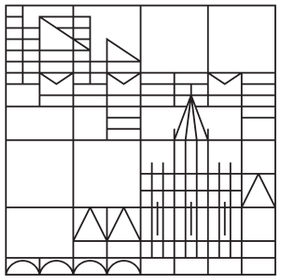
Universität
Konstanz



Database System Architecture and Implementation

Module 8

Query Optimization

January 7, 2019

Orientation

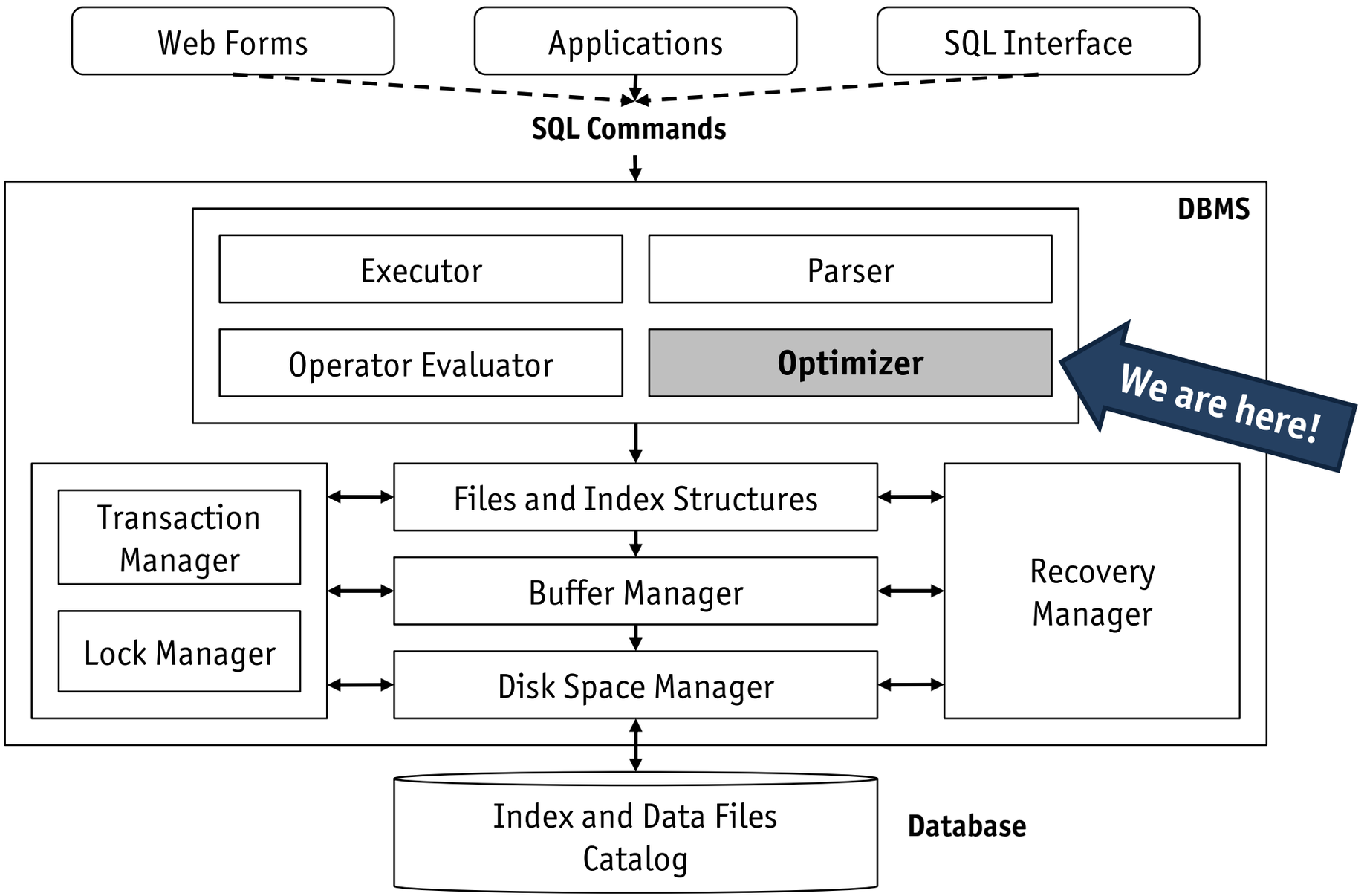


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Translating queries into algebra
- Relational algebra equivalences
- Plan enumeration
- Example query optimizers
- Cost estimation and histograms
- Nested sub-queries

Outline

"Query optimization is not rocket science.

When you flunk out of query optimization,
we make you go build rockets."

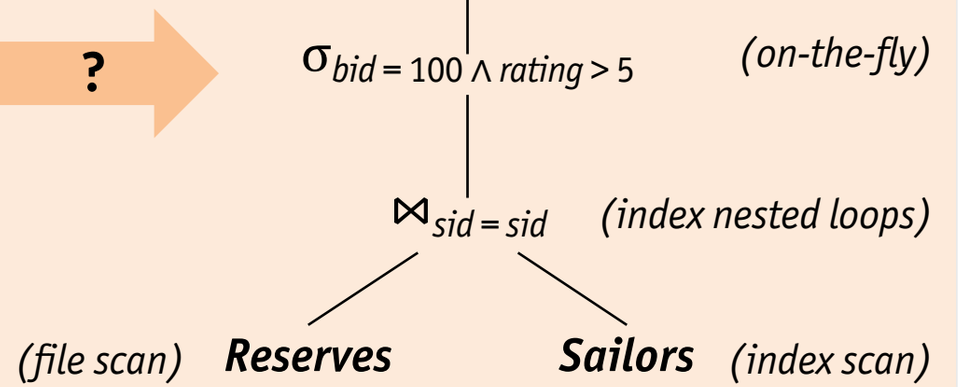
– *Anonymous Quote*

Slide Credit: David DeWitt, "PASS Summit Keynote", 2010.

