Gungnir

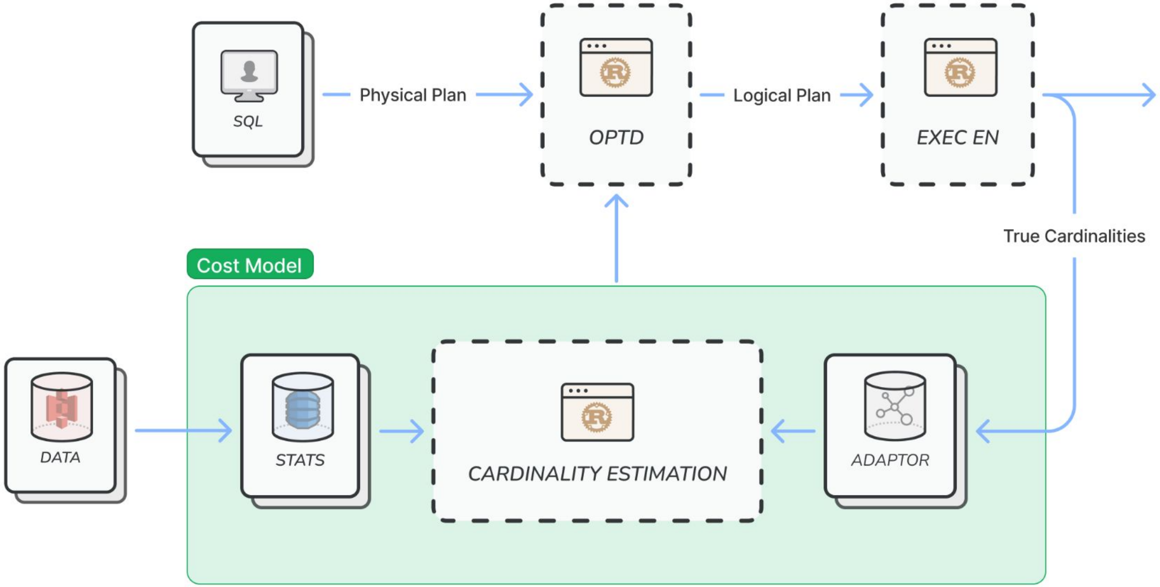
Patrick Wang, David Guo, Alexis Schlomer

Previously, on Gungnir

Goal: Estimate query run time *statically*.

Recipe 📝

1. *Extract* Statistics
2. *Estimate* Cardinality
3. *Infer* Query Cost



✅: Done, ⌛: Doing, ❌: Not Started

Today

75%

100%

125%

T-Digest ✅
HLL ✅ *MCVs* ✅

Parallelize T-Digest ❌
HLL ❌ *MCVs* ❌

Filter ✅ *Limit* ✅
Cardinality

Join ✅ *Aggregation* ✅
Cardinality

Getting TPC-H to run ✅

Cardinality benchmarking ⌛

Group Cardinality Caching ❌

Semantic Correlation ❌

✅: Done, ⌛: Doing, ❌: Not Started

Today

75% | 100% | 125%

T-Digest ✅
HLL ✅ *MCVs* ✅

Parallelize T-Digest ✅
HLL ✅ *MCVs* ✅

Filter ✅ *Limit* ✅
Cardinality

Join ✅ *Aggregation* ✅
Cardinality

Getting TPC-H to run ✅

Cardinality benchmarking ✅

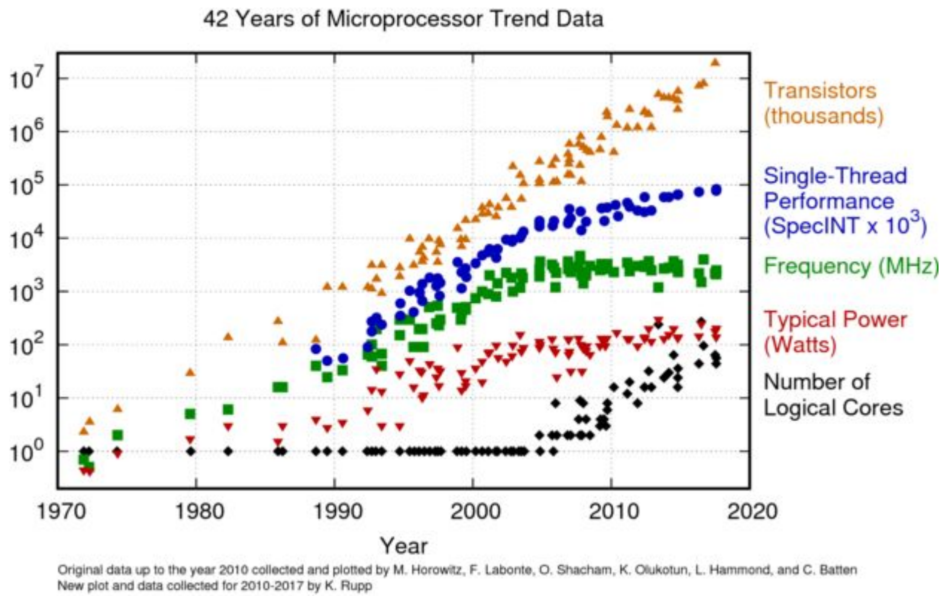
Group Cardinality Caching ✅

Semantic Correlation ✅

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!

