Gungnir
Patrick Wang, David Guo, Alexis Schlomer
Previously, on Gungnir

**Goal:** Estimate query run time *statically.*

**Recipe**

1. *Extract* Statistics
2. *Estimate* Cardinality
3. *Infer* Query Cost
<table>
<thead>
<tr>
<th>Task</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Digest</td>
<td>✅</td>
</tr>
<tr>
<td>HLL</td>
<td>✅</td>
</tr>
<tr>
<td>MCVs</td>
<td>✅</td>
</tr>
<tr>
<td>Parallelize T-Digest</td>
<td>❌</td>
</tr>
<tr>
<td>HLL</td>
<td>❌</td>
</tr>
<tr>
<td>MCVs</td>
<td>❌</td>
</tr>
<tr>
<td>Getting TPC-H to run</td>
<td>✅</td>
</tr>
<tr>
<td>Cardinality benchmarking</td>
<td>⌚</td>
</tr>
<tr>
<td>Group Cardinality Caching</td>
<td>❌</td>
</tr>
<tr>
<td>Semantic Correlation</td>
<td>❌</td>
</tr>
</tbody>
</table>
Statistics

Motivation: processors are *not* getting faster!

1. Building statistics is **CPU bound**.
2. OLAP systems must support **+16 PB** of data (*Redshift*).

Solutions:

1. Sampling (ex. only use 1% of data).
2. Parallel sketching algorithms.

⇒ OUR FOCUS!
Statistics

A new ANALYZE paradigm: fold-reduce associativity!

- Goal 1: **inter**-node parallelism.
- Goal 2: **intra**-node parallelism.
- Goal 3: **flexible linear scaling** w/ nodes.

Problems:

1. How/on what do you combine?
2. Arrow *poorly* exploits parallelism!
Statistics

Roadmap: unleashing the power of modern processors.

1. Find sketches that satisfy the fold fold/reduce paradigm: HLL (n-distinct), TDigest (distribution), MisraGries (most-common-values).
2. Implement these algorithms from scratch.
3. Expose more parallelism from Arrow to have parallel scanners.
4. Rely on a modular thread-pool to split the tasks into smaller jobs (Rayon).
5. Optimize like a German.

Results: single node +10 Gbps throughput (on SOTA hardware).

or… +1PB/day with only 10 nodes.
Cardinality Formulas

- **Filter** selectivity
  - $t1.\text{colA} \ [=, \neq, <, \leq, >, \geq] \ \text{constant}$
  - AND/OR/NOT
  - $\text{colA IN (“advanced”, “database”, “systems”)}$
  - $\text{colA LIKE “%abc%” using MCVs}$
  - CAST

- **Join** selectivity
  - Join types (Inner, Outer, Cross)
  - Join conditions vs. join filters
    - $t1.\text{colA} = t2.\text{colB} \ \text{vs.} \ t1.\text{colA} < 2$
    - Detects *semantic correlation*

- **Aggregation, Limit**
Detect **Semantic Correlation**

These are the *SAME* column

```
t.id = mc.tid
```

```
t.id = mi.tid
```

```
t
mc
mi
```
Detect **Semantic Correlation**

Conceptually, it’s a “multi-equality"

\[ t.tid = mi.tid = mc.tid \]
Detect **Semantic Correlation**

By contrast, these are **NOT** the same
Detect **Semantic Correlation**

Semantic correlation distinguishes between these two cases:

- $t.id = mi[tid] = mc[tid]\$
- $mc[kid] = mk[id]\$

$t \quad mc \quad mi$

$mc \quad mc \quad mi$

$mc \quad mk$
Detect **Semantic Correlation**

This one feature decreased our Q-Error on JOB by $100x$
Adaptivity through **Group Cardinality Caching**

- JOB-light Q27a **Q-Error: 10k → 600, 17x**

```
SELECT *
FROM title t,
     movie_info mi,
     movie_companies mc,
     cast_info ci,
     movie_keyword mk
WHERE t.id=mi.movie_id
    AND t.id=mc.movie_id
    AND t.id=ci.movie_id
    AND t.id=mk.movie_id
    AND ci.role_id=2
    AND mi.info_type_id=16
    AND t.production_year>2000
    AND t.production_year<2010
    AND mk.keyword_id=7084;
```

