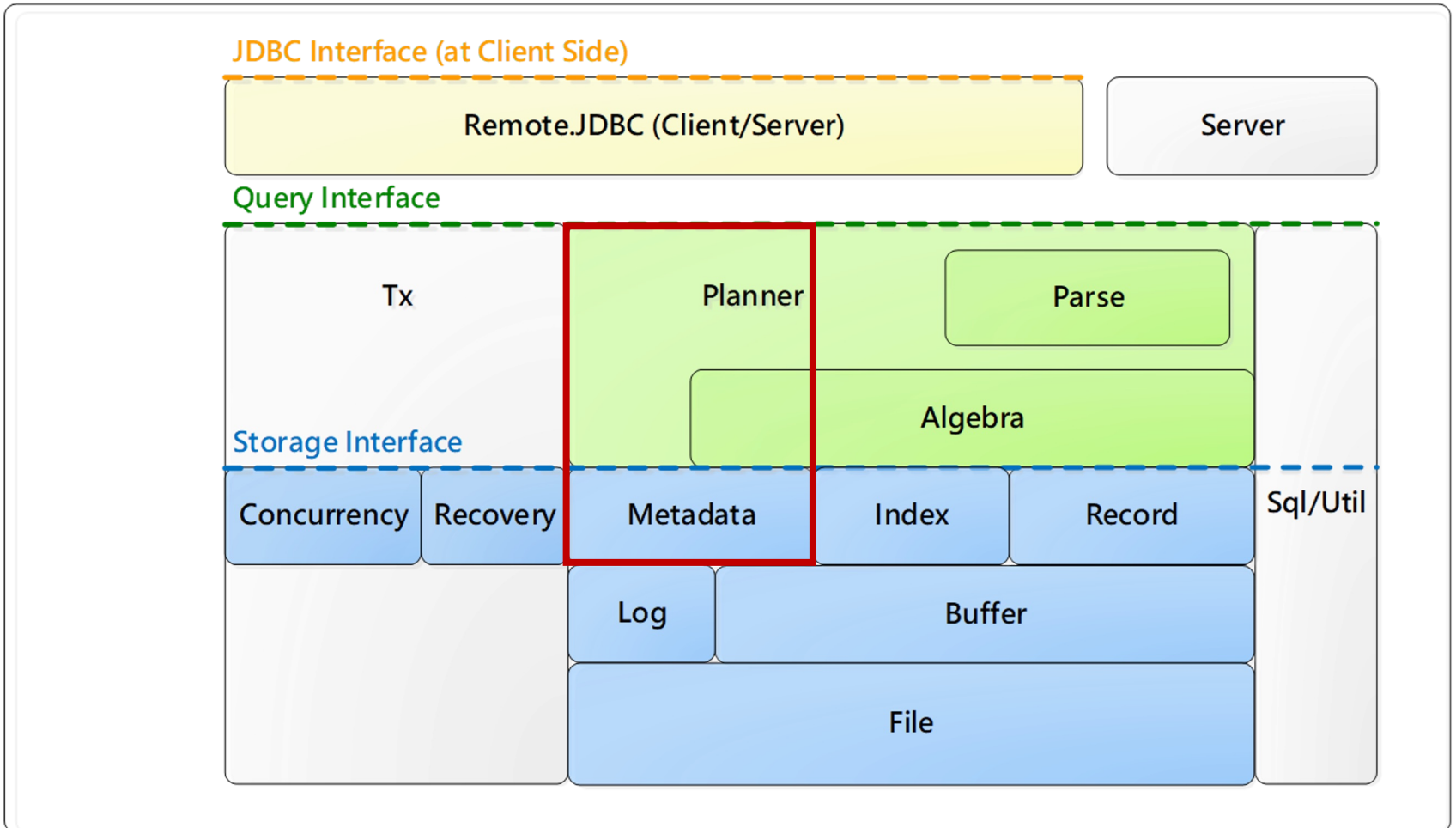


Query Optimization

Shan-Hung Wu and DataLab
CS, NTHU

Where Are We?

VanillaCore



Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

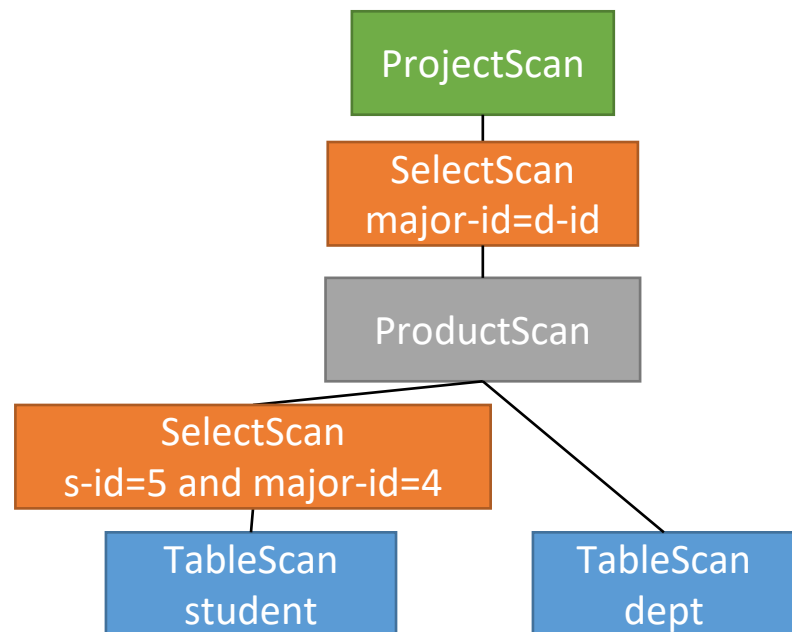
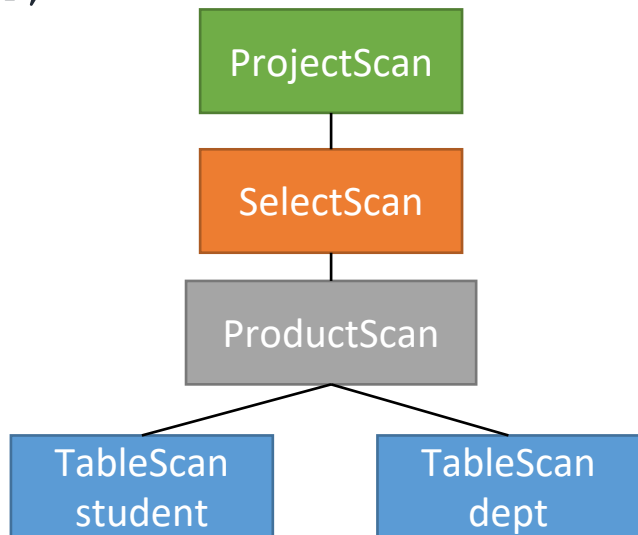
Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

SQL and Relational Algebra

- A SQL command can be expressed as multiple trees in relational algebra

```
SELECT sname FROM student, dept
WHERE major-id = d-id AND s-id = 5 AND major-id =
4;
```



Query Optimization

- A good scan tree can be faster than a bad one for orders of magnitude
- Query optimizer:
 1. Generate candidate plan trees
 2. Estimate cost of each corresponding scan tree (not discussed yet)
 3. Pick and open the “best” one to execute query
- Goal (ideally): find the one with least cost
- Goal (in practice): avoid bad trees

Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Push Select Down
 - Join Order Problem
 - Heuristic Planner in VanillaCore
- Selinger-Style Query Optimizer

Metric for Cost

- Cost of a query?
 - To user: query delay
 - Low delay also implies better system throughput
-
- Typically, I/O delay dominates query delay

Cost Estimation

- For each plan/table p , we estimate $B(p)$
 - #blocks accessed by the corresponding scan
- Usually, estimating $B(p)$ requires more knowledge:
 - $R(p)$: #records output
 - *Search cost* (#blocks) of index, if used
 - $V(p,f)$: #distinct values for field f in p