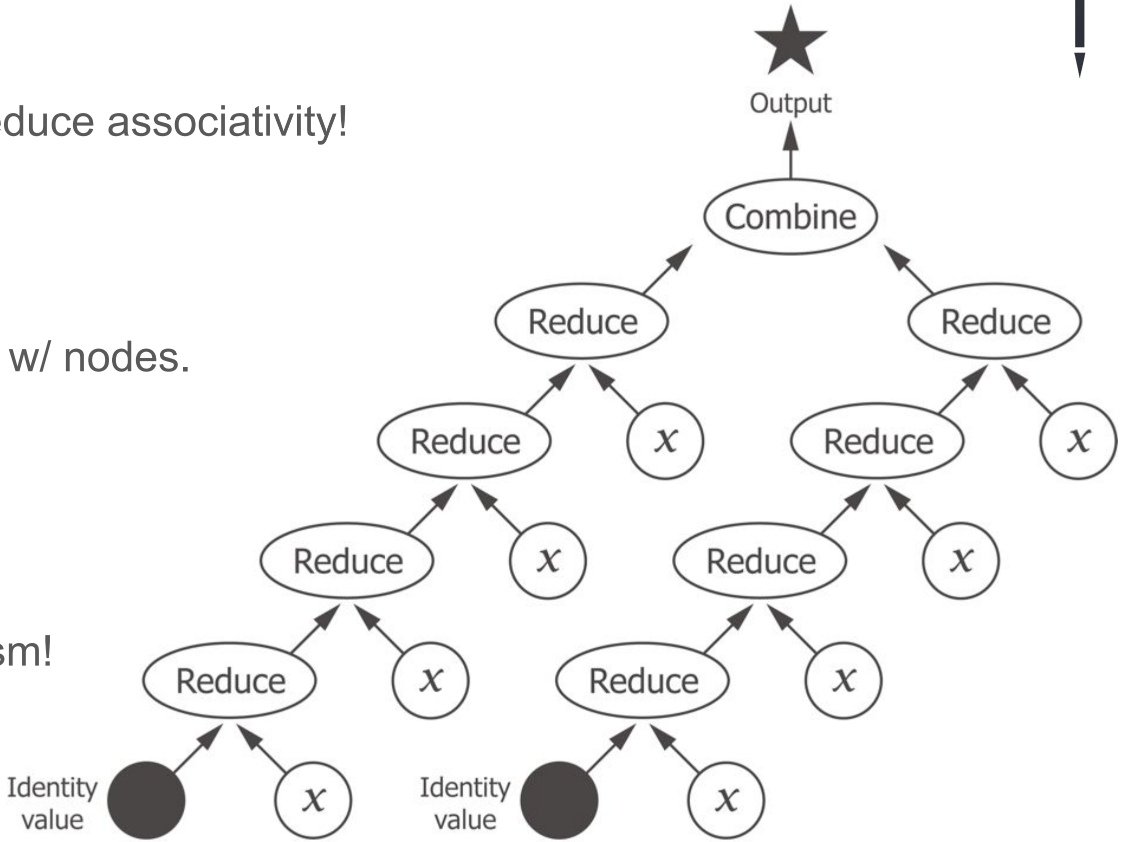
Statistics

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.~~

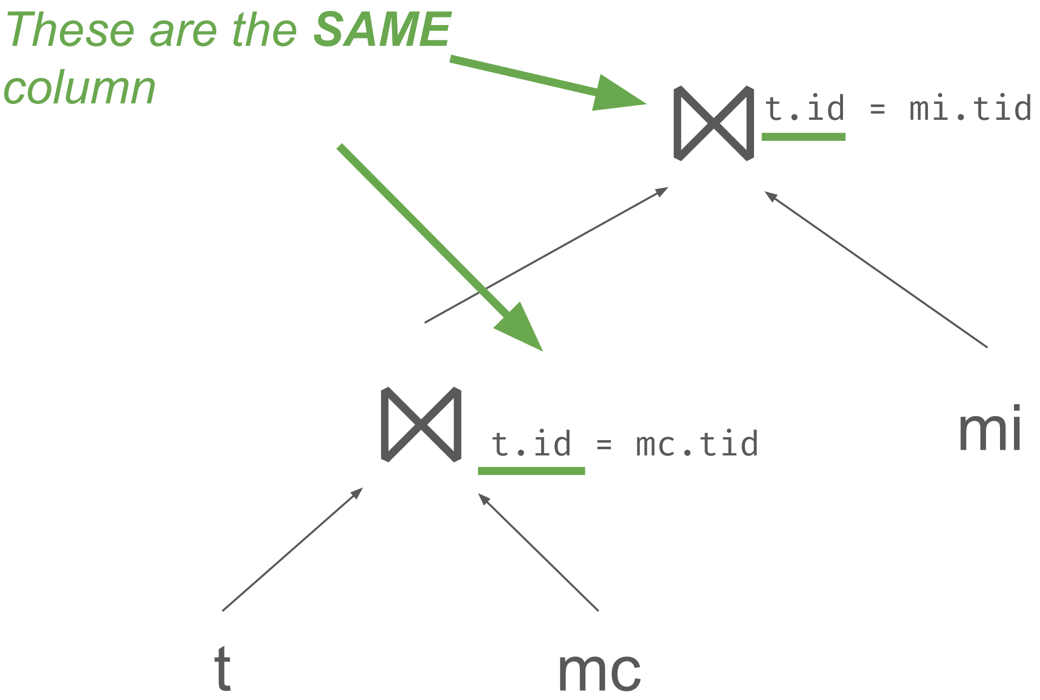
Results: *single* node ***+10 Gbps*** throughput (on SOTA hardware).

