Columnar Store for PostgreSQL using cstore_fdw

Marco Slot
marco@citusdata.com

This Webinar

Introduction for new users:

- What is cstore_fdw?
- Getting started with cstore_fdw
- Typical workloads (analytics, archival)

Dive into technical details of cstore_fdw:

- Storage format
- Skip indexes
- Query processing
- Performance numbers

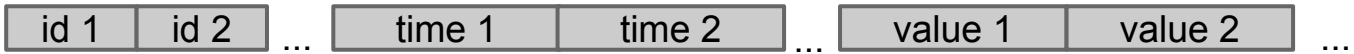
Advanced uses of cstore_fdw:

- Compress bulky JSONB fields
- Combine row-based and columnar storage in PostgreSQL 9.5

What is columnar storage?

PostgreSQL tables are stored as a sequence of tuples on disk, optimized for *operations on a small number of tuples* and *data ingestion*.

Columnar storage stores all the values in a column sequentially on disk, optimized for *operations on a small number of columns* and *compression*.



What is cstore_fdw?

cstore_fdw is an open source *extension* for PostgreSQL that provides columnar storage through a *foreign data wrapper* with *compression*.

https://github.com/citusdata/cstore_fdw/

Format is based on *ORC files in Hadoop/Hive*, but not compatible.

It's meant for *append-only* data through COPY or INSERT INTO … SELECT …

Automatically builds range-based indexes to skip over blocks of data when possible.

cstore_fdw tables vs. regular tables

Pros:

- Reduce I/O for queries that use few columns, but many rows
- Save on storage space through compression
- Fit more data into memory through compression
- Cheap projection

Cons:

- Indexes only effective on sorted columns
- Cannot update/delete/insert individual rows
- Cost of (de)compression
- Expensive selection

Installing cstore_fdw

Install protobuf library:

```
sudo yum install protobuf-c-devel
```

Get code:

```
git clone https://github.com/citusdata/cstore_fdw.git
```

Build code:

```
PATH=/usr/local/postgresql/bin/:$PATH make
sudo PATH=/usr/local/postgresql/bin/:$PATH make install
```

Update postgresql.conf:

```
shared_preload_libraries = 'cstore_fdw'    # (change requires restart)
```

Restart PostgreSQL.

Getting started with cstore_fdw

Load extension: `CREATE EXTENSION cstore_fdw;`

Create server object : `CREATE SERVER cstore_server FOREIGN DATA WRAPPER cstore_fdw;`

Create foreign table:

```
CREATE FOREIGN TABLE events (
    event_date date,
    …
) SERVER cstore_server OPTIONS(compression 'pglz');
```

Load data:

```
COPY events FROM '/tmp/events-2015-09-09.csv' WITH CSV;
```

Use-case: Analytics

Analytical queries often process many rows, but only a few columns.

Get the revenue by country in 2015:

```
SELECT country, count(*) FROM events WHERE event_date BETWEEN date
'2015-01-01' AND date '2015-12-31' GROUP BY country ORDER BY 2 DESC;
```

On a regular table, this query reads all data for 2015.

On a columnar table, this query only reads the country and event_date columns for 2015.

This can easily amount to a significant reduction in I/O, and thus significant speed-ups.

Use-case: Data archival

PostgreSQL does not built-in table compression method other than TOAST, and compression doesn’t work well for row-based storage.

Say we have an events and an archive table:

```
CREATE TABLE events (...);
CREATE FOREIGN TABLE archive (...) SERVER cstore_server OPTIONS(compression 'pglz');
```

Can archive data into compressed cstore table:

```
WITH migrate AS (DELETE FROM events WHERE event_date < date '2015-01-01' RETURNING *)
INSERT INTO archive SELECT * FROM migrate;
```

Typically 50-90% reduction in storage space.

Data format

Inspired by Optimized Row Columnar (ORC) format developed by Hortonworks.

Data file contains stripes.

Stripes contain blocks and metadata on blocks for each column.

Blocks contain null bitmap and (compressed) data.

Footer file contains metadata on stripes.