Estimating $B(p)$

p	$B(p)$
TablePlan	Actual #blocks cached by StatMgr (via periodic table scanning)
ProjectPlan(c)	$B(c)$
SelectPlan(c)	$B(c)$
IndexSelectPlan(t)	$\text{IndexSearchCost}(R(t), R(p)) + R(p)$
ProductPlan(c1, c2)	$B(c1) + (R(c1) * B(c2))$
IndexJoinPlan(c1, t2)	$B(c1) + (R(c1) * \text{IndexSearchCost}(R(t2), 1)) + R(p)$

- $B(c)$ is evaluated recursively down to the table level

For Any p , We Need to Estimate $R(p)$ and Index Search Cost

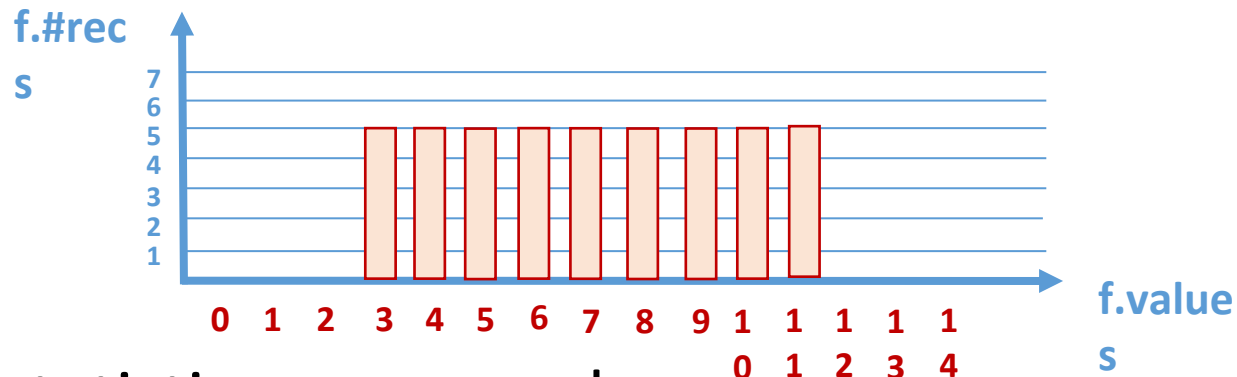
- Index Search Cost:
 - `HashIndex.searchCost()`
 - `BTreeIndex.searchCost()`
- Estimating $R(p)$ is called ***cardinality estimation***

Outline

- Overview
- Cost Estimation
 - **Cardinality Estimation**
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Naïve Approach

- Uniform assumption
 - All values in field appear with the same probability



- Few statistics are enough:

R(c)	#records in child plan c
V(c, f)	#distinct values in field f in c
Max(c, f)	Max value in field f in c
Min(c, f)	Min value in field f in c

$p = \text{Select}(c, f=x)$

- $R(p)$?

$R(c)$	#records in child plan c
$V(c, f)$	#distinct values in field f in c
$\text{Max}(c, f)$	Max value in field f in c
$\text{Min}(c, f)$	Min value in field f in c

- $\text{Selectivity}(f=x) = \frac{1}{V(c, f)}$
- $R(p) = \text{Selectivity}(f=x) * R(c)$

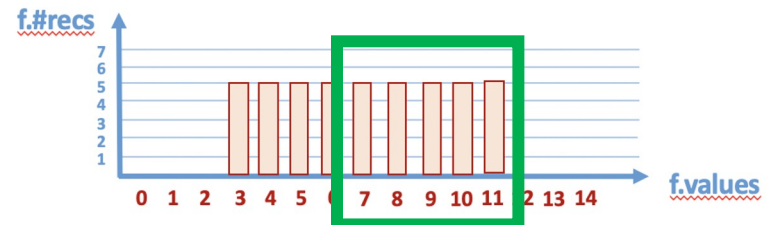


$p = \text{Select}(c, f > x)$

- $R(p)$?

$R(c)$	#records in child plan c
$V(c, f)$	#distinct values in field f in c
$\text{Max}(c, f)$	Max value in field f in c
$\text{Min}(c, f)$	Min value in field f in c

- $\text{Selectivity}(f > x) = \frac{\text{Max}(c, f) - x}{\text{Max}(c, f) - \text{Min}(c, f)}$
- $R(p) = \text{Selectivity}(f > x) * R(c)$

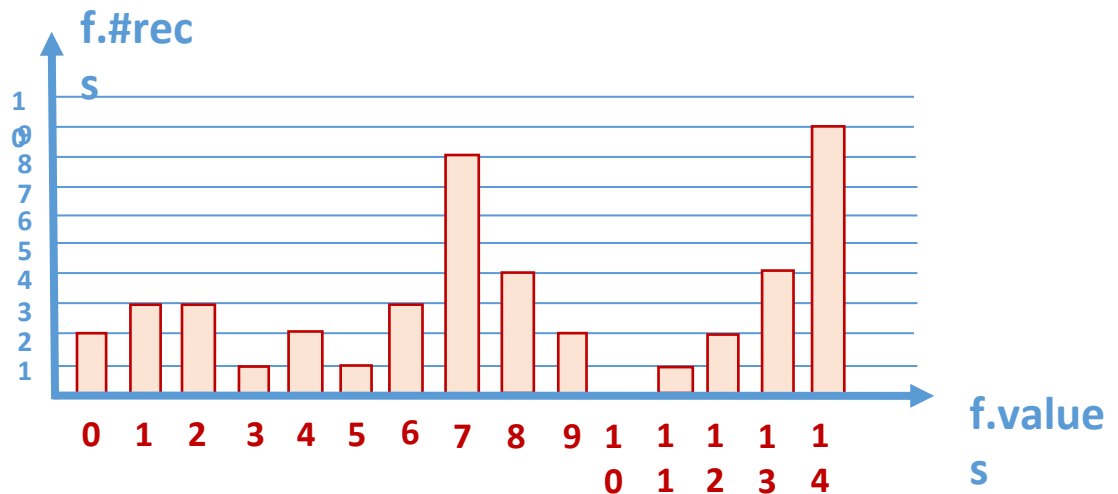


Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - **Histogram-based Estimation**
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

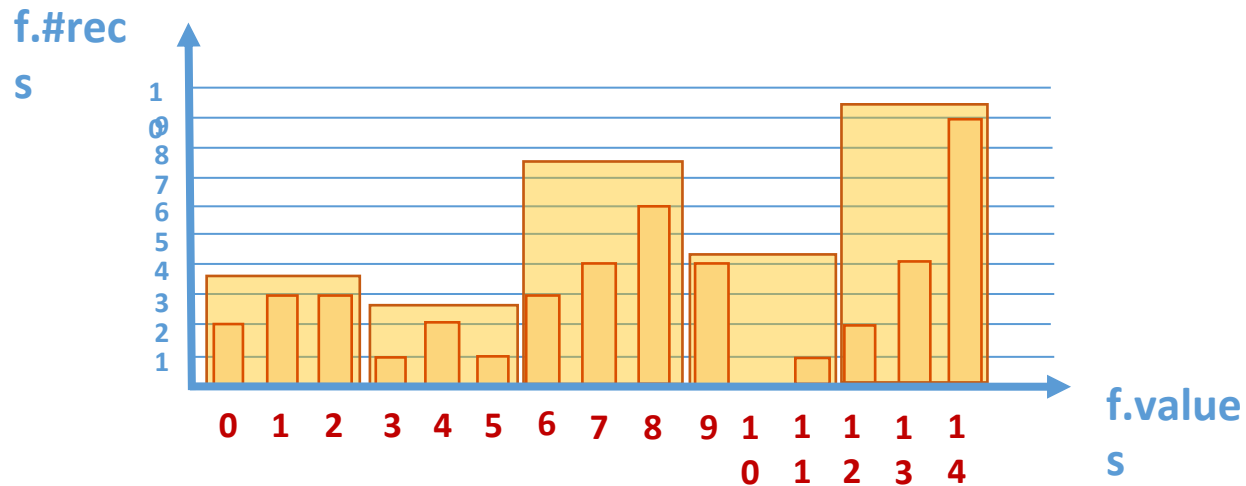
Naïve Estimation is Inaccurate

- In the real world, values in a field are seldom uniform distributed
- $p = \text{Select}(c, f=14)$
- Estimated $R(p) = \frac{1}{15} * R(c) = 3$
- Actually, $R(p) = 9$