Outline

From SQL to a query execution plan

```
SELECT sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND
       R.bid = 100 AND
       S.rating > 5
```



- Recall that there may be **many ways** to answer a given SQL query
  - different query trees (pushing selections and projections, join orders)
  - different physical operators for the same logical operator
  - different parameters (block size, buffer allocation, ...)
- Task of finding the **best** execution plan is crucial to any database implementation

Outline

- To evaluate a given (SQL) query *Q*, a database management system performs the following steps
  1. parse and analyze *Q*
  2. derive a **relational algebra** expression *E* that computes *Q*
  3. generate a set of **logical plans** *L* by transforming and simplifying *E*
  4. generate a set of **physical plans** *P* by annotating each plan in *L* with access methods and operator algorithms
  5. for each plan in P, **estimate the quality** (cost) of the plan and choose the best plan as the final evaluation plan
- Query optimizer encompasses the last three steps
  - **plan enumeration** in Steps 3 and 4
  - **algebraic** (or rewrite) query optimization in Step 3
  - **non-algebraic** (or cost-based) query optimization in Steps 4 and 5

Outline

- **Query parser** performs syntactical and semantical analysis and checks
- **Plan enumeration**
  - **rewrite optimization** uses heuristics independent of the current database state
  - **cost-based optimization** relies on plan cost estimation to annotate plan with access path and operator algorithms
- **Plan cost estimation** assigns a cost to a plan based on
  - cost model (I/O cost, CPU cost, etc.)
  - current database state (table sizes, index availability, etc.)
- **Evaluator** executes the resulting plan using the relational operators presented in the previous two modules

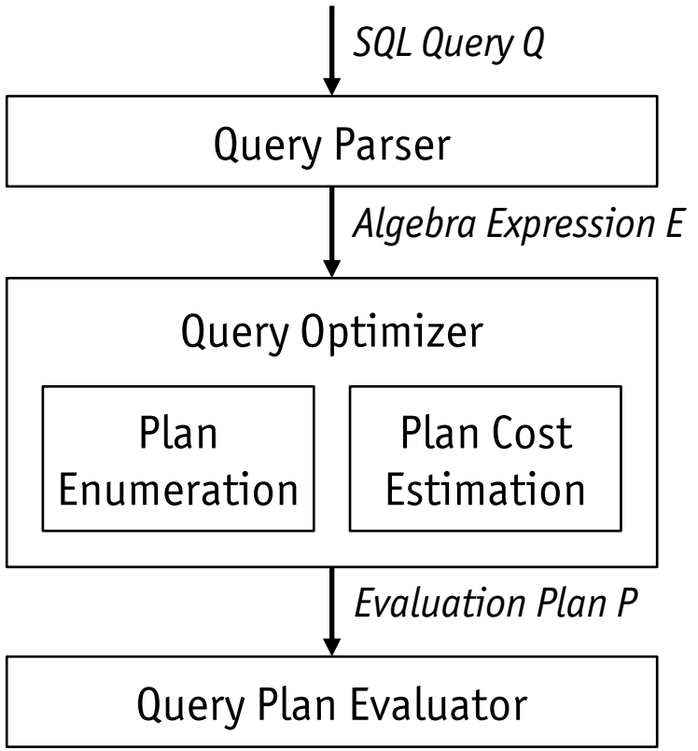
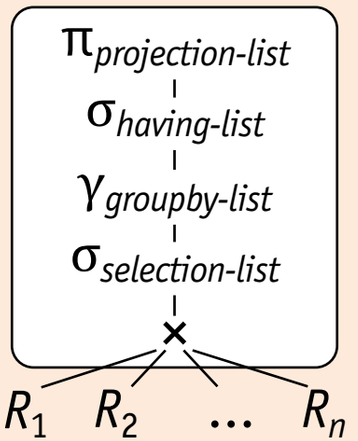


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Parser

Deriving a query block from an SQL statement

SELECT *projection-list*
FROM $R_1, R_2, \ldots, R_n$
WHERE *selection-list*
GROUP BY *groupby-list*
HAVING *having-list*



query block

- After checking its syntactical and semantical correctness, parser creates **internal representation** of the input query
  - internal representation resembles the original query
  - each SELECT-FROM-WHERE clause is translated to a **query block**
  - each R_i can be a base relation or another query block

Query Optimizer

- Query optimizer considers each query block and chooses an evaluation plan for this block
  - for time being the focus is on **single-block queries**
  - optimization of **nested queries** will be discussed separately
- In order to find the best evaluation plan, query optimizer explores so-called **search space** of possible alternative plans
  - **logical level:** relational algebra equivalences
  - **physical level:** access methods and operator algorithms

Relational Algebra Equivalences

- Recall that each query block is a relation algebra expression consisting of cross-product, selection, and projection
- Rewrite optimization applies **relational algebra equivalences** to transform a query block into an equivalent expression
  - convert cross-product to join
  - choose different join orders
  - push selections and projections ahead of joins
- Two relational algebra expressions E_1, E_2 are **equivalent** if they generate the same set of tuples on every legal database instance
- Such equivalences imply **equivalence rules** of the form $E_1 \equiv E_2$
  - optimizer may apply equivalence rules in both directions ($\rightarrow$, $\leftarrow$)
  - these equivalence rules are first introduced in the course "Database Systems" (INF-12040)

Relational Algebra Equivalences

- Selections
  1. conjunctive selections can be deconstructed into a sequence of individual selections (**cascading selections**)

$$\sigma_{c1 \wedge c2 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots\sigma_{cn}(R))\ldots)$$

  2. selections are **commutative**

$$\sigma_p(\sigma_q(R)) \equiv \sigma_q(\sigma_p(R))$$

- Projections
  3. only the last projection in a sequence of projections is needed, the others can be omitted (**cascading projections**)

$$\pi_{a1}(R) \equiv \pi_{a1}(\pi_{a2}(\ldots(\pi_{an}(R))\ldots))$$

  where $a_i \subseteq a_{i+1}$ for $i = 1, \ldots, n-1$

Relational Algebra Equivalences

- Cross-products and natural joins
  4. are both **commutative**

$$R \times S \equiv S \times R$$
$$R \bowtie S \equiv S \bowtie R$$

  5. are both **associative**

$$R \times (S \times T) \equiv (R \times S) \times T$$
$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

- General joins
  6. are **associative**

$$(R \bowtie_p S) \bowtie_{q \wedge r} T \equiv R \bowtie_{p \wedge q} (S \bowtie_r T)$$

  where predicate r involves attributes of S and T only

Relational Algebra Equivalences

Importance of these equivalences

What is the relevance of these two properties of cross-products and joins with respect to query optimization?