Is 414 rows in reality
We underestimate by **20x**
Adaptivity through **Group Cardinality Caching**

- **JOB-light Q27a**  
  **Q-Error:** $10k \rightarrow 600$, 17x

```sql
SELECT *  
FROM title t,  
    movie_info mi,  
    movie_companies mc,  
    cast_info ci,  
    movie_keyword mk  
WHERE t.id=mi.movie_id  
AND t.id=mc.movie_id  
AND t.id=ci.movie_id  
AND t.id=mk.movie_id  
AND ci.role_id=2  
AND mi.info_type_id=16  
AND t.production_year>2000  
AND t.production_year<2010  
AND mk.keyword_id=7084;
```

```sql
SELECT *  
FROM movie_keyword mk  
WHERE mk.keyword_id=7084;
```

**Run this first!**

No longer underestimating  
Leads to 17x better Q-Error
## Results - **Cardinality Estimation** Accuracy

### TPC-H (SF1)

<table>
<thead>
<tr>
<th></th>
<th>PG</th>
<th>Optd</th>
</tr>
</thead>
<tbody>
<tr>
<td># Better</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td># Tied</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>p50</td>
<td>3.50</td>
<td>1.00</td>
</tr>
<tr>
<td>p90</td>
<td>1203.0</td>
<td>100.00</td>
</tr>
<tr>
<td>p99</td>
<td>1517.5</td>
<td>31250</td>
</tr>
</tbody>
</table>

### JOB*

<table>
<thead>
<tr>
<th></th>
<th>PG</th>
<th>Optd</th>
</tr>
</thead>
<tbody>
<tr>
<td># Better</td>
<td>21</td>
<td>39</td>
</tr>
<tr>
<td># Tied</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>p50</td>
<td>209.33</td>
<td>80.00</td>
</tr>
<tr>
<td>p90</td>
<td>8546.2</td>
<td>128548</td>
</tr>
<tr>
<td>p99</td>
<td>42963</td>
<td>4.0e11</td>
</tr>
</tbody>
</table>

### JOB-light

<table>
<thead>
<tr>
<th></th>
<th>PG</th>
<th>Optd</th>
</tr>
</thead>
<tbody>
<tr>
<td># Better</td>
<td>7</td>
<td>51</td>
</tr>
<tr>
<td># Tied</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p50</td>
<td>5.73</td>
<td>3.10</td>
</tr>
<tr>
<td>p90</td>
<td>69.31</td>
<td>13.28</td>
</tr>
<tr>
<td>p99</td>
<td>7887.4</td>
<td>7382.1</td>
</tr>
</tbody>
</table>

*with stats from Python csv2parquet script

**Shows we do well in...**

- operator variety
- complex predicates
- pure join estimation
Detect **Semantic Correlation**

Solution: Keep track of equal columns as a group’s logical property with **Union-Find**

\{mc.mid, mi.mid\}
Detect **Semantic Correlation**

Solution: Keep track of equal columns as a group’s logical property with **Union-Find**

\[
\{mc.\text{mid}, \; mi.\text{mid}, \; t.\text{id}\}
\]
Detect **Semantic Correlation**

Solution: Keep track of equal columns as a group’s logical property with **Union-Find**

\[
\{mc.\text{mid}, mi.\text{mid}, t.\text{id}\}
\]

**Selectivity Adjustment Factor**

s.t. total selectivity = \(1 / \text{(product of # distinct of N - 1 most selective columns)}\)
Benchmarking

- Made TPC-H and JOB queries **not crash** opt-d
  - internal repr for more data types and exprs
- **Robust, fast, and easy-to-use framework**
  - Ergonomic CLI and detailed output
  - Robust caches for data+queries, truecard, optd stats, and pgdata
  - Compatible with Postgres in a container or a different machine
Getting **TPC-H, JOB, JOB-light** to run

Getting **TPC-H, JOB, JOB-light** to not crash optd

- More data types
  - Various Int types
  - Date
  - Serialized
  - IntervalMonthDateNano

- More expressions
  - Like
  - InList
  - Cast

Result: **13 / 22** for TPC-H, **93 / 113** for JOB, **58 / 70** for JOB-light
Row count with **EXPLAIN**