Histogram

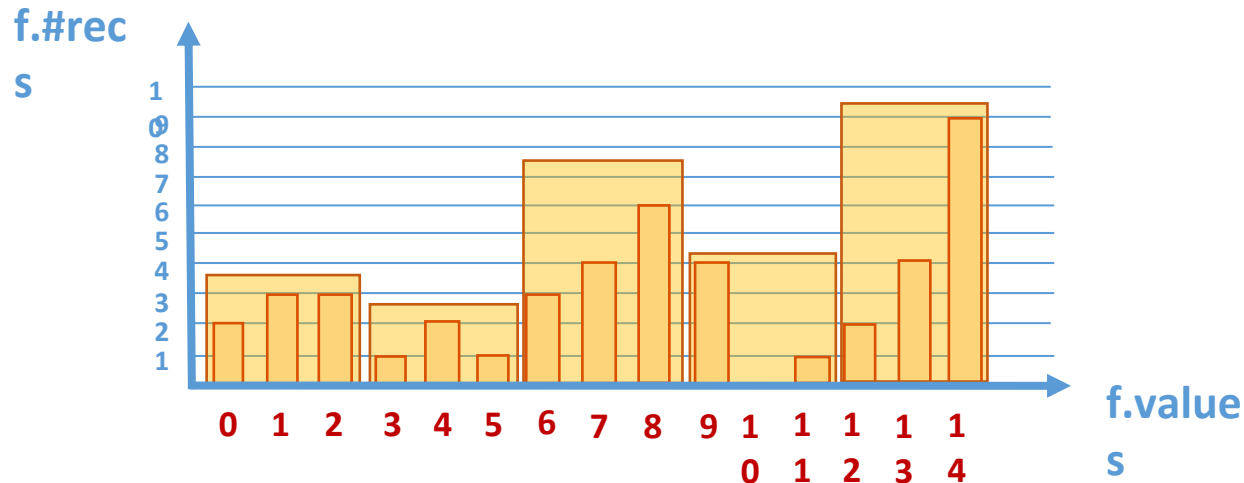
- Approximates value distribution in every field
- Partitions field values into a set of *buckets*



- More #buckets, more accurate approximation
 - Tradeoff between accurate and storage cost

Buckets

- Each bucket b collects statistics of a value range
 - Assumes ***uniform distribution of records and values*** in b



- $R(p, f, b)$: #records
- $V(p, f, b)$: #distinct values
- $\text{Range}(p, f, b)$: value range

Cardinality Estimation

- Not matter what p is, we have

$$R(p) = \sum_{b \in p.hist.buckets(f)} R(p, f, b)$$

for any f

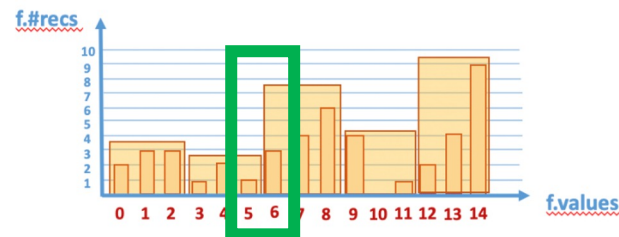
- Problem: how to construct the histogram?

Range Selection (1/2)

- $p = \text{Select}(c, f \text{ in Range})$
- For each bucket b *in f*:
 - $\text{Selectivity} = \frac{|\text{Range}(c, f, b) \cap \text{Range}|}{|\text{Range}(c, f, b)|}$
 - $\text{Range}(p, f, b) = \text{Range}(c, f, b) \cap \text{Range}$
 - $V(p, f, b) = V(c, f, b) * \text{selectivity}$
 - $R(p, f, b) = R(c, f, b) * \text{selectivity}$
- Assumptions:
 - #Records in a bucket are uniformly distributed
 - Values in a bucket are uniformly distributed

Given $\forall f, b$:

$\text{Range}(c, f, b)$
 $V(c, f, b)$
 $R(c, f, b)$



Range Selection (2/2)

- $p = \text{Select}(c, f \text{ in Range})$
- For each bucket b in $f' \neq f$:
 - Reduction = $\frac{\sum_b R(p, f', b)}{R(c)}$
 - $\text{Range}(p, f', b) = \text{Range}(c, f', b)$
 - $R(p, f', b) = R(c, f', b) * \text{Reduction}$
 - $V(p, f', b) = \min(V(c, f', b), R(p, f', b))$
- Assumptions:
 - Values in different fields are independent with each other

Given $\forall f, b$:

$\text{Range}(c, f, b)$

$V(c, f, b)$

$R(c, f, b)$

Product

- $p = \text{Product}(c1, c2)$
- For each (b,f) in $c1$:
 - $\text{Range}(p,f,b) = \text{Range}(c1,f,b)$
 - $V(p,f,b) = V(c1,f,b)$
 - $R(p,f,b) = R(c1,f,b) * R(c2)$
- For each (b,f) in $c2$:
 - $\text{Range}(p,f,b) = \text{Range}(c2,f,b)$
 - $V(p,f,b) = V(c2,f,b)$
 - $R(p,f,b) = R(c2,f,b) * R(c1)$

Given $\forall f,b$:

$\text{Range}(c1, f, b)$

$V(c1, f, b)$

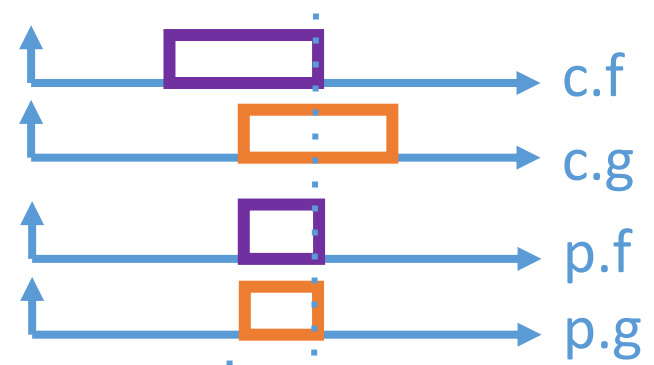
$R(c1, f, b)$

$\text{Range}(c2, f, b)$

$V(c2, f, b)$

$R(c2, f, b)$

Join Selection (1/2)



- $p = \text{Select}(c, f=g) \text{ or } \text{Join}(a, b, a.f=b.g)$
- For each bucket **b1** in f and **b2** in g:
 - $\text{Range}(p,f,b1) = \text{Range}(p,g,b2) = IR = \text{Range}(c,f,b1) \cap \text{Range}(c,g,b2)$
 - $V(p,f,b1) = V(p,g,b2) = \min V = \min\left(\frac{|IR| * V(c,f,b1)}{|\text{Range}(c,f,b1)|}, \frac{|IR| * V(c,g,b2)}{|\text{Range}(c,g,b2)|}\right)$
 - $R1 = R(c,f,b1) * \frac{\text{Match rate with recs in } b2}{V(c,f,b1)} * \frac{1}{V(c,g,b2)} * \frac{R(c,g,b2)}{R(c)}$

Match rate with b2
 - $R2 = R(c,g,b2) * \frac{\min V}{V(c,g,b2)} * \frac{1}{V(c,f,b1)} * \frac{R(c,f,b1)}{R(c)}$
 - $R(p,f,b1) = R(p,g,b2) = \min(R1, R2)$
- Assumptions:
 - #Records & values in bucket are uniformly distributed
 - All values in the range having smaller number of values appear in the range having larger number of values
 - Values in different fields are independent with each other

Join Selection (2/2)

- $p = \text{Select}(c, f=g)$
- For each bucket b in $f' \notin \{f, g\}$:
 - Reduction = $\frac{\sum_b R(p, f, b)}{R(c)}$
 - $R(p, f', b) = R(c, f', b) * \text{Reduction}$
 - $V(p, f', b) = \min(V(c, f', b), R(p, f', b))$
 - $\text{Range}(p, f', b) = \text{Range}(c, f', b)$
- Assumptions:
 - Values in different fields are independent with each other