Relational Algebra Equivalences

- Selections, cross-product, and join
  7. selections can be **combined with cross-product** to form a join
  $$\sigma_p(R \times S) \equiv R \bowtie_p S$$
  8. selections can be **combined with join**
  $$\sigma_p(R \bowtie_q S) \equiv R \bowtie_{p \wedge q} S$$
  9. selections **commute** with cross-products and joins, if predicate p involves attributes of R only
  $$\sigma_p(R \times S) \equiv \sigma_p(R) \times S$$
  $$\sigma_p(R \bowtie_q S) \equiv \sigma_p(R) \bowtie_q S$$
  10. selections **distribute** over cross-product and join, if predicate p only involves attributes of R and predicate q only involves attributes of
  $$\sigma_{p \wedge q}(R \times S) \equiv \sigma_p(R) \times \sigma_p(S)$$
  $$\sigma_{p \wedge q}(R \bowtie_r S) \equiv \sigma_p(R) \bowtie_r \sigma_p(S)$$

Relational Algebra Equivalences

- Projections, selections, cross-product, and join
  11. projection and selection **commute** if the selection predicate p only involves attributes retained by the projection list a

$$\pi_a(\sigma_p(R)) \equiv \sigma_p(\pi_a(R))$$

  12. projection **distributes** over cross-product

$$\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$$

  where a_1 is the subset of attributes in a that appear in R and a_2 is the subset of attributes that appear in S

  13. projection **distributes** over join

$$\pi_a(R \bowtie_p S) \equiv \pi_{a1}(R) \bowtie_p \pi_{a2}(S)$$

  where a_1 is the subset of attributes in a that appear in R, a_2 is the subset of attributes that appear in S, and predicate p only involves attributes in $a_1 \cup a_2$

Relational Algebra Equivalences

- Union and intersection
  14. are both **commutative**

$$R \cup S \equiv S \cup R$$
$$R \cap S \equiv S \cap R$$

  15. are both **associative**

$$(R \cup S) \cup T \equiv R \cup (S \cup T)$$
$$(R \cap S) \cap T \equiv R \cap (S \cap T)$$

- Projection and union
  16. projection **distributes** over union

$$\pi_a(R \cup S) \equiv \pi_a(R) \cup \pi_a(S)$$

Relational Algebra Equivalences

- Selection, union, intersection, and difference
  17. are **distributive**

$$\sigma_p(R \cup S) \equiv \sigma_p(S) \cup \sigma_p(R)$$
$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap \sigma_p(R)$$
$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus \sigma_p(R)$$

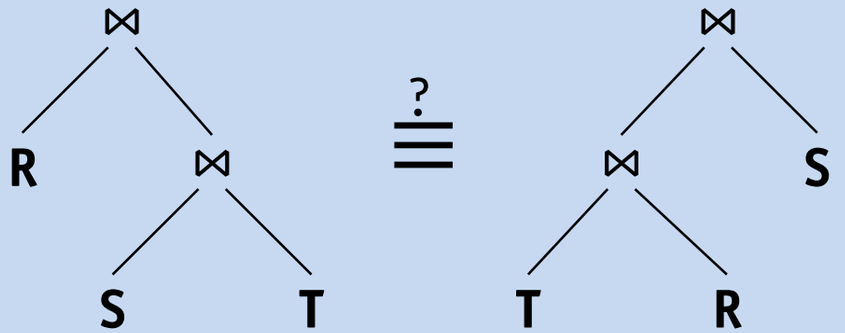
  18. intersect and differences are **commutative** (does **not** apply for union)

$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap R$$
$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus R$$

Relational Algebra Equivalences

Verification of equivalence

Using relational algebra equivalences, show that the following two query trees are equivalent.



Rewrite Optimization

- Query optimizers use relational algebra equivalence rules to improve expected performance of a query in **most cases**
- Optimization is guided by the following **heuristics**
  1. **Break apart conjunctive selections** into a sequence of simpler selections (rule [1], preparatory step for second step)
  2. **Move selections down the query tree** to reduce the cardinality of intermediate results as early as possible (rules [2], [9], and [17])
  3. **Replace selection–cross-product pairs with joins** to avoid large intermediate results (rule [7])
  4. **Break lists of projection attributes apart**, **move them down the query tree**, and **create new projections where possible** to reduce tuple widths as early as possible (rules [3], [13], and [16])
  5. **Perform joins with the smallest expected result first** (cost-based heuristic)

Rewrite Optimization

Example query

```
SELECT *
  FROM Sailors S
 WHERE S.rating * 100 > 50
```

Example query after predicate simplification

```
SELECT *
  FROM Sailors S
 WHERE S.rating > 0.5
```

Importance of predicate simplification

What is the main reason why the query optimizer rewrites the selection predicate as shown above?

Slide Credit: Torsten Grust, University of Tübingen, Germany

Rewrite Optimization

Example query

```
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c
```

Example query after implicit join predicate expansion

```
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c AND A.a = C.c
```

- Implicit join predicates can be turned into explicit one to enable more join orders
- In the example, the rewrite makes the join tree $(\mathbf{A} \bowtie \mathbf{C}) \bowtie \mathbf{B}$ feasible, which otherwise would have needed a cross-product

Slide Credit: Torsten Grust, University of Tübingen, Germany

Plan Enumeration

- Given a query, an optimizer **enumerates** a certain set of plans and chooses the plan with the least estimated cost
- The **search space** of alternative plans is given by
  - relational algebra equivalences
  - choice of implementation technique for relational operators
  - choice of access method based on presence of indexes
  - other available resources, e.g., number of buffer pages, etc.
- In general, it is impossible to enumerate all plans as it would be prohibitively expensive for all but the simplest queries
- Two important cases can be distinguished
  - single-relation queries
  - multiple relation queries

Single-Relation Queries

- Optimizer enumerates **all possible plans** and assesses their cost
- Main decision in a single-relation plan is the access method used to retrieve the tuples of the relation
  - plans without index
  - plans utilizing an index
- Observe that different single-relation plans might have different **physical properties**, i.e., sort order to the produced tuples
- For each set of physical properties, the plan with the least estimated cost is retained

Single-Relation Queries

- Plans without index
  1. heap file scan on (single) relation
  2. apply selection and projection (without duplicate elimination) on-the-fly
  3. sort according to **`GROUP BY`** clause
  4. apply aggregation and **`HAVING`** clause on-the-fly to each group
- Single-index access path
  1. choose index that is estimated to retrieve the fewest pages
  2. apply projections and non-primary selections
  3. proceed to compute grouping and aggregation operations by sorting
- Multiple-index access path (for index entry variants ❷ and ❸)
  1. retrieve and intersect rid sets and sort results by page id
  2. retrieve tuples that satisfy primary condition of all indexes
  3. apply projections and non-primary selection terms, followed by grouping and aggregation operations

Single-Relation Queries

- Sorted-index access path
  1. retrieve tuples in order required by **`GROUP BY`** clause
  2. apply selection and projection to each retrieved tuple on-the-fly
  3. compute aggregate operations for each group on-the-fly
- Index-only access path
  1. retrieve matching tuples or perform an index-only scan
  2. apply (non-primary) selection conditions and projections to each retrieved tuple on-the-fly
  3. possibly, sort the result to achieve grouping
  4. compute aggregate operations for each group on-the-fly

Single-Relation Queries

The Real World

All major RDBMS recognize the importance of **index-only plans** and look for such plans whenever possible.