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

Cardinality Formulas

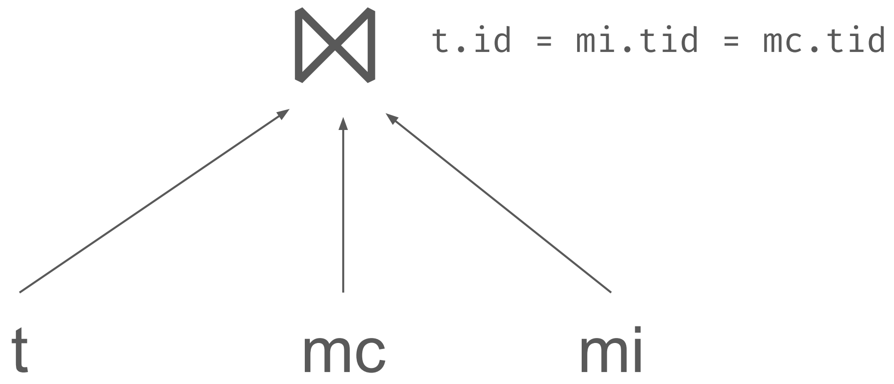
- **Filter** selectivity
  - `t1.colA [=, ≠, <, ⩽, >, ⩾] constant`
  - `AND/OR/NOT`
  - `colA IN (“advanced”, “database”, “systems”)`
  - `colA LIKE “%abc%”` using MCVs
  - `CAST`
- **Join** selectivity
  - Join types (Inner, Outer, Cross)
  - Join conditions vs. join filters
    - `t1.colA = t2.colB` vs. `t1.colA < 2`
  - Detects ***semantic correlation***
- **Aggregation**, **Limit**

Detect ***Semantic Correlation***

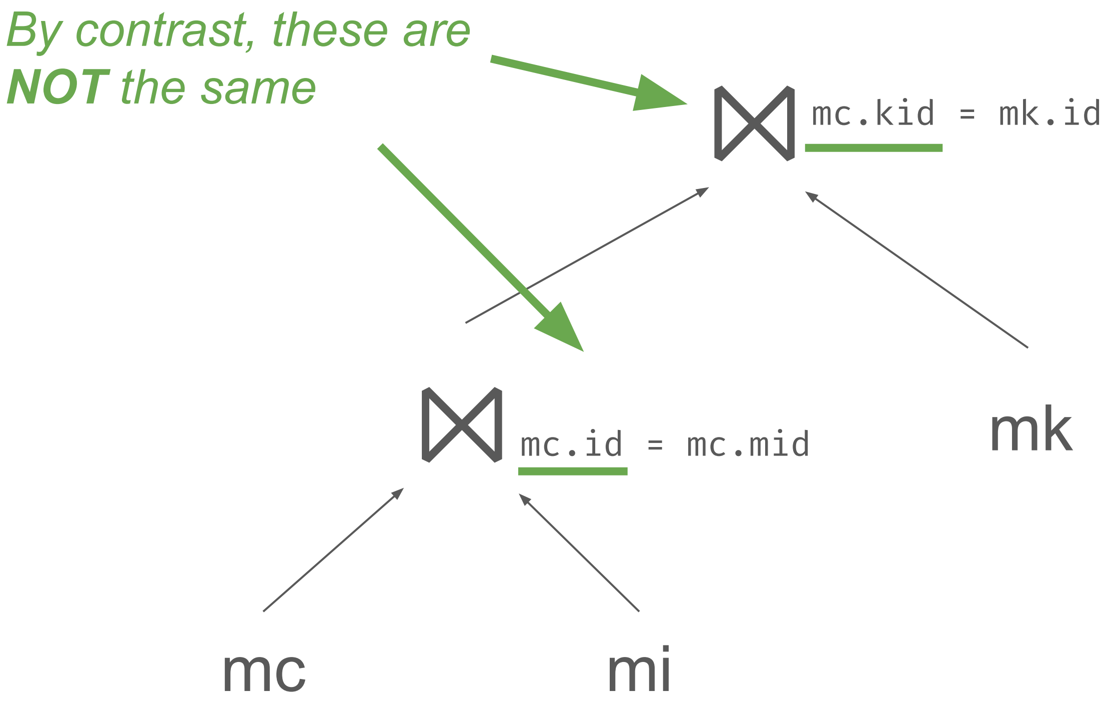


Detect ***Semantic Correlation***

Conceptually, it's a "multi-equality"

