PostgreSQLとArrowとGPUで楽々大量データ処理

HeteroDB,Inc
Chief Architect & CEO
KaiGai Kohei <kaigai@heterodb.com>

自己紹介／会社紹介

ヘテロジニアスコンピューティング技術を**データベース**領域に適用し、
誰もが使いやすく、シンプルで高速な大量データ分析基盤を提供する。

会社概要

- 商号　ヘテロDB株式会社
- 創業　2017年7月4日
- 拠点　品川区北品川5-5-15
  大崎ブライトコア4F
- 事業内容　高速データ処理製品の開発・販売
  GPU＆DB領域の技術コンサルティング

NVIDIA INCEPTION PROGRAM　IPA 情報処理推進機構

代表者プロフィール

- 海外　浩平（KaiGai Kohei）
- OSS開発者コミュニティにおいて、PostgreSQLやLinux kernelの開発に15年以上従事。主にセキュリティ・FDW等の分野でPostgreSQLのメインライン機能の強化に貢献。
- IPA未踏ソフト事業において“天才プログラマー”認定 (2006)
- GPU Technology Conference Japan 2017でInception Awardを受賞

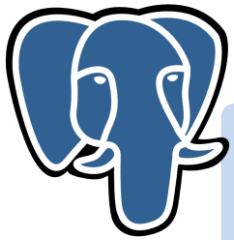
PG-Strom – GPUを用いてSQLを並列処理する

実行結果

SQL パーサ

SQL オプティマイザ

SQL エグゼキュータ



トランザクション　共有バッファ

各種関数・演算子　プロセス間通信

PostgreSQLの各種インフラ

PG-Strom拡張モジュール

代替実行計画作成

GPUコード生成

データの読み出し

GPUカーネル実行

GPUデバイス管理

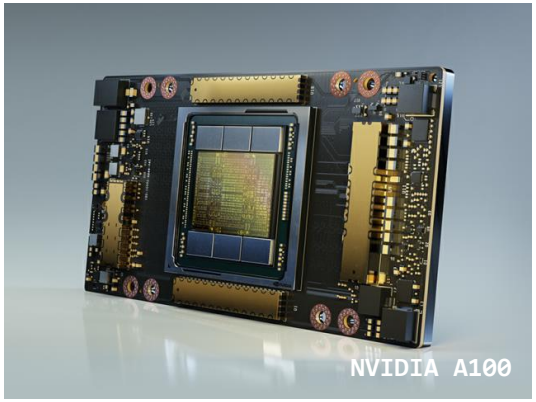
JITコンパイル

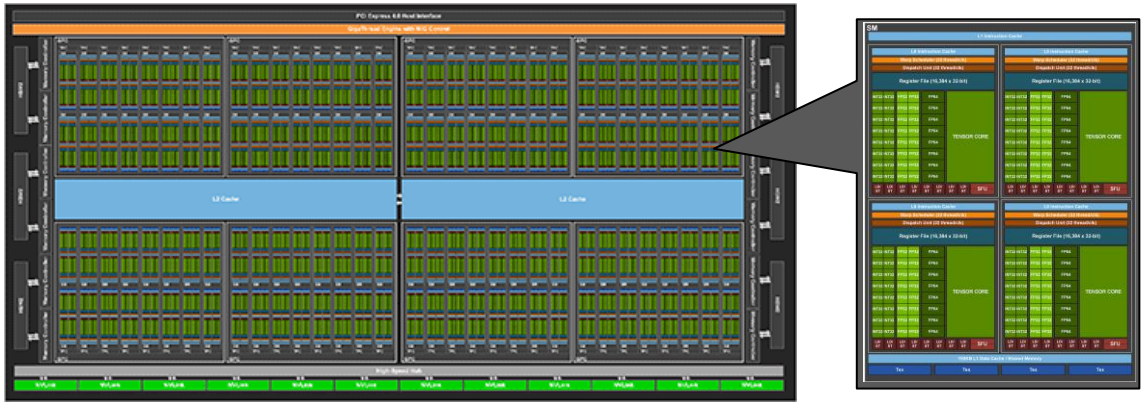
GPU-Direct SQL

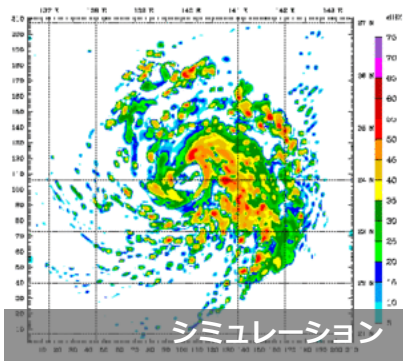
PG-Stromのインフラ機能

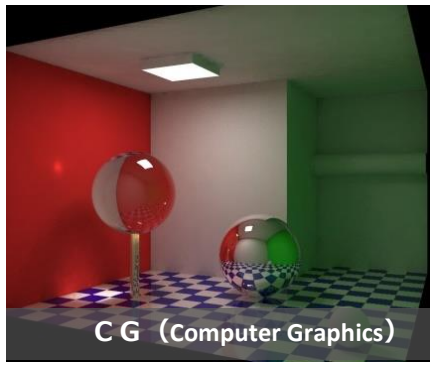
GPUとはどんなプロセッサなのか？

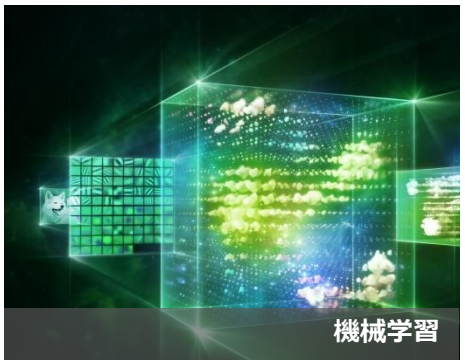
主にHPC分野で実績があり、機械学習用途で爆発的に普及





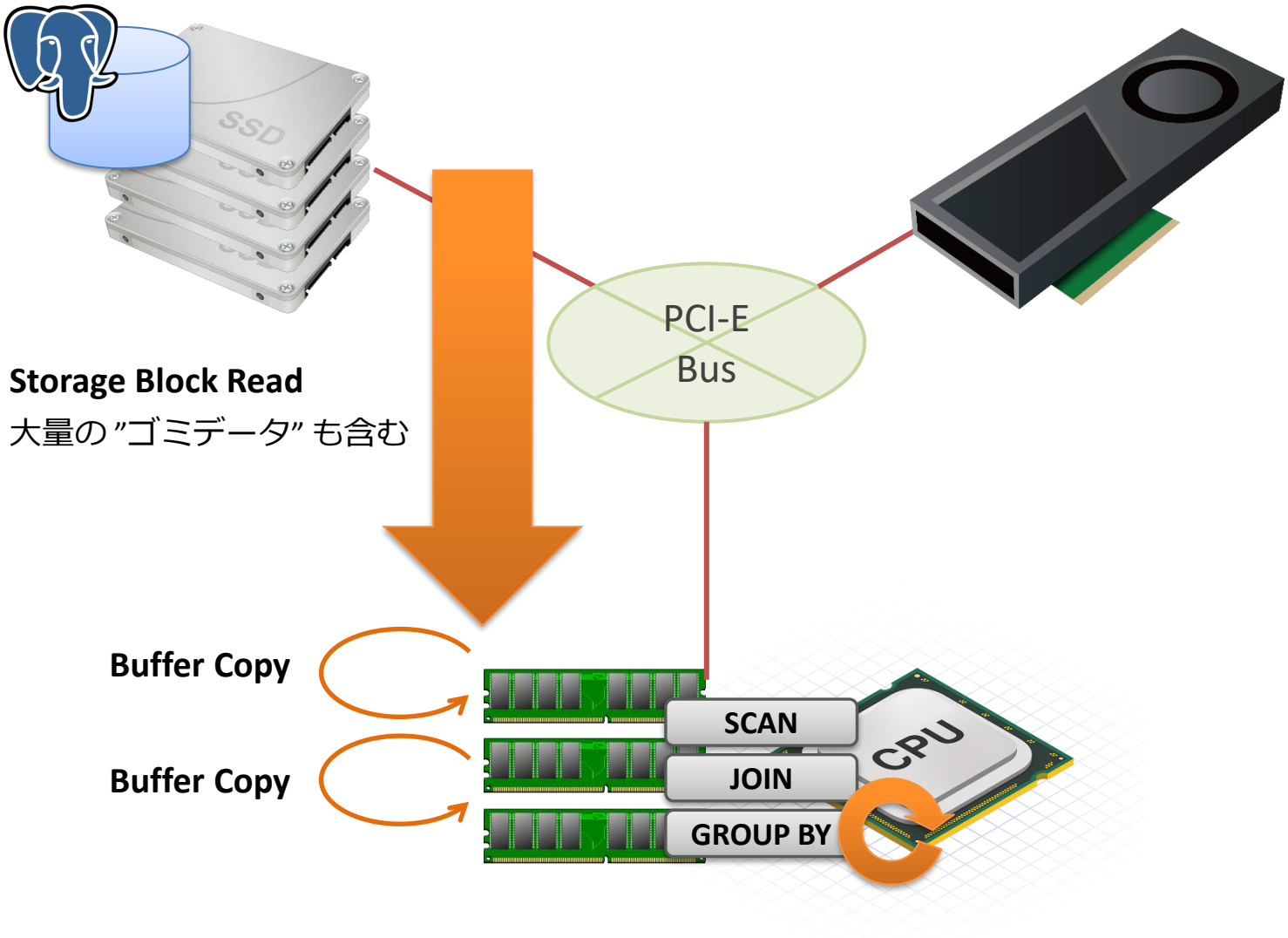






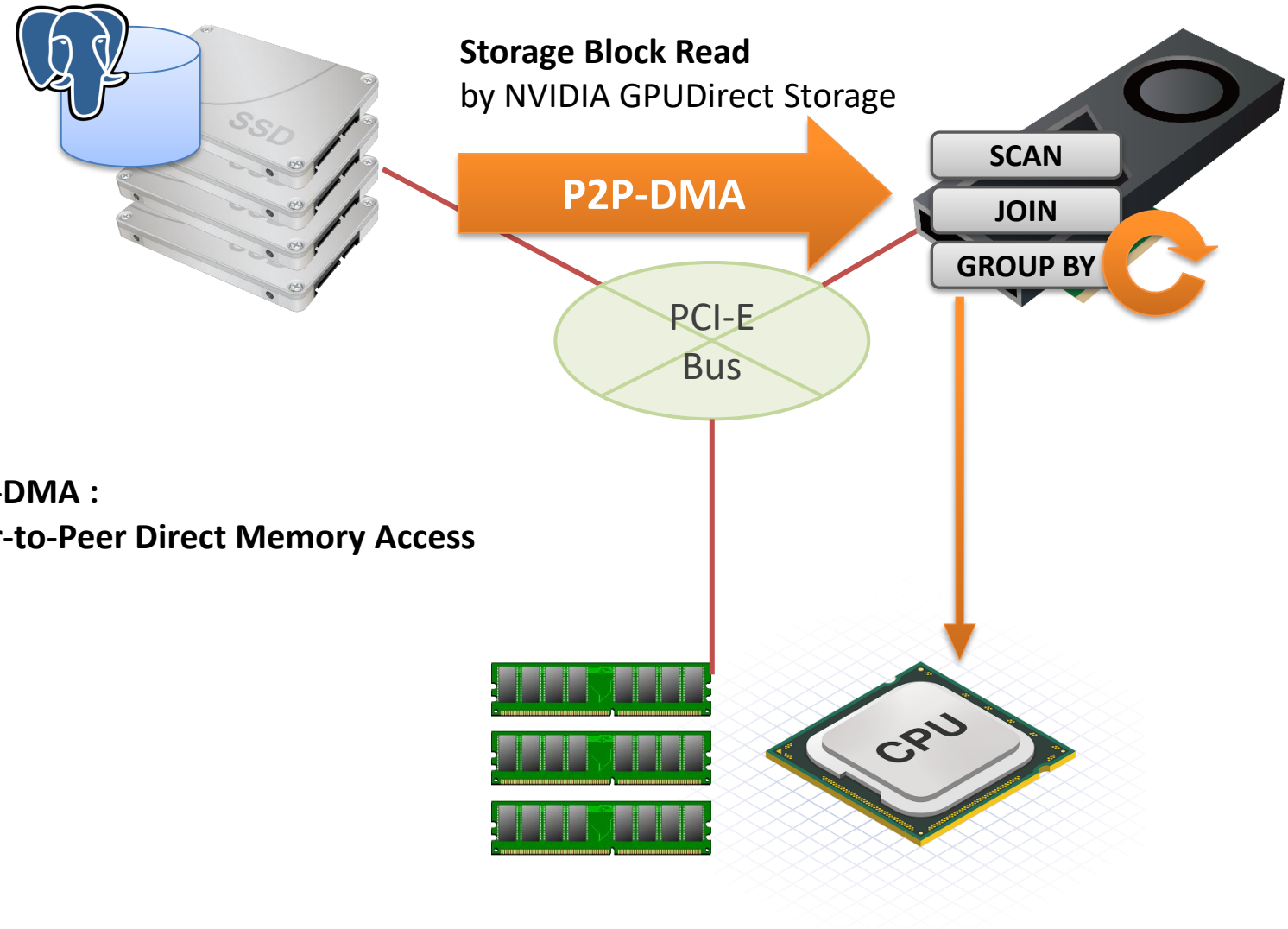
数千コアの並列処理ユニット、TB/sのスループットに達する広帯域メモリを
ワンチップに実装した半導体デバイス。
“**同じ計算を大量のデータに並列実行**” を最も得意とする