- **IBM DB2**
  - users can specify a set of **include columns** that are kept in the index, but are not part of the index key
  - this feature enables a richer set of index-only queries to be handled
- **Microsoft SQL Server**
  - exploits indexes for partially matching selection predicates by **joining index entries** on the rid of the data record

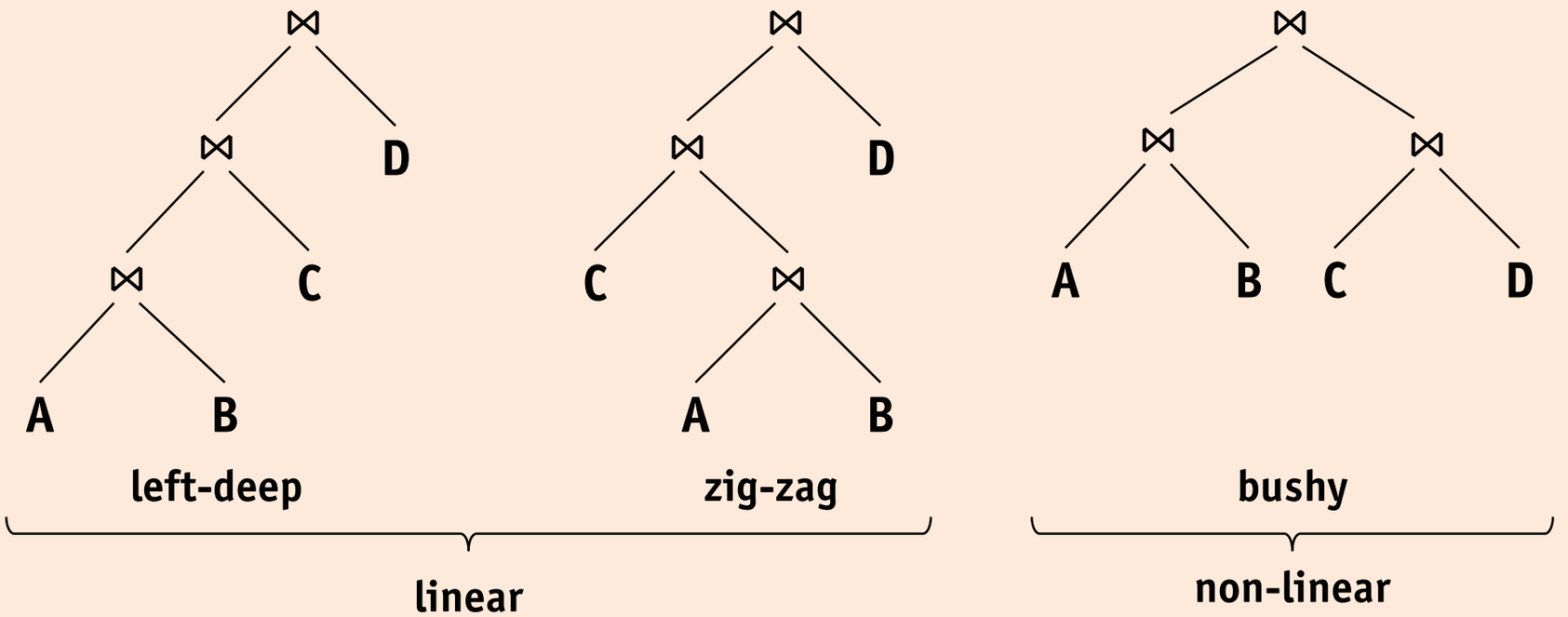
Multiple-Relation Queries

- Queries over multiple relations require joins (or cross-products)
- Finding a good plan for such queries is very important
  - these queries can be quite expensive (since joins are expensive)
  - queries involving joins are the normal case in a normalized database
- Size of **final result** can be estimated by taking product of
  - sizes of all relations
  - reduction factors of all selection predicates
- Size of **intermediate relations** can vary substantially depending on order in which relations are joined
- Therefore multiple-relation plans can have **very** different costs

Multiple-Relation Queries

Enumerating multiple-relation plans

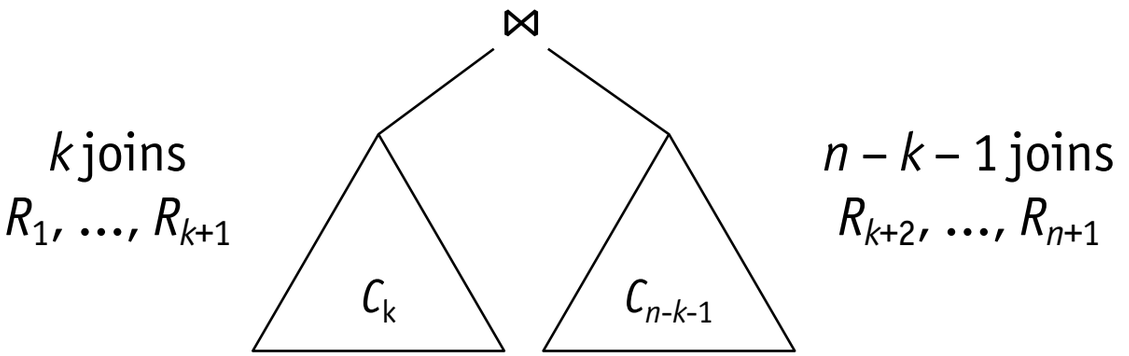
Using relational algebra equivalences (joins commute), the following equivalent plans can be enumerated for the query **A ⋈ B ⋈ C ⋈ D**.



Of course, these are **not all** possible join orders for a four-way join. In fact, there are many more...