Display estimated cost with EXPLAIN VERBOSE

```sql
PhysicalSort
    └── exprs:SortOrder { order: Desc }
        ├── #1
        │    └── cost: weighted=34061549.96, row_cnt=25.00, compute=26400304.96, io=7661245.00
        │        └── PhysicalProjection { exprs: [ #0, #1 ], cost: weighted=34061466.46, row_cnt=25.00, compute=26400221.46, io=7661245.00 }
        │            └── PhysicalAgg
        │                 └── aggs:Agg(Sum)
        │                             └── Mul
        │                                 └── Sub
        │                                     └── #1
        │                                         └── #0
        │                                             └── groups: [ #2 ]
        │                                                 └── cost: weighted=34061465.16, row_cnt=25.00, compute=26400220.16, io=7661245.00
        │                                                     └── PhysicalProjection { exprs: [ #0, #1, #2 ], cost: weighted=34055530.75, row_cnt=100.34, compute=26394285.75, io=7661245.00 }
        │                                                         └── PhysicalProjection { exprs: [ #0, #1, #4, #5, #6 ], cost: weighted=34055523.65, row_cnt=100.34, compute=26394278.65, io=7661245.00 }
        │                                                            └── PhysicalProjection { exprs: [ #2, #3, #5, #6, #7, #8, #9 ], cost: weighted=34055512.50, row_cnt=100.34, compute=26394267.50, io=7661245.00 }
        │                                                                   └── PhysicalProjection { exprs: [ #0, #3, #4, #5, #6, #7, #8, #9, #10, #11 ], cost: weighted=34055497.30, row_cnt=100.34, compute=26394252.30, io=7661245.00 }
        │                                                                       └── PhysicalProjection { exprs: [ #1, #2, #4, #5, #6, #7, #8, #9, #10, #11 ], cost: weighted=34055476.02, row_cnt=100.34, compute=26394231.02, io=7661245.00 }
```
Ergonomic & Robust Benchmarking Framework

- **Ergonomic:** 8 CLI, detailed outputs
  - All CLI options have sensible defaults
  - Outputs: per-query, aggregate, and comparative Q-Error

- **Robust:** consistent caches in all partial failure scenarios
  - Caching gives 70x speedup on TPC-H SF1

---

Usage: optd-perftest cardtest [OPTIONS] [BENCHMARK_NAME]

Arguments:
  [BENCHMARK_NAME] [default: tpch] [possible values: tpch, job, joblight]

Options:
  --scale-factor <SCALE_FACTOR> [default: 0.01]
  --seed <SEED> [default: 15721]
  --query-ids <QUERY_IDS>... The queries to get the Q-error of
  --rebuild-cached-optd-stats Whether to use the cached optd stats/cache generated stats
  --adaptive Whether to enable adaptivity for optd
  --pguser <PGUSER> The name of a user with superuser privileges [default: default_user]
  --pgpassword <PGPASSWORD> The name of a user with superuser privileges [default: password]
  -h, --help Print help
Results - *Benchmark Subsystem* Performance

Compares **Q-Error** with PostgreSQL

*Caches* statistics and true cardinalities → **70x speedup**

- DF build stats (150.45s)
- PG load tables (63.24s)
- PG stats (40.41s)
- Gen TPC-H (35.76s)

DF & PG cost models (3.71s)