Cost Estimation in VanillaCore

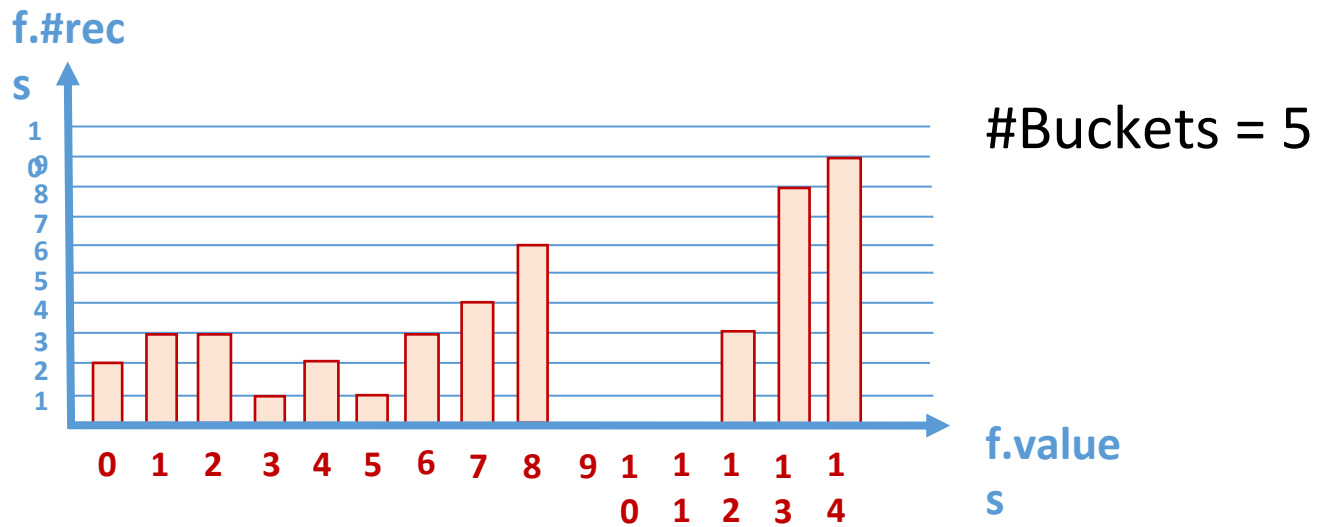
- $B(p)$: `p.blocksAccessed()`
- Histogram-based cardinality estimation:
 - $R(p)$: `p.histogram().recordsOutput()`
 - $V(p,f)$: `p.histogram().distinctVaues(f)`
- Each plan builds its own histogram in constructor
- Important utility methods to trace:
 - `SelectPlan.constantRangeHistogram()`
 - `ProductPlan.productHistogram()`
 - `SelectPlan.joinFieldHistogram()`
 - `AbstractJointPlan.joinHistogram()`

Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

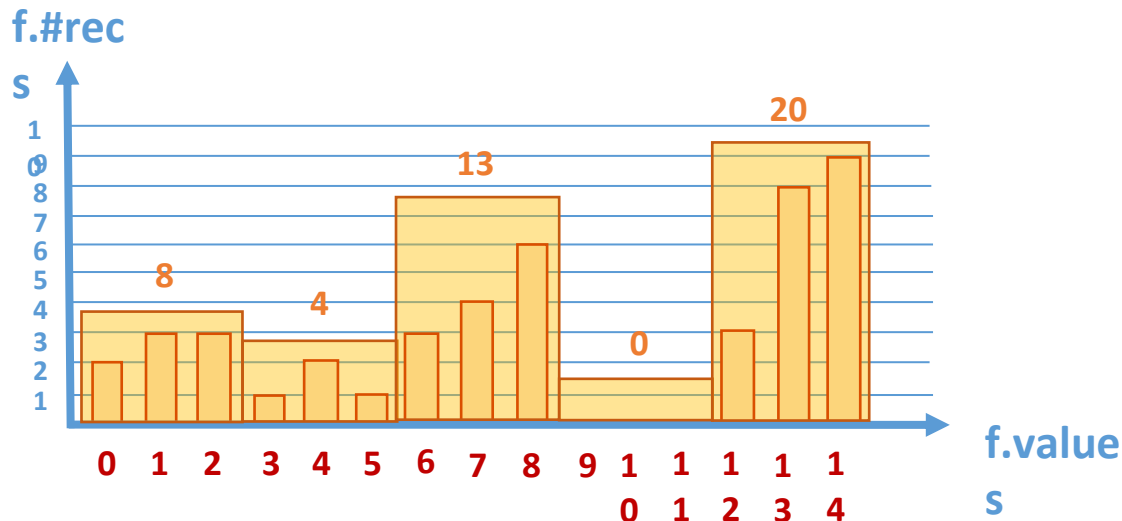
Table Histogram at Lowest-Level

- Data structure that approximates value distribution
- Partitions field values into a set of *buckets*
- Each bucket b collects statistics of a value range
 - Assumes *uniform distribution of records and values* in b
- Given a fixed #buckets, how to decide bucket ranges?



Equi-Width Histogram

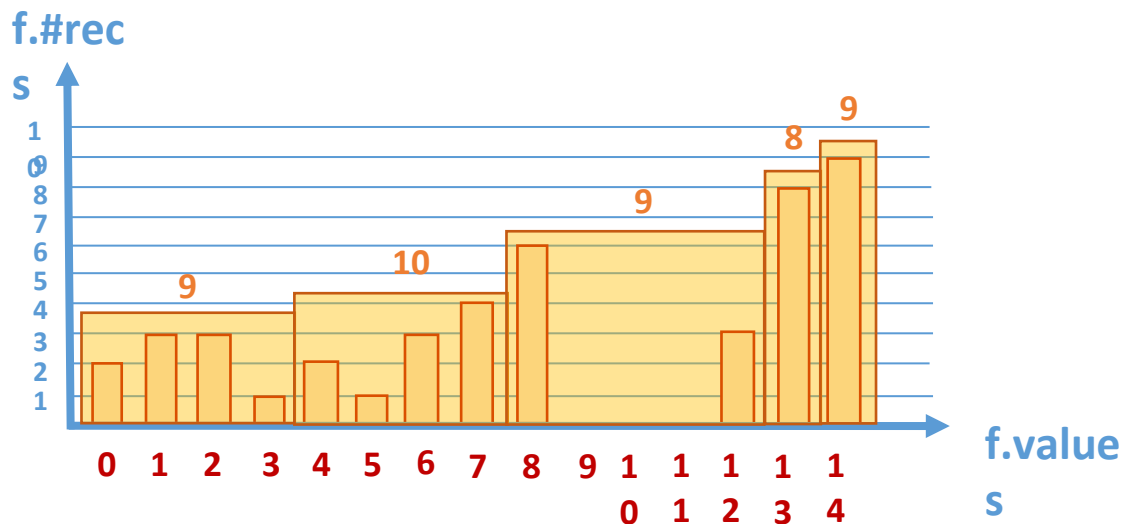
- Partition strategy: all buckets have the same range
- $|\text{Range}(b)| = \frac{\text{Max}(p,f) - \text{Min}(p,f) + 1}{\#Buckets}$



- Problem: some buckets may be wasted

Equi-Depth Histogram

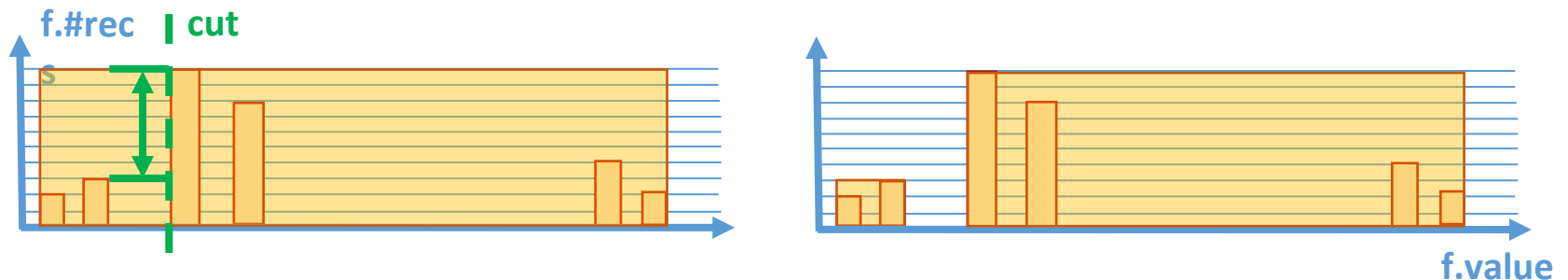
- Partition strategy: all buckets have the same #recs
- Depth = $\frac{R(p)}{\#Buckets}$