GPU-Direct SQL（1/4）



GPU-Direct SQL（2/4）

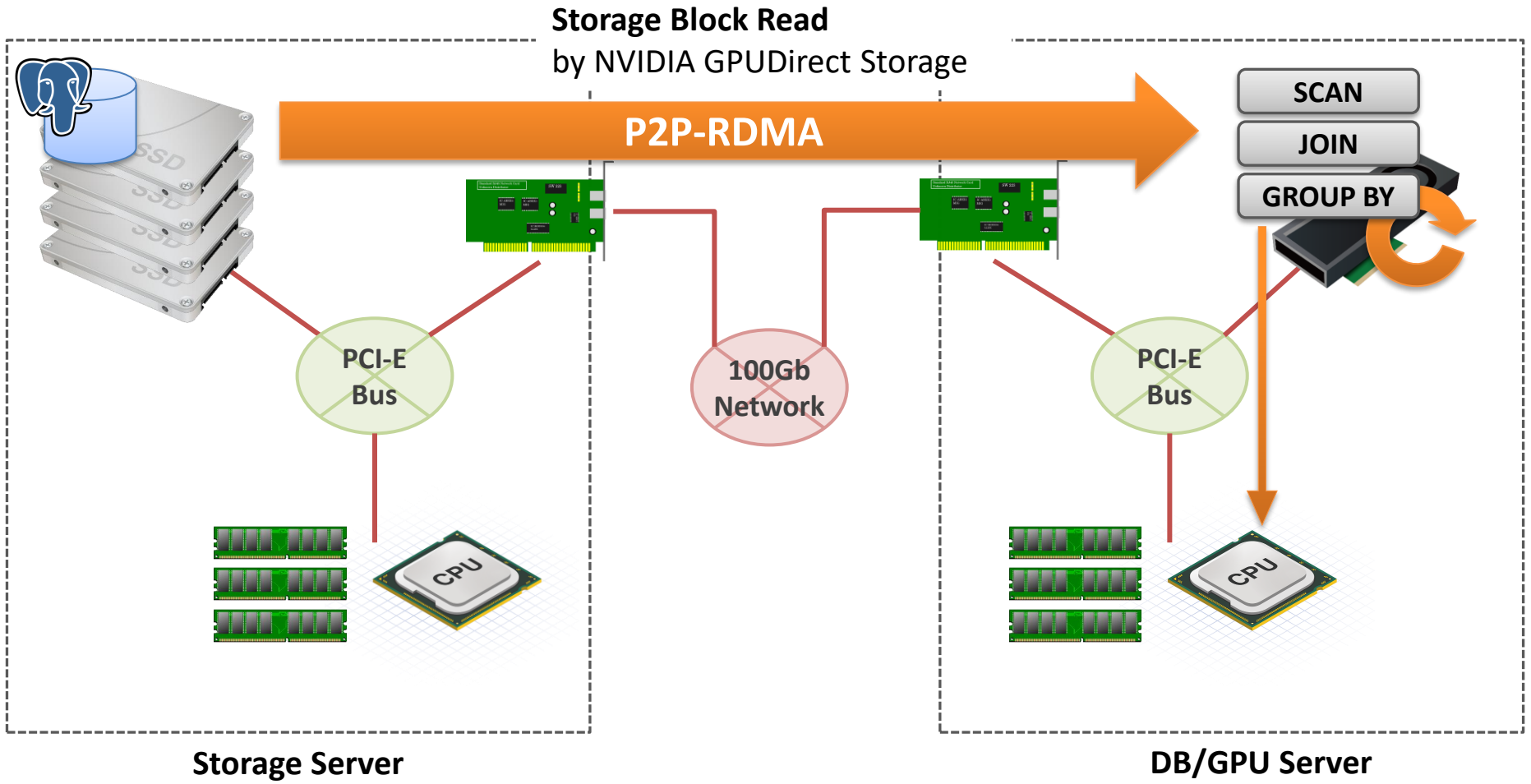
P2P-DMAを利用し、NVME-SSDとGPUを直結してデータ転送



P2P-DMA :
Peer-to-Peer Direct Memory Access

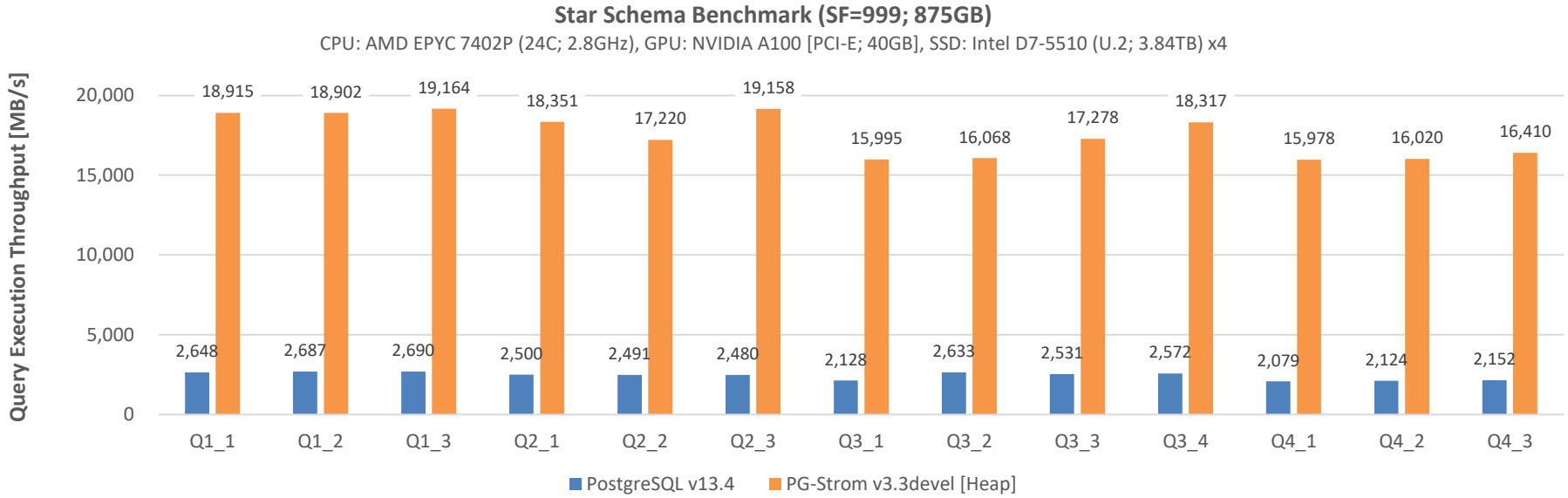
GPU-Direct SQL（3/4）

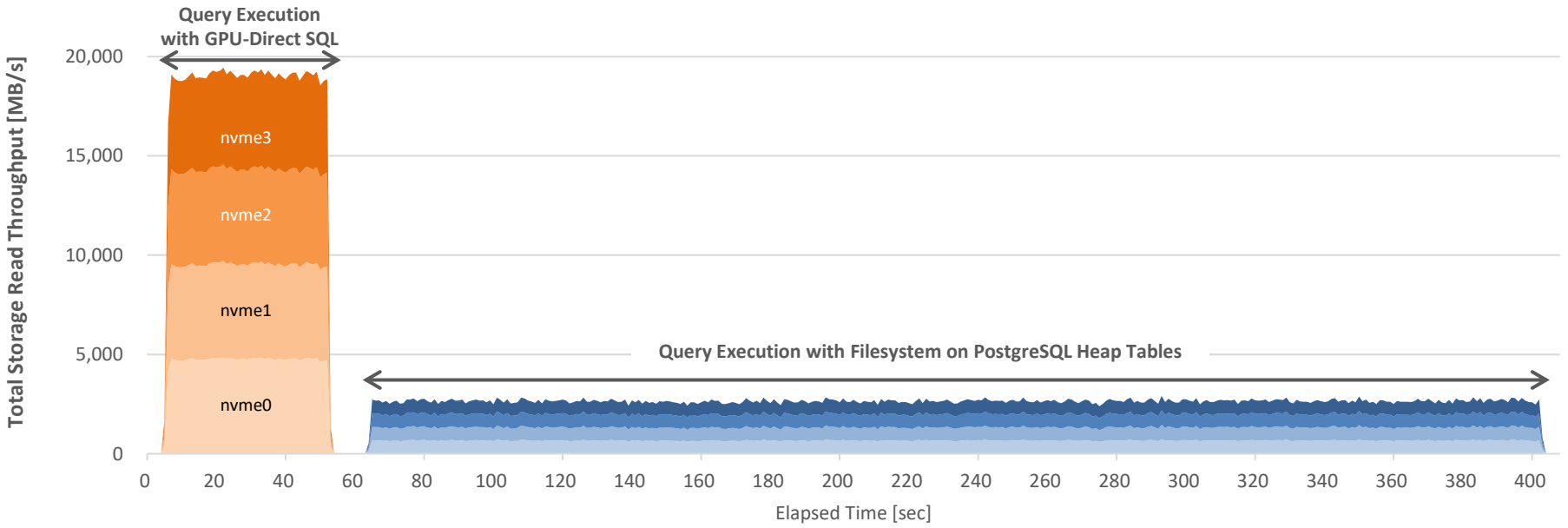
P2P-RDMAを用いて、他ノードのストレージから直接読み出しも可



P2P-RDMA :
Peer-to-Peer Remote Direct Memory Access

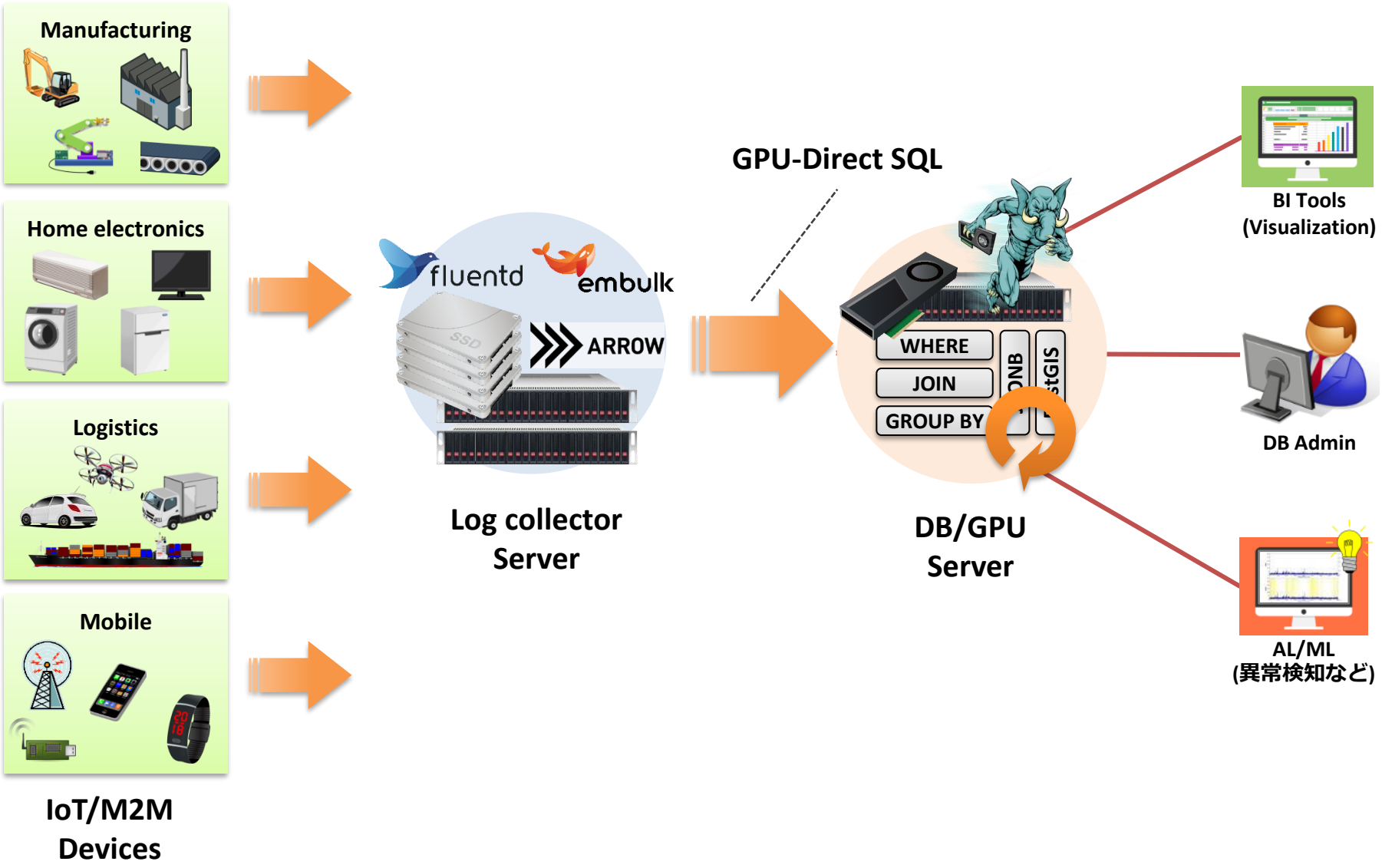
GPU-Direct SQL（4/4）





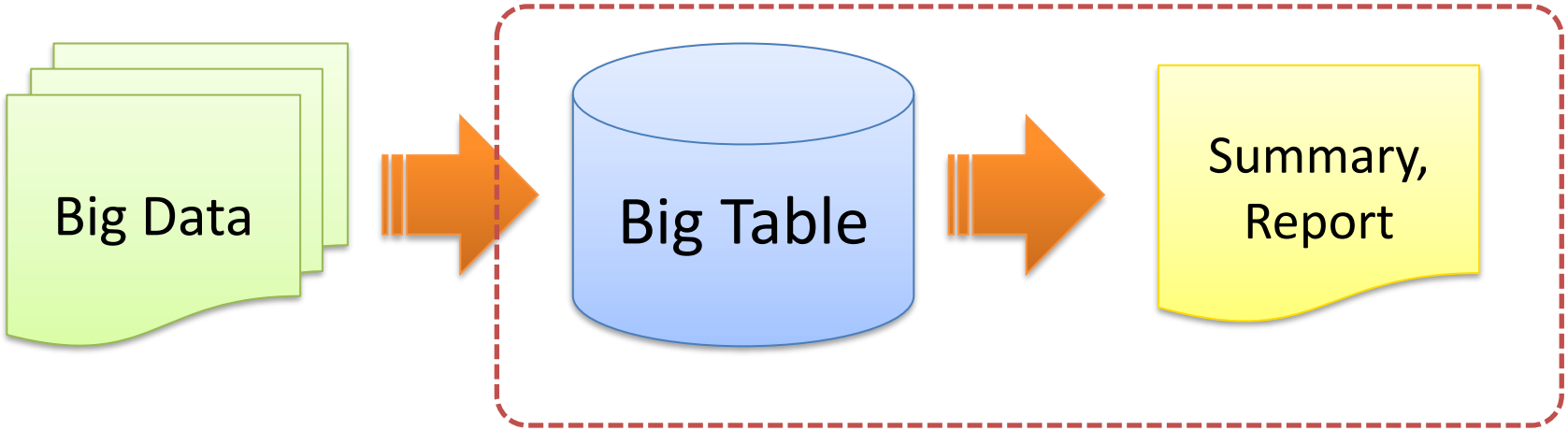
ココまでが前振りです

大量データの処理を考える（1/3）



大量データの処理を考える（2/3）

検索・集計処理がツラい…。