Multiple-Relation Queries

- A join over $n + 1$ relations $R_1, \ldots, R_{n+1}$ requires n **binary joins**
- Its **root-level operator** joins sub-plans of k and $n - k - 1$ join operators ($0 \leq k \leq n - 1$)



- Let C_i be the **number of possibilities** to construct a binary tree of i inner nodes (join operators)

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$

Multiple-Relation Queries

- It turns out that this recurrence is satisfied by **Catalan numbers** describing the number of ordered binary trees with n + 1 leaves

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)! \cdot n!}$$

- For each of these trees, the input relations $R_1, \ldots, R_{n+1}$ can be permuted, which adds a factor $(n+1)!$

Number of possible join trees for an (n + 1)-way relational join

$$\frac{(2\mathrm{n})!}{(\mathrm{n}+1)! \cdot \mathrm{n}!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

Multiple-Relation Queries

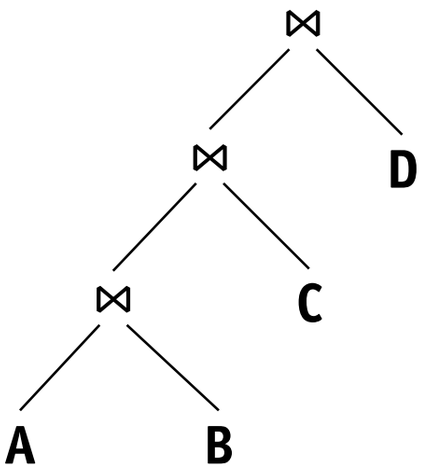
Explosion of the search space

number of relations n	C_{n-1}	join trees
2	1	2
3	2	12
4	5	120
5	14	1,680
6	42	30,240
7	132	665,280
8	429	17,297,280
10	4,862	17,643,225,600

- Remarks
  - formula only include **logical** query evaluation plans
  - considering the use of k **different join algorithms** adds a factor k^{n-1}

Multiple-Relation Queries

- It is impossible to enumerate all plan and that is a fact
  - optimizer will **not** be able to find the overall best plan
  - optimizer can do damage control by trying to avoid **really bad** plans
- Restrict the search space by only considering **left-deep plans**
  - left-deep plans can be translated to **fully pipelined** plans because inner relation is already materialized
  - some join algorithms may benefit from index on inner relation
- Number of possible left-deep joins orders for an *n*-way join is "only" *n*!



Multiple-Relation Queries

- If the optimizer cannot find the overall best plan, at least it needs to ensure that is does not miss the best left-deep plan
- There are still some degrees of freedom left
  - for each base relation in the query, consider all **access methods**
  - for each join operation in the left-deep tree, select a **join algorithm**

How many possible query plans are left now?

Back-of-envelope calculation for a query with n relations, assuming j available join algorithms and i indexes per relation