Each COPY or INSERT INTO … SELECT appends one or more new stripes and updates footer file.

When querying, stripes are buffered in memory.

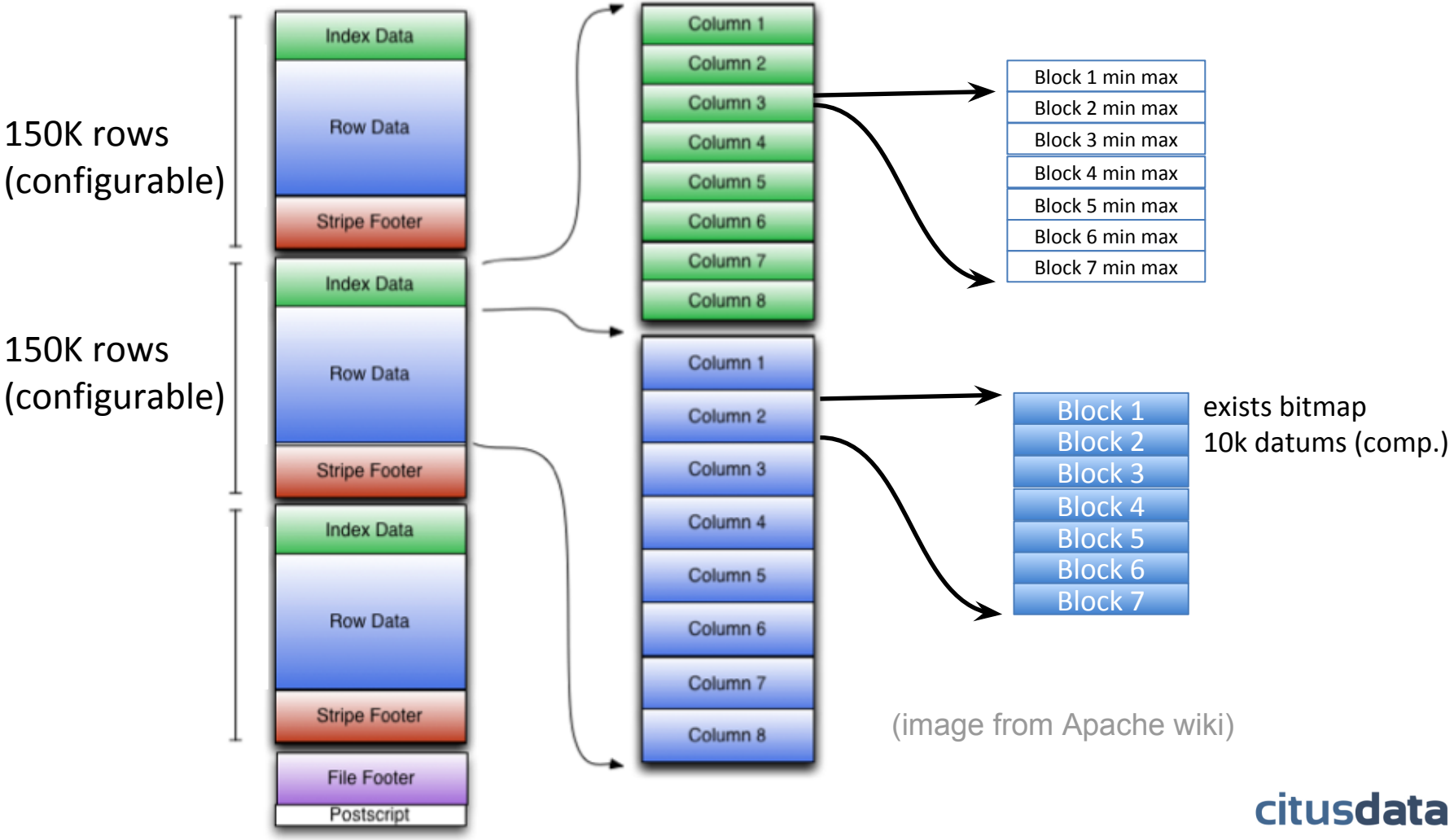
Data format

Data file:

-> Stripes	- data for up to 150,000 rows (configurable)
-> Skip indexes	- min and max values of blocks
-> Columns	- any number of columns
-> Blocks	- data for up to 10,000 rows (configurable)
-> Exists bitmap	- for null values
-> Data	- compressed PostgreSQL datums
-> Stripe footer	- offsets/lengths for each skip node and block

Footer file:

-> Table footer	- offsets/lengths for each stripe
-> Postscript	- versions, footer length


150K rows
(configurable)
150K rows
(configurable)
Index Data
Row Data
Stripe Footer
Index Data
Row Data
Stripe Footer
Index Data
Row Data
Stripe Footer
File Footer
Postscript
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8
Block 1 min max
Block 2 min max
Block 3 min max
Block 4 min max
Block 5 min max
Block 6 min max
Block 7 min max
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
exists bitmap
10k datums (comp.)
(image from Apache wiki)
citusdata

Skip indexes

For each block, cstore_fdw keeps track of min and max values.

At query time, check if “min <= column AND column <= max” is refuted by the WHERE clause (using predicate_refuted_by) to determine whether a block can be skipped.

Works well for sorted (often time-series) data. For example, skip indexes on date:

```
block 1 [min 2014-12-01, max 2014-12-31]
block 2 [min 2015-01-01, max 2015-01-31]
…
```

Then block 1 would be skipped for all columns when querying:

```
SELECT event_id FROM events WHERE event_date >= date ‘2015-01-01’;
```

SELECT processing

Start a foreign scan (BeginForeignScan)

- Read table footer

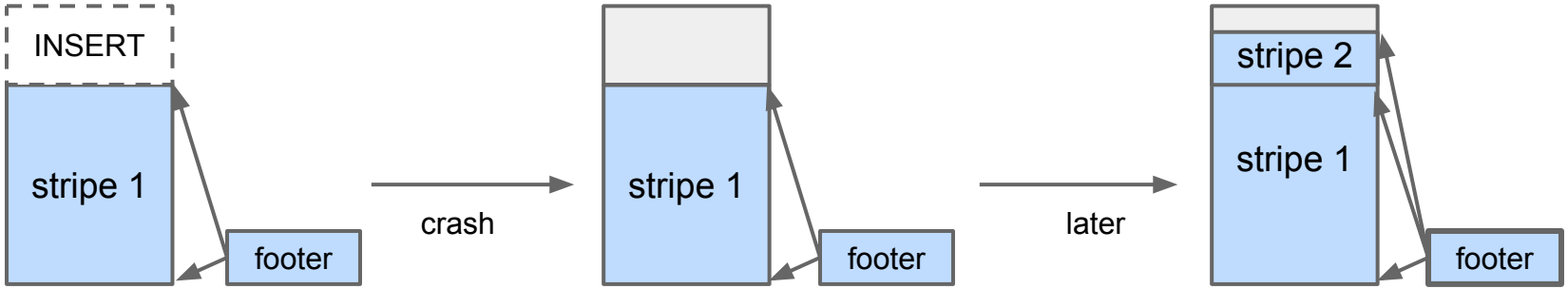
Read a row (IterateForeignScan)

- If no stripe buffered
  - Read stripe footer
  - Read skip indexes
  - Find skippable blocks using skip indexes and WHERE clause
  - Read non-skippable blocks for columns in projection
- If current block not yet deserialized
  - Decompress and deserialize next block
- Return a tuple containing columns in projection (NULL for other values)

Crash-resilience

Writes to cstore_fdw don’t go through the WAL. What happens if a crash occurs?

Footer file is not updated, so new data is ignored in subsequent reads/writes.



Writes to stripes are always synced to disk before updating footer.

Footer files are atomically updated by renaming a hard link.