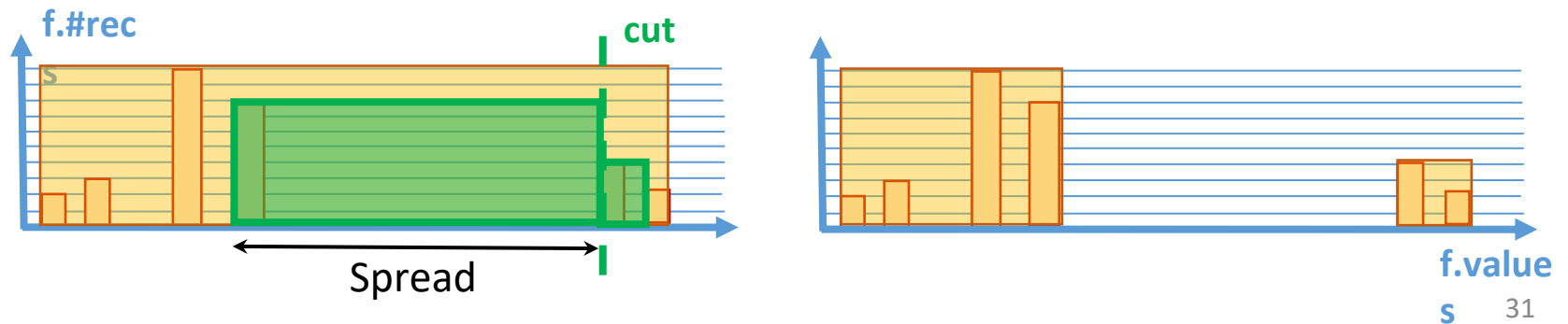
- Problem: records/values in a bucket may **not** be uniformly distributed

Max-Diff Histogram

- Partition strategy: split buckets at values with max. diff in #rec (MaxDiff(F)) or area (MaxDiff(A)):
 1. #recs: uniform #records in each bucket

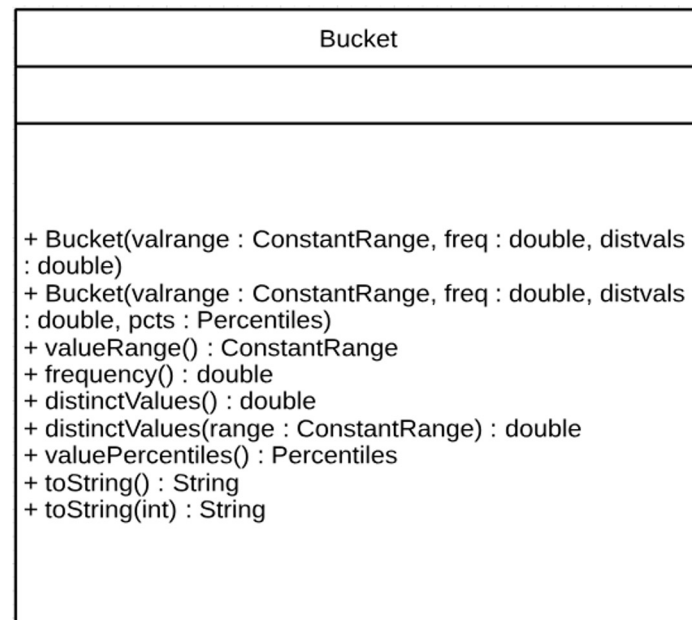
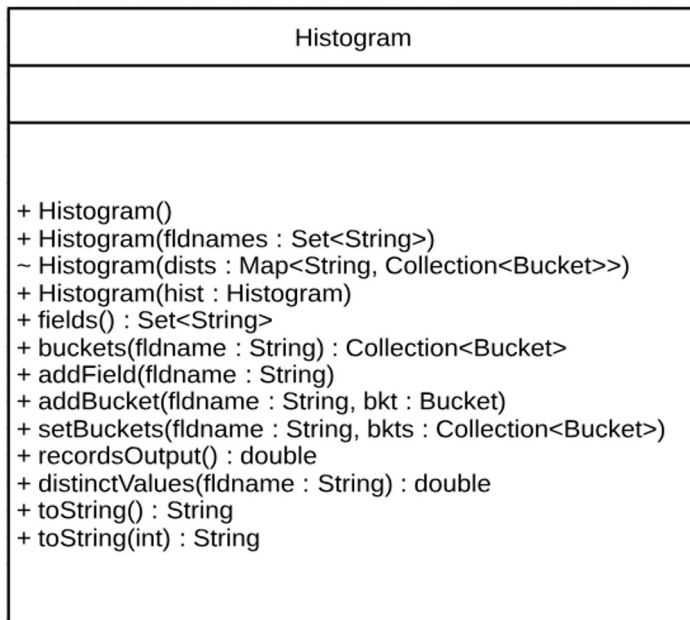


1. Area: uniform #records *and values* in each bucket ^s



Histogram in VanillaCore

- Table histograms are statistics metadata
 - org.vanilladb.core.storage.metadata.statistics
- Accessed (by TablePlan) via StatMgr.getTableStatInfo()



Building Histogram (1/2)

- When system starts up:
- StatMgr:
 - Scans table and calls `SampledHistogramBuilder.sample()`
 - When done, calls `SampledHistogramBuilder.newMaxDiffHistogram()`
- Histogram types:
 - `MaxDiff(A)` : when field value is numeric
 - `MaxDiff(F)` : otherwise

Building Histogram (2/2)

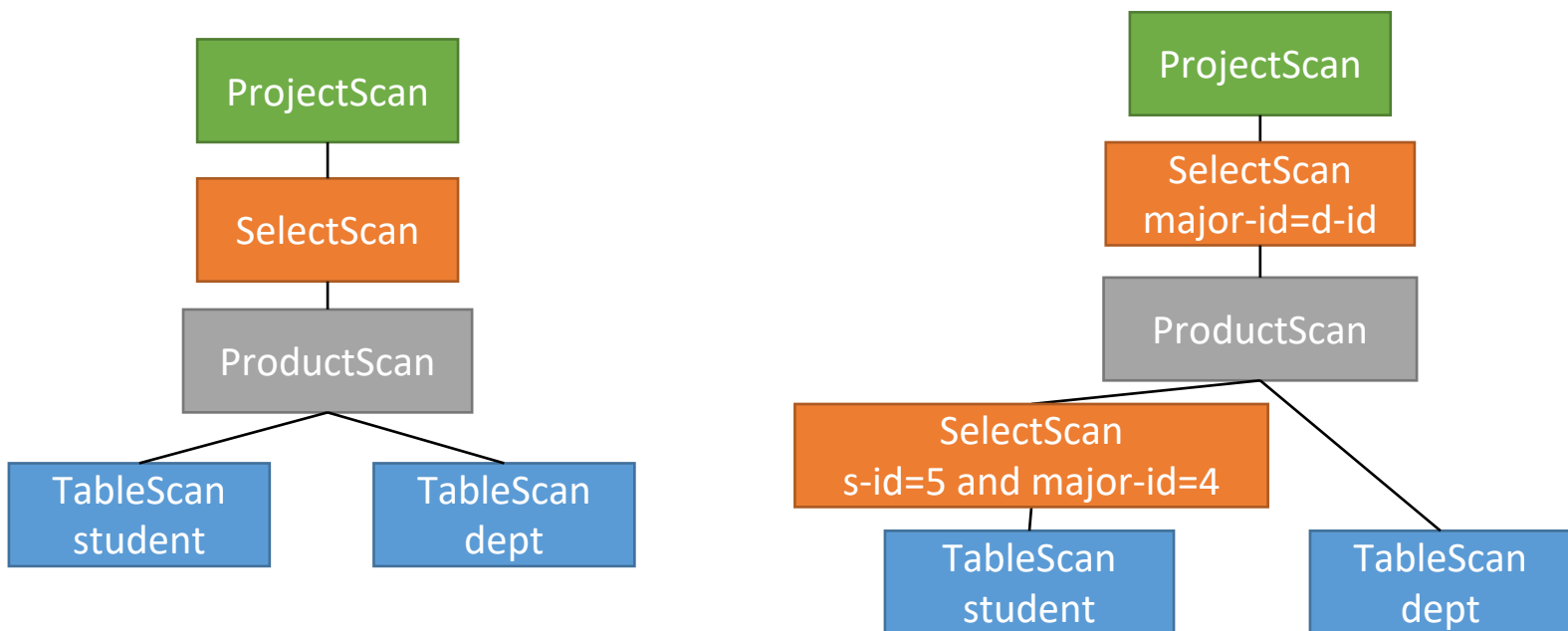
- At runtime:
- StatMgr tacks #recs updated for each table
 - QueryPlanner calls StatMgr.countRecordUpdates() after executing modify/insert/delete queries
- Rebuilds histogram *in background* when StatMgr.getTableStatInfo() is called
 - If #recs updated > threshold (e.g., 100)
- StatisticsRefreshTask:
 - Scans table and calls SampledHistogramBuilder.sample()
 - When done, calls SampledHistogramBuilder.newMaxDiffHistogram()

