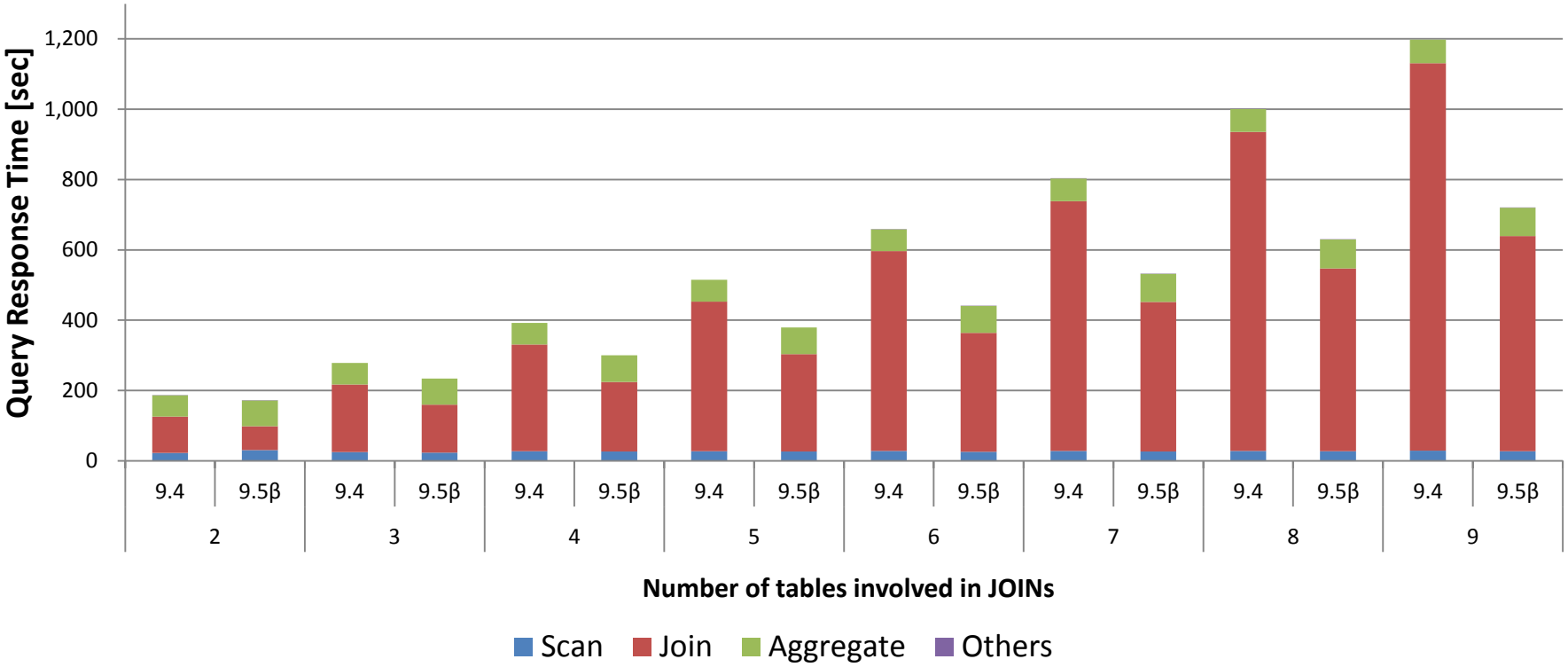
この辺を頑張ると、
体感パフォーマンスが
ぐっと上がる（ハズ）

アジェンダ

1. TPC-DSベンチマークとは？
2. ベンチマーク結果と分析
3. **改善アプローチ**
   - **Sustaining Innovations**
   - **Upper Planner Path-Ification**
   - **Parallelism – scale-up**
   - **Parallelism – scale-out**
   - **Distributed Aggregation**
4. その先の未来

v9.5 Sustaining Innovations (1/2) – HashJoin

- Improve in-memory hash performance

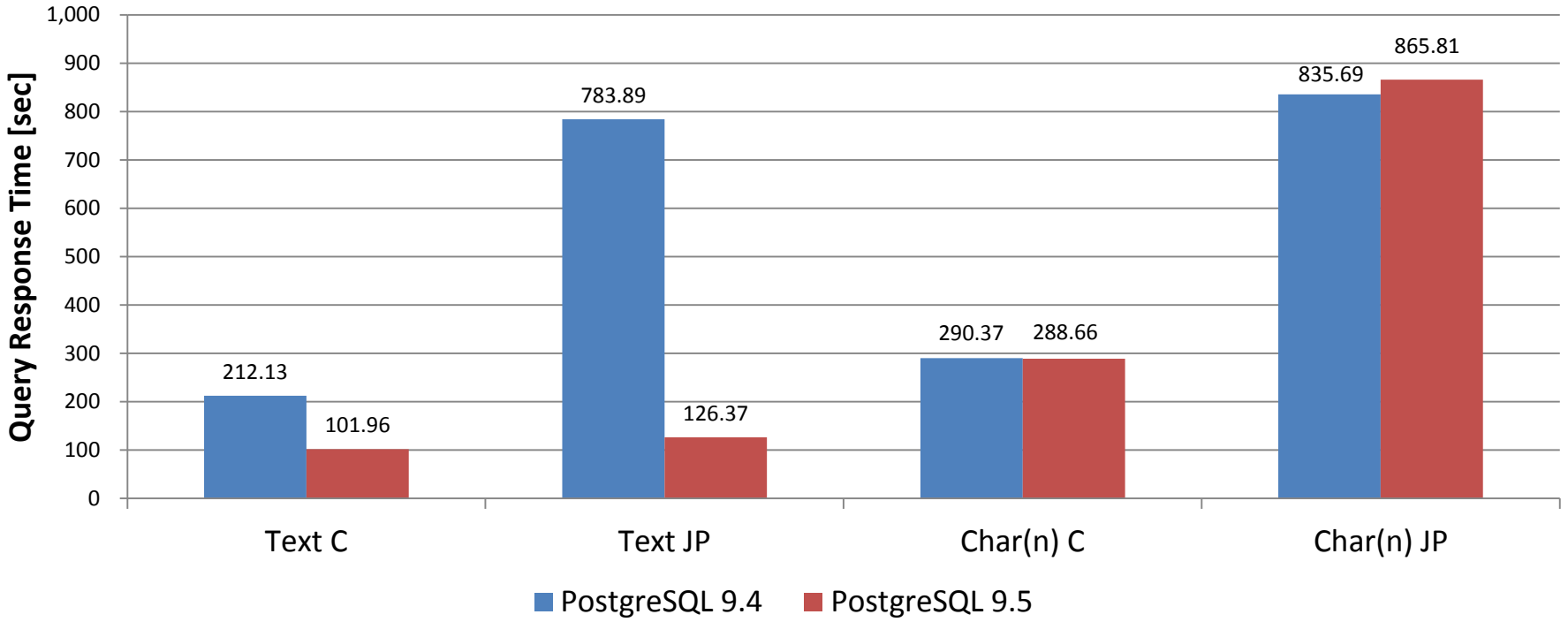



```
SELECT cat, AVG(x) FROM t0 NATURAL JOIN t1 [, ...] GROUP BY cat;
```

- t0: 100M rows, t1~t10: 100K rows for each, all the data was preloaded.
- CPU: Xeon E5-2670v3, RAM: 384GB, Red Hat Enterprise Linux 7.0

v9.5 Sustaining Innovations (2/2) – SortSupport

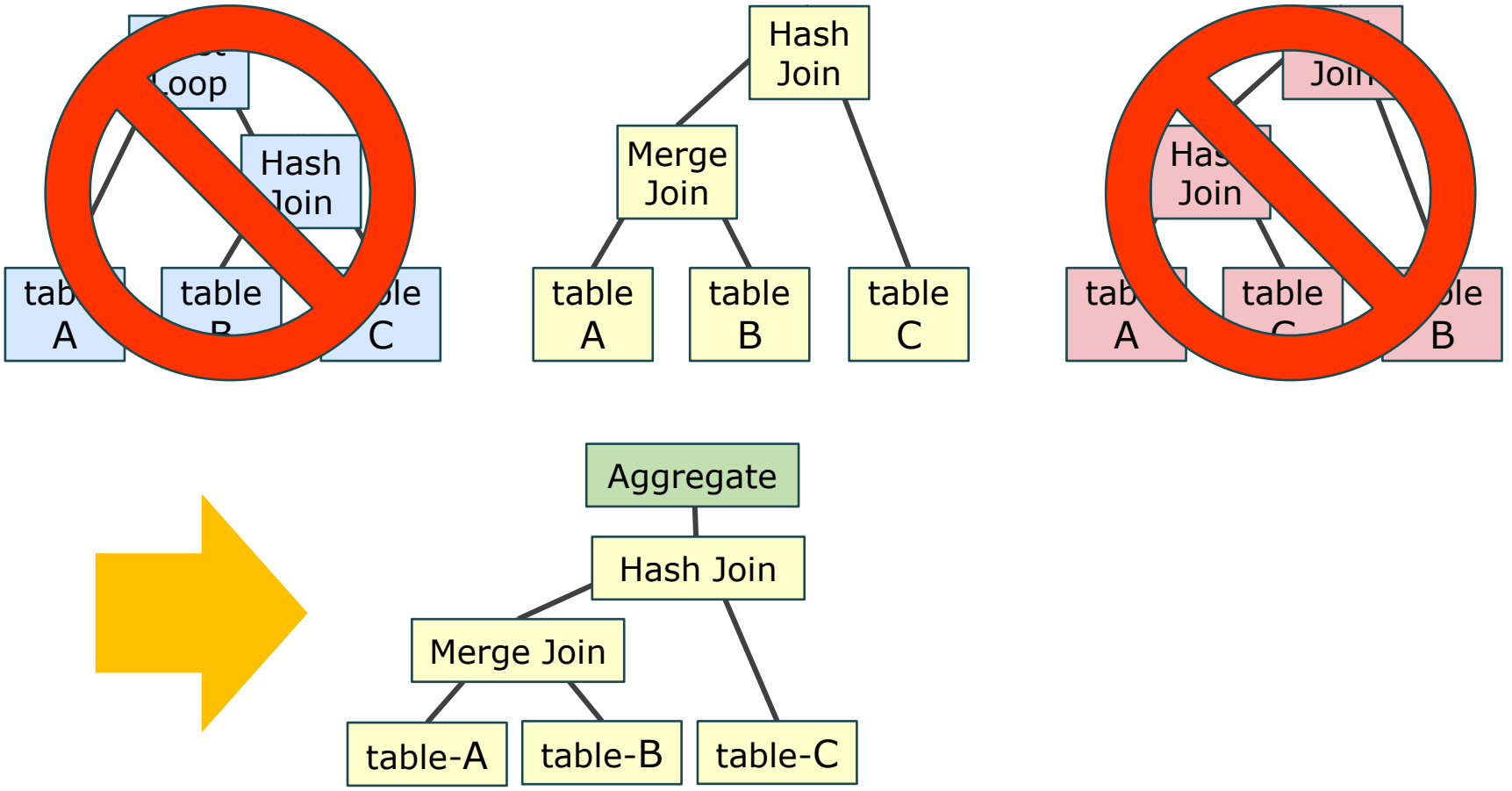
- Improve the speed of sorting VARCHAR, TEXT, and NUMERIC fields
- Extend the infrastructure that allows sorting to be performed by inlined...




```
SELECT * FROM tbl_[text|char] ORDER BY val;
```