ワークロードの特性

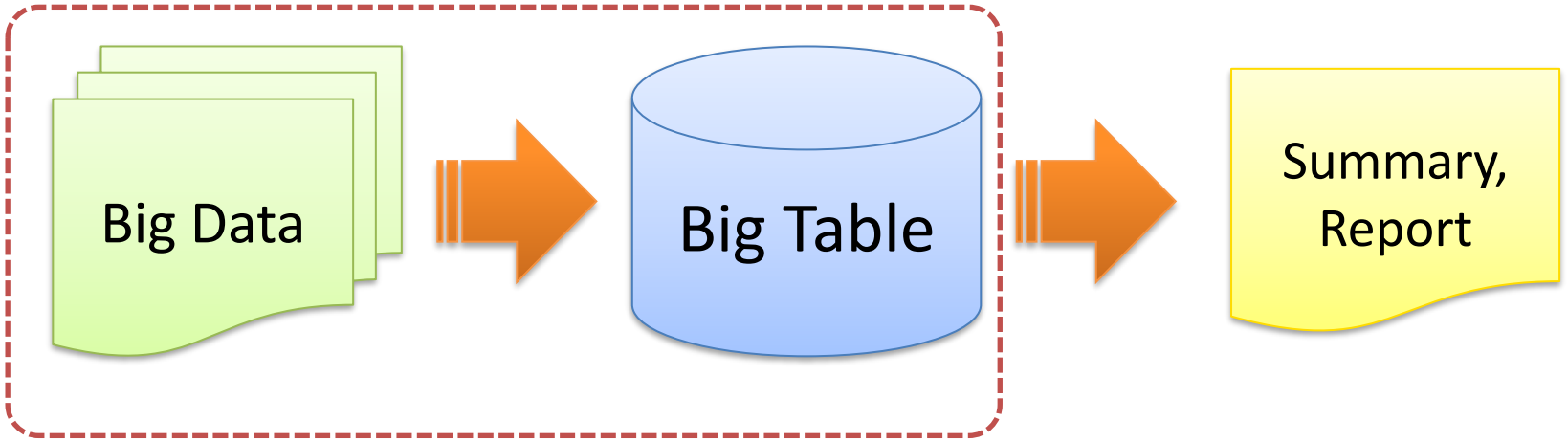
- 多くの場合、テーブルのフルスキャンを伴う。
- テーブル単位のデータ分割が一般的（パーティションなど）
- ストレージからの読み出し速度で律速

どうする？

- **列指向データ**の利用　…　参照しないデータを読み出さない

大量データの処理を考える（3/3）

そもそもDBへのインポートもツラい…。



ワークロードの特性

- 大量の元データを読み出して、**DBの内部形式**に変換してテーブルへ書き込み。
- データ形式を変換するCPU、ストレージへの書き込み速度で律速
- 並列度を上げにくいことも多い（順次読み出し前提のデータ形式）

どうする？

- **外部データ**を “そのまま” 読み出す　　…　インポート自体の必要性をなくす

Apache Arrowデータ形式（1/3）

列指向の構造化データ形式

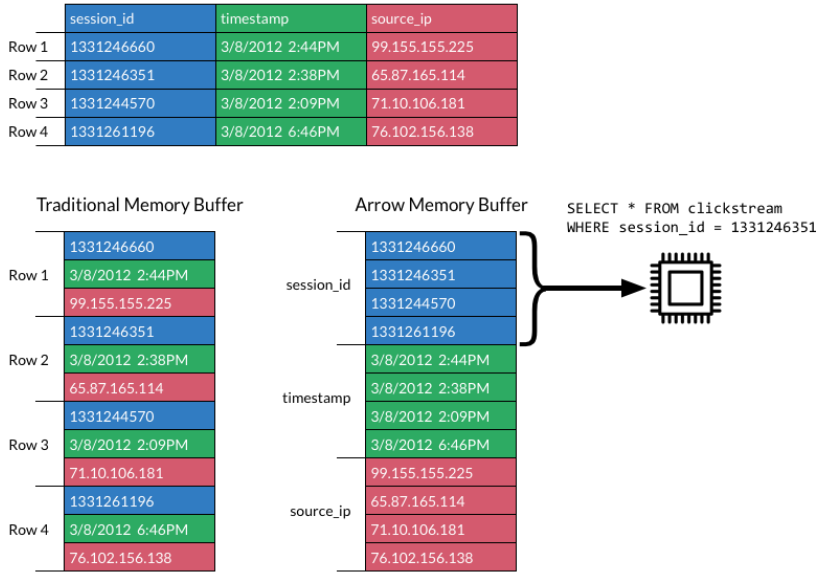
- 被参照列のみ読み出す（= I/O量の削減）が可能
- データが隣接しているため、SIMD命令やGPUでの処理性能を引き出しやすい