Outline

- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- **Heuristic Query Optimizer**
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Query Optimization

- Query optimizer:
 1. Generate candidate plan trees
 2. Estimate cost of each corresponding scan tree
 3. Pick and open the “best” one to execute query

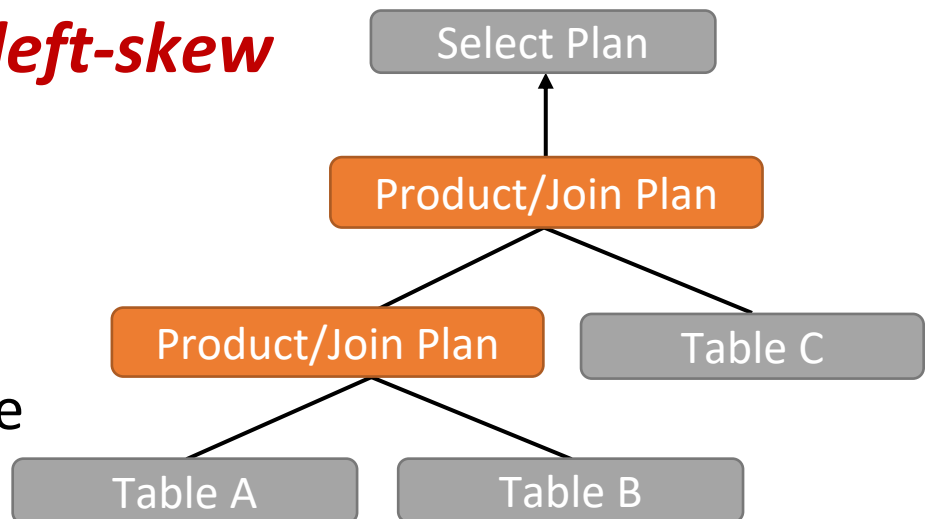


In Reality...

- Generating all candidate plan trees are too costly
 - #trees with n products/joins = Catalan number:

$$\frac{1}{n+1} \binom{2n}{n}$$

- Compromise: consider *left-skew* candidate trees only
- Query planner's goal
 - Avoiding bad trees
 - Not finding the best tree



Why Left-Skew Trees Only?

- Tend to be better than plans of other shapes
- Because many join algorithms scan right child c_2 multiple times
- Normally, we don't want c_2 to be a complex subtree

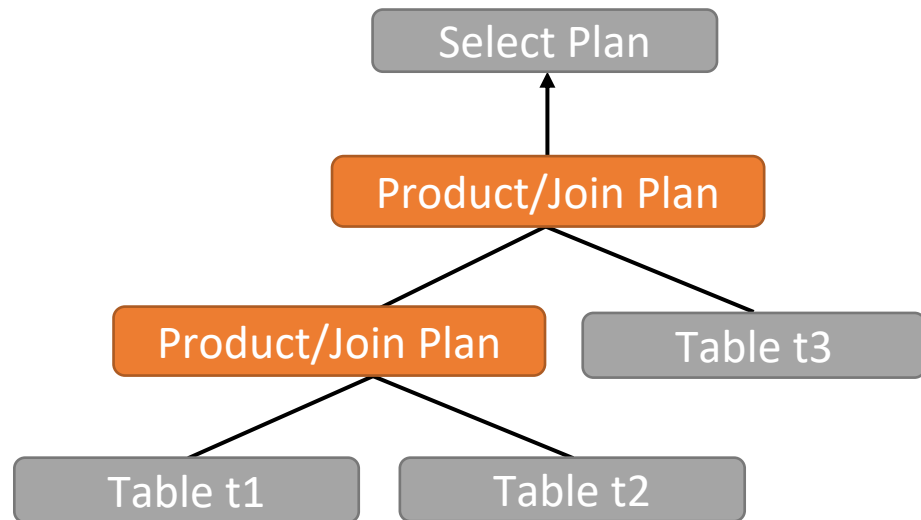
BasicQueryPlanner

```
public Plan createPlan(QueryData data, Transaction tx) {
    // Step 1: Create a plan for each mentioned table or view
    List<Plan> plans = new ArrayList<Plan>();
    for (String tblname : data.tables()) {
        String viewdef = VanillaDb.catalogMgr().getViewDef(tblname, tx);
        if (viewdef != null)
            plans.add(VanillaDb.newPlanner().createQueryPlan(viewdef, tx));
        else
            plans.add(new TablePlan(tblname, tx));
    }
    // Step 2: Create the product of all table plans
    Plan p = plans.remove(0);
    for (Plan nextplan : plans)
        p = new ProductPlan(p, nextplan);
    // Step 3: Add a selection plan for the predicate
    p = new SelectPlan(p, data.pred());
    // Step 4: Add a group-by plan if specified
    if (data.groupFields() != null) {
        p = new GroupByPlan(p, data.groupFields(), data.aggregationFn(), tx);
    }
    // Step 5: Project onto the specified fields
    p = new ProjectPlan(p, data.projectFields());
    // Step 6: Add a sort plan if specified
    if (data.sortFields() != null)
        p = new SortPlan(p, data.sortFields(), data.sortDirections(), tx);
    // Step 7: Add a explain plan if the query is explain statement
    if (data.isExplain())
        p = new ExplainPlan(p);
    return p;
}
```

- Product/join order follows what's written in SQL

Cost & Bottlenecks

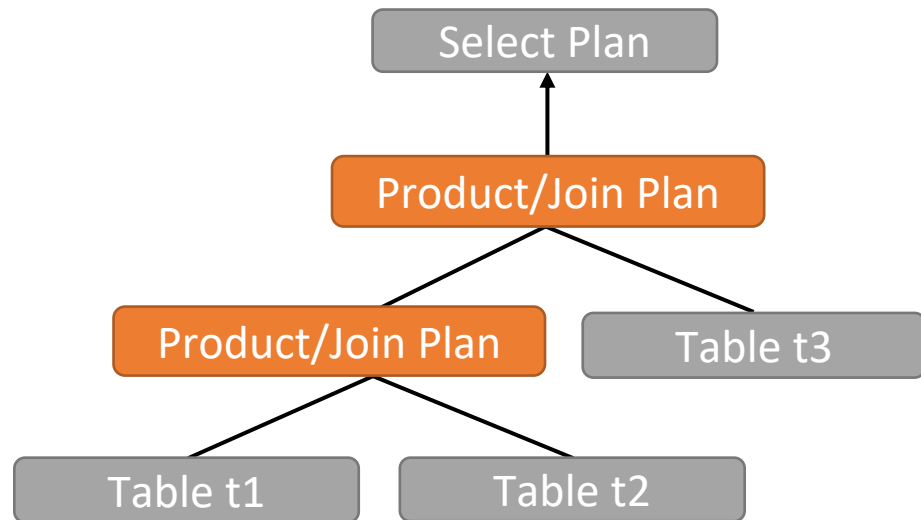
```
SELECT *  
FROM t1, t2, t3  
WHERE t1.f1 = t2.f2  
AND t2.f3 = t3.f4  
AND t1.f5 = x
```