True cardinalities (4.25s)
Code Quality - Modularity

- Pluggable cost model, stats, and DBMSs in benchmarking framework via traits

```rust
pub trait CostModel<T: RelNodeType>:
    'static + Send + Sync {
        fn compute_cost(
            &self,
            node: &G,
            data: &Option<value>?,
            children: &[(Cost),
            context: Option<RelNodeContext>],
            // one reason we need the optimizer is to traverse children nodes to build up an expression tree
            optimizer: Option<&CascadesOptimizer<T>>,
        ) -> Cost;
    }

    fn compute_plan_node_cost(&self, node: &RelNode<T>) -> Cost;

    fn explain(&self, cost: &Cost) -> String;

    fn accumulate(&self, total_cost: &mut Cost, cost: &Cost);

    fn sum(&self, self_cost: &Cost, inputs: &[(Cost)]) -> Cost {
        let mut total_cost = self_cost.clone();
        for input in inputs {
            total_cost += input;
        }
        total_cost
    }

    fn zero(&self) -> Cost;
}
```

```rust
pub trait Distribution: 'static + Send + Sync {
    // Give the probability of a random value sampled from the distribution being <= 'value'
    fn cdf(&self, value: &Value) -> f64;
}
```

```rust
pub trait MostCommonValues: 'static + Send + Sync {
    // It is true that we could just expose freq_over_pred() and use that for freq() and total_freq()
    // however, freq() and total_freq() each have potential optimizations (freq() is O(1) instead of O(n) and total_freq() can be cached)
    // additionally, it makes sense to return an Option<f64> for freq() instead of just 0 if value doesn't exist
    // thus, I expose three different functions
    fn freq(&self, value: &ColumnCombValue) -> Option<f64>;
    fn total_freq(&self) -> f64;
    fn freq_over_pred(&self, pred: Box<dyn Fn(&ColumnCombValue) -> bool>) -> f64;

    // returns the # of entries (i.e. value + freq) in the most common values structure
    fn cnt(&self) -> usize;
}
```

```rust
pub trait CardtestRunnerDBMSHelper {
    // get_name() has &self so that we're able to do Box<dyn CardtestRunnerDBMSHelper>
    fn get_name(&self) -> &str;

    // The order of queries in the returned vector has to be the same between all databases,
    // and it has to be the same as the order returned by TruecardGetter.
    async fn eval_benchmark_estcards(
        &mut self,
        benchmark: &Benchmark,
    ) -> anyhow::Result<Vec<usize>>;
}
```
Code Quality - Readability

- Tons of comments

/// A predicate set defines a "multi-equality graph", which is an unweighted undirected graph. The
/// nodes are columns while edges are predicates. The old graph is defined by 'past_eq_columns'
/// while the 'predicate' is the new addition to this graph. This unweighted undirected graph
/// consists of a number of connected components, where each connected component represents columns
/// that are set to be equal to each other. Single nodes not connected to anything are considered
/// standalone connected components.
///
/// The selectivity of each connected component of N nodes is equal to the product of 1/indistinct of
/// the k-1 nodes with the highest indistinct values. You can see this if you imagine that all columns
/// being joined are unique columns and that they follow the inclusion principle (every element of the
/// smaller tables is present in the larger table). When these assumptions are not true, the selectivity
/// may not be completely accurate. However, it is still fairly accurate.
///
/// However, we cannot simply add 'predicate' to the multi-equality graph and compute the selectivity of
/// the entire connected component, because this would be "double counting" a lot of nodes. The Join(s)
/// before this join would already have a selectivity value. Thus, we compute the selectivity of the
/// Join(s) before this join (the first block of the function) and then the selectivity of the connected
/// component after this join. The quotient is the "adjustment" factor.
///
/// NOTE: This function modifies 'past_eq_columns' by adding 'predicate'.

/// The core logic of join selectivity which assumes we've already separated the expression
/// into the on conditions and the filters.
///
/// Hash join and NLJ reference right table columns differently, hence the
/// 'right_col_ref_offset' parameter.
///
/// For hash join, the right table columns indices are with respect to the right table,
/// which means #0 is the first column of the right table.
///
/// For NLJ, the right table columns indices are with respect to the output of the join.
/// For example, if the left table has 3 columns, the first column of the right table
/// is #3 instead of #0.

/// The expr_tree input must be a "mixed expression tree".
///
/// An "expression node" refers to a RelNode that returns true for is_expression()
///
/// A "full expression tree" is where every node in the tree is an expression node
///
/// A "mixed expression tree" is where every base-case node and all its parents are expression nodes
///
/// A "base-case node" is a node that doesn't lead to further recursion (such as a BinOp(Eq))
///
/// The schema input is the schema the predicate represented by the expr_tree is applied on.
///
/// The output will be the selectivity of the expression tree if it were a "filter predicate".
///
/// A "filter predicate" operates on one input node, unlike a "join predicate" which operates on two input nodes.
/// This is why the function only takes in a single schema.