Backups

Common techniques:

- SQL-level back-up using pg_dump
- File system-level back-up by copying database directory

Alternatively, back-up the following files:

$PGDATA/cstore_fdw/[database oid]/[table oid].footer (footer file)

$PGDATA/cstore_fdw/[database oid]/[table oid] (data file)

To recover, create a cstore table with an existing data file:

CREATE FOREIGN TABLE … OPTIONS(filename '[data file]', …)

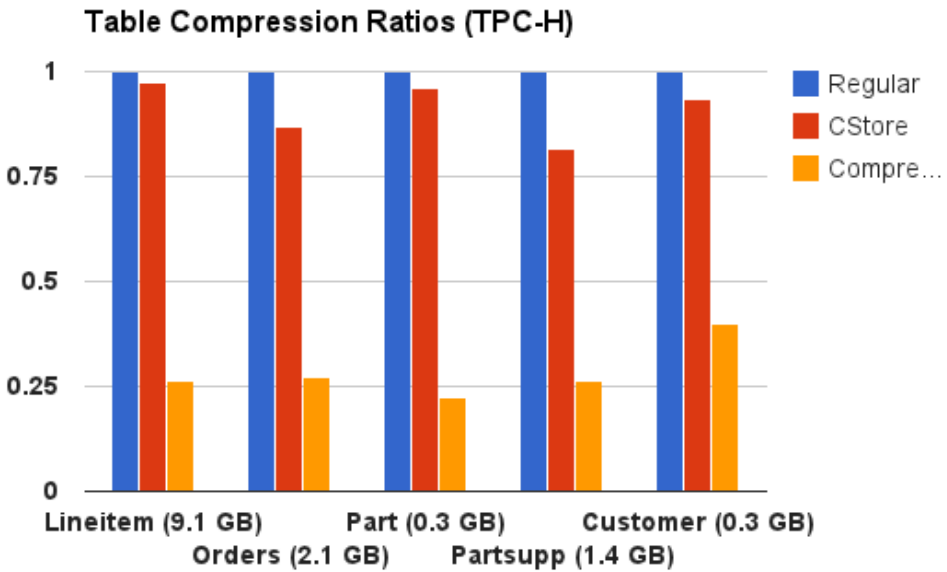
Compression ratio

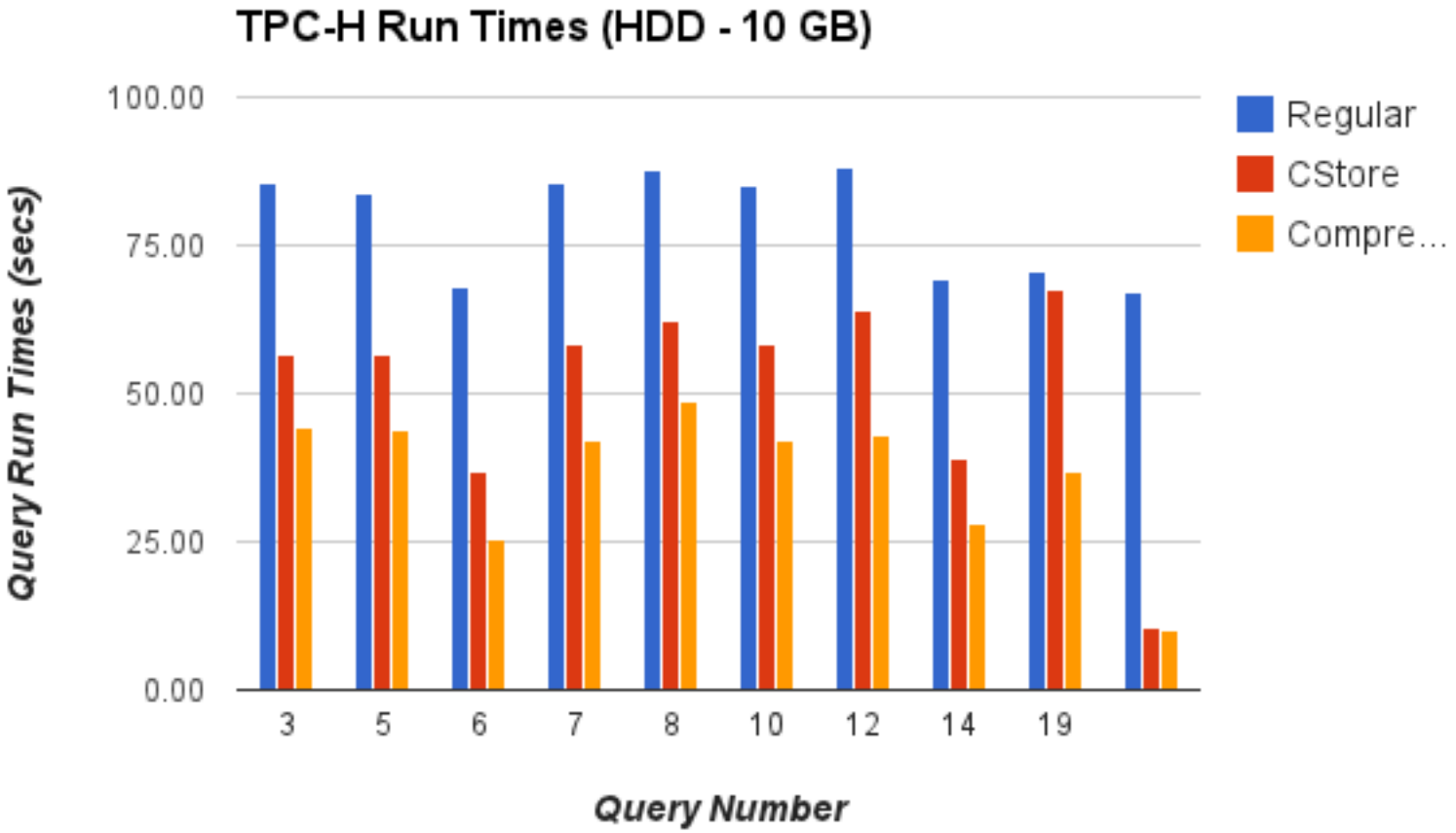
Uses PG_LZ in PostgreSQL core.