- B(root) dominated by *#recs* of product/join ops
 - $B(\text{Product}(c1, c2)) = B(c1) + (R(c1) * B(c2))$
 - $B(\text{IndexJoin}(c1, c2)) = B(c1) + (R(c1) * \text{SearchCost}(\dots)) + \dots$

Optimizations

```
SELECT *  
FROM t1, t2, t3  
WHERE t1.f1 = t2.f2  
AND t2.f3 = t3.f4  
AND t1.f5 = x
```

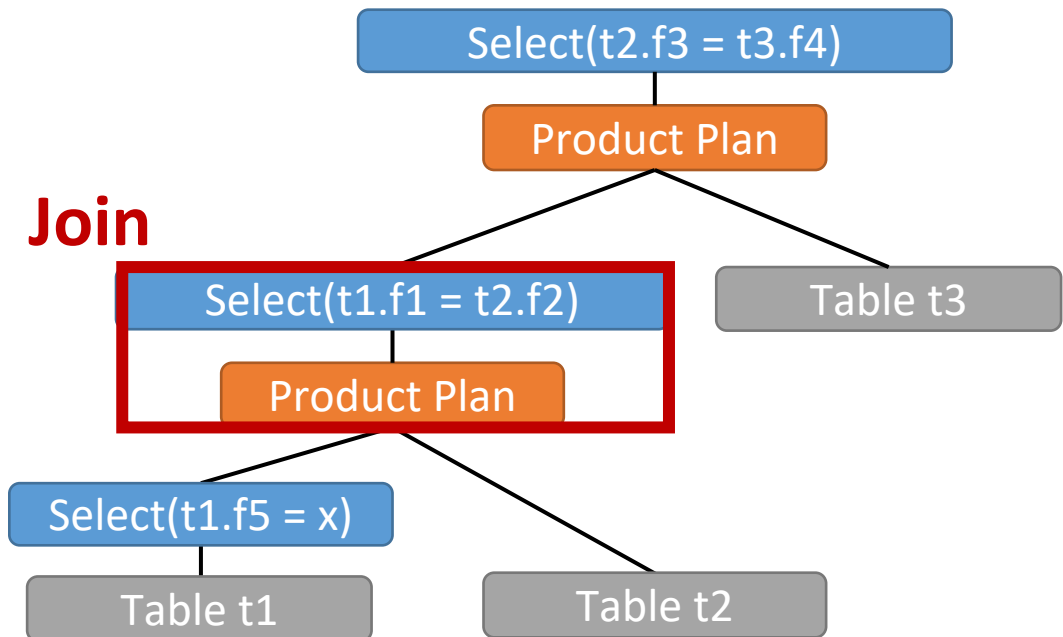


- Goal $\downarrow B(\text{root})$ reduced to $\downarrow R(c1)$
- Heuristics:
 - Pushing Select ops down
 - Greedy Join ordering

Pushing Select Ops Down

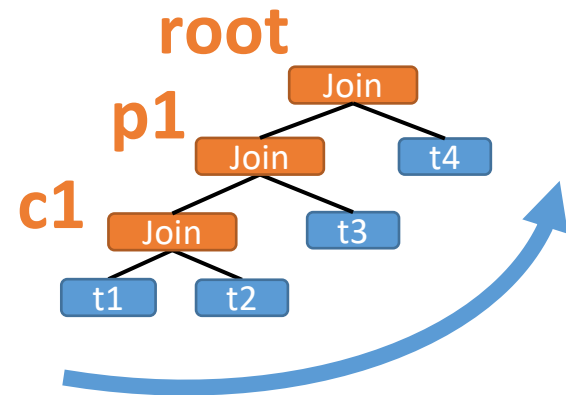
- Execute Select ops as early as possible
- ↓R of each product/join op

```
SELECT *  
FROM t1, t2, t3  
WHERE t1.f1 = t2.f2  
AND t2.f3 = t3.f4  
AND t1.f5 = x
```



Greedy Join Ordering

- $B(\text{root}) = B(p1) + (R(p1) * \dots) + \dots$
 - $\downarrow B(\text{root})$ implies $\downarrow(p1)$
- $B(p1) = B(c1) + (R(c1) * \dots) + \dots$
 - $\downarrow B(\text{root})$ also implies $\downarrow(c1)$
- ...
- $B(\text{root}) \propto R(p1) + R(c1) + \dots$



- **Greedy Join ordering**: repeatedly add table to the “trunk” that result in lowest $R(\text{trunk})$

HeuristicPlanner in VanillaCore

```
public Plan createPlan(QueryData data, Transaction tx) {
    // Step 1: Create a TablePlanner object for each mentioned table/view
    int id = 0;
    for (String tbl : data.tables()) {
        String viewdef = VanillaDb.catalogMgr().getViewDef(tbl, tx);
        if (viewdef != null)
            views.add(VanillaDb.newPlanner().createQueryPlan(viewdef, tx));
        else {
            TablePlanner tp = new TablePlanner(tbl, data.pred(), tx, id);
            tablePlanners.add(tp);
        }
        id += 1;
    }
    // Step 2: Choose the lowest-size plan to begin the trunk of join
    Plan trunk = getLowestSelectPlan();
    // Step 3: Repeatedly add a plan to the join trunk
    while (!tablePlanners.isEmpty() || !views.isEmpty()) {
        Plan p = getLowestJoinPlan(trunk);
        if (p != null)
            trunk = p;
        else
            // no applicable join
            trunk = getLowestProductPlan(trunk);
    }
    // Step 4: Add a group by plan if specified
    // Step 5. Project on the field names
    // Step 6: Add a sort plan if specified
    // Step 7: Add a explain plan if the query is explain statement
}
```