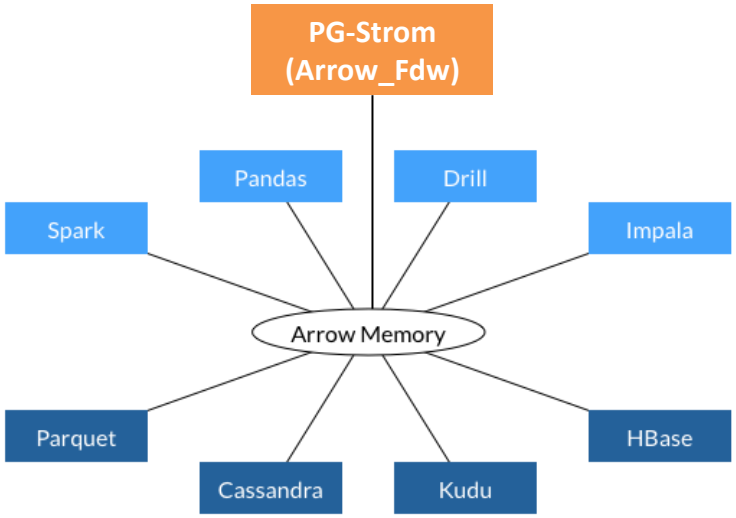
多くのビッグデータ系OSSで対応が進んでいる

- SparkやDrill、Python(PyArrow)など。PG-Strom (Arrow_Fdw) もその一つ。

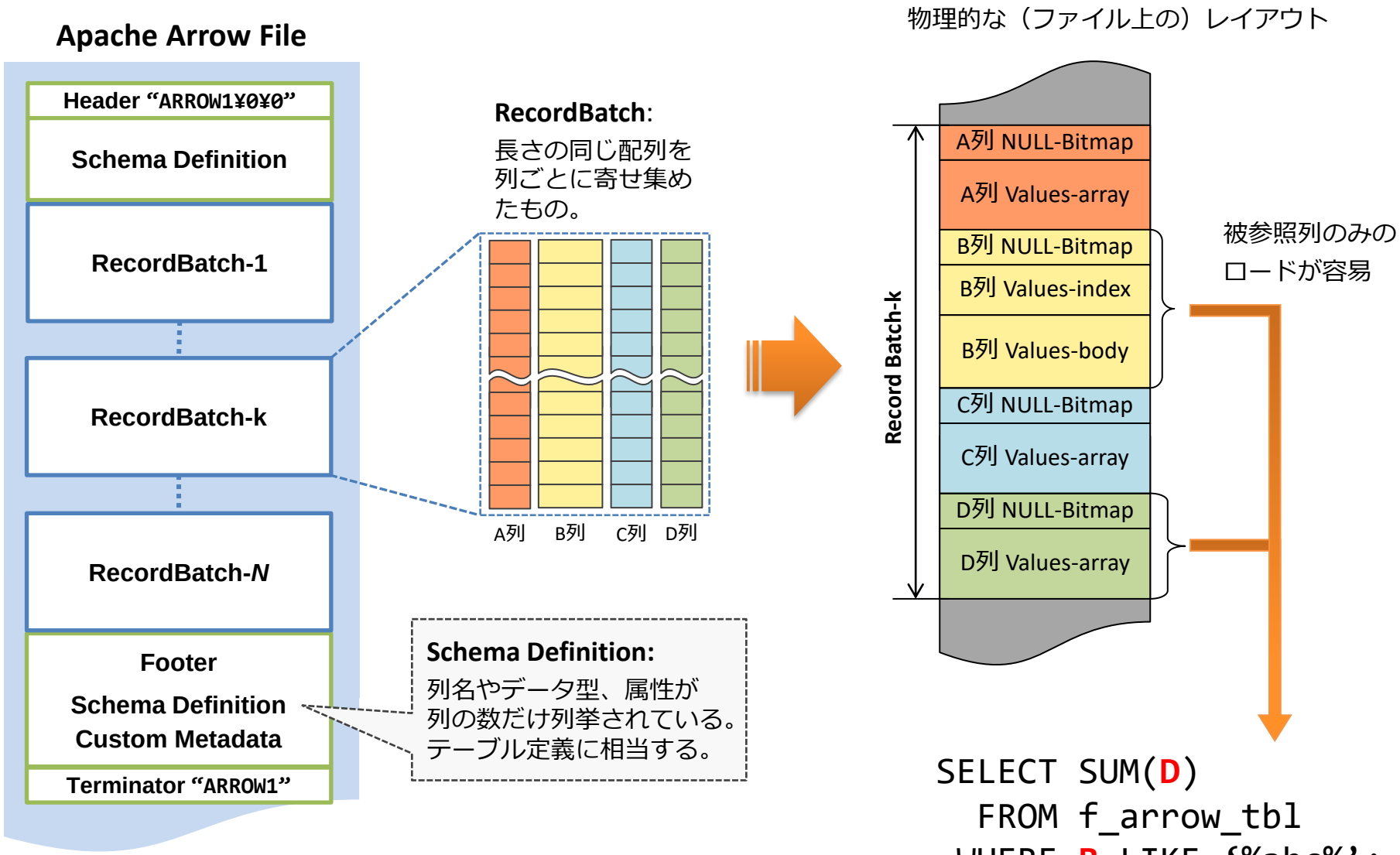
留意点

- 更新はほぼ不可能。追記のみ（Insert-Only）と考えた方がよい



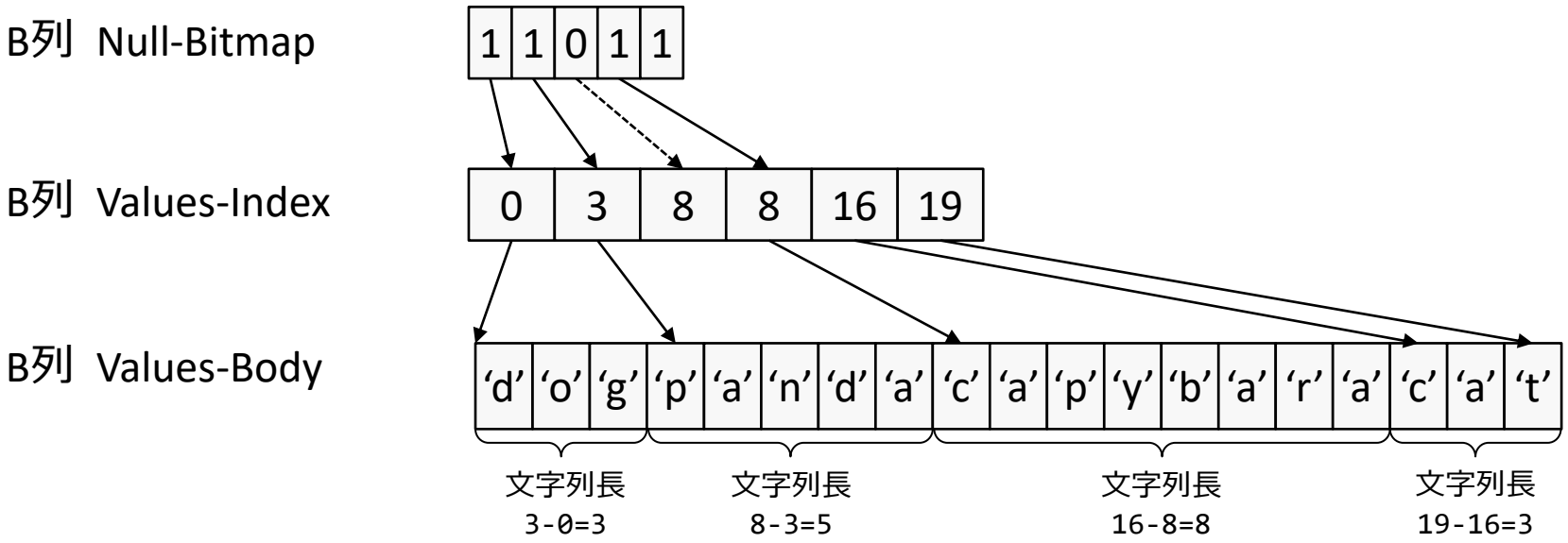


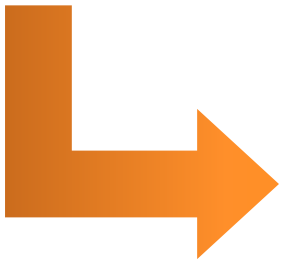
Apache Arrowデータ形式（2/3）




```
SELECT SUM(D)
  FROM f_arrow_tbl
 WHERE B LIKE ‘%abc%’;
```

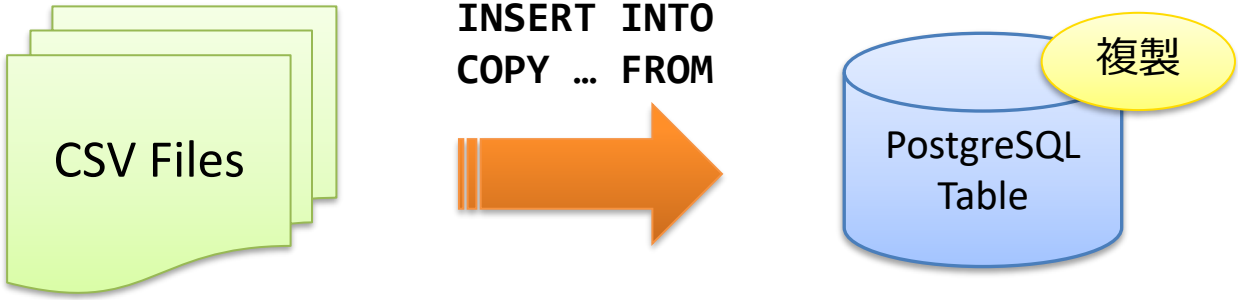
Apache Arrowデータ形式（3/3）　補足：可変長データ





B[0] = ‘dog’
B[1] = ‘panda’
B[2] = NULL
B[3] = ‘capybara’
B[4] = ‘cat’

Apache Arrowならデータのインポートが不要



- ✓ CSVなど外部ファイルを、読み出し、データ形式を変換し、PostgreSQLのテーブルに複製を書き込むことになる。
- ✓ データの件数が多い場合、非常に時間を要する処理になる。

- ✓ PostgreSQLのFDW機能（Arrow_Fdw）を使用して、Arrowファイルに紐付けた外部テーブル（Foreign Table）を定義する。
- ✓ データの移動を伴わないため、一瞬で完了する。（Arrowファイル自体の移動は OS 上のファイルコピー）

Arrow_Fdwによる外部テーブルの定義（1/3）

```
postgres=# drop foreign table f_mytest ;
DROP FOREIGN TABLE
postgres=# CREATE FOREIGN TABLE f_mytest (
             id int,
             ts timestamp,
             x  float8,
             y  text,
             z  numeric
           ) SERVER arrow_fdw
             OPTIONS (file '/tmp/mytest.arrow');
CREATE FOREIGN TABLE

postgres=# SELECT * FROM f_mytest;
 id |            ts             |         x          |      y       |    z
----+---------------------------+--------------------+--------------+---------
  1 | 2022-08-02 00:34:44.210589 |   77.4344383633856 | 65ac7f6      | 38.6218
  2 | 2019-05-11 03:06:16.353798 |  95.33235230265761 | 9105319395   | 51.8267
  3 | 2015-01-04 11:02:25.66779  |  93.67415248121794 | b56930f7834  | 84.9033
  :             :                           :                :           :
 24 | 2022-09-24 06:26:02.058316 | 17.668632938372266 | c5e35        | 55.9739
 25 | 2016-08-08 18:16:12.248363 |   92.2211769466387 | fa889dd51692 |  19.246
(25 rows)
```

Arrow_Fdwによる外部テーブルの定義（2/3）

IMPORT FOREIGN SCHEMAで、Arrowファイルのスキーマ定義ごと取り込むのが楽

```
postgres=# import foreign schema f_mytest
             from server arrow_fdw
             into public options (file '/tmp/mytest.arrow');
IMPORT FOREIGN SCHEMA

postgres=# ¥d f_mytest
                          Foreign table "public.f_mytest"
 Column |            Type             | Collation | Nullable | Default | FDW options
--------+-----------------------------+-----------+----------+---------+-------------
 id     | integer                     |           |          |         |
 ts     | timestamp without time zone |           |          |         |
 x      | double precision            |           |          |         |
 y      | text                        |           |          |         |
 z      | numeric                     |           |          |         |
Server: arrow_fdw
FDW options: (file '/tmp/mytest.arrow’)
```

Arrow_Fdwによる外部テーブルの定義（3/3）

ファイル形式は共通なので、当然、Pythonでも普通に読める

```
$ python3 -q
>>> import pyarrow as pa
>>> X = pa.RecordBatchFileReader('/tmp/mytest.arrow')
>>> X.schema
id: int32
  -- field metadata --
  min_values: '1'
  max_values: '25'
ts: timestamp[us]
x: double
y: string
z: decimal(30, 8)
-- schema metadata --
sql_command: 'SELECT * FROM mytest'
>>> X.get_record_batch(0).to_pandas()
    id                         ts          x             y           z
0    1 2022-08-02 00:34:44.210589  77.434438       65ac7f6  38.62180000
1    2 2019-05-11 03:06:16.353798  95.332352    9105319395  51.82670000
2    3 2015-01-04 11:02:25.667790  93.674152   b56930f7834  84.90330000
3    4 2017-02-10 21:05:26.631906  90.553723          2103  86.78170000
:    :                          :         :             :           :
22  23 2019-05-19 02:06:38.668702  47.295458  6e00a6e037ad  11.02260000
23  24 2022-09-24 06:26:02.058316  17.668633         c5e35  55.97390000
24  25 2016-08-08 18:16:12.248363  92.221177  fa889dd51692  19.24600000
```

Arrowファイルの作り方（1/4）　RDBMSから

pg2arrowによる PostgreSQL からのダンプ

```
[kaigai@magro ~]$ pg2arrow -d postgres ¥
                          -c "select * from lineorder ¥
                                  where lo_orderpriority in ('1-URGENT','2-HIGH','3-MEDIUM') ¥
                                  order by lo_orderdate" ¥
                          -o /tmp/flineorder.arrow --progress ¥
                          --stat=lo_orderdate
RecordBatch[0]: offset=1712 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[1]: offset=268438792 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[2]: offset=536875872 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[3]: offset=805312952 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[4]: offset=1073750032 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[5]: offset=1342187112 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[6]: offset=1610624192 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[7]: offset=1879061272 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[8]: offset=2147498352 length=76206296 (meta=920, body=76205376) nitems=369928
$ ls -lh /tmp/flineorder.arrow
-rw-r--r--. 1 kaigai users 2.1G Oct 12 20:17 /tmp/flineorder.arrow
```

- ✓ -c オプションで指定したクエリの実行結果をArrowファイルとして保存
- ✓ 単純なダンプと異なり、検索条件を指定したり、ソートやJOINなども可能
- ✓ デフォルトで 256MB 毎に RecordBatch を作る（-s オプションで変更可能）
- ✓ --stat オプションはmin/max統計情報の採取を有効にする。（後述）