$$\#\text{plans} \approx n! \cdot j^{n-1} \cdot (i+1)^n$$

↳ **Example** with $n = 3$ relations, $j = 3$, and $i = 2$

$$\#\text{plans} \approx 3! \cdot 3^2 \cdot 3^3 = 1458$$

Multiple-Relation Queries

Step-by-step example: Setup and assumptions

```
SELECT S.sname, R.rname, B.bname
  FROM Sailors S, Reserves R, Boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

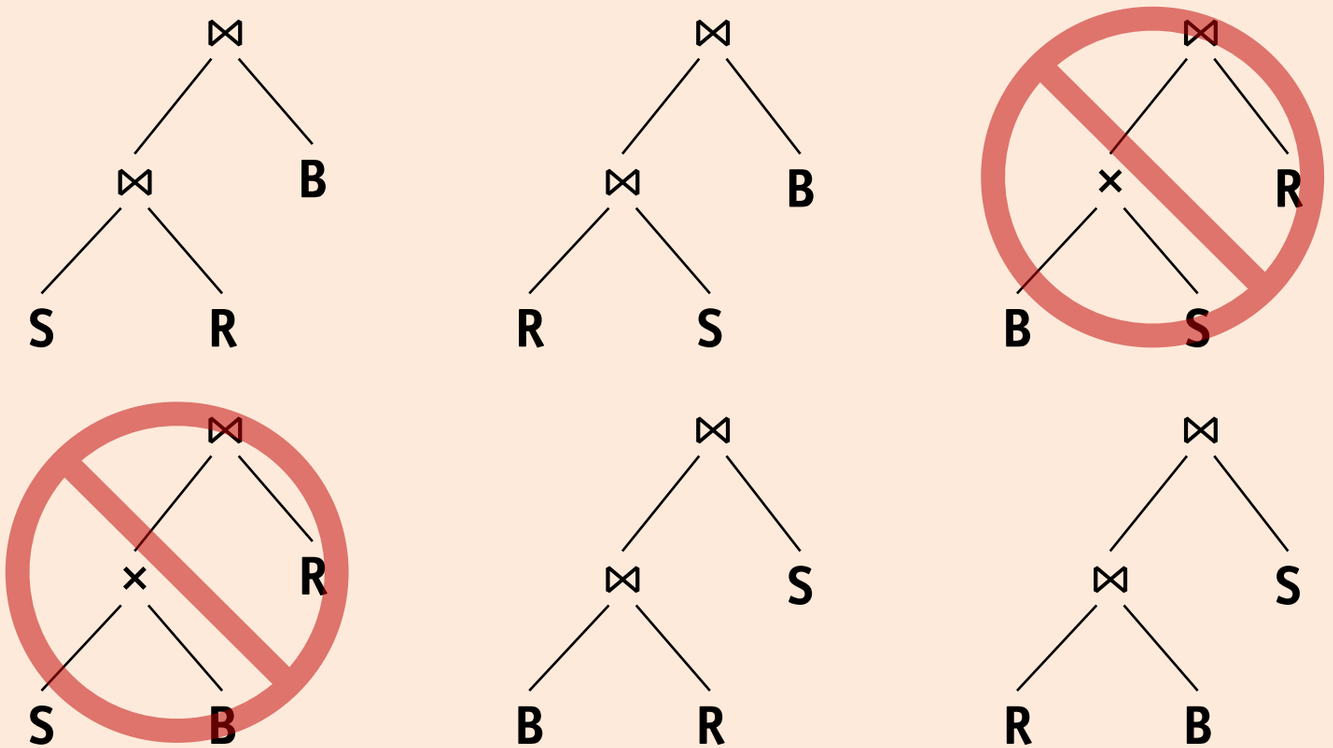
Assumptions

- Available join algorithms
  - hash join
  - block nested loops join
  - index nested loops join
- Available indexes
  - clustered B+ tree index **I** on **R.sid**, *INPages*(**I**) = 50
- Relation cardinality
  - *NPages*(**S**) = 500 pages, 80 tuples/page
  - *NPages*(**R**) = 1000 pages, 100 tuples/page
  - *NPages*(**B**) = 10 pages
- 100 (**R** ⋈ **S**)-tuples fit on a page

Multiple-Relation Queries

Step-by-step example: Enumerate candidate plans

- Enumerate $n!$ left-deep join trees ($3! = 6$)

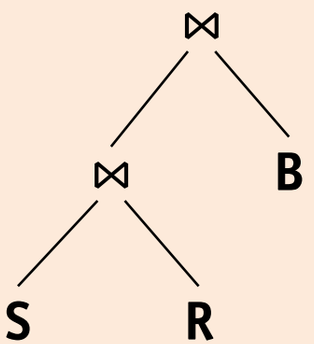


- **Prune** plans that need a cross-product immediately (no join predicate between **S**, **B**)
- Four **candidate plans** remain

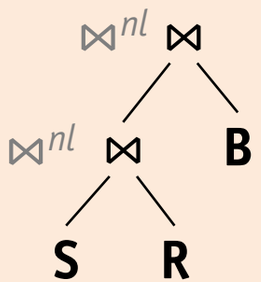
Multiple-Relation Queries

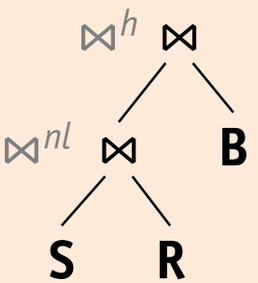
Step-by-step example: Choose join algorithms

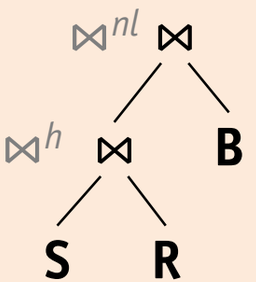
Candidate plan

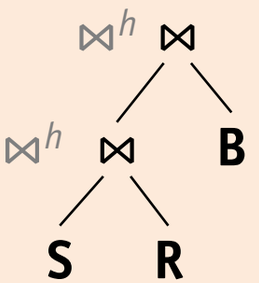


Possible join algorithm choices for this candidate plan







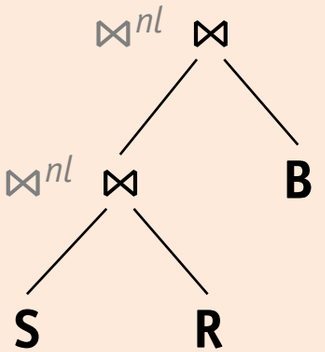


↳ Repeat for remaining three candidate plans

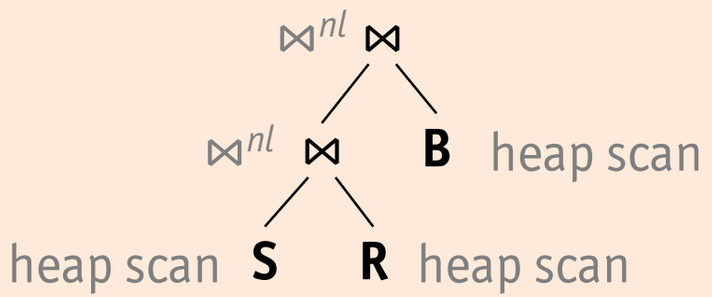
Multiple-Relation Queries

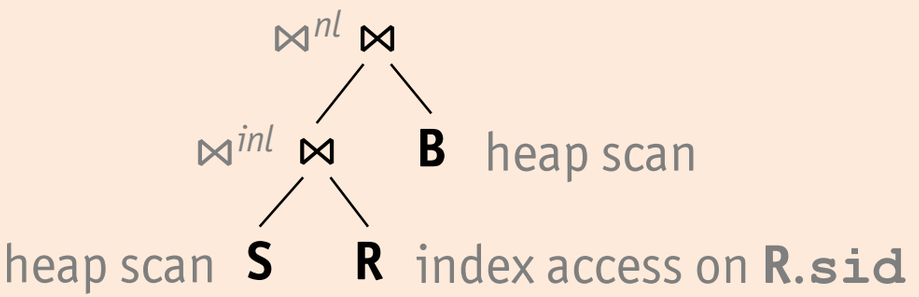
Step-by-step example: Choose access methods

Candidate plan



Possible access method choices for this candidate plan



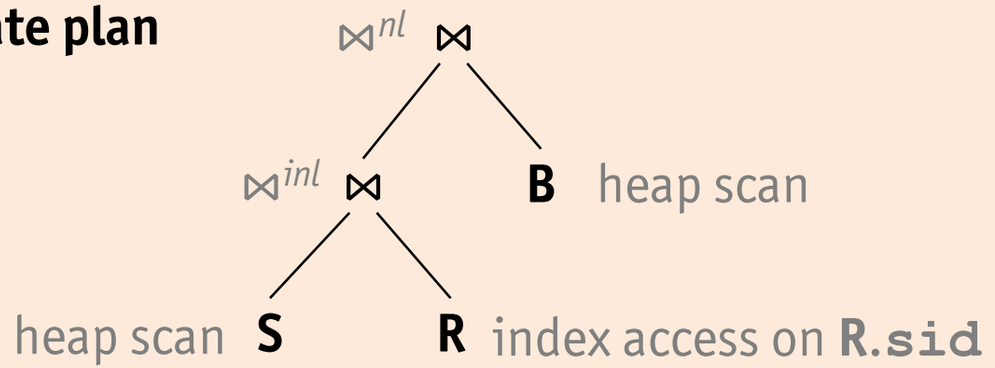


↳ Repeat for remaining candidate plans

Multiple-Relation Queries

Step-by-step example: Cost estimation

Candidate plan



Cost estimation for this candidate plan

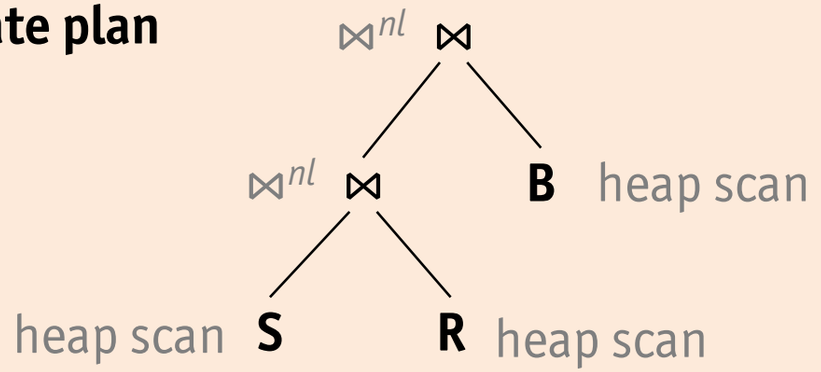
- cost of heap scan on **S**: 500 pages
- cost of **S** ⋈ **R** (**R.sid** is a foreign key)
  - *NTuples*(**S**) · *sel*(**S.sid** = **R.sid**) · (*NPages*(**R**) + *NPages*(**I**))
  - 40,000 · 1/40,000 · (1000 + 50) = 1050 pages
- *NPages*(**S** ⋈ **R**) = *NTuples*(**S** ⋈ **R**)/100 = *NTuples*(**R**)/100 = 100,000/100 = 1000 pages
- cost of (**S** ⋈ **R**) ⋈ **B**: *NPages*(**S** ⋈ **R**) · *NPages*(B) = 1000 · 10 = 10,000 pages

Total estimated cost: 500 + 1050 + 10,000 = 11,550 pages

Multiple-Relation Queries

Step-by-step example: Cost estimation

Candidate plan



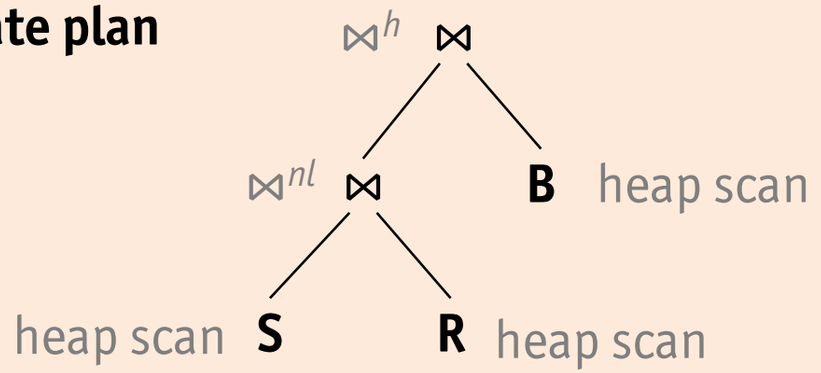
Total estimated cost for candidate plan

- $NPages(\mathbf{S}) + NPages(\mathbf{S}) \cdot NPages(\mathbf{R}) + NPages(\mathbf{S} \bowtie \mathbf{R}) \cdot NPages(\mathbf{B})$
- $500 + 500 \cdot 1000 + 1000 \cdot 10 =$ 510,500 pages

Multiple-Relation Queries

Step-by-step example: Cost estimation

Candidate plan



Assumption

- $\bowtie^h$ requires two passes

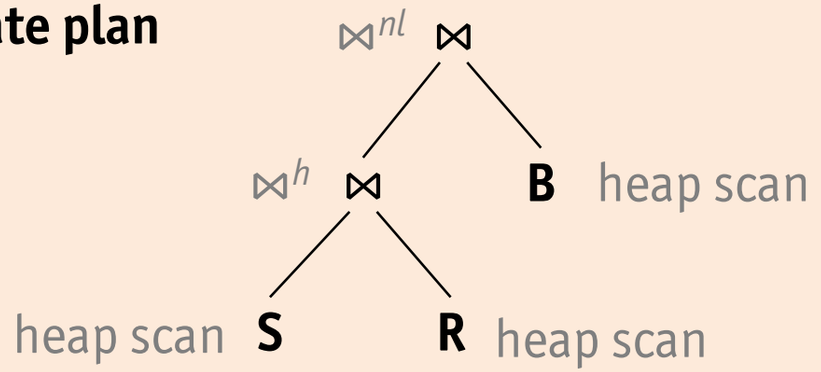
Total estimated cost for candidate plan

- $NPages(\mathbf{S}) + NPages(\mathbf{S}) \cdot NPages(\mathbf{R}) + 2 \cdot (NPages(\mathbf{S} \bowtie \mathbf{R}) + NPages(\mathbf{B}))$
- $500 + 500 \cdot 1000 + 2 \cdot (1000 + 10) =$ 502,520 pages

Multiple-Relation Queries

Step-by-step example: Cost estimation