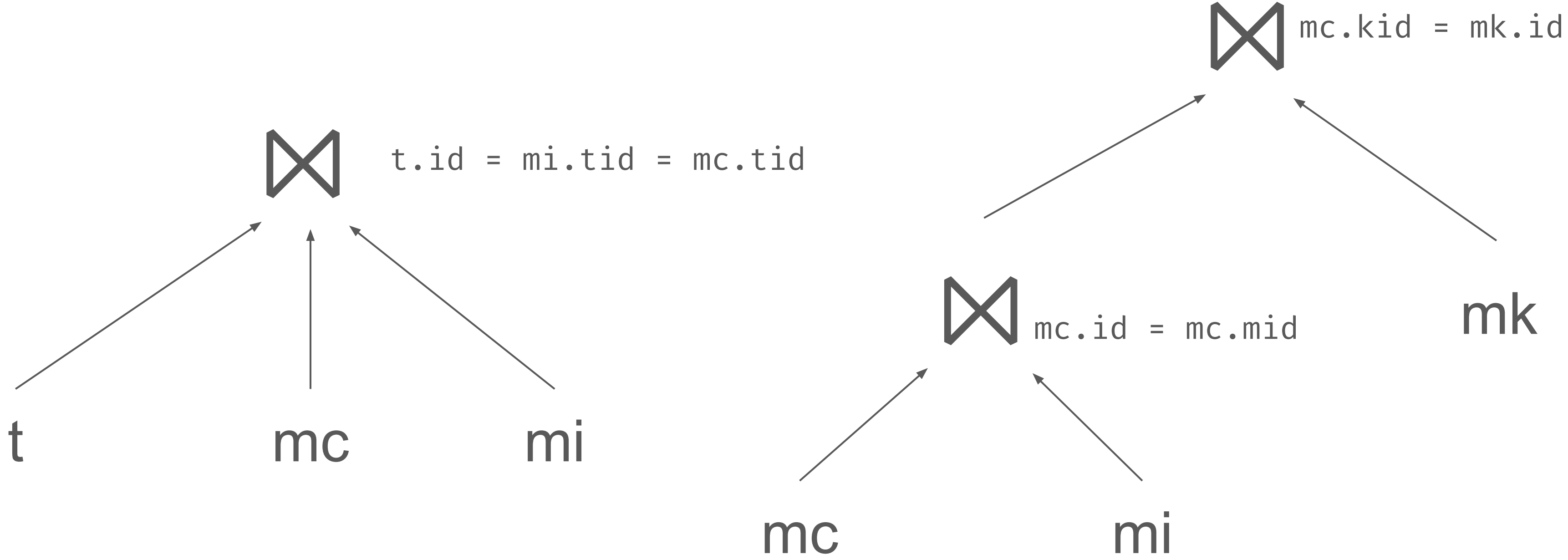


Detect ***Semantic Correlation***



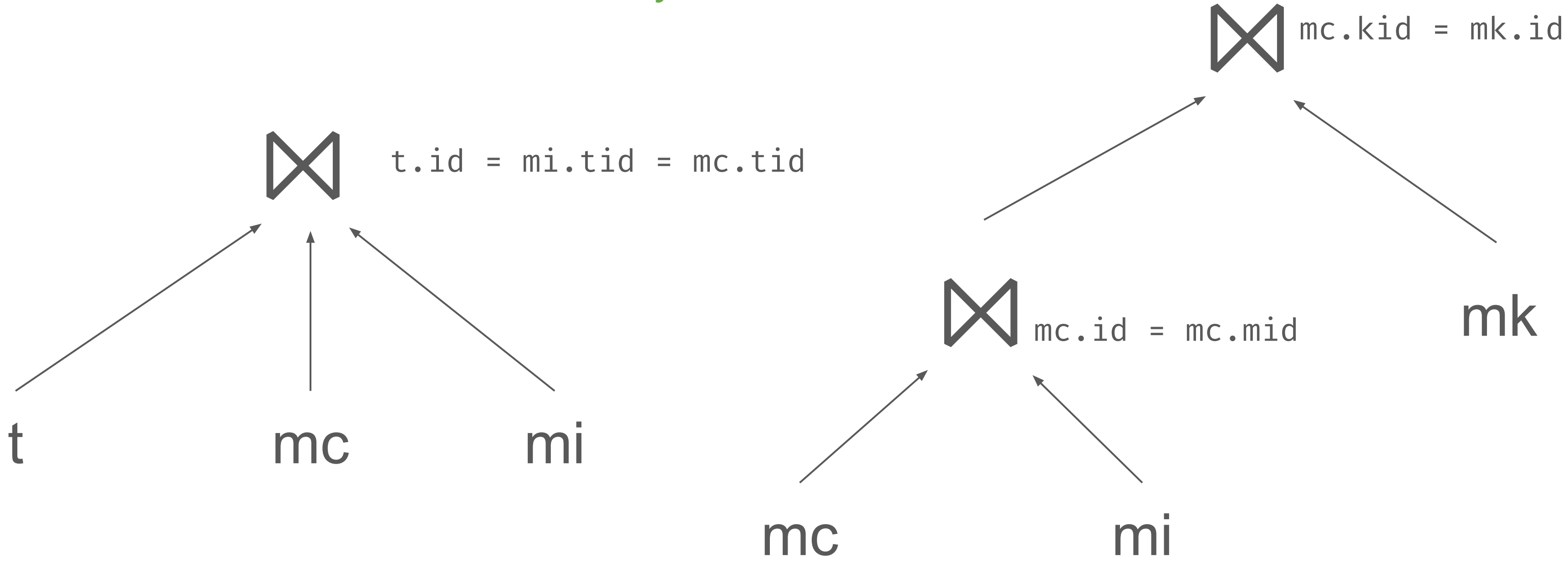
Detect ***Semantic Correlation***

Semantic correlation distinguishes between these two cases



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;
```

```
PhysicalFilter
├── cond:Eq
│   ├── Cast { cast_to: Int64, expr: #2 }
│   └── 7084(i64)
├── cost: weighted=45420266.24,row_cnt=21.63,compute=40896336.24,io=4523930.00
└── PhysicalScan { table: movie_keyword, cost: weighted=4523930.00,row_cnt=4523930.00,
```

Is 414 rows in reality

We underestimate by 20x

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;
```

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

Run this first!

```
PhysicalFilter
├── cond:Eq
│   ├── Cast { cast_to: Int64, expr: #2 }
│   └── 7084(i64)
├── cost: weighted=45420256.20,row_cnt=414.00,compute=40896327.20,io=4523929.00
└── PhysicalScan { table: movie_keyword, cost: weighted=4523929.00,row_cnt=4523929.00,
```

No longer underestimating
Leads to 17x better Q-Error

Results - *Cardinality Estimation* Accuracy

TPC-H (SF1)

	PG	Optd
# Better	1	**3**
# Tied	***9***	***9***
p50	3.50	**1.00**
p90	1203.0	**100.00**
p99	**1517.5**	31250

JOB*

	PG	Optd
# Better	21	**39**
# Tied	***33***	***33***
p50	209.33	**80.00**
p90	**8546.2**	128548
p99	**42963**	4.0e11

JOB-light

	PG	Optd
# Better	7	**51**
# Tied	***0***	***0***
p50	5.73	**3.10**
p90	69.31	**13.28**
p99	7887.4	**7382.1**