Compression ratios depend on data.

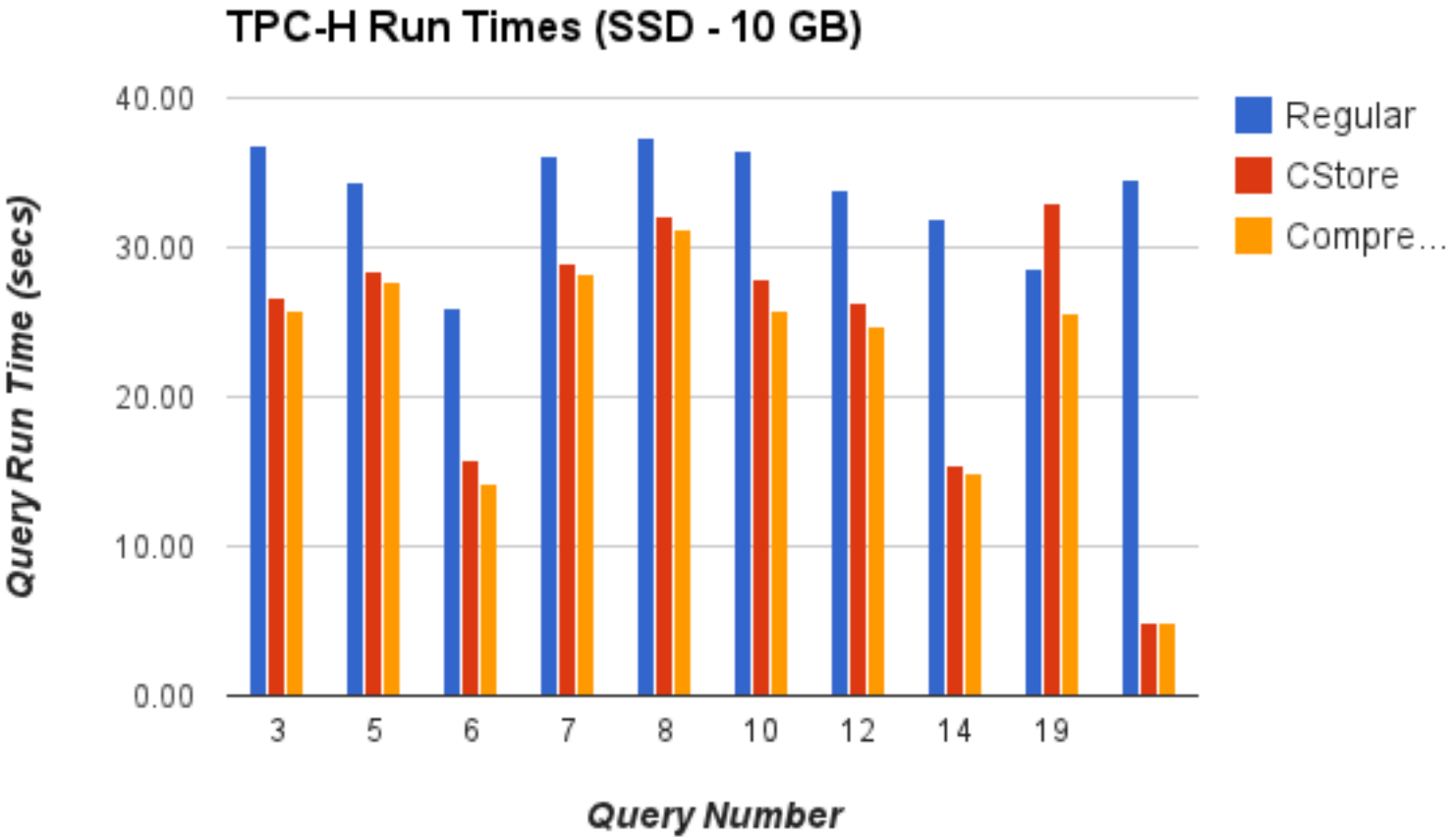
TPC-H:	4x
customer reviews JSONB:	6x
github events JSONB:	2x