/// Get a dbname that deterministically describes the "data" of this benchmark.
/// Note that benchmarks consist of "data" and "queries". This name is only for the data
/// For instance, if you have two TPC-H benchmarks with the same scale factor and seed
/// but different queries, they could both share the same database and would thus
/// have the same dbname.
///
/// This name must be compatible with the rules all databases have for their names, which
/// are described below:
///
/// Postgres' rules:
/// - The name can only contain A-Z a-z 0-9 _ and cannot start with 0-9.
/// - There is a weird behavior where if you use CREATE DATABASE to create a database,
///   Postgres will convert uppercase letters to lowercase. However, if you use pos to
///   then connect to the database, Postgres will not convert capital letters to
///   lowercase. To resolve the inconsistency, the names output by this function will
///   not contain uppercase letters.

/// This trait defines helper functions to enable cardinality testing on a DBMS.
/// The reason "get true card" is not a function here is because we don't need to call
/// "get true card" for all DBMS we are testing, since they'll all return the same
/// answer. We also cache true cardinalities instead of executing queries every time
/// since executing OLAP queries could take minutes to hours. Due to both of these
/// factors, we conceptually view getting the true cardinality as a completely separate
/// problem from getting the estimated cardinalities of each DBMS.
///
/// When exposing a "get est card" interface, you could do it on the granularity of
/// a single SQL string or on the granularity of an entire benchmark. I chose the
/// latter for a simple reason: different DBMS might have different SQL strings
/// for the same conceptual query (see how open in tpc-kit takes in DBMS as an input).
/// When more performance tests are implemented, you would probably want to extract
/// get_name) into a generic "DBMS" trait.
Code Quality - Rustic

- Functional style
Code Quality - Testing

- **Unit tests**
  - 53 for selectivity
  - 15 for stats
  - 2.5k testing LoC
  - 90% coverage over 5.4K feature LoC

- **Integration tests**
  - SQL planner tests
  - Automated test for benchmarking

Running 53 tests

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>test::hyperloglog::tests::hll_small_strings</td>
<td>ok</td>
</tr>
<tr>
<td>test::mismatch::tests::aggregate_double</td>
<td>ok</td>
</tr>
<tr>
<td>test::counter::tests::accumulate</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_small_164</td>
<td>ok</td>
</tr>
<tr>
<td>test::mismatch::tests::aggregate_simple</td>
<td>ok</td>
</tr>
<tr>
<td>test::counter::tests::merge</td>
<td>ok</td>
</tr>
<tr>
<td>test::utils::arith_encoder::tests::encode_tests</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::weighted_merge</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::uniform_merge_sequence</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::uniform_merge_parallel</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::mismatch::tests::aggregate_zipfian</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_big</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_massive_parallel</td>
<td>ok</td>
</tr>
</tbody>
</table>

Running 15 tests

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>test::stats::hyperloglog::tests::hll_small_strings</td>
<td>ok</td>
</tr>
<tr>
<td>test::mismatch::tests::aggregate_double</td>
<td>ok</td>
</tr>
<tr>
<td>test::counter::tests::accumulate</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_small_164</td>
<td>ok</td>
</tr>
<tr>
<td>test::mismatch::tests::aggregate_simple</td>
<td>ok</td>
</tr>
<tr>
<td>test::counter::tests::merge</td>
<td>ok</td>
</tr>
<tr>
<td>test::utils::arith_encoder::tests::encode_tests</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::weighted_merge</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::uniform_merge_sequence</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::tdigest::tests::uniform_merge_parallel</td>
<td>ok</td>
</tr>
<tr>
<td>test::stats::mismatch::tests::aggregate_zipfian</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_big</td>
<td>ok</td>
</tr>
<tr>
<td>test::hyperloglog::tests::hll_massive_parallel</td>
<td>ok</td>
</tr>
</tbody>
</table>
Code Quality - Improvements

- Repetitive code for downloading/loading benchmark data
- Stats should be a logical property for stats propagation
- Robust Parquet generation
Future Tasks

- Stats propagation
- Multi-column stats (halfway supported)
- Sampling
- Integration: generate stats with ANALYZE + store in catalog
- Expression inlining, e.g. YEAR(col) < 2001
- Update statistics when data changes