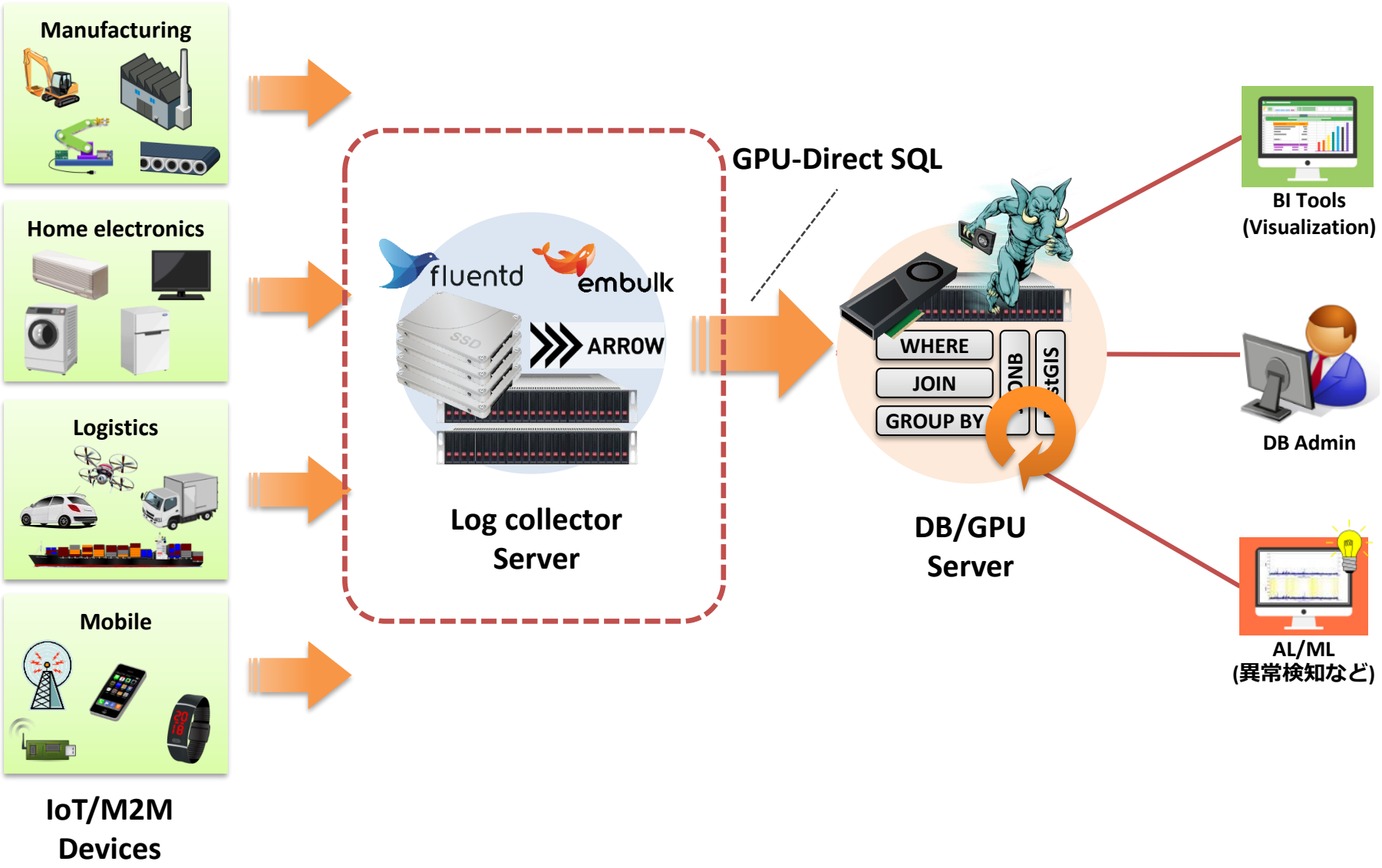
Arrowファイルの作り方（2/4）　NICから

pcap2arrowによるネットワークキャプチャ

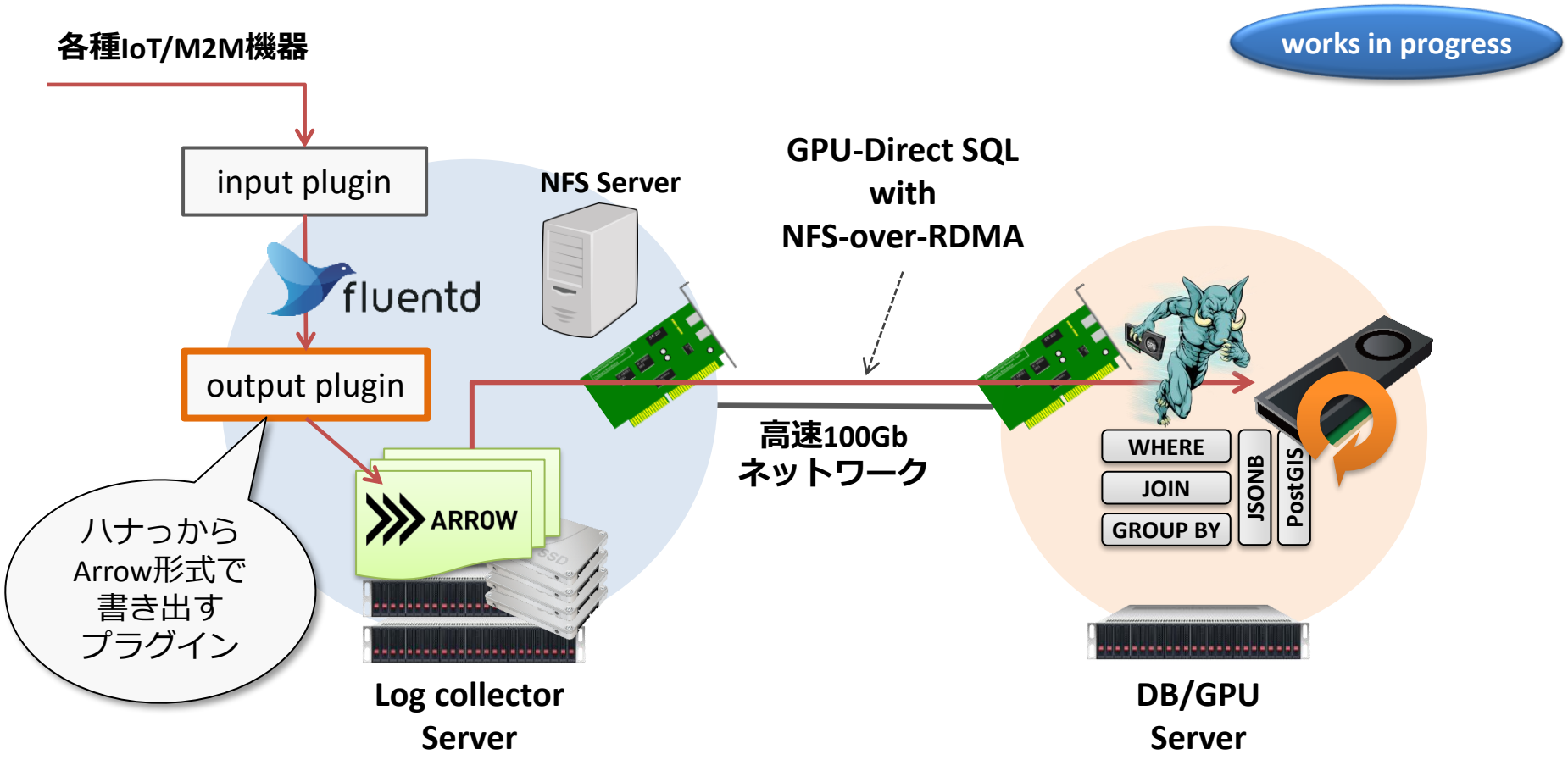
```
# pcap2arrow -i lo,eno1 -o /tmp/my_packets.arrow -p 'tcp4,udp4,icmp4’
^C
# ls -lh /tmp/my_packets.arrow
-rw-r--r--. 1 root root 57K 10月 12 22:19 /tmp/my_packets.arrow

$ psql postgres
postgres=# import foreign schema my_packets from server arrow_fdw
                        into public options (file '/tmp/my_packets.arrow');
IMPORT FOREIGN SCHEMA
postgres=# select timestamp, src_addr, dst_addr, protocol, length(payload) from my_packets;
         timestamp          |    src_addr    |    dst_addr     | protocol | length
----------------------------+----------------+-----------------+----------+--------
 2021-10-12 13:18:22.684989 | 192.168.77.72  | 192.168.77.106  | 6        |      6
 2021-10-12 13:18:31.531827 | 192.168.77.72  | 192.168.77.106  | 6        |     36
 2021-10-12 13:19:00.000605 | 192.168.77.254 | 239.255.255.250 | 17       |    472
 2021-10-12 13:19:00.000929 | 192.168.77.254 | 239.255.255.250 | 17       |    448
           :                          :                :          :          :
```

- ✓ -i で指定したネットワークデバイスからパケットをキャプチャし、Arrowファイルとして書き出す。
- ✓ スキーマ構造は -p で指定したプロトコルのヘッダ構造に準ずる。
- ✓ PF_RINGドライバを使用しており、50Gbps近辺までのキャプチャまでは実証済。

Arrowファイルの作り方（3/4）



Arrowファイルの作り方（4/4）汎用ログコレクタから【開発中】



- ✓ IoT/M2M機器から受信したログを、直接 Arrow ファイルとして出力（現在、fluentd向けプラグインを開発中）
- ✓ Arrowファイルは、高速100Gbネットワークを介してGPU-Direct SQLでDB/GPUサーバに転送する。

Apache Arrowを使うメリット

列形式の構造化データ

- 分析処理の際に、被参照列のみを読み出す事が容易であるため、
  大量データの分析時にボトルネックとなる I/O ワークロードの軽減に繋がる。
  ※ 特に、センサデータの場合フィールド数が非常に多い事がある。
- 固定長データの場合、GPUやSIMD命令でプロセッサの性能を引き出しやすい
  構造をしている。

インポートが不要

- 外部のデータファイルを「そのまま」読み出せるため、わざわざ、
  PostgreSQL内部のデータ形式に変換して複製を作る必要はない。

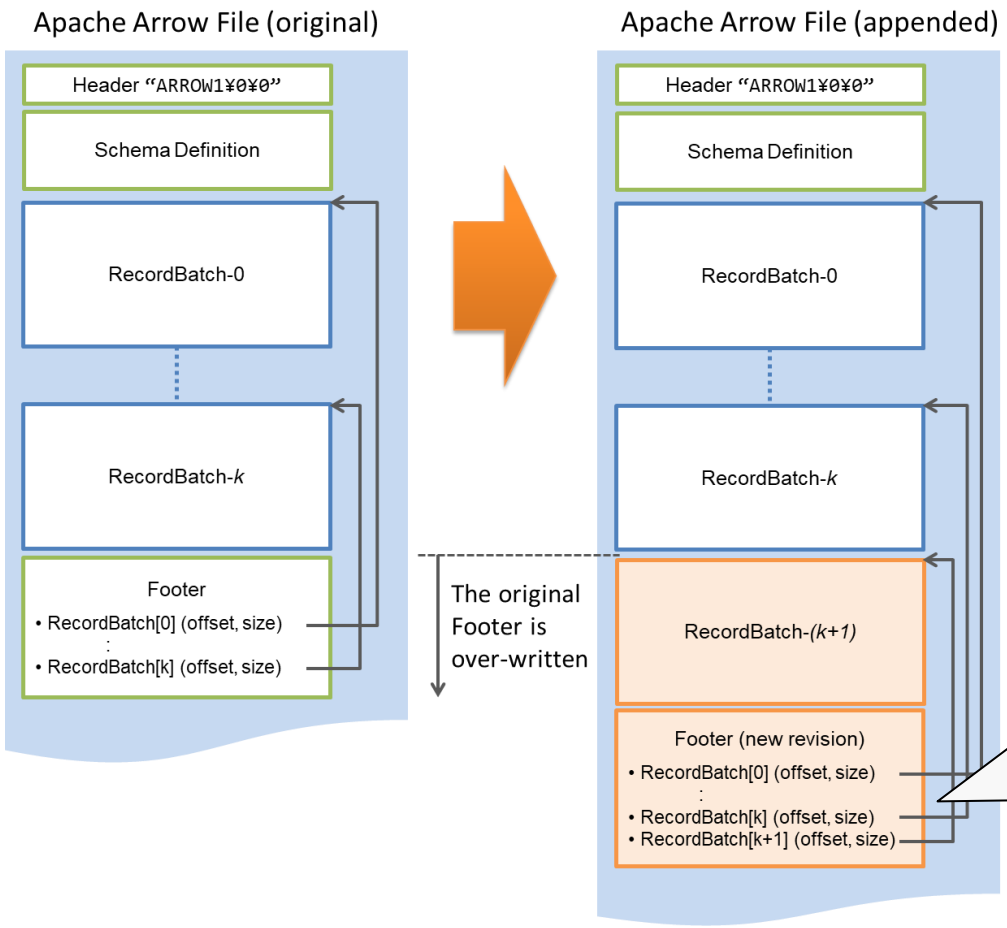
長期保管にも適する

- 長期保管の際は、zipで固めて安価なストレージに放り込んでおくだけでよい。
- 圧縮自体をフォーマットに組み込んだ Parquet よりは一手間増えるが、
  検索・分析処理フェーズでの展開処理とのトレードオフ
- ファイル自体がスキーマ定義を内包するため、当時のスキーマ定義散逸したり
  して、読めなくなるという事がない。

Apache Arrowを使う上での留意点

追記のみ（Insert-only）のデータ形式である事

- RDBMSのような行単位の更新・削除は実装が難しい
- Arrow_Fdwでも追記または全削除（Truncate）にしか対応していない。



追記は簡単

元のフッタを上書きして、新たにRecordBatch[k+1]のポインタを持つフッタを付ければよい。

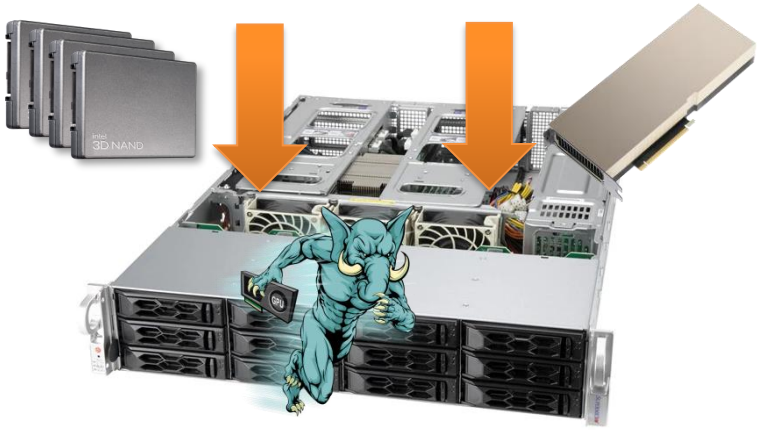
上書き前のフッタを保持しておき、元の位置に書き戻してファイルサイズを元に戻せば、ロールバック処理も簡単に実装できる。

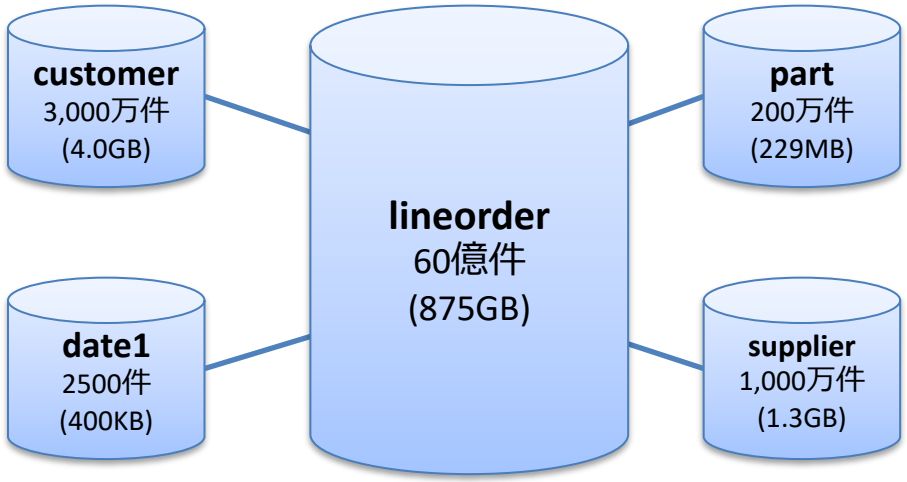
PG-Strom + Arrowのベンチマーク（1/3）

NVIDIA A100x1 と PCIe 4.0世代のNVME-SSD 4枚のストライピング構成

SeqRead: 6500MB/s
Intel SSD D7-5510 [3.84TB] x4

6912コア搭載
NVIDIA A100 [40GB; PCI-E]