- tbl_[text|char]: contains 50M rows, MD5 random string
- CPU: Xeon E5-2670v3, RAM: 384GB, Red Hat Enterprise Linux 7.0

Upper Planner Path-Ification (1/2) – 現在のプラン生成

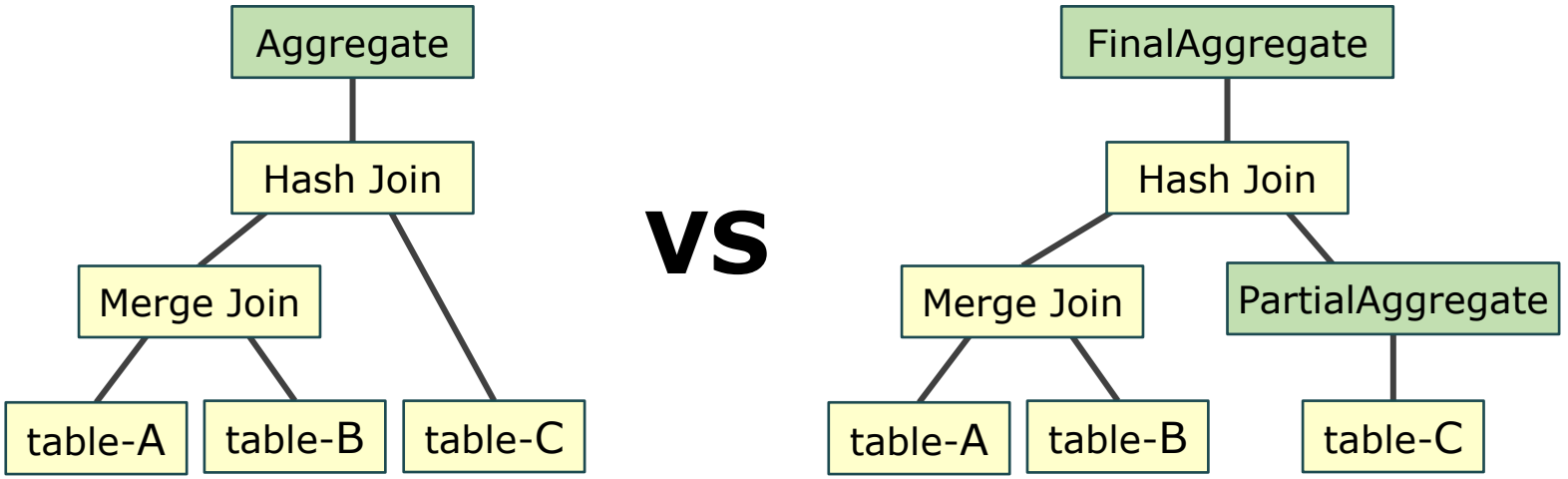


- 最初にScan+Joinの組合せをトライ
- 最も推定コストの小さな実行パスにAggregateなどを乗せる

➔ 途中でAggregateを挟んだ方がよいケースを上手く扱えない。

Upper Planner Path-Ification (2/2) – あるべき姿

ScanやJoin同様にAggregate等を含むパスを検討する機能



- 何が可能になるか？
  - JOINの前に部分集約を挟み、JOINすべき行数を削減
  - ワーカープロセス側の処理で部分集約を実行し、データ量を削減
  - ....など
- 各候補パスをコストベースで比較検討できる事がポイント

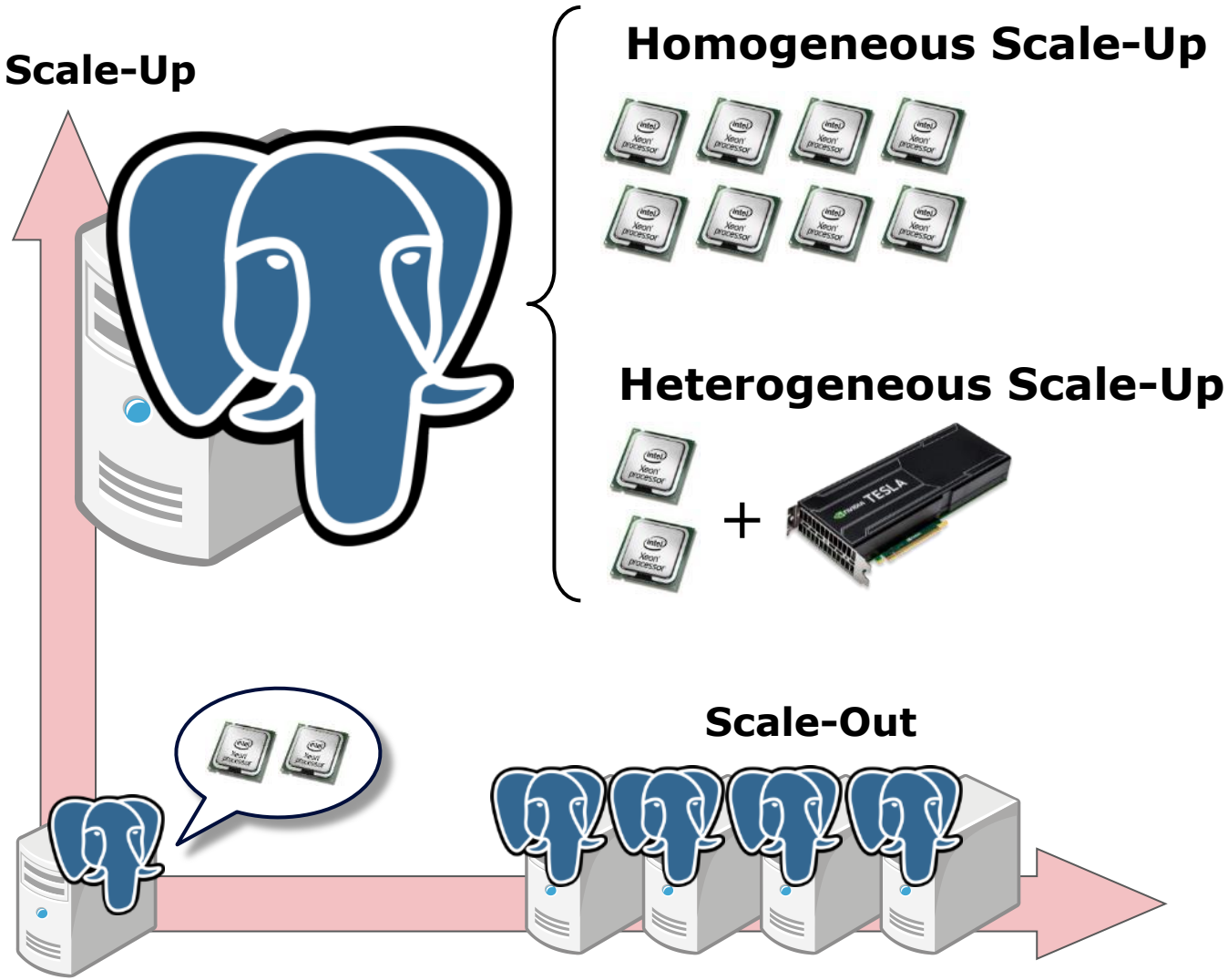
その他のプラナー改善

- 大規模なプラナー改善は Path-Ification機能のマージを待っている状況

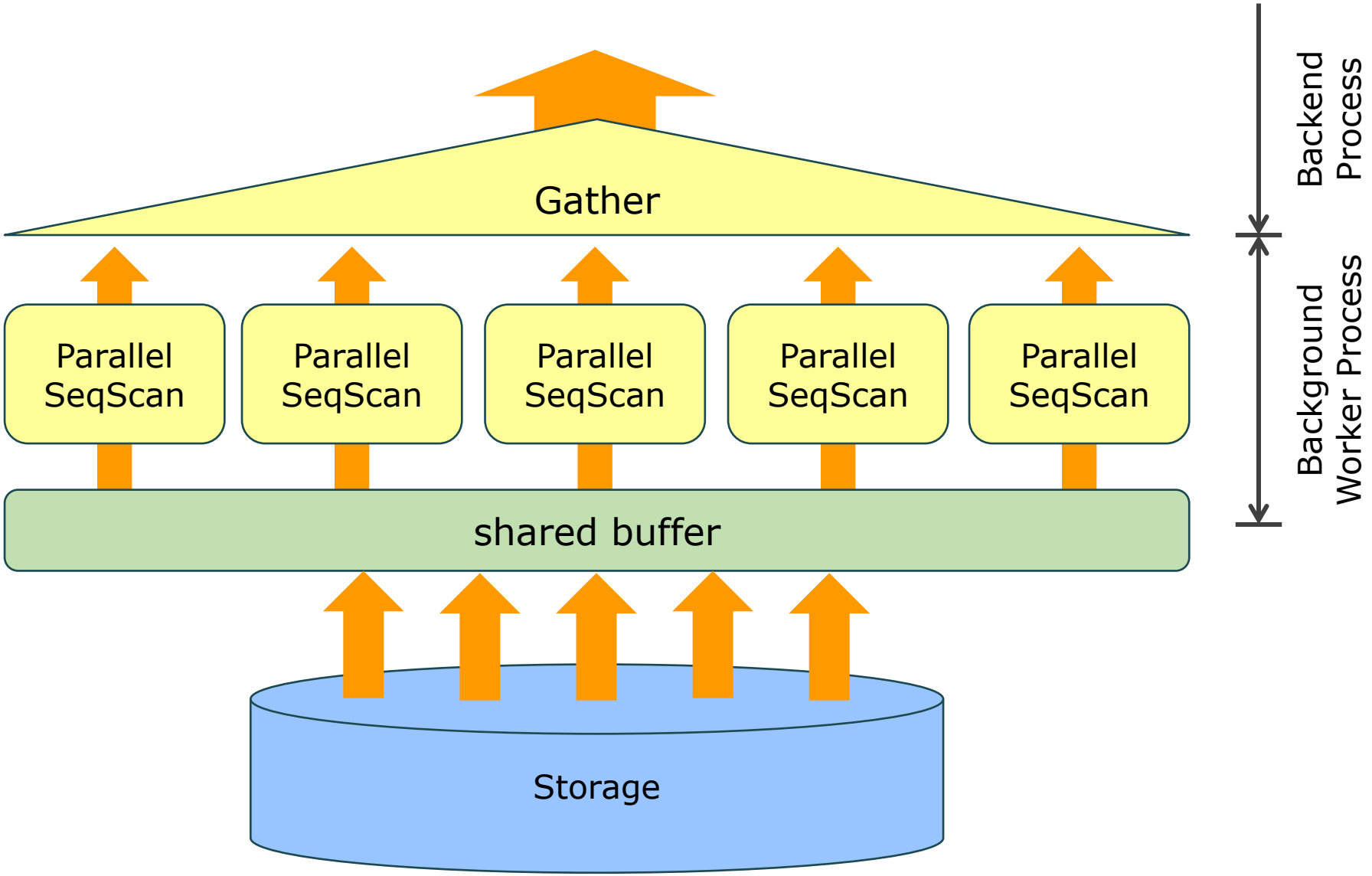
- インテリジェントな SubLinkのPull-up
- 行推定の変動幅を考慮したプラン選択
  - ✓ 現時点では具体的な取り組みに落ち込んでいない
  - ➔ pgsql-hackersへの参加者求む！


WELCOME TO
THE PARALLEL
WORLD

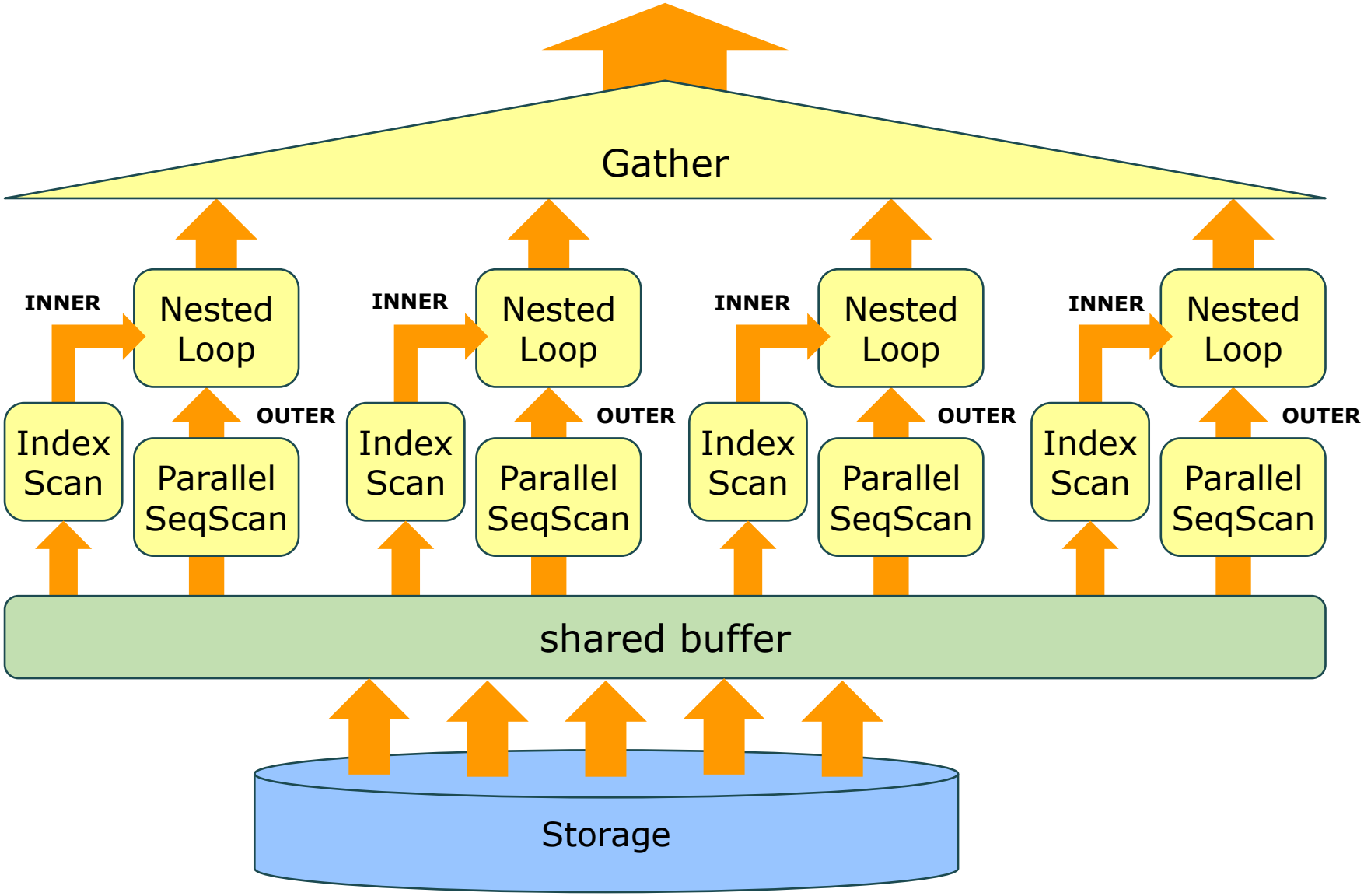
並列処理と開発者コミュニティでの動向



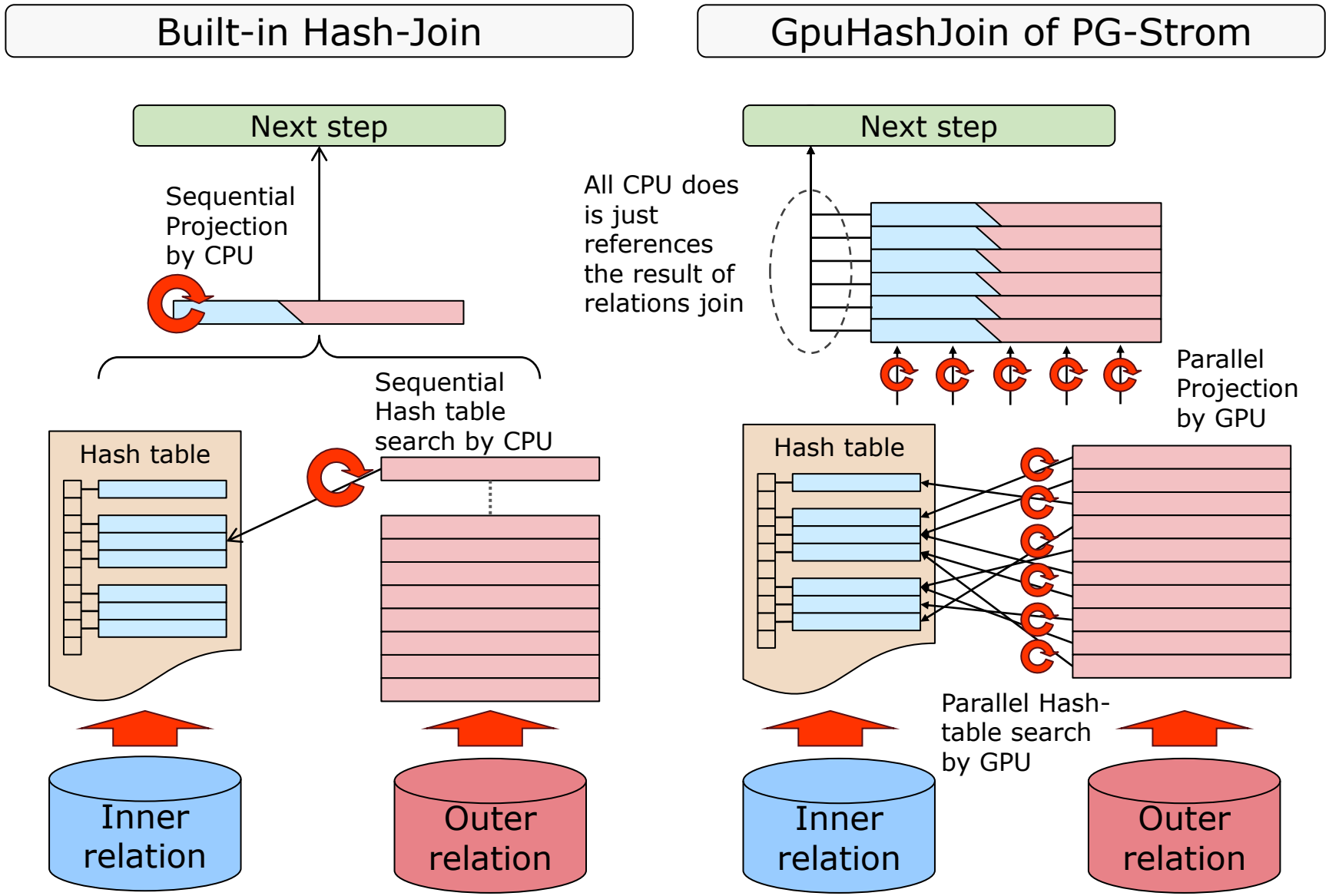
なぜ並列処理が Scan 高速化につながるのか？



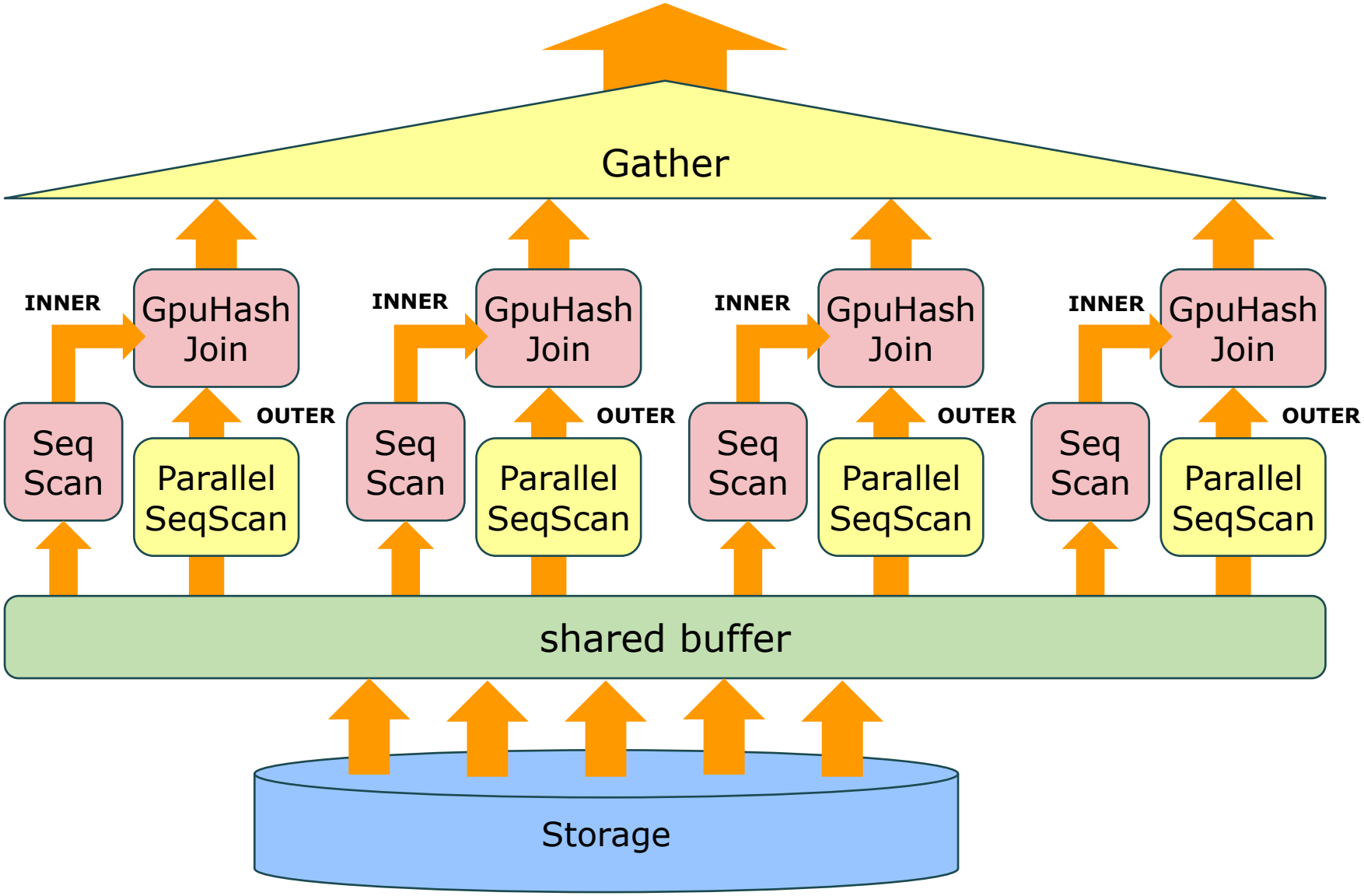
なぜ並列処理が Join 高速化につながるのか？



GpuHashJoin – より細粒度での並列処理



なぜCPU+GPU並列処理が Join 高速化につながるのか？

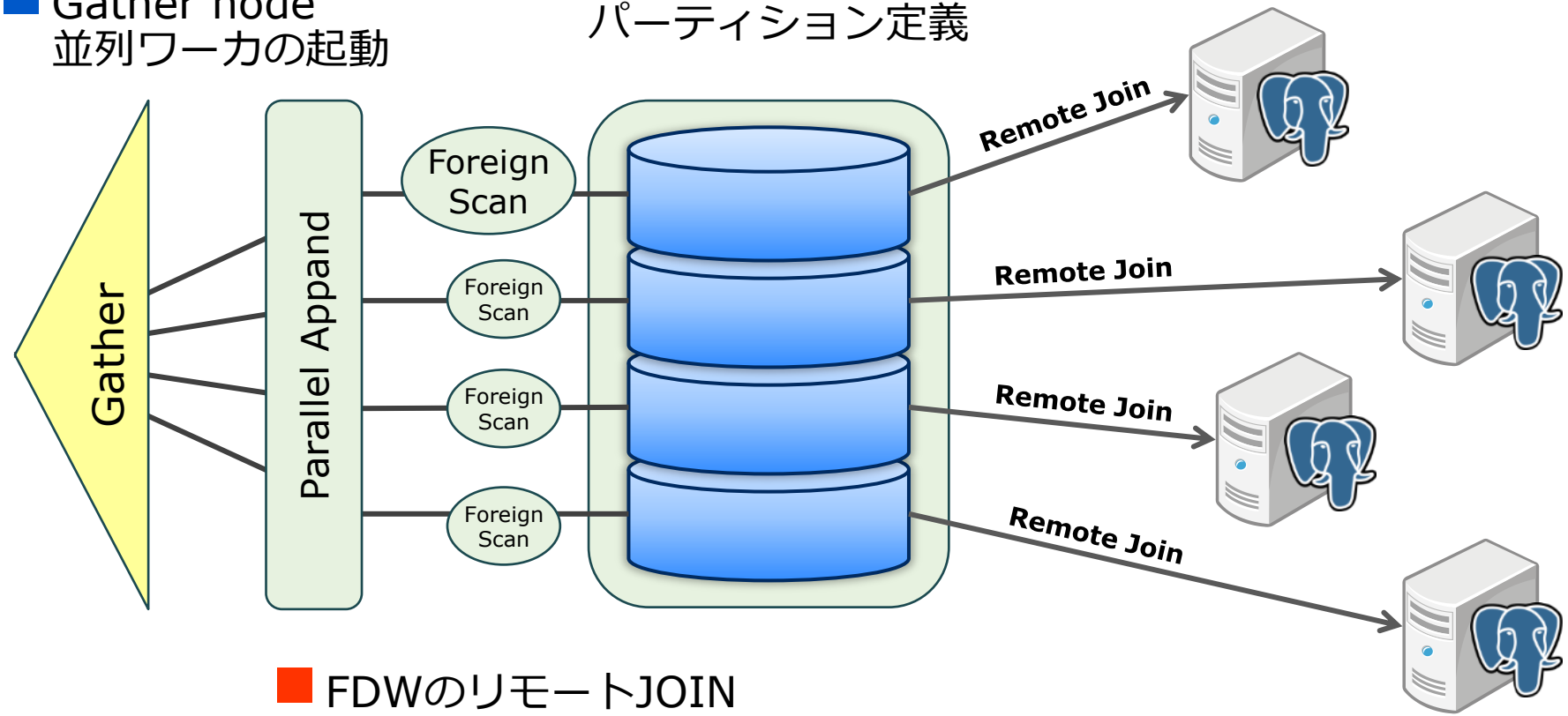


スケールアウト (1/2) – FDW をベースとした分散DB

v9.5 機能
v9.6 機能

Gather node
並列ワーカの起動

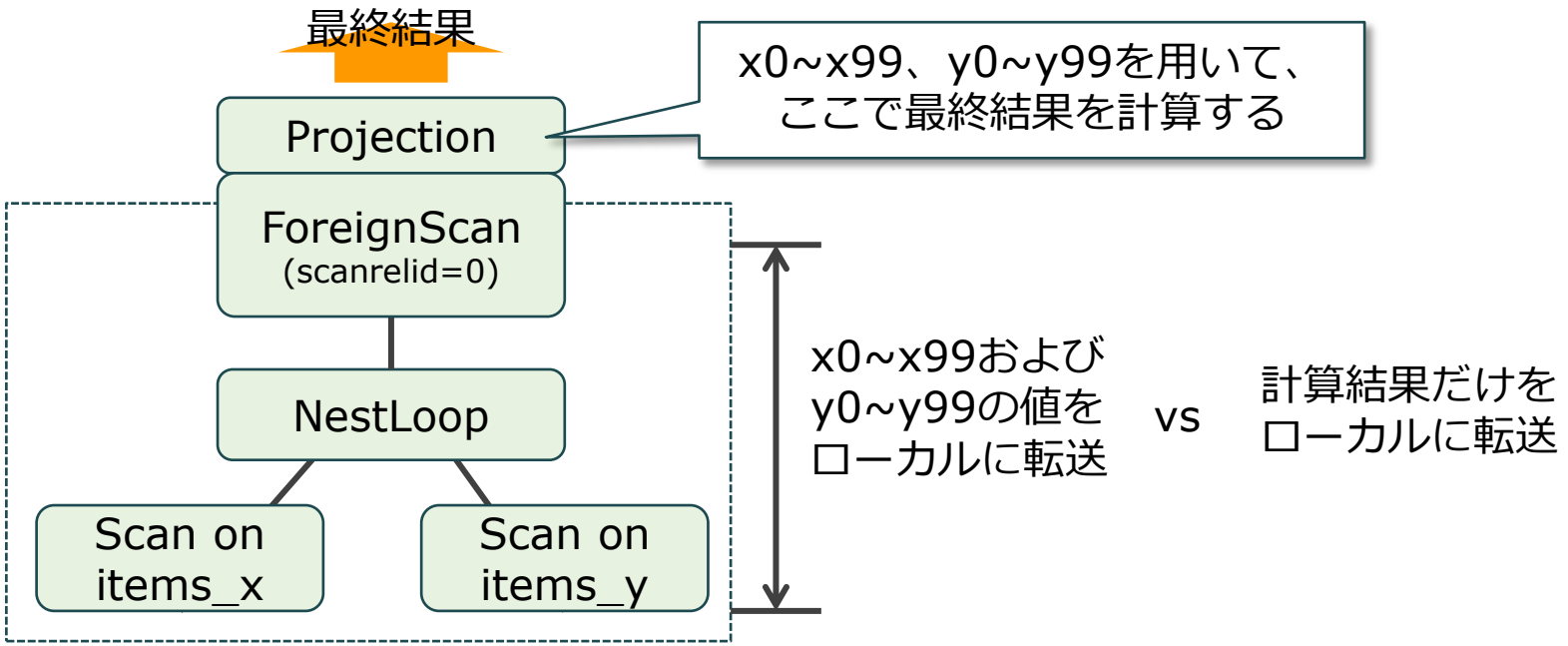
外部表を含む
パーティション定義



FDWのリモートJOIN
AppendとJoinの順序入替え

スケールアウト (2/2) – Target List Push Down

```
SELECT sqrt((x0-y0)^2 + ... + (x99-y99)^2) dist FROM
items_x, items_y WHERE x_id != y_id;
```



- プロジェクションが複雑なクエリでは、ローカルCPUを節約し、外部の計算機資源を使用した方が性能を向上できると考えられる。
  - ✓ 外部計算機資源：CPU（他プロセス、外部サーバ）、GPU、FPGA(?)など
  - ✓ Upper Planner Path-Ification でプラナーのインフラが改善された次のステップ

Distributed Aggregation (1/4) – Map-Reduce

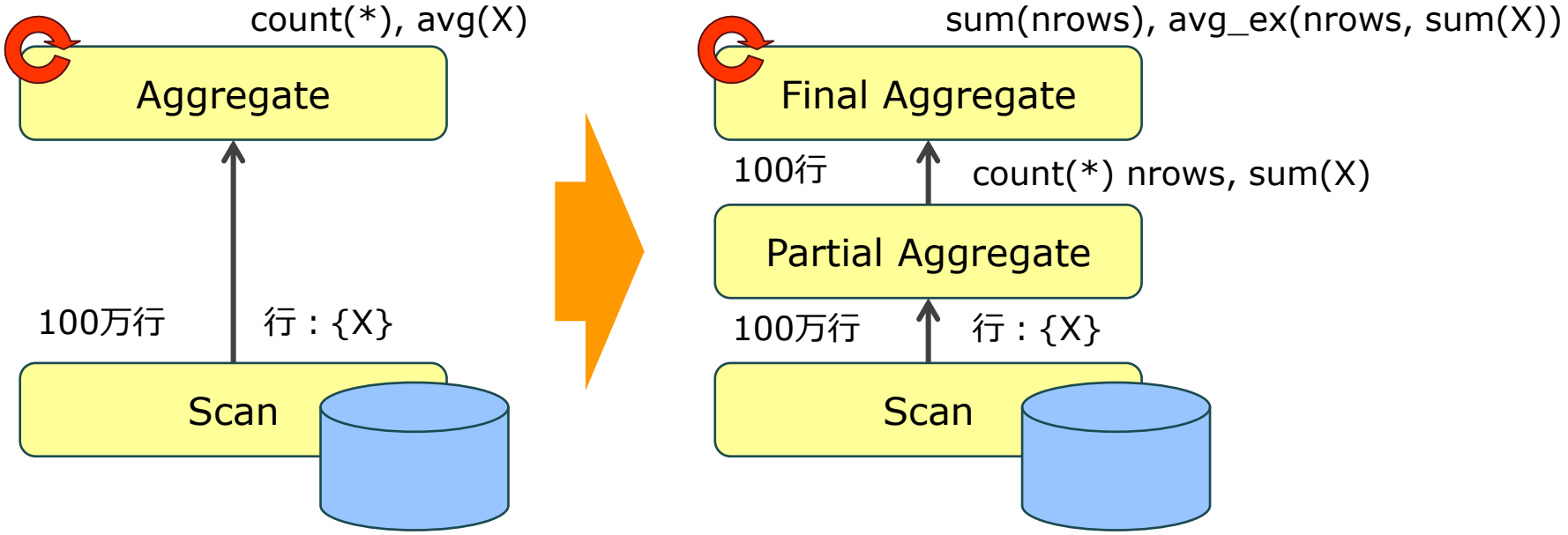
平均値の定義

$$\frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{N}\left(\sum_{i=1}^{k_1} x_i + \cdots + \sum_{i=k_{j-1}}^{N} xi\right) \quad \ldots \left(1 < k_j, k_j < N\right)$$

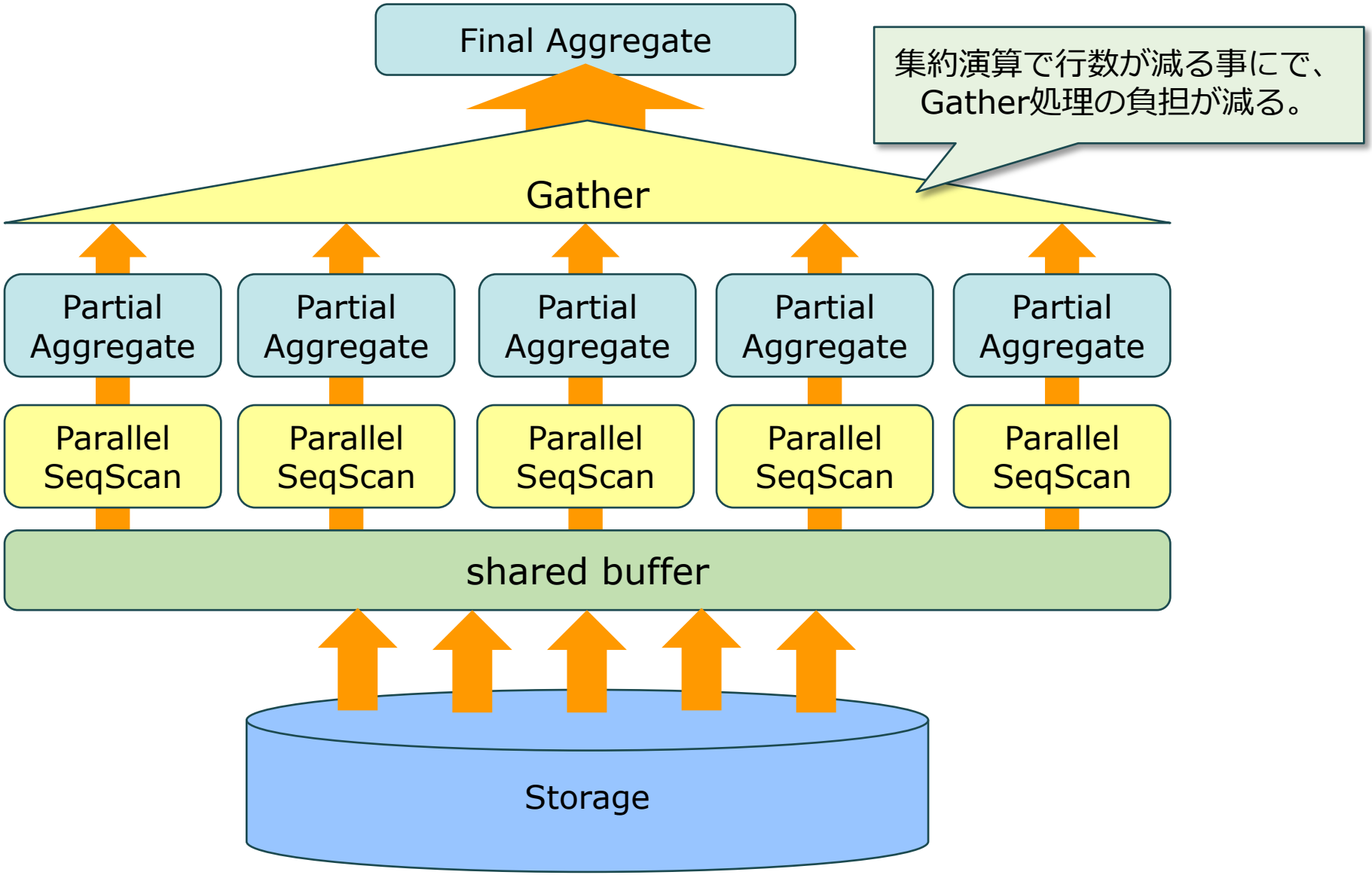
➔ 各Σ項を複数に分割、独立に計算しても最終結果は同じ。

- 実は浮動小数点計算誤差も小さくなる。

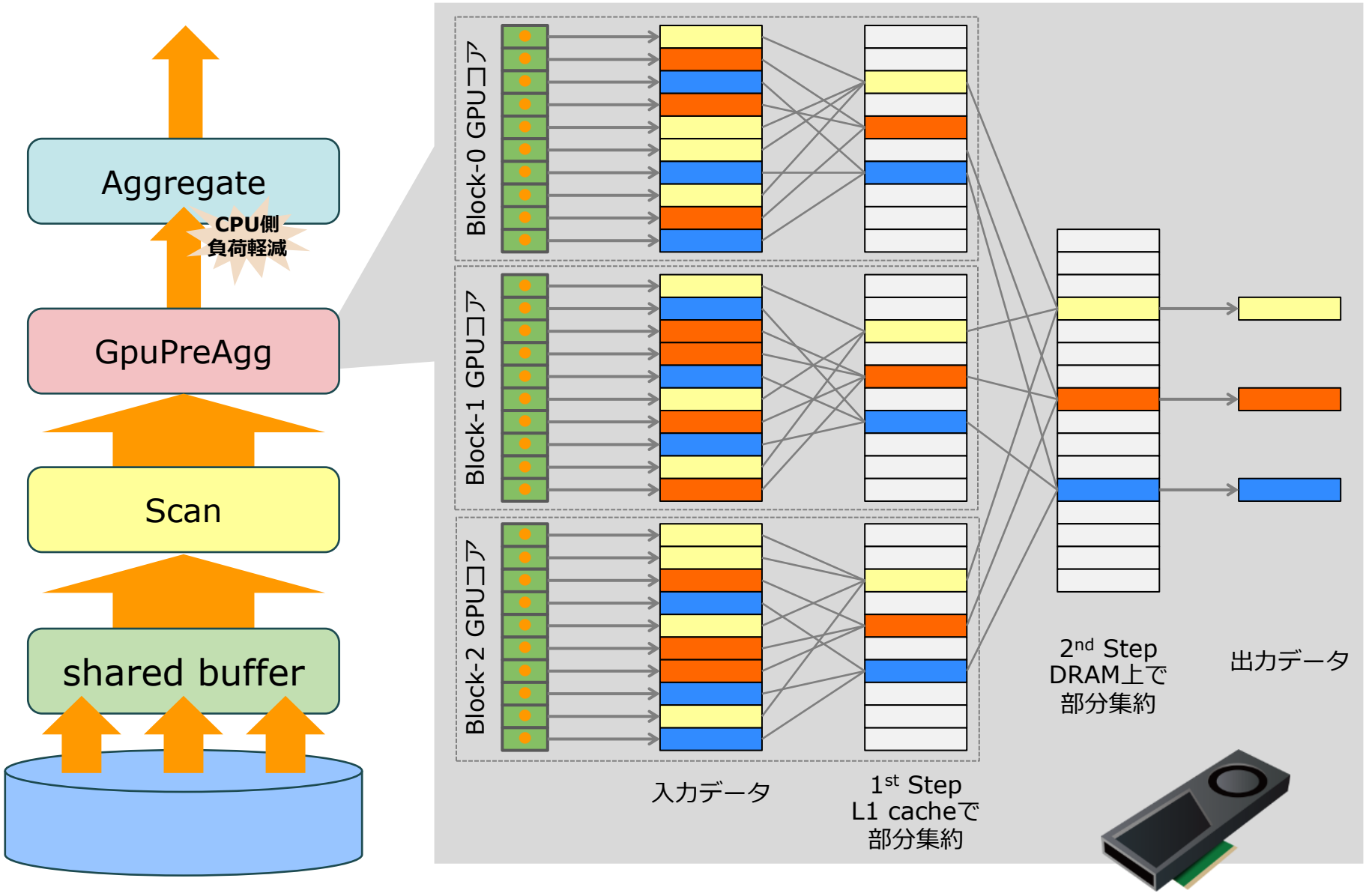
Partial Aggregate + Final Aggregate



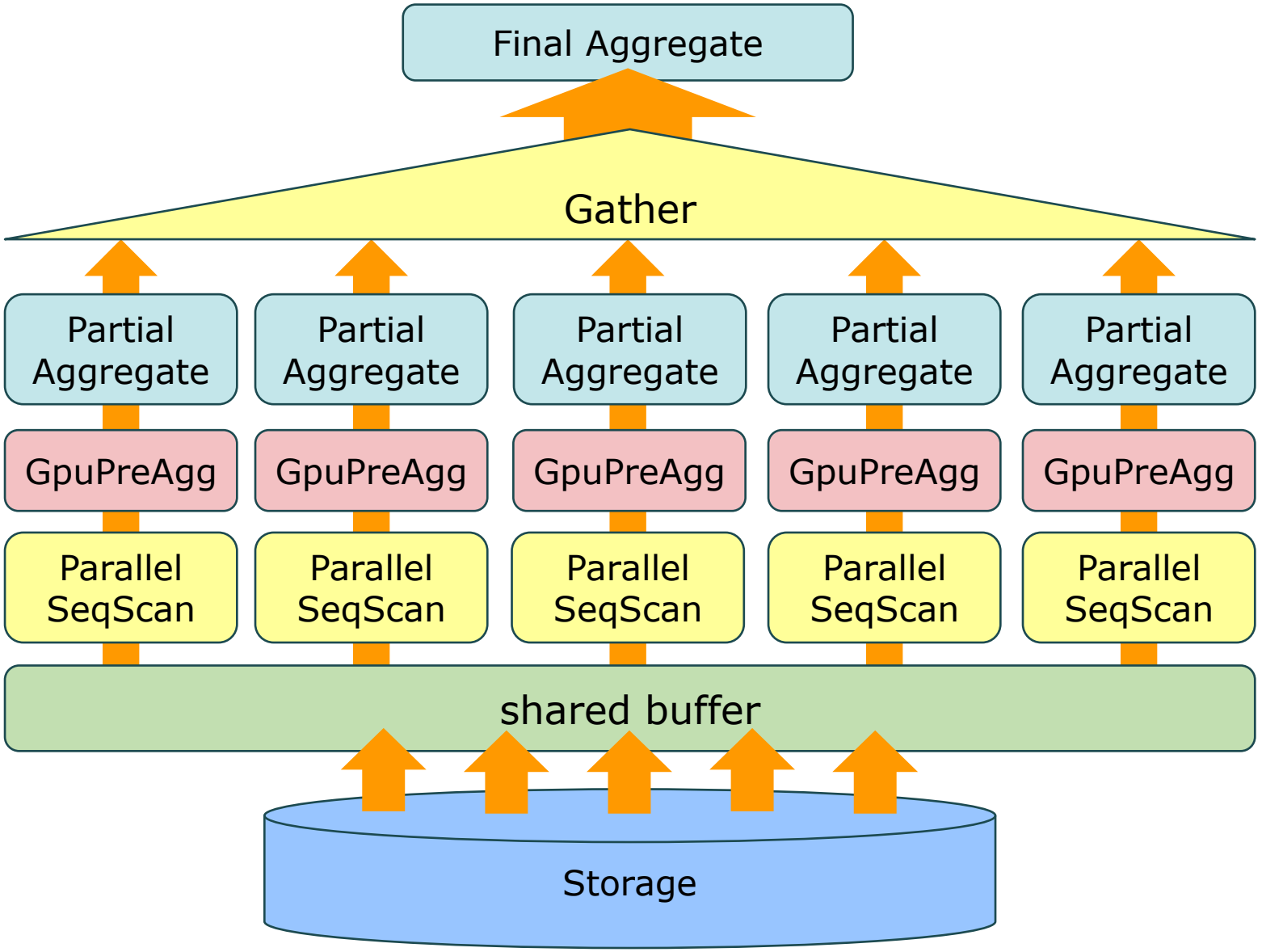
Distributed Aggregation (2/4) – CPU並列



Distributed Aggregation (3/4) – GPU並列による実装



Distributed Aggregation (4/4) – CPU+GPUハイブリッド実装



アジェンダ

1. TPC-DSベンチマークとは？
2. ベンチマーク結果と分析
3. 改善アプローチ
4. **その先の未来**

再掲）TPC-DSの結果から得られた知見

前提：TPC-DSは世間一般のBIワークロードを反映している

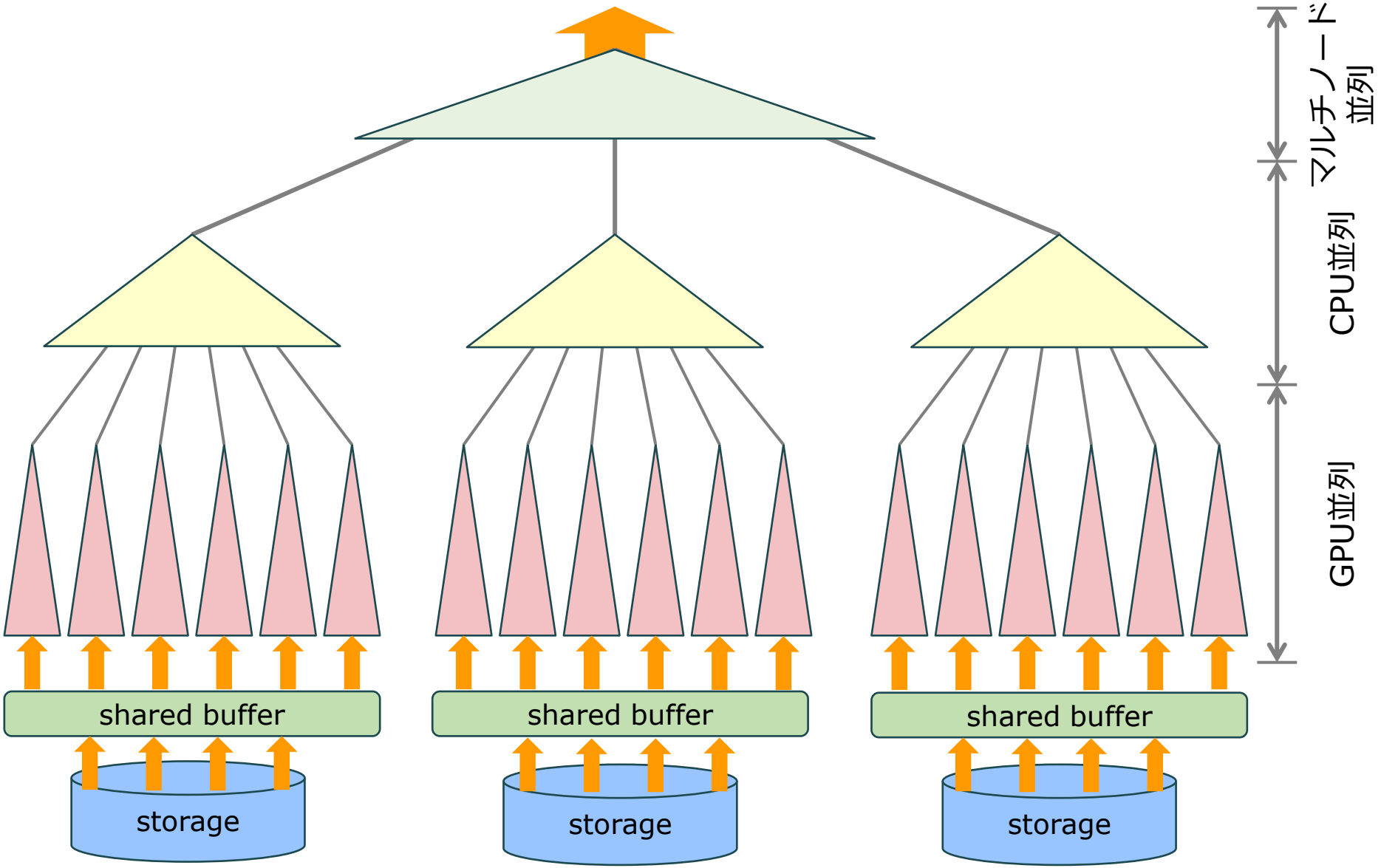
- 裏ボス
  - Planner
- ラスボス
  - Scan
  - Join
  - Aggregation
- 中ボス
  - Sort
  - SetOp
  - Window関数

1st Step: Upper Planner Path-Ification
➔ これをインフラとして、より高度な実行計画を生成できるようにする。

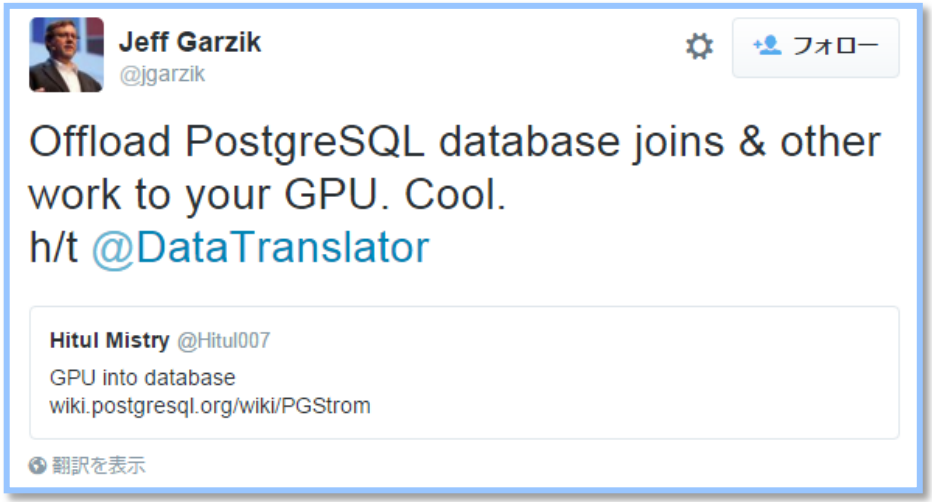
インテリジェントなプラナーが処理を分散し、様々な粒度で並列処理に落とし込むのが基本路線。

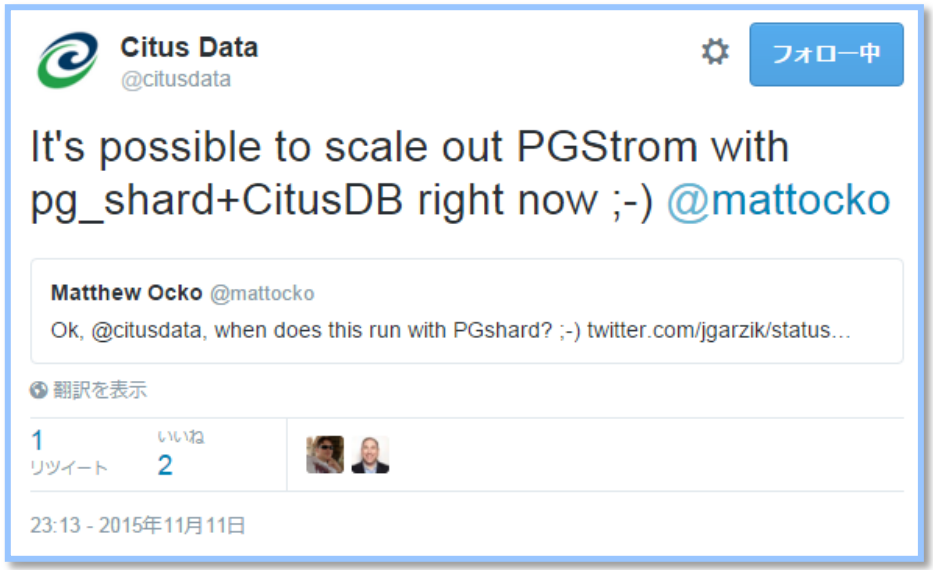
PostgreSQL開発者コミュニティでは、以下の全てが開発されている。
- CPU並列
- GPU並列
- マルチノード並列

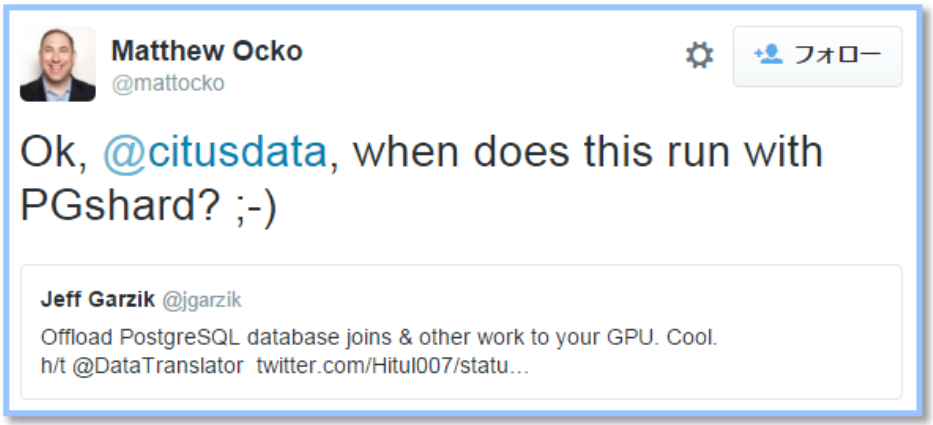
全部組み合わせるとどうなるか？



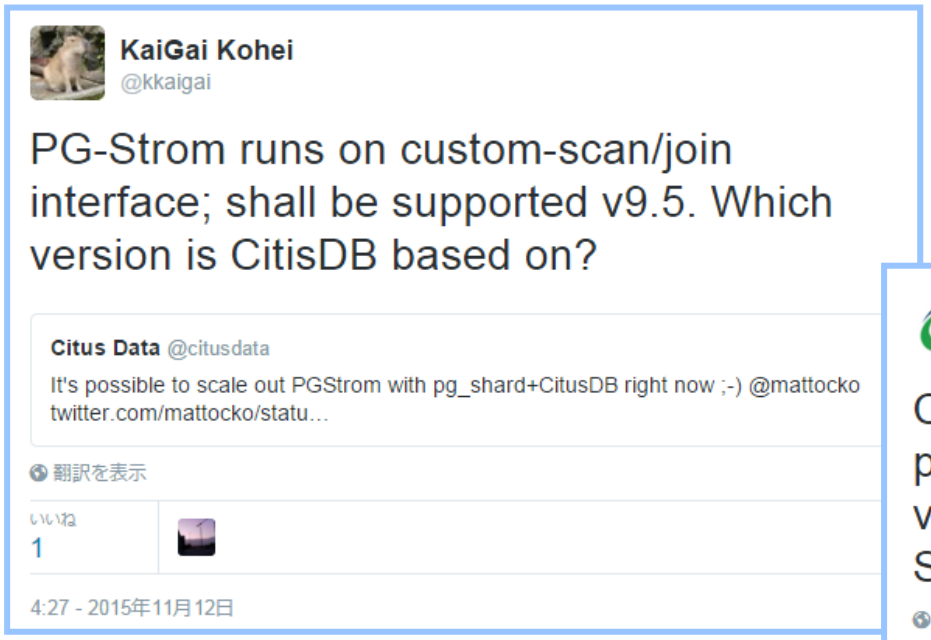
余談）全部組み合わせたらどうなるか？

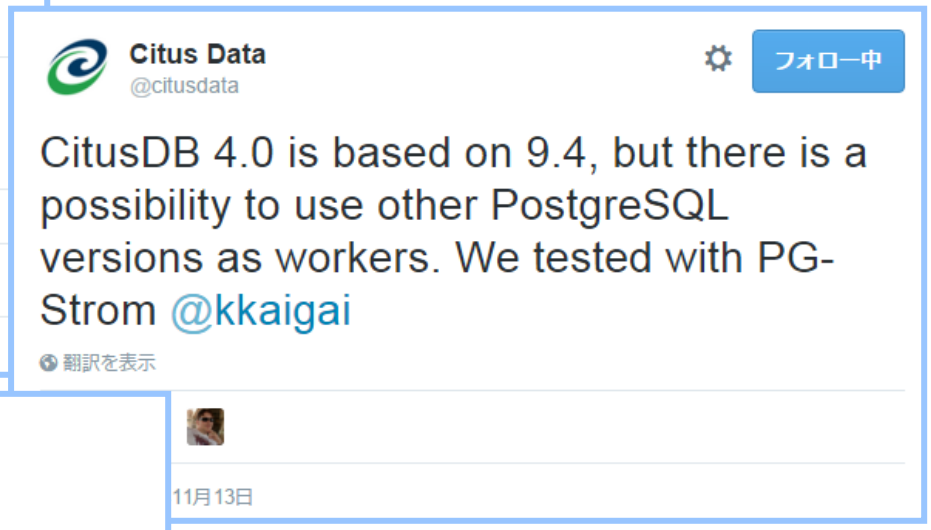


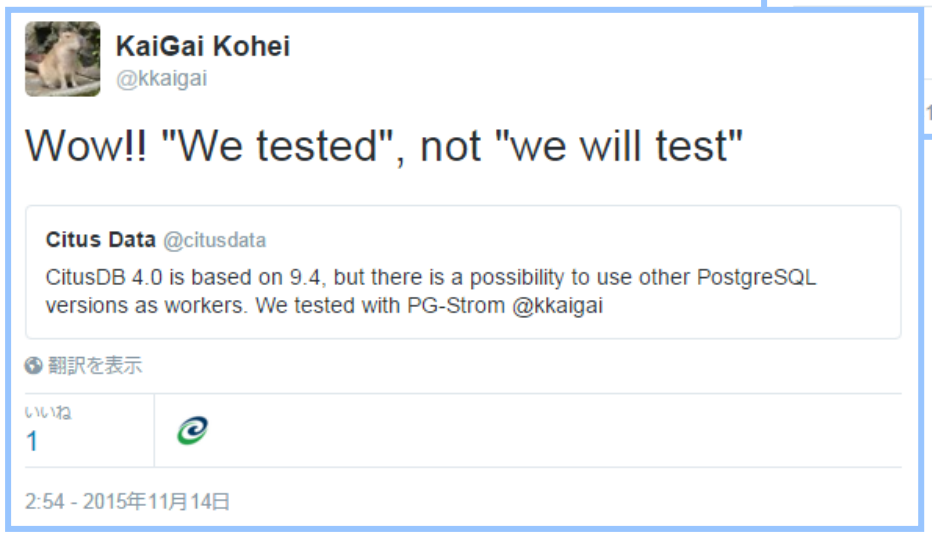




余談）全部組み合わせたらどうなるか？



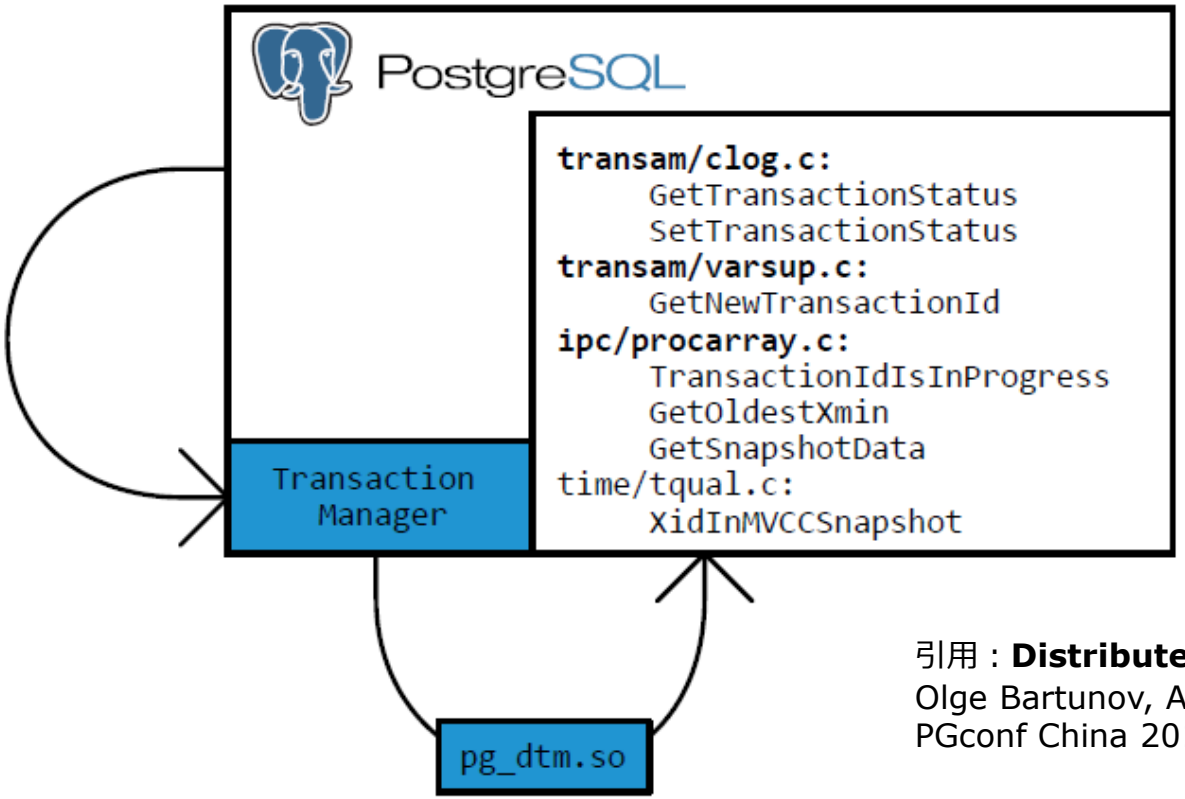




分散トランザクションマネージャ (DTM)

eXtensible Transaction Manager

- ノード間でMVCC一貫性を担保するには、複数ノードに対応したトランザクション状態の調停機構が必要。（最終的にはロックも）
- トランザクション管理を拡張可能とする枠組みが提案されている。

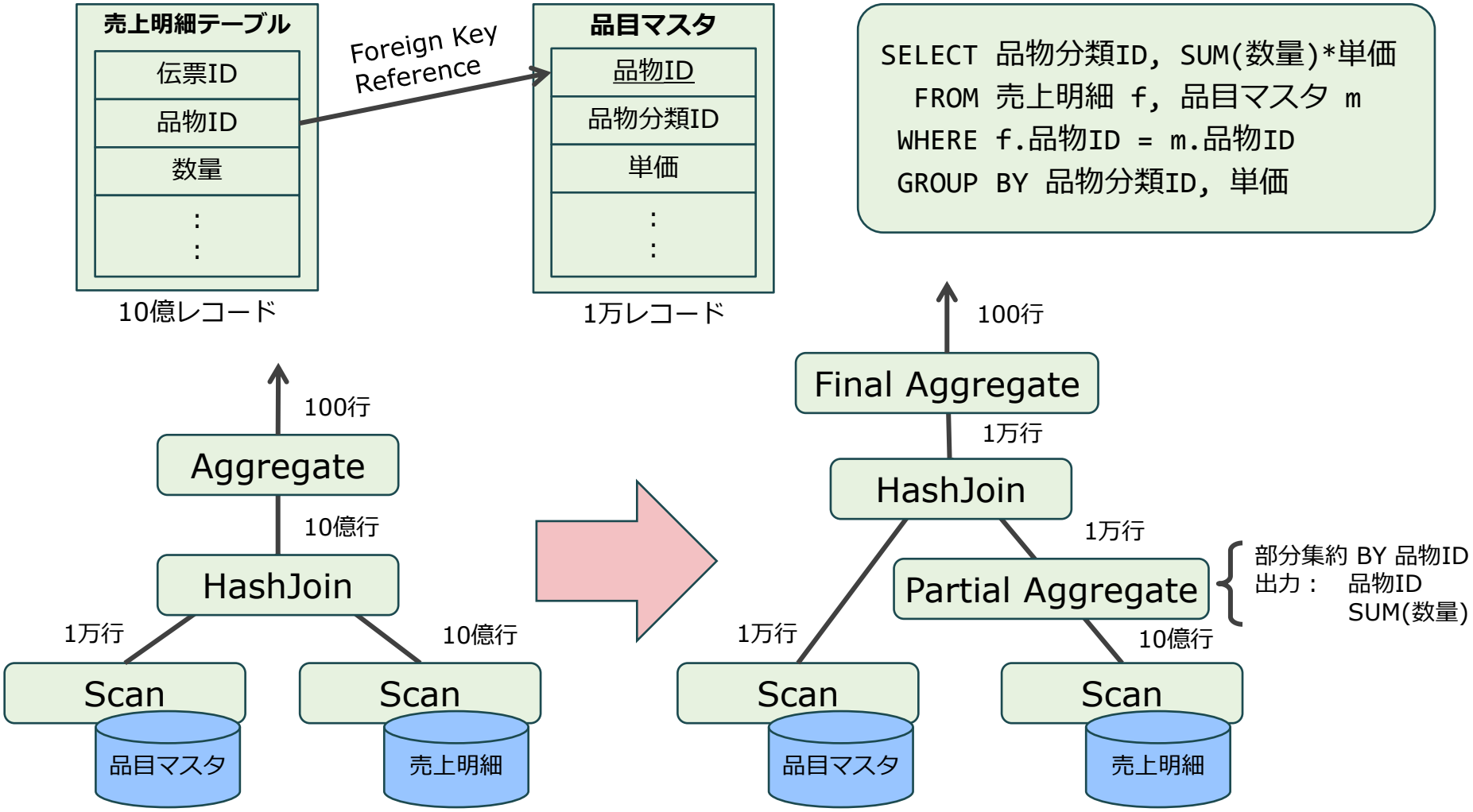


引用：**Distributed Transaction Manager**
Olge Bartunov, Alexander Korotkov,
PGconf China 2015 (Beijing)

Aggregation Before Join

Distributed Aggregationの派生形

- 一定の条件下で、Joinの前に部分集約を作る事ができる。



列指向ストレージ

特徴

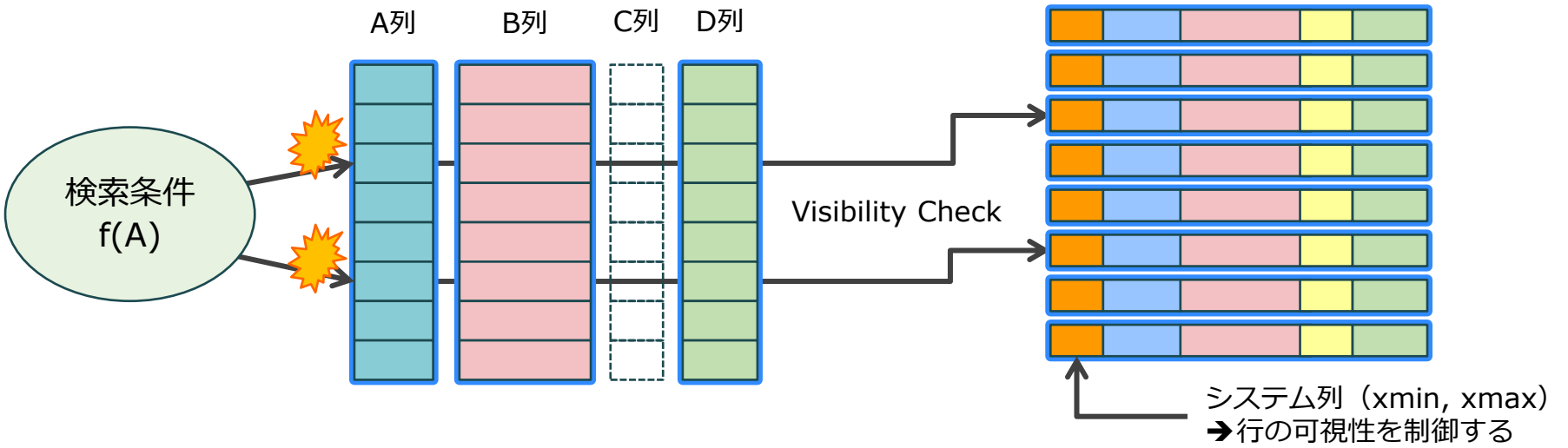
- 必要なカラムだけを取り出すため、I/O量が小さくなり、圧縮を効かせやすい。
- 更新系(OLTP)ワークロードは圧倒的に苦手

FDWベースの実装

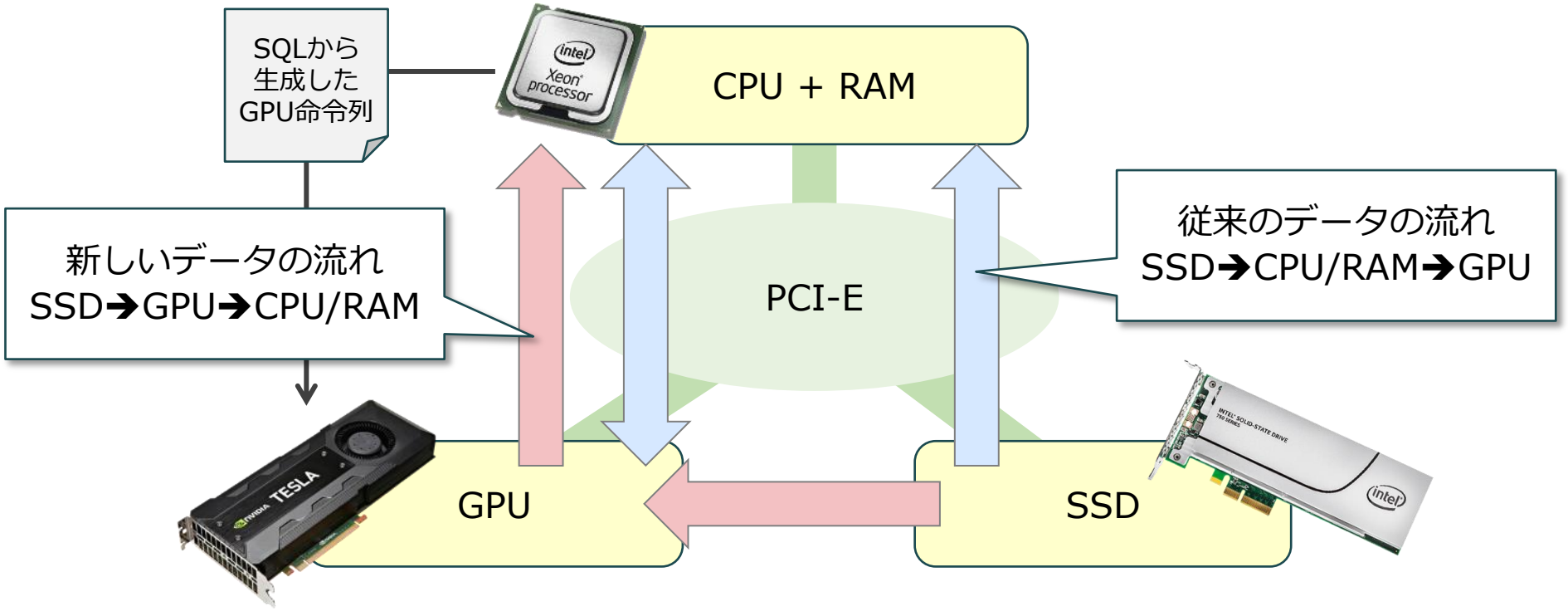
- cstore_fdw ... CitusDataによるFDWモジュールの実装
- v1.3がJul-2015に提供済みだが、INSERT/UPDATE/DELETEなど制約もあり。

Native columnar storage

- Alvaro Herrera/Tomáš Vondra(2ndQuadrant)が提案中
- MVCCのvisibility checkをどうするか？（visibility mapを使うしかない？）



SSD-to-GPU Direct



SSD-to-GPU Directにより期待できる効果

- 条件句に一致しない行を事前に除去できる。
- クエリで参照されない列を事前に除去できる。
- CPU/RAMにロードされた時点で既にJOIN済み・集約済み

Features improvement step by step...


Orchestrating a brighter world