Shows we do well in…

operator variety

complex predicates

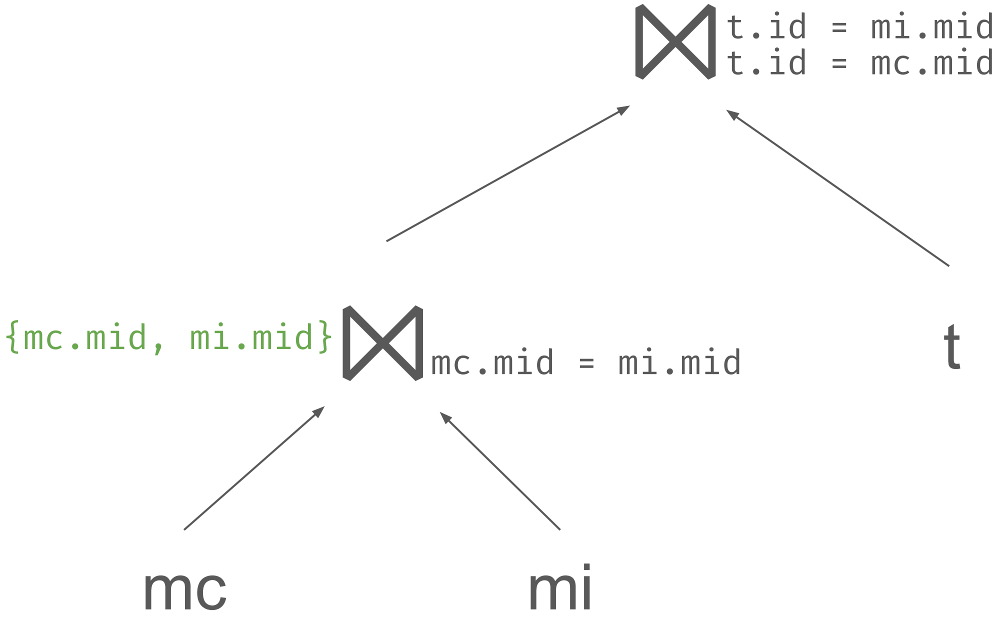
pure join estimation

**with stats from Python csv2parquet script*

Detect ***Semantic Correlation***

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

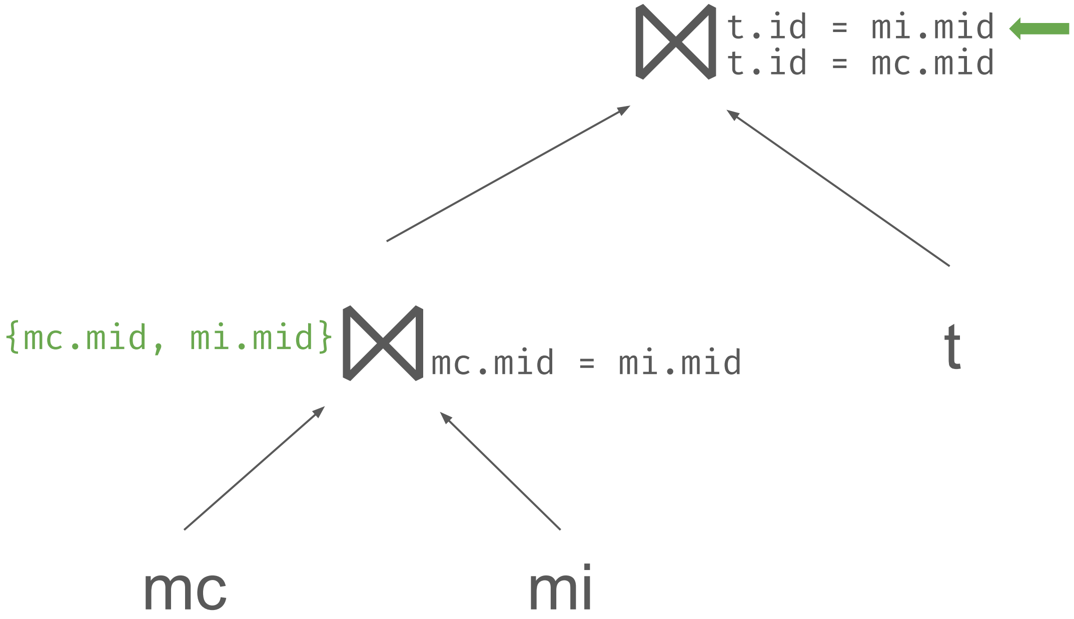
{mc.mid, mi.mid}



Detect ***Semantic Correlation***

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

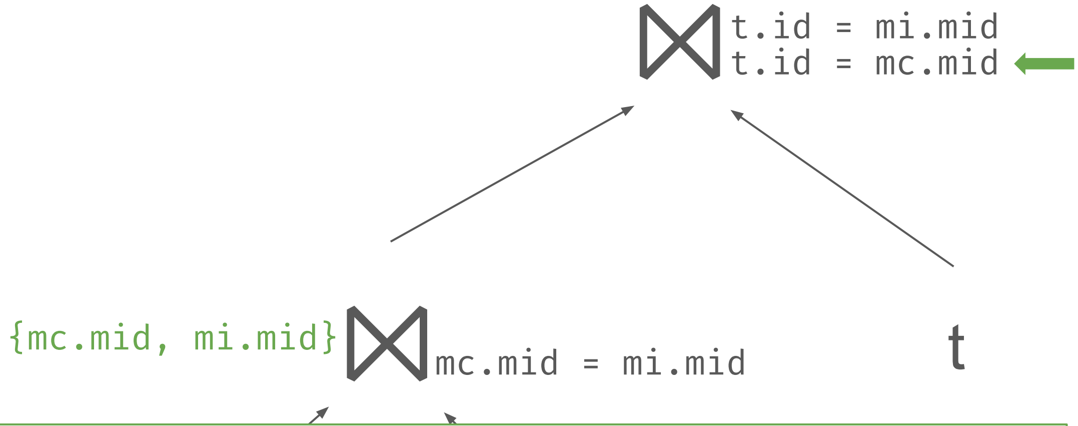
`{mc.mid, mi.mid, t.id}`



Detect *Semantic Correlation*

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

{mc.mid, mi.mid, t.id}



Selectivity Adjustment Factor

s.t. total selectivity = 1 / (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`

```
    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
            ├── aggrs:Agg(Sum)
            │   └── Mul