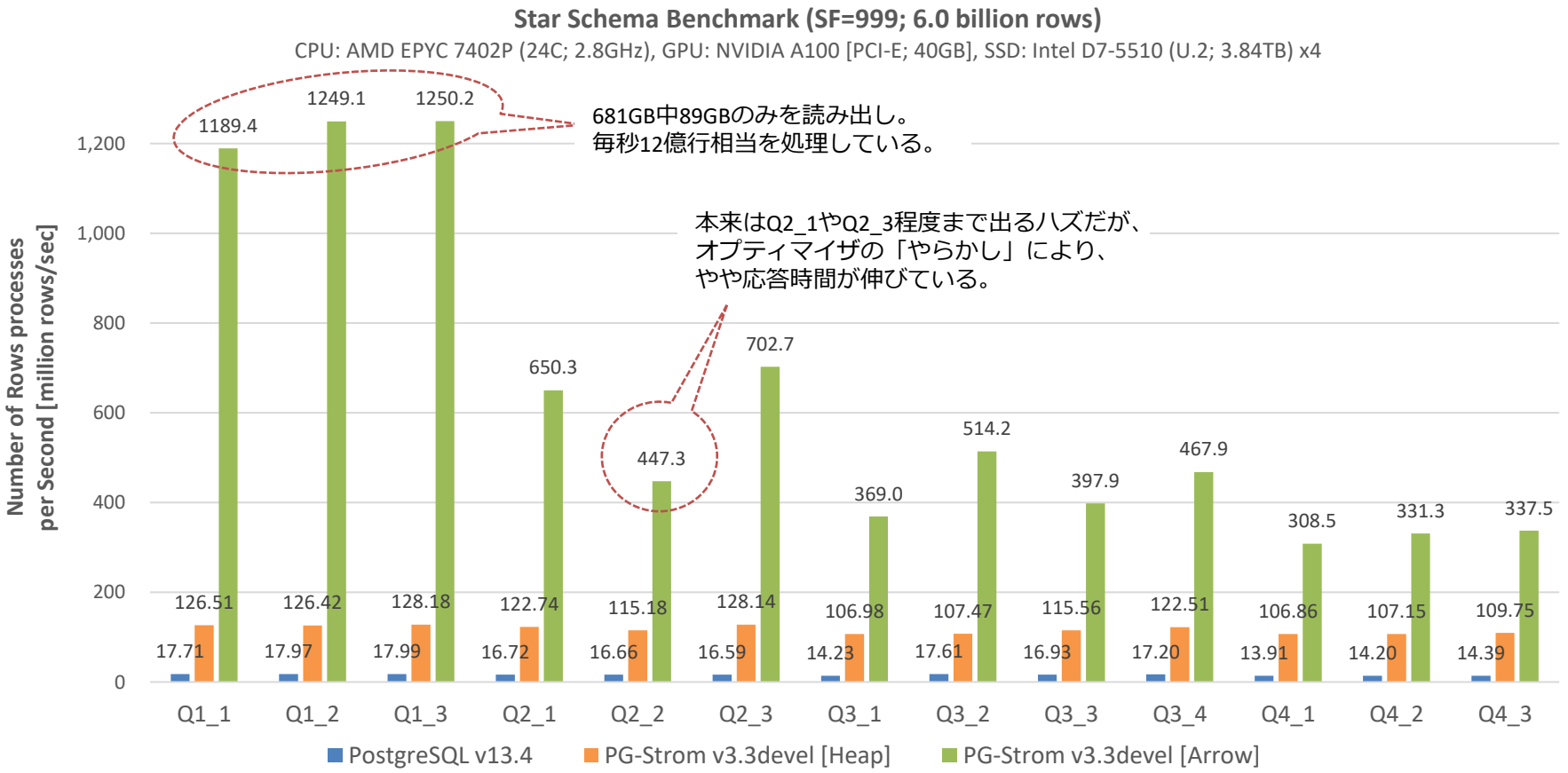


測定環境	
Chassis	Supermicro AS-2014CS-TR
CPU	AMD EPYC 7402P (24C/2.85GHz)
RAM	128GB [16GB DDR4-3200; ECC] x8
GPU	NVIDIA A100 (6912C; 40GB; PCI-E) x1
SSD	Intel SSD D7-5510 (U.2; 3.84TB) x4
OS	Red Hat Enterprise Linux 8.4 CUDA 11.5 + driver 495.29.05
DB	PostgreSQL v13.x + PG-Strom v3.3devel

■ クエリ例

```
SELECT sum(lo_revenue), d_year, p_brand1
  FROM lineorder, date1, part, supplier
 WHERE lo_orderdate = d_datekey
   AND lo_partkey = p_partkey
   AND lo_suppkey = s_suppkey
   AND p_category = 'MFGR#12‘
   AND s_region = 'AMERICA‘
 GROUP BY d_year, p_brand1
 ORDER BY d_year, p_brand1;
```

PG-Strom + Arrowのベンチマーク（2/3）



- PostgreSQL（CPU並列）より圧倒的に速かった GPU-Direct SQL より圧倒的に速い GPU-Direct SQL on Apache Arrow
- 約681GB、60億件のArrowファイルの処理に要したのは5～20秒程度
- 列データ構造とGPUでの並列処理、PCI-E 4.0の広帯域の効果といえる。

ところで、、、

さっきのコレ、覚えてますか？

```
[kaigai@magro ~]$ pg2arrow -d postgres ¥
                          -c "select * from lineorder ¥
                                 where lo_orderpriority in ('1-URGENT','2-HIGH','3-MEDIUM') ¥
                                 order by lo_orderdate" ¥
                          -o /tmp/flineorder.arrow --progress ¥
                          --stat=lo_orderdate

RecordBatch[0]: offset=1712 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[1]: offset=268438792 length=268437080 (meta=920, body=268436160) nitems=1303085
RecordBatch[2]: offset=536875872 length=268437080 (meta=920, body=268436160) nitems=1303085
      :               :               :               :         :                 :
```

コレ

- ✓ --stat オプションはmin/max統計情報の採取を有効にする。

どういうことか？

- ✓ 各RecordBatchが保持する約130件ごとに、**lo_orderdate**列の最大値／最小値をそれぞれ記録し、カスタムメタデータとして埋め込んでおく。
- ✓ 最大値／最小値が分かれば、明らかに検索条件に合致しないRecordBatchを読み飛ばすことができる。
- ✓ ログデータのタイムスタンプには効果が高い。

独自機能：min/max統計情報（1/3）

CustomMetadataフィールドを利用して、統計情報を Arrow ファイルに埋め込む。

```
$ python3 -q
>>> import pyarrow as pa
>>> X = pa.RecordBatchFileReader('/tmp/flineorder.arrow’)
>>> X.num_record_batches
9
>>> X.schema
lo_orderkey: decimal(30, 8)
lo_linenumber: int32
lo_custkey: decimal(30, 8)
lo_partkey: int32
lo_suppkey: decimal(30, 8)
lo_orderdate: int32
  -- field metadata --
  min_values: '19920101,19921018,19930804,19940521,19950307,19951223,1996' + 22
  max_values: '19921018,19930804,19940521,19950307,19951223,19961009,1997' + 22
lo_orderpriority: fixed_size_binary[15]
lo_shippriority: fixed_size_binary[1]
lo_quantity: decimal(30, 8)
lo_extendedprice: decimal(30, 8)
lo_ordertotalprice: decimal(30, 8)
lo_discount: decimal(30, 8)
lo_revenue: decimal(30, 8)
lo_supplycost: decimal(30, 8)
  :                 :
```

lo_orderdate列のRecordBatchごと最大値／最小値を、カスタムメタデータとして埋め込んでいる。
未対応のアプリケーションの場合、単純に無視される。

独自機能：min/max統計情報（2/3）

明らかに検索条件に合致しない RecordBatch を読み飛ばしているのが分かる

```
postgres=# import foreign schema flineorder
                    from server arrow_fdw
                    into public options (file '/tmp/flineorder.arrow');
IMPORT FOREIGN SCHEMA

postgres=# explain (analyze, costs off)
           select count(*),sum(lo_revenue)
             from flineorder
            where lo_orderdate between 19930101 and 19931231;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Aggregate (actual time=36.967..36.969 rows=1 loops=1)
   ->  Custom Scan (GpuPreAgg) on flineorder (actual time=36.945..36.950 rows=1 loops=1)
         Reduction: NoGroup
         Outer Scan: flineorder (actual time=10.693..20.367 rows=2606170 loops=1)
         Outer Scan Filter: ((lo_orderdate >= 19930101) AND (lo_orderdate <= 19931231))
         Rows Removed by Outer Scan Filter: 966184
         referenced: lo_orderdate, lo_revenue
         Stats-Hint: (lo_orderdate >= 19930101), (lo_orderdate <= 19931231) [loaded: 2, skipped: 7]
         files0: /tmp/flineorder.arrow (read: 206.00MB, size: 2120.69MB)
 Planning Time: 0.175 ms
 Execution Time: 42.146 ms
(11 rows)
```

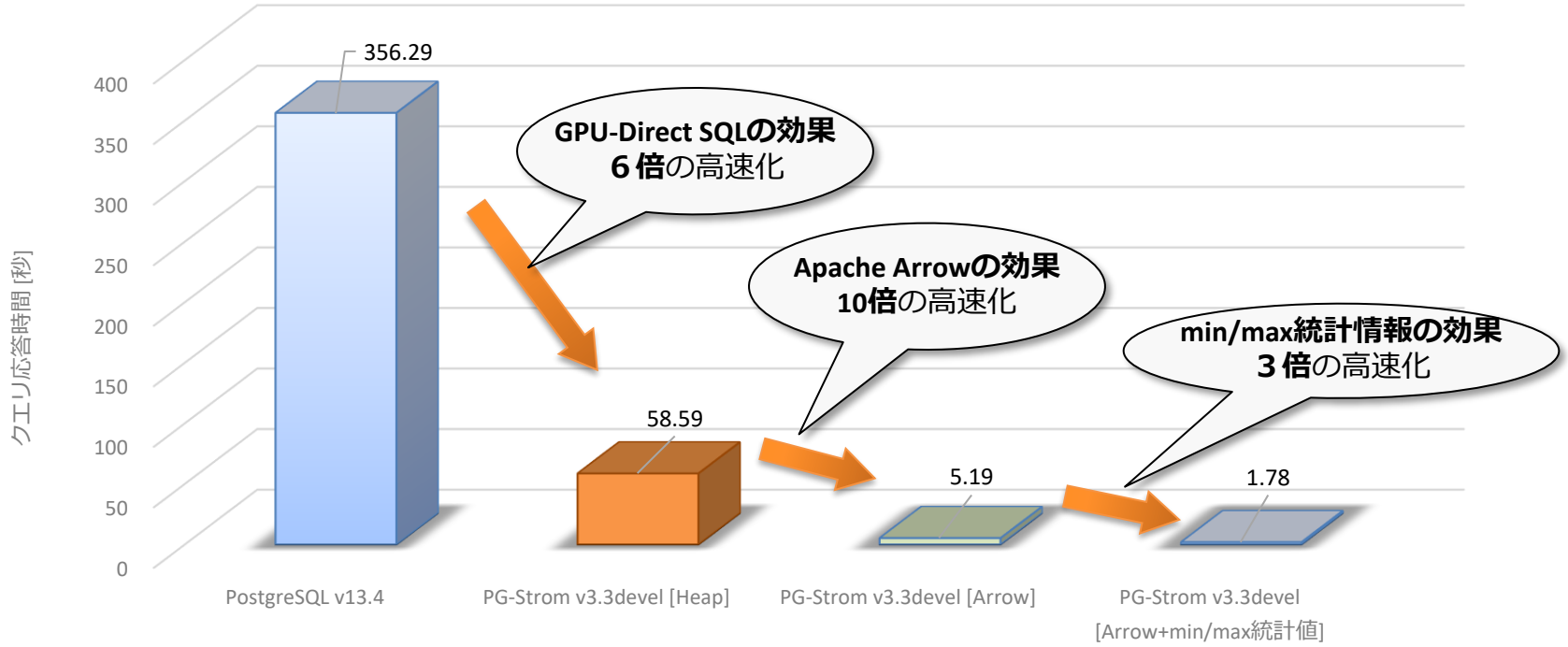
コレ

独自機能：min/max統計情報（3/3）

統計情報を活用するよう、SSBMクエリQ1_1を微修正

```
select sum(lo_extendedprice*lo_discount)
from lineorder,date1
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between 1 and 3
and lo_quantity < 25;
```

```
select sum(lo_extendedprice*lo_discount)
from lineorder
where lo_orderdate between 19930101
                       and 19931231
and lo_discount between 1 and 3
and lo_quantity < 25;
```