Candidate plan



Assumption

- $\bowtie^h$ requires two passes

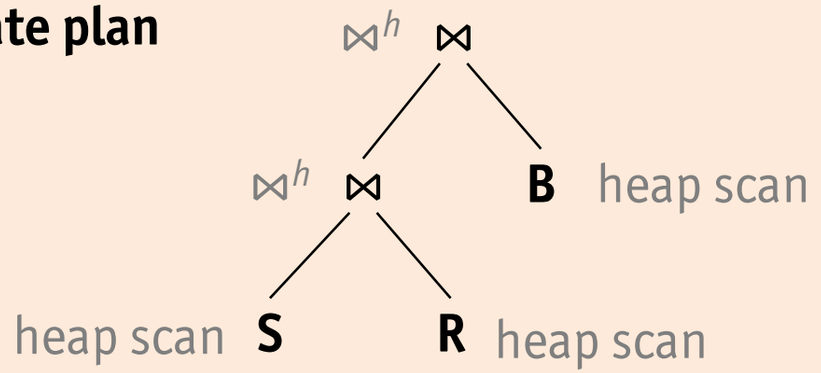
Total estimated cost for candidate plan

- $2 \cdot (NPages(\mathbf{S}) + NPages(\mathbf{R})) + NPages(\mathbf{S} \bowtie \mathbf{R}) \cdot NPages(\mathbf{B})$
- $2 \cdot (500 + 1000) + 1000 \cdot 10 =$ 13,000 pages

Multiple-Relation Queries

Step-by-step example: Cost estimation

Candidate plan



Assumption

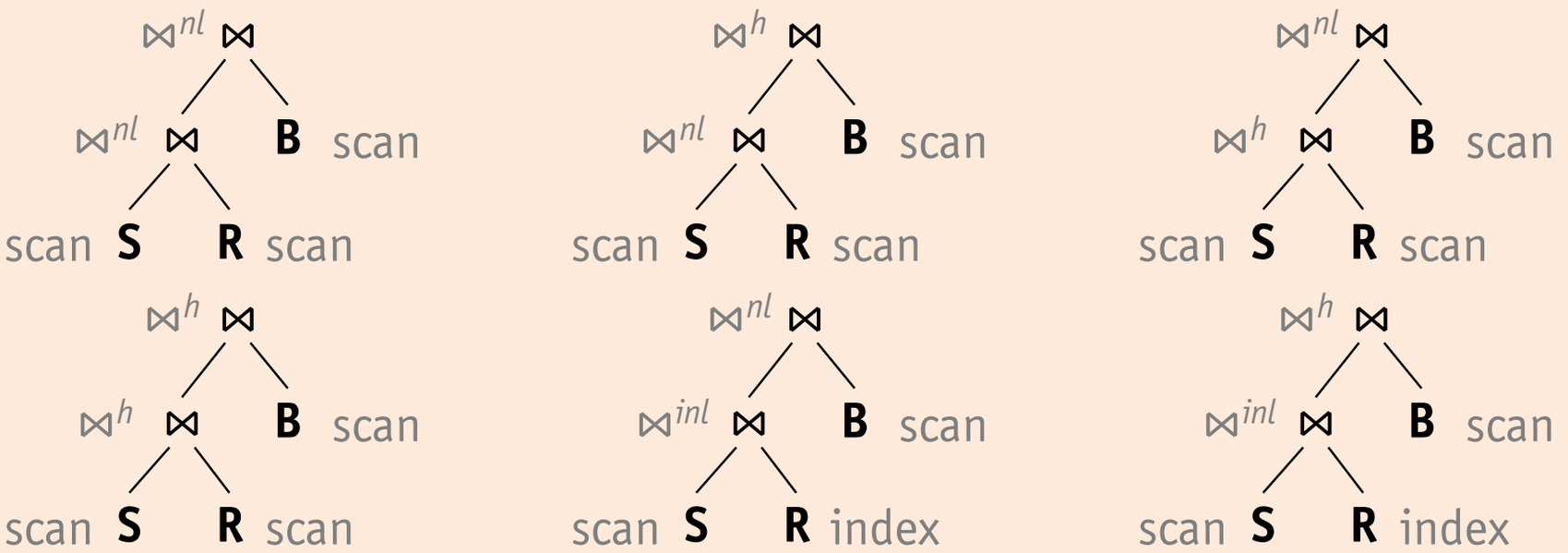
- $\bowtie^h$ requires two passes

Total estimated cost for candidate plan

- $2 \cdot (NPages(\mathbf{S}) + NPages(\mathbf{R})) + 2 \cdot (NPages(\mathbf{S} \bowtie \mathbf{R}) + NPages(\mathbf{B}))$
- $2 \cdot (500 + 1000) + 2 \cdot (1000 + 10) =$ 5020 pages

Multiple-Relation Queries

Repeated enumeration of identical sub-plans



- Plan enumeration reconsiders the same sub-plans over and over
  - cost and result size of sub-plan are **independent** of embedding plan
  - optimizer needs to avoid **regenerating** and **reassessing** such sub-plans
- Dynamic programming **memoizes** already considered sub-plans

Dynamic Programming

- Dynamic programming approach was pioneered by System R
  - find the cheapest plan for an *n*-way join in *n* **passes**
  - in each pass *k*, find the best plan for all ***k*-relation sub-plans**
  - **construct** the plans in pass *k* by joining another relation to the best (*k* – 1)-relation sub-plans found in earlier passes

Principle of Optimality

Assumption

↳ To find the optimal **global plan**, it is sufficient to only consider the optimal plans for all its possible **sub-plans**

Dynamic Programming

Pass 1 (all 1-relation plans)
- find best 1-relation plan for each relation
- this pass mainly consists of selecting access method
- also see discussion on single-relation queries

Keep the best 1-relation plans for each set of physical properties

Pass 2 (all 2-relation plans)
- find best way to join sub-plans from Pass 1 to another relation
- generate left-deep trees with sub-plans from Pass 1 as outer relation in these joins

Again, keep the best 2-relation plans for each set of physical properties

⋮

Pass *n* (all n-relation plans)
- find best way to join sub-plans of Pass $n - 1$ to the nth relation
- sub-plans of Pass $n - 1$ appear as outer relation in this join

Return the **overall best** plan

Dynamic Programming

Number of join orderings enumerated

Ignoring the physical properties of a plan, the presence of different join algorithms, and the availability of indexes, how many plans does the dynamic programming strategy enumerate to find the best plan for **A** ⋈ **B** ⋈ **C** ⋈ **D**?

Dynamic Programming

- Dynamic programming records **cost** and **result size estimates** for each retained plan
- **Pruning** for each subset of joined relations
  - keep cheapest sub-plan **overall**
  - keep cheapest sub-plans that generate an intermediate result with an **interesting order** of tuples
  - discard sub-plans that involve **cross-products** due to lack of a join condition between two relations
- **Interesting order** determined by
  - presence of SQL **`ORDER BY`** clause in the query
  - presence of SQL **`GROUP BY`** clause in the query
  - join attributes of subsequent equi-joins (prepare for merge join)

Dynamic Programming

Step-by-step example: Setup and assumptions