            │       ├── #0
            │       └── Sub
            │           ├── 1
            │           └── #1
            ├── 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=1
00.34,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: weight
ed=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, #1
0, #11, #12, #13 ], 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

Query 27b

DBMS	Q-Error	Est. Card.	True Card.
DataFusion	3310.7654320987654	81	268172
Postgres	7887.411764705882	34	268172

Aggregate Q-Error Information

DBMS	Median	P90	P95	P99	# Inf	Mean	Min	Max
DataFusion	3.10	13.28	152.55	7382.12	0	350.17	1.02	9202.55
Postgres	5.73	69.31	107.62	7887.41	0	436.41	1.06	10166.63

Comparative Q-Error Information

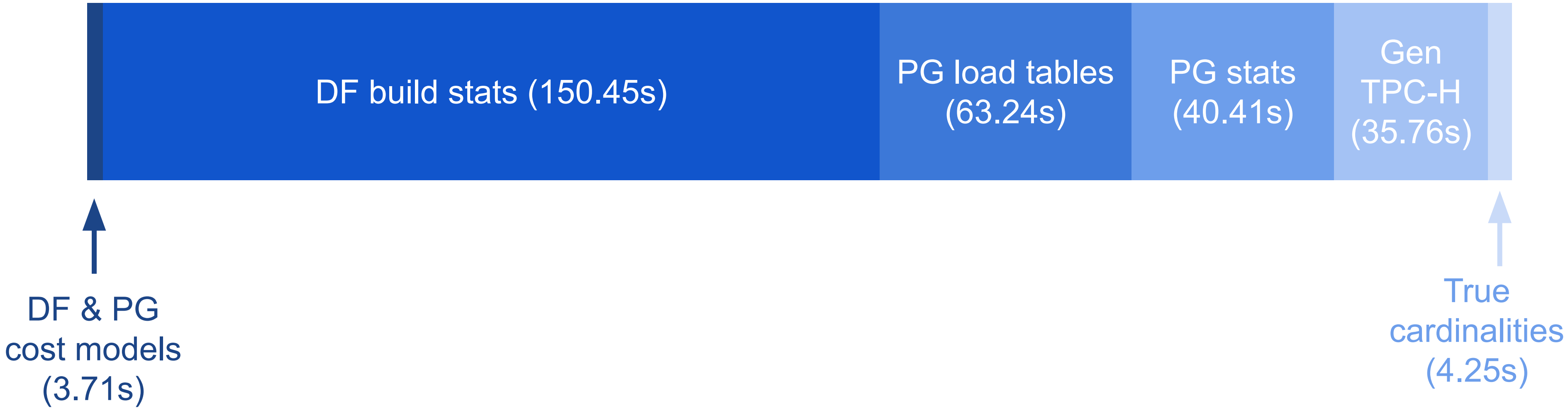
DBMS	# Best	# Tied Best
DataFusion	51	0
Postgres	7	0