実ワークロードにおける効果
～国立天文台様との共同検証～

背景

国立天文台　ハワイ観測所　すばる望遠鏡

（画像提供：国立天文台様）

観測データ
（高解像度画像）

観測データ
（画像と天体のメタデータ）

様々な検索条件で
画像データを抽出

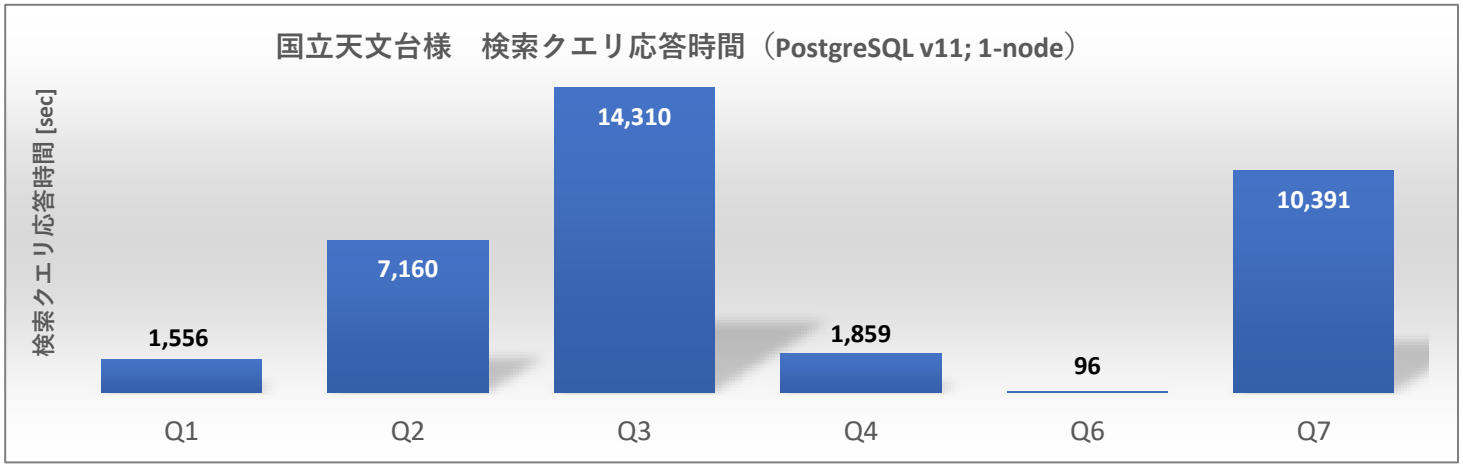
研究者

観測データの管理

- 高解像度の画像データには、一意なIDを付けてBLOBとしてファイルシステムに保存。
- 画像とそこに写る無数の天体の特徴を表すメタデータはPostgreSQLに保存し、研究者は、画像や天体情報に紐づいた様々な属性に基づいて検索を行う事ができる。
- 天体メタデータは数千列×数億～数十億行の規模。
- 画像メタデータだけでもかなりのサイズ。

課題 – 検索時間がものすごくかかる

一回の検索に数十分～数時間を要し、研究活動に支障を生じる



要因①　データサイズが巨大

- いくつかのテーブルに分割されているとはいえ、1.0TB～3.0TB 程度のテーブルに対して、複雑な検索条件（= インデックスの効かない）で問い合わせる。

➔ テーブルのスキャンに要する I/O 負荷が大きすぎる。

要因②　列数が多すぎる

- メタデータに含まれる列数が2000を越える。
- PostgreSQLのHeapテーブルの制限値は MaxHeapAttributeNumber (=1600)

➔ 複数のテーブルに水平分割し、実行時に JOIN せざるを得ない。

検証シナリオ（1/3）

観測データをArrow化し、データサイズおよび列数制限に関わる問題を解消した上で、GPUによる並列処理を適用

課題の分析

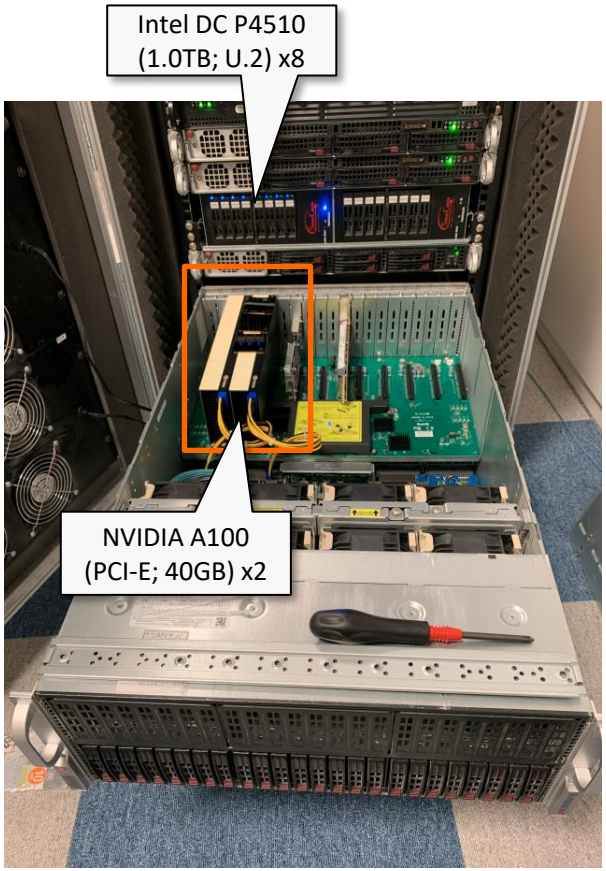
- 2000以上の列定義を有するものの、当然、検索クエリで同時に参照する列ははるかに少ない。
  - ➔列形式データであれば、1～2%程度を読み出すだけで十分では？
- テーブルを水平分割するのは、PostgreSQL Heapテーブルの列数制限に起因するワークアラウンド
  - ➔では、より多くの列を定義できるデータ形式を使用すれば？
- 観測データの更新／削除はあるか？
  - ➔ ない。日々の新しい観測データを追加する事はある。

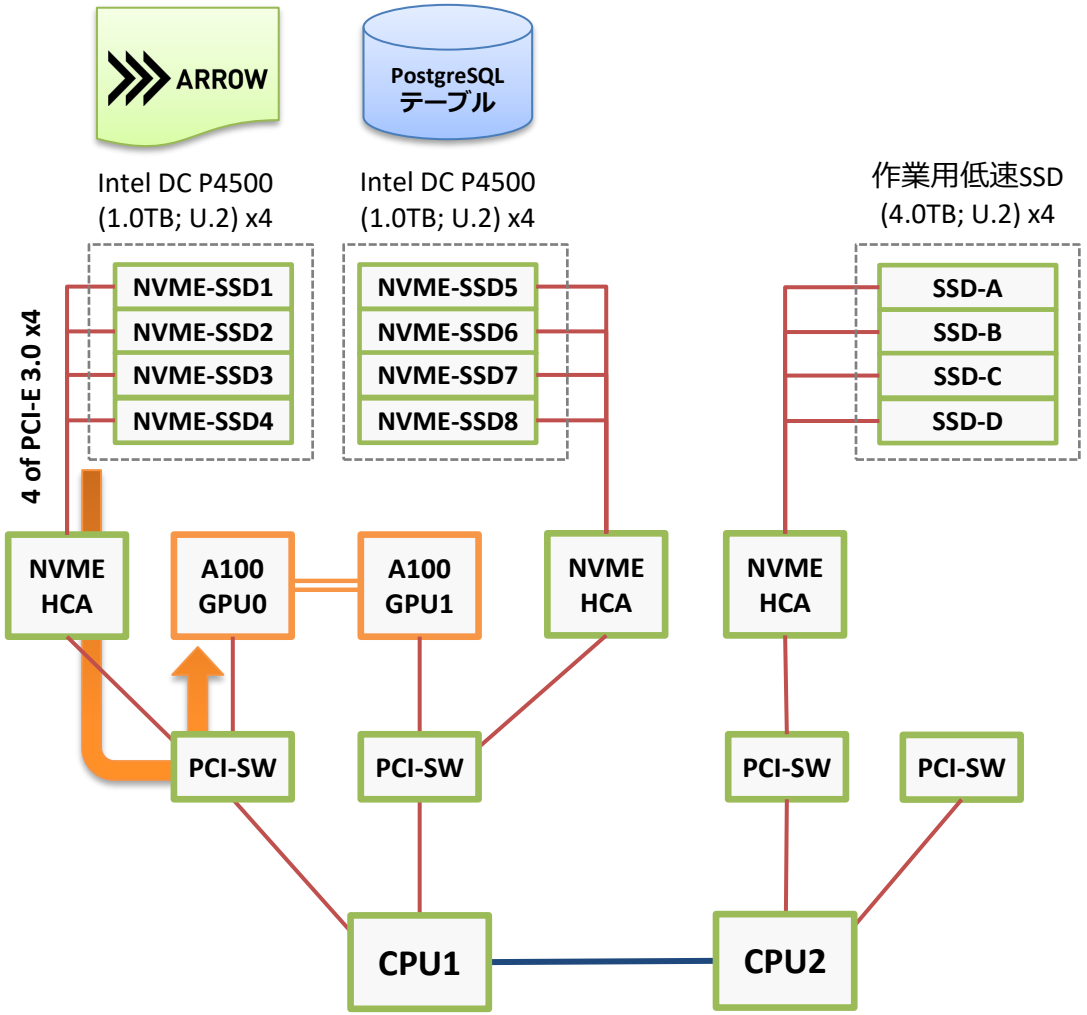
解決策

- 現在、PostgreSQLで管理している観測データを pg2arrow を用いてArrowファイルに変換し、それに対して同一の検索クエリを実行する。

検証シナリオ（2/3）

NVME-SSDx4 (計4.0TB) に対して、GPUx1をPCI-SW経由で直結させる構成



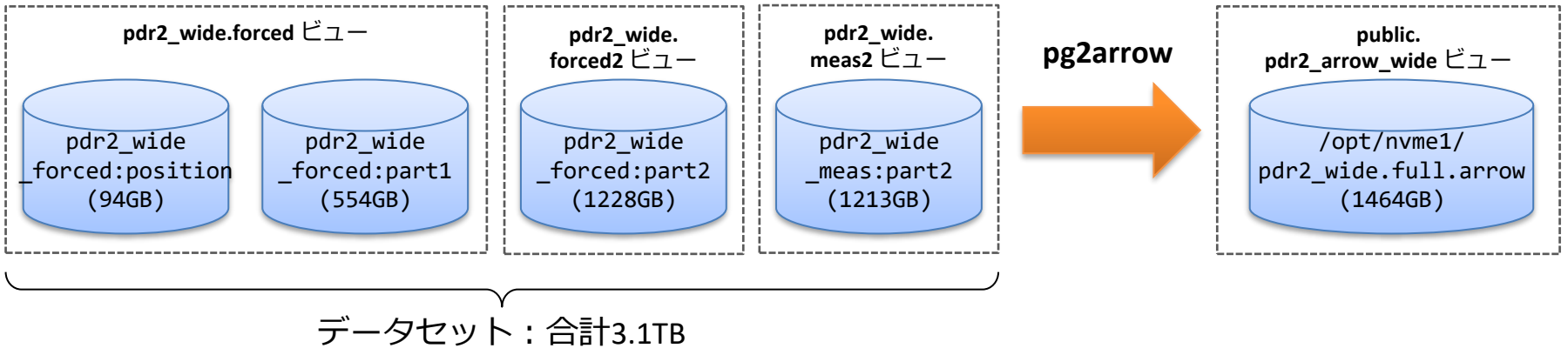


検証シナリオ（3/3）

- Arrow形式に列数制限はないため、全ての必要な属性を予めフラット化
- SSD容量の都合で、データ量は約半分に切り下げ。

```
[Query-03; Original]
SELECT object_id,
       f1.ra,
         :
       f2.r_psfflux_mag - f2.i_psfflux_mag AS r_i
  FROM
       pdr2_wide.forced AS f1
  LEFT JOIN
       pdr2_wide.forced2 AS f2 USING (object_id)
  LEFT JOIN
       pdr2_wide.meas2  AS m2 USING (object_id)
 WHERE f2.i_psfflux_mag < 16.2
         :
   AND f1.isprimary;
```

```
[Query-03; PG-Strom]
SELECT object_id,
       f1.ra,
         :
       f2.r_psfflux_mag - f2.i_psfflux_mag AS r_i
  FROM
       pdr2_arrow_wide
 WHERE f2.i_psfflux_mag < 16.2
         :
   AND f1.isprimary;
```