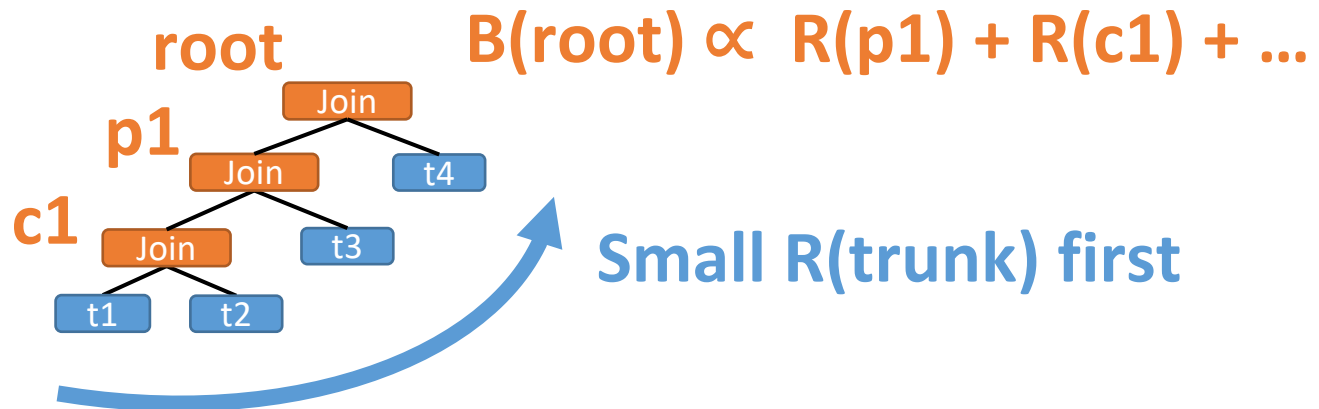
Feasible Select ops applied



Outline

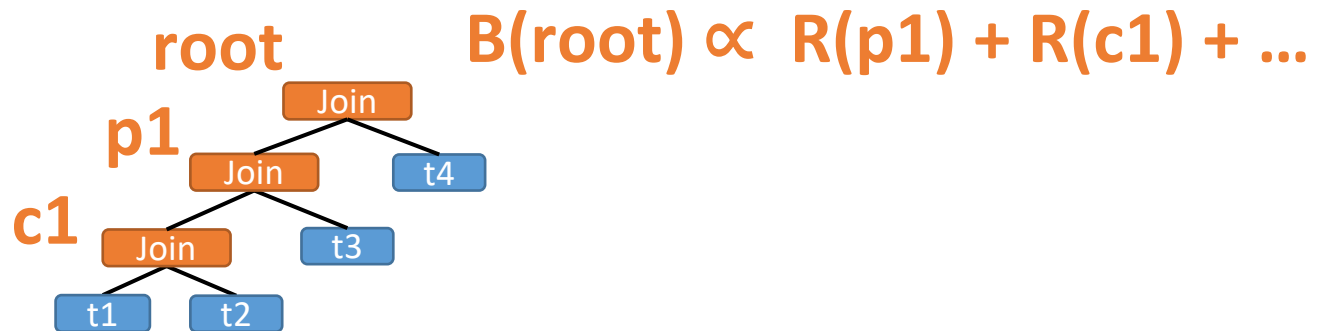
- Overview
- Cost Estimation
 - Cardinality Estimation
 - Histogram-based Estimation
 - Types of Histograms
- Heuristic Query Optimizer
 - Basic Planner
 - Pushing Select Down
 - Join Ordering
 - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Why not HeuristicPlanner?



- Assumption: $\downarrow R(c1)$ implies $\downarrow R(p1)$
- May **not** be true: joint rate also matters
- Exhaustively searching the best join order?
 - #Candidates: $O(n!)$ for n joins (e.g., $8! = 40320$)

Selinger-Style Optimizer




- Recursion:
 - $B^*({t1, t2, t3}) = \min(B^*({t1, t2} \bowtie t3), B^*({t1, t3} \bowtie t2), B^*({t2, t3} \bowtie t1))$
- Sub-optimality :
 - If $B^*({t1, t2}) = B(\mathbf{t1} \bowtie \mathbf{t2}) \leq B(\mathbf{t2} \bowtie \mathbf{t1})$
 - Then $B^*({t1, t2} \bowtie t3) = B(\mathbf{t1} \bowtie \mathbf{t2} \bowtie t3) \leq B(\mathbf{t2} \bowtie \mathbf{t1} \bowtie t3)$
- We can use **dynamic programming** to avoid repeating computations

Selinger Optimizer Example (1/3)

- Consider 3 relations to join: X, Y, Z
- Step 1: compute the $B(t)$ of each table t
 - with proper selection ops

1-Set	Best Plan	R
{X}	Index Select Plan	10
{Y}	Table Plan	30
{Z}	Select Plan	20

Selinger Optimizer Example (2/3)

- Step 2: compute the cost of 2-way join
 - Estimate all left-deep permutation using the single-relation cost just cached
- E.g. $\{X, Y\} =$
 - $B(\{X\} \bowtie Y)$: 159 
 - $B(\{Y\} \bowtie X)$: 189

Because the $R(X \bowtie Y)$, $R(Y \bowtie X)$ is the same, we can only keep one in K-set

1-Set	Best Plan	Cost
{X}	Index Select Plan	10
{Y}	Table Plan	30
{Z}	Select Plan	20

2-Set	Best Plan	Cost
{X, Y}	$X \bowtie Y$	159
{X, Z}	$Z \bowtie X$	98
{Y, Z}	$Z \bowtie Y$	77

Selinger Optimizer Example (3/3)

- Step 3: compute the cost of 3-way join
 - Estimate all left-deep tree permutation using the 2-set costs
- E.g. $\{X, Y, Z\} =$
 - $B(\{X, Y\} \bowtie Z) = 259$
 - $B(\{X, Z\} \bowtie Y) = 100$ ✓
 - $B(\{Y, Z\} \bowtie X) = 111$

2-Set	Best Plan	Cost
$\{X, Y\}$	$X \bowtie Y$	159
$\{X, Z\}$	$Z \bowtie X$	98
$\{Y, Z\}$	$Z \bowtie Y$	77

3-Set	Best Plan	Cost
$\{X, Y, Z\}$	$Z \bowtie X \bowtie Y$	100

Complexity (Simplified)

$$\binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = O(2^n)$$

- E.g., when $n=8$:
- Exhaustively search: $8! = 40320$ candidates
- Selinger-style planner: $2^8=256$ candidates

SelingerLikeQueryPlanner in VanillaCore

- Package: org.vanilladb.core.query.planner.opt

```
private Plan getAllCombination(Plan viewTrunk) {
    long finalKey = 0;

    // for layer = 1, use select down strategy to construct
    for (TablePlanner tp: tablePlanners) {
        Plan bestPlan = null;
        if (viewTrunk != null) {
            bestPlan = tp.makeJoinPlan(viewTrunk);
            if (bestPlan == null)
                bestPlan = tp.makeProductPlan(viewTrunk);
        }
        else
            bestPlan = tp.makeSelectPlan();

        AccessPath ap = new AccessPath(tp, bestPlan);
        lookupTbl.put(ap.getAPId(), ap);

        // compute final access path id
        finalKey += ap.getAPId();
    }

    .
    .
    .
}
```

```

// construct all combination layer by layer
for (int layer = 2; layer <= tablePlanners.size(); layer++) {
    Set<Long> keySet = new HashSet<Long>(lookupTbl.keySet());

    for (TablePlanner rightOne: tablePlanners) {
        for (Long key: keySet) {
            AccessPath leftTrunk = lookupTbl.get(key);

            // cannot join with table which (layer-1) combination already included
            if (leftTrunk.isUsed(rightOne.getId()))
                continue;

            // do join
            Plan bestPlan = rightOne.makeJoinPlan(leftTrunk.getPlan());
            if (bestPlan == null)
                bestPlan = rightOne.makeProductPlan(leftTrunk.getPlan());

            AccessPath candidate = new AccessPath(leftTrunk, rightOne, bestPlan);
            AccessPath ap = lookupTbl.get(candidate.getAPIId());

            // there is no access path contains this combination
            if (ap == null) {
                lookupTbl.put(candidate.getAPIId(), candidate);
            }
            // check whether new access path is better than previous
            else {
                if (candidate.getCost() < ap.getCost())
                    lookupTbl.put(candidate.getAPIId(), candidate);
            }
        }
    }

    // remove the elements belong to layer-1
    // because in the next layer we only need this layer's combination
    for (Long key: keySet)
        lookupTbl.remove(key);
}

return lookupTbl.get(finalKey).getPlan();

```

- Iterate all table planners to join with all existing (layer-1) combination to construct this layer

```
public class AccessPath {
    private Plan p;
    private AccessPathId apId;
    private long cost = 0;
    private ArrayList<Integer> tblUsed = new ArrayList<Integer>();
```

```
public class AccessPathId {
    long id;

    AccessPathId(TablePlanner tp) {
        this.id = (long) Math.pow(2, tp.getId());
    }

    AccessPathId(AccessPath ap, TablePlanner tp) {
        this.id = ap.getAPId() + (long) Math.pow(2, tp.getId());
    }

    public long getID() {
        return id;
    }
}
```

```
public AccessPath (TablePlanner newTp, Plan p) {
    this.p = p;
    this.tblUsed.add(newTp.getId());
    this.apId = new AccessPathId(newTp);
    this.cost = p.recordsOutput();
}

public AccessPath (AccessPath preAp, TablePlanner newTp, Plan p) {
    this.p = p;
    this.tblUsed.addAll(preAp.getTblUsed());
    this.tblUsed.add(newTp.getId());
    this.apId = new AccessPathId(preAp, newTp);

    // approximate cost = previous cost + new cost
    this.cost = preAp.getCost() + p.recordsOutput();
}
```

- apID is the key of the lookup table
- Use **sum of pow(2, tp.id)** to represent the k-set in an access path

- Approximate B(root) using $R(p1) + R(c1)...$

Reference

- <https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws1011/db2/db2-selectivity.pdf>
- <https://www.cise.ufl.edu/~adobra/approxqp/histograms2>
- <https://pdfs.semanticscholar.org/b024/0a44105fa0a0967d96d109aac9f021902ebb.pdf>