```
SELECT S.sname, R.rname, B.bname
  FROM Sailors S, Reserves R, Boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

Assumptions

- Available join algorithms
  - merge join
  - block nested loops join
  - index nested loops join
- Available indexes
  - clustered B+ tree index **I** on **S.sid**, *height*(**I**) = 3, *INPages*(**I**) = 500
- Relation cardinality
  - *NPages*(**S**) = 10,000 pages, 5 tuples/page
  - *NPages*(**R**) = 10 pages, 10 tuples/page
  - *NPages*(**B**) = 10 pages, 20 tuples/page
- 10 (**R** ⋈ **S**)-tuples fit on a page, 10 (**B** ⋈ **R**)-tuples fit on a page

Dynamic Programming

Step-by-step example: Pass 1 (1-relation plans)

- access methods for **S**
  1. heap scan
     cost = *NPages*(**S**) = 10,000
  2. index scan on **S.sid**, index **I**
     cost = *INPages*(**I**) + *NPages*(**S**) = 10,500

$opt_1(\mathbf{S})$

$opt_2(\mathbf{S})$

- access method for **R**
  1. heap scan
     cost = *NPages*(**R**) = 10

opt(**R**)

- access method for **B**
  1. heap scan
     cost = *NPages*(**B**) = 10

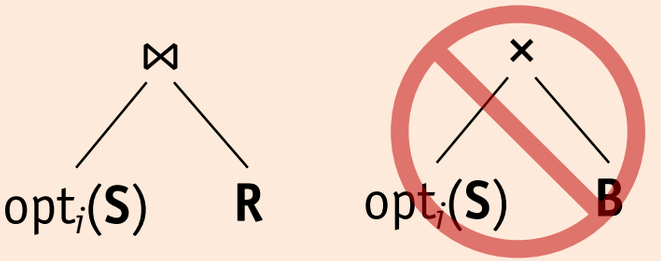
opt(**B**)

Note that both $opt_1(\mathbf{S})$ and $opt_2(\mathbf{S})$ are kept, since $opt_2(\mathbf{S})$ has an **interesting order** on attribute **sid**, which is a join attribute

Dynamic Programming

Step-by-step example: Pass 2 (2-relation plans)

Plans with $\text{opt}_i(\mathbf{S})$ as outer relation



- $\text{opt}_i(\mathbf{S})$ with **R** as inner relation
  1. **block nested loops join**, $\text{opt}_1(\mathbf{S})$ using **heap scan**
     $cost = 10{,}000 + 10{,}000 \cdot NPages(\mathbf{R}) = \underline{110{,}000}$
  2. **merge join** (with two-way sort), $\text{opt}_1(\mathbf{S})$ using **heap scan**
     $cost = 10{,}000 + 2 \cdot 10{,}000 + 2 \cdot NPages(\