```
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**



Code Quality - Modularity

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

```
pub trait CostModel<T: RelNodeTyp>: 'static + Send + Sync {
    fn compute_cost(
        &self,
        node: &T,
        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 {
            self.accumulate(&mut total_cost, input);
        }
        total_cost
    }

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

```
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;
}
```

```
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;
}
```

```
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/ndistinct of
/// the N-1 nodes with the highest ndistinct 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 tables). 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` to it.
```

```
/// 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 psql 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 DBMSs 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 DBMSs might have different SQL strings
///   for the same conceptual query (see how qgen in tpch-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

```rust
let (hlls, mgs, null_cnts) = receiver
    .into_iter()
    .par_bridge()
    .fold(Self::first_pass_stats_id(nb_stats), |local_stats, batch| {
        let mut local_stats = local_stats?;

        match batch {
            Ok(batch) => {
                let (hlls, mgs, null_cnts) = &mut local_stats;
                let comb = Self::get_column_combs(&batch, &comb_stat_types);
                Self::generate_partial_stats(&comb, mgs, hlls, null_cnts);
                Ok(local_stats)
            }
            Err(e) => {
                println!("Err: {:?},, {:?}", e, comb_stat_types.len());
                Err(e.into())
            }
        }
    })
    .reduce(
        Self::first_pass_stats_id(nb_stats),
        |final_stats, local_stats| {
            let mut final_stats = final_stats?;
            let local_stats = local_stats?;

            let (final_hlls, final_mgs, final_counts) = &mut final_stats;
            let (local_hlls, local_mgs, local_counts) = local_stats;

            for i in 0..nb_stats {
                final_hlls[i].merge(&local_hlls[i]);
                final_mgs[i].merge(&local_mgs[i]);
                final_counts[i] += local_counts[i];
            }

            Ok(final_stats)
        },
    )?;
```

```rust
let (distrs, cnts, row_cnts) = receiver
    .into_iter()
    .par_bridge()
    .fold(
        Self::second_pass_stats_id(&comb_stat_types, &mgs, nb_stats),
        |local_stats, batch| {
            let mut local_stats = local_stats?;

            match batch {
                Ok(batch) => {
                    let (distrs, cnts, row_cnts) = &mut local_stats;
                    let comb = Self::get_column_combs(&batch, &comb_stat_types);
                    Self::generate_full_stats(&comb, cnts, distrs, row_cnts);
                    Ok(local_stats)
                }
                Err(e) => Err(e.into()),
            }
        },
    )
    .reduce(
        Self::second_pass_stats_id(&comb_stat_types, &mgs, nb_stats),
        |final_stats, local_stats| {
            let mut final_stats = final_stats?;
            let local_stats = local_stats?;

            let (final_distrs, final_cnts, final_counts) = &mut final_stats;
            let (local_distrs, local_cnts, local_counts) = local_stats;

            for i in 0..nb_stats {
                final_cnts[i].merge(&local_cnts[i]);
                if let (Some(final_distr), Some(local_distr)) =
                    (&mut final_distrs[i], &local_distrs[i])
                {
                    final_distr.merge(local_distr);
                    final_distr.norm_weight += local_distr.norm_weight;
                }

                final_counts[i] += local_counts[i];
            }

            Ok(final_stats)
        },
    )?;
```

```rust
base_col_refs
    .into_iter()
    .map(|base_col_ref| {
        match self.get_column_comb_stats(&base_col_ref.table, &[base_col_ref.col_idx]) {
            Some(per_col_stats) => per_col_stats.ndistinct,
            None => DEFAULT_NUM_DISTINCT,
        }
    })
    .map(|ndistinct| 1.0 / ndistinct as f64)
    .sorted_by(|a, b| {
        a.partial_cmp(b)
            .expect("No floats should be NaN since n-distinct is never 0")
    })
    .take(num_base_col_refs - 1)
    .product()
```