Interestingly, storage size can be lower even without compression.

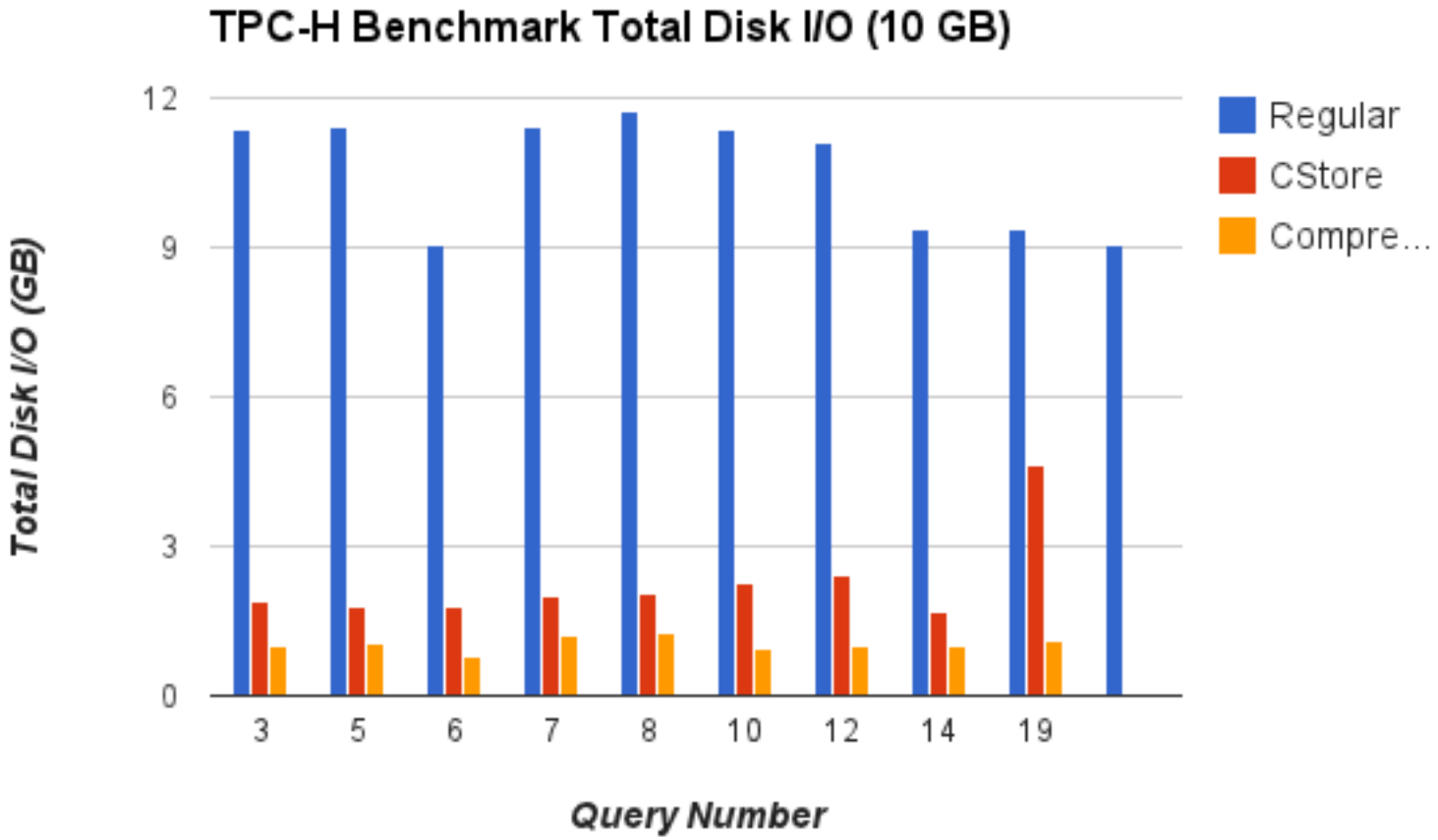




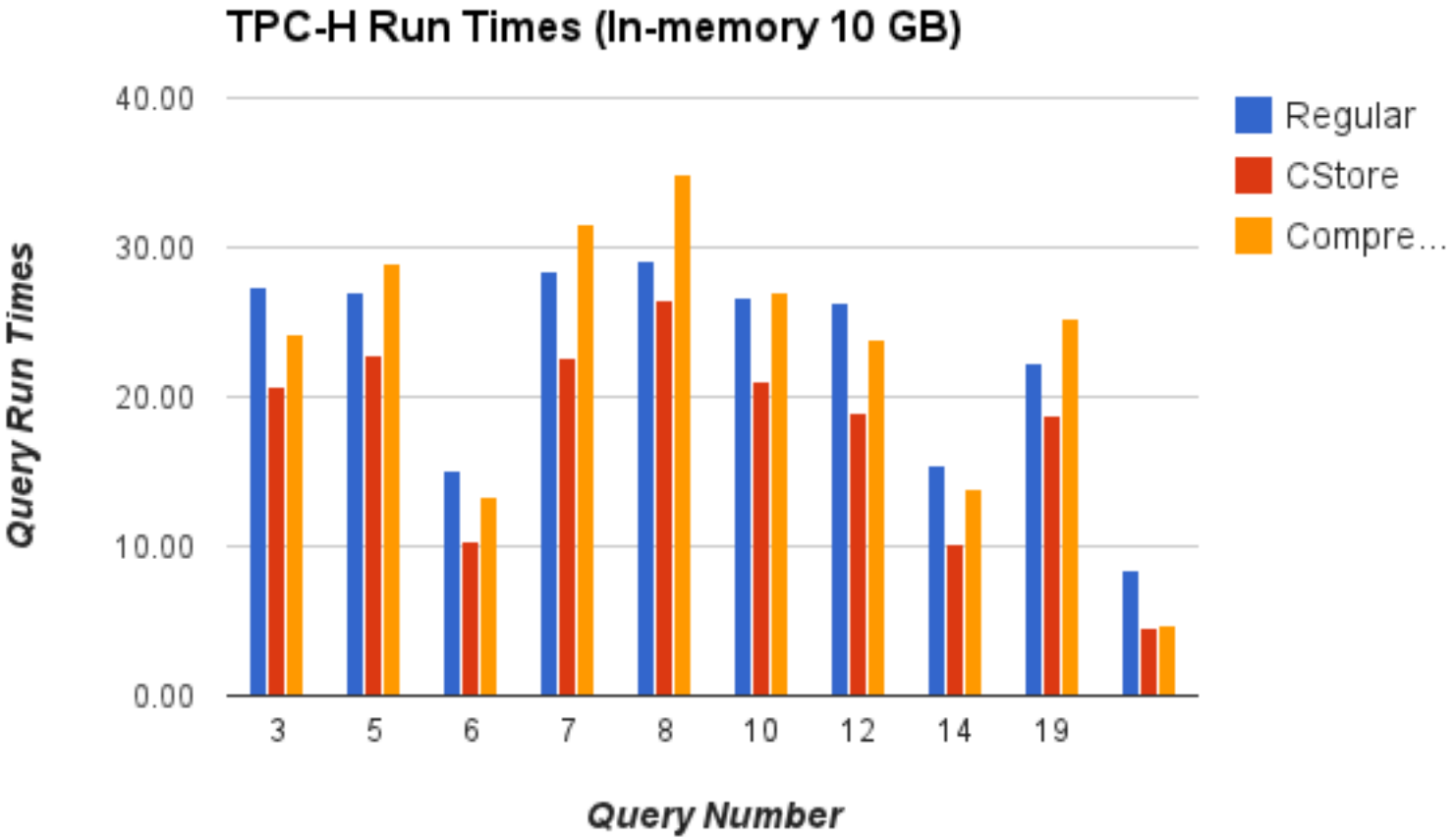
10GB of uncached data on m2.2xlarge



10GB of uncached data on m3.2xlarge



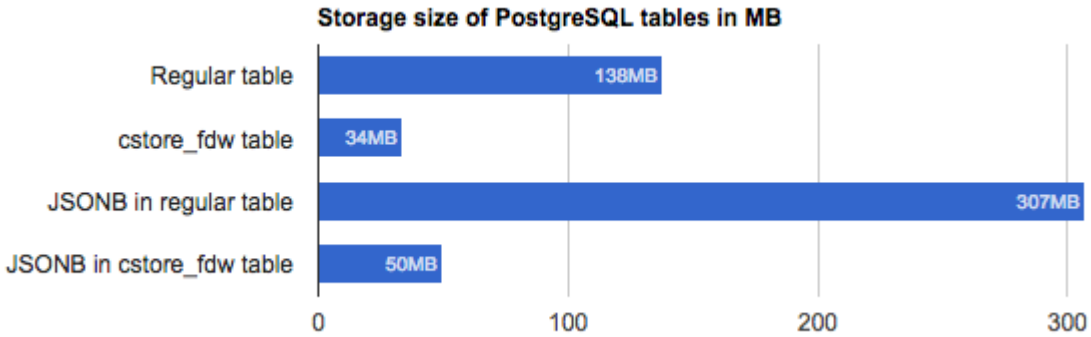
Total disk I/O measured with iotop



10GB of cached data on m2.2xlarge

cstore_fdw with JSONB

Field names are repeated very often in JSONB fields, compresses well with cstore_fdw



However, depends on structure of the data.

PostgreSQL 9.5: Foreign table inheritance

Example:

```
CREATE TABLE events (...);

CREATE TABLE events_current () INHERITS(main);

CREATE FOREIGN TABLE events_201501 (
    CHECK(20150101 <= event_date AND event_date <= 20150131)
)
INHERITS(main) SERVER cstore_server OPTIONS(compression 'pglz');
```

Fast INSERT and look-ups into current table.

Periodically move data to archive table for compression.

Query both via main table.

Summary

Discussed cstore_fdw internals:

- Storage format — Stripes, blocks and footers
- Skip indexes — Automatic index on sorted columns
- Query processing — Read column-by-column, buffer stripes in memory

Performance numbers:

- Great for I/O bound workloads
- 50% or more compression

Demo:

- Foreign table inheritance: combined row-based and columnar store!

Questions?

Marco - marco@citusdata.com

General - engage@citusdata.com

https://www.citusdata.com/blog