問題①：PostgreSQLの1600列制限について（1/2）

```
struct HeapTupleHeaderData
{
    union
    {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    }           t_choice;

    ItemPointerData t_ctid;     /* current TID of this or newer tuple (or a
                                 * speculative insertion token) */

    /* Fields below here must match MinimalTupleData! */
    uint16      t_infomask2;    /* number of attributes + various flags */
    uint16      t_infomask;     /* various flag bits, see below */
    uint8       t_hoff;         /* sizeof header incl. bitmap, padding */

    /* ^ - 23 bytes - ^ */
    bits8       t_bits[FLEXIBLE_ARRAY_MEMBER];  /* bitmap of NULLs */

    /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

- PostgreSQLの行データは、HeapTupleHeaderData とそれに続く NULL-bitmap + ペイロードの組み合わせとして表現されている。
- ペイロード部分は (char *)((char *)htup + htup->t_hoff) で高速に参照できるが、t_hoffは8bit変数かつ64bitアラインなので、最大でも +248 でペイロードが始まる。
- ヘッダは最低23バイト。OIDが4バイトを取る可能性があるため、NULL-bitmapの最大長は 248 - 23 - 4 = 221byte = 1768bit となる。
  将来の拡張可能性など多少のマージンを考慮し、列数上限が1600と定義されている。

問題①：PostgreSQLの1600列制限について（2/2）

pg2arrowで1600列を越えるArrowファイルを生成する

- `SELECT ... FROM t1 NATURAL JOIN t2 ...;` で結果が制限値を越えるとエラー。

➔ DB側で実行できないなら、クライアント側でJOINを実行すればよい。

```
$ pg2arrow -d postgres -o /tmp/test1.arrow ¥
           -c ‘SELECT * FROM t_a’          ¥
           --inner-join 'SELECT b1,b2,b3,b4 FROM t_b WHERE $(id) = id'
```

- メインのSQLコマンド（-cオプション）の結果からid列を取り出し、その値を$(id)と置き換えて --inner-join コマンドを実行。その結果と結合して Arrow ファイルを生成する。
- 詳しくは、海外の俺メモ：Pg2Arrowに『ぐるぐるSQL』モードをつけてみた。
  https://kaigai.hatenablog.com/entry/2021/02/09/011940

1600列を越える外部テーブルを定義する

- 本来は HeapTuple 形式の制限なので、外部テーブルに対しては無意味な制限ではあるが…。
- 通常の CREATE FOREIGN TABLE で1600列を越える外部テーブルを定義しようとすると、エラーが発生して怒られが発生する。

➔ 直接、システムカタログを更新してしまえば良いじゃないか。

```
=# pgstrom.arrow_fdw_import_file(‘f_mytable’,          -- relation name
                                 ‘/tmp/mytable.arrow’); -- file name
```

問題②：contrib/{cube, earthdistance} モジュール

PG-Stromでの未対応データ型／演算子

- 一部の検索条件が {cube, earthdistance} モジュールに由来する型・演算子を用いている。

```
SELECT object_id, ...
  FROM t_observation
 WHERE coord <@ '(123.456, ...., 456.789)'::cube
   AND ...;
```

- coord列 ... earthdistance型の列
- <@演算子 ... {cube, earthdistance} 型同士の包含関係を判定する。

Apache Arrowでの cube データ型の扱い

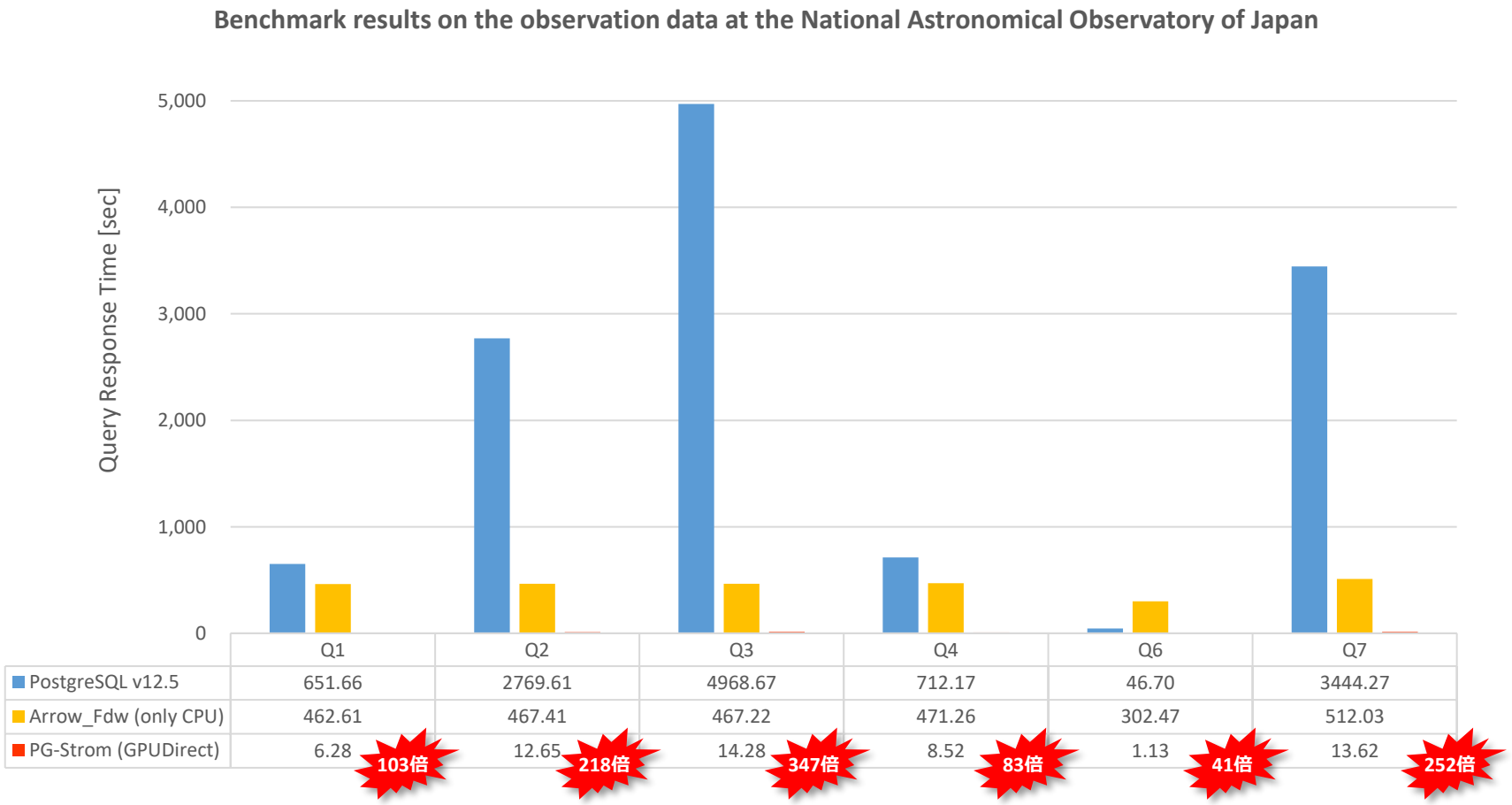
- Apache Arrowの基本データ型ではない。
- CustomMetadataを利用し、'pg_type=cube' を付加した Binary 型としてダンプしている。
- 他に類似の対応：inet型、macaddr型

GPUで利用できる演算子の扱い

- 特定ワークロードのため、専用演算子のサポートを追加する事になると、PG-Stromのコードベースに収拾がつかなくなってしまう。
- 『PG-Stromに対する拡張モジュール』で、GPU演算子・関数を追加できるように。

➔ 細かいAPIを調整の上、v4.0では正式機能化の予定。

検証：ベンチマーク結果



	Q1	Q2	Q3	Q4	Q6	Q7
PostgreSQL v12.5	651.66	2769.61	4968.67	712.17	46.70	3444.27
Arrow_Fdw (only CPU)	462.61	467.41	467.22	471.26	302.47	512.03
PG-Strom (GPUDirect)	6.28	12.65	14.28	8.52	1.13	13.62

- ✓ 対PostgreSQLで100～350倍程度の実行性能を記録
- ✓ クエリの種類によらず、ほぼ計算量とI/O量に比例した応答時間を記録
- ➔ 「数時間」が「十数秒」レベルに改善している。

検証まとめ

結論

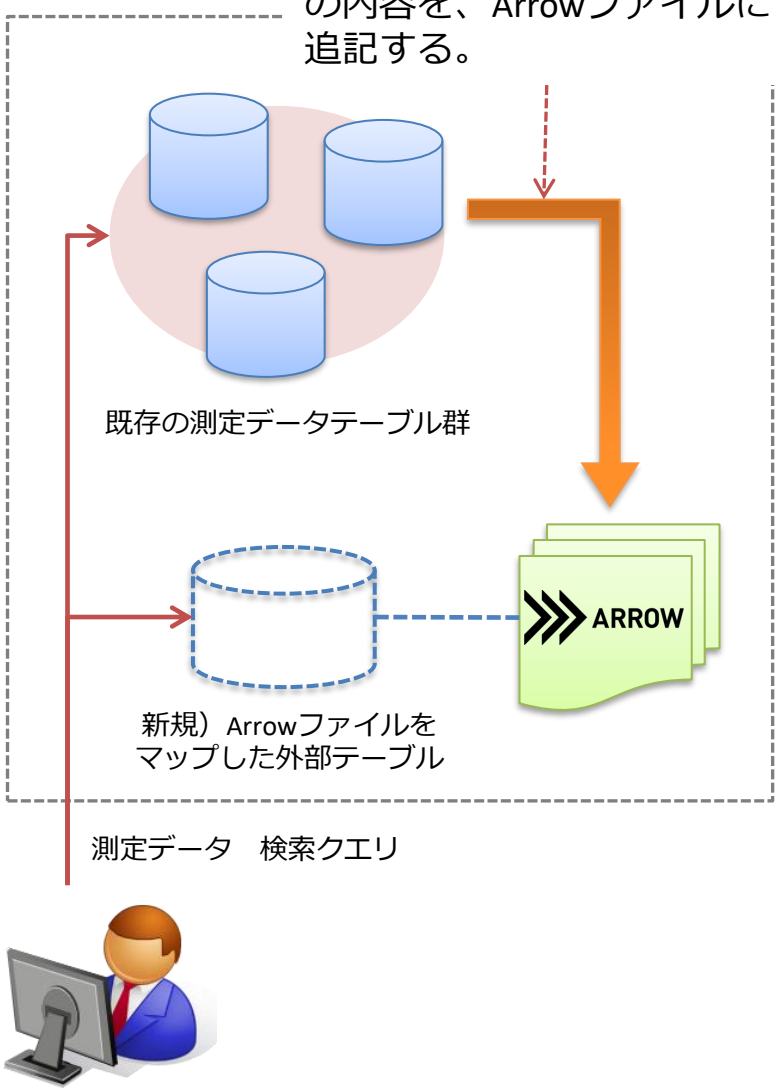
- 元々の課題
  - ✓ データサイズの巨大化に伴う、検索時の I/O 負荷の増大
  - ✓ PostgreSQLの列数制限に伴う JOIN の発生

➔ これらは、Apache Arrow の適用により大幅な負荷軽減に。
（「数時間」➔「数分」レベル）

- GPUの適用と並列処理の効果により、さらに「十数秒」レベルにまで応答時間を短縮する事を実証。

望ましいシステム構成

- この結果を踏まえると、次のようなシステム構成が望ましいと考える。
  - ✓ 観測データが蓄積するたびに、適宜、PostgreSQLテーブルから、Arrowファイルに移送する。
  - ✓ 検索時は（継承などを使い）両方のテーブルを同時に検索する。



まとめ

まとめ

IoT/M2Mログデータに Apache Arrowを使うと…。

- 列データなので、必要最小限のデータを読み出すだけで検索・分析が可能。
- 特に、GPUやSIMD命令の処理性能を最大化するようなデータ配置。
- 外部から「データを取り込む」場合でも、ファイルコピーのみで完了。
- 一方で、更新/削除はできないので『追記のみ（Insert-only）』なデータである事が必須。（トランザクション向きではない）

PG-Strom (Arrow_Fdw) を使うと…？

- GPU-Direct SQLで「ほぼほぼH/W限界値」な処理速度を実現できる。
- 使い慣れたSQLで操作でき、他のデータと組み合わせた分析も容易。
- 独自機能の min/max統計情報 は、列だけでなく、行方向の絞り込みを可能にするインデックスとして機能する。
- pg2arrow, pcap2arrowなど。fluentd向けArrow出力プラグインも開発中。

実際のワークロードでの検証

- 特にフィールド数の多いセンサデータであり、Arrow化の効果が極めて大。
- GPUの並列処理効果も併せ、元の処理速度から比較して最大350倍の高速化。

➔ センサデータの検索・分析にPG-Strom + Arrow が有効であることを実証した。

オモシロ技術を、カタチにする。