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 15 tests
test stats::hyperloglog::tests::hll_small_strings ... ok
test stats::misragries::tests::aggregate_double ... ok
test stats::counter::tests::aggregate ... ok
test stats::hyperloglog::tests::hll_small_u64 ... ok
test stats::misragries::tests::aggregate_simple ... ok
test stats::murmur2::tests::murmur_string ... ok
test stats::counter::tests::merge ... ok
test utils::arith_encoder::tests::encode_tests ... ok
test stats::tdigest::tests::weighted_merge ... ok
test stats::tdigest::tests::uniform_merge_sequential ... ok
test stats::tdigest::tests::uniform_merge_parallel ... ok
test stats::misragries::tests::aggregate_zipfian ... ok
test stats::misragries::tests::merge_zipfians ... ok
test stats::hyperloglog::tests::hll_big ... ok
test stats::hyperloglog::tests::hll_massive_parallel ... ok
```

```
     Running unittests src/lib.rs (target/debug/deps/optd_datafusion_repr-cf772101246a0024)

running 53 tests
test cost::base_cost::filter::tests::test_cast_colref_eq_colref ... ok
test cost::base_cost::filter::tests::test_cast_colref_eq_value ... ok
test cost::base_cost::filter::tests::test_colref_eq_constint_in_mcv ... ok
test cost::base_cost::filter::tests::test_colref_eq_cast_value ... ok
test cost::base_cost::filter::in_list::tests::test_in_list ... ok
test cost::base_cost::filter::tests::test_and ... ok
test cost::base_cost::filter::tests::test_colref_eq_constint_not_in_mcv_no_nulls ... ok
test cost::base_cost::filter::tests::test_colref_eq_constint_not_in_mcv_with_nulls ... ok
test cost::base_cost::filter::tests::test_colref_geq_constint_no_nulls ... ok
test cost::base_cost::filter::tests::test_colref_geq_constint_with_nulls ... ok
test cost::base_cost::filter::tests::test_colref_gt_constint_no_nulls ... ok
test cost::base_cost::filter::tests::test_colref_gt_constint_with_nulls ... ok
test cost::base_cost::filter::tests::test_colref_leq_constint_no_mcvs_in_range ... ok
test cost::base_cost::filter::tests::test_colref_leq_constint_no_mcvs_in_range_with_nulls ... ok
test cost::base_cost::filter::tests::test_colref_leq_constint_with_mcv_at_border ... ok
test cost::base_cost::filter::tests::test_colref_leq_constint_with_mcvs_in_range_not_at_border ... ok
test cost::base_cost::filter::tests::test_colref_lt_constint_no_mcvs_in_range ... ok
test cost::base_cost::filter::tests::test_colref_lt_constint_no_mcvs_in_range_with_nulls ... ok
test cost::base_cost::filter::tests::test_colref_lt_constint_with_mcv_at_border ... ok
test cost::base_cost::filter::like::tests::test_like_with_nulls ... ok
test cost::base_cost::filter::like::tests::test_like_no_nulls ... ok
test cost::base_cost::filter::tests::test_colref_lt_constint_with_mcvs_in_range_not_at_border ... ok
test cost::base_cost::filter::tests::test_colref_neq_constint_in_mcv ... ok
test cost::base_cost::filter::tests::test_const ... ok
test cost::base_cost::filter::tests::test_not_no_nulls ... ok
test cost::base_cost::filter::tests::test_not_with_nulls ... ok
test cost::base_cost::filter::tests::test_or ... ok
test cost::base_cost::join::tests::test_inner_colref_eq_colref_same_table_is_not_oncond ... ok
test cost::base_cost::join::tests::test_inner_and_of_filters ... ok
test cost::base_cost::join::tests::test_inner_const ... ok
test cost::base_cost::join::tests::test_inner_and_of_oncond_and_filter ... ok
test cost::base_cost::join::tests::test_inner_oncond ... ok
test cost::base_cost::join::tests::test_inner_and_of_onconds ... ok
test cost::base_cost::join::tests::test_join_which_connects_two_components_together ... ok
test cost::base_cost::join::tests::test_outer_nonunique_oncond_inner_sometimes_lt_rowcnt ... ok
test cost::base_cost::join::tests::test_outer_nonunique_oncond_inner_always_geq_rowcnt ... ok
test cost::base_cost::join::tests::test_outer_unique_oncond_filter ... ok
test cost::base_cost::join::tests::test_outer_unique_oncond ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_1_0_2_expects ... ok
test plan_nodes::macros::test::test_explain_complex_data ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_1_1_2_expects ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_1_0_2_1_2_expects ... ok
test properties::column_ref::tests::test_eq_base_table_column_sets ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_1_expects ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_2_1_2_expects ... ok
test rules::filter_pushdown::tests::filter_merge ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_1_2_expects ... ok
test cost::base_cost::join::tests::test_three_table_join_for_initial_join_on_conds::_0_2_expects ... ok
test rules::filter_pushdown::tests::push_past_agg ... ok
test rules::filter_pushdown::tests::push_past_sort ... ok
test rules::filter_pushdown::tests::push_past_proj_basic ... ok
test rules::filter_pushdown::tests::push_past_proj_adv ... ok
test rules::filter_pushdown::tests::push_past_join_conjunction ... ok

test result: ok. 53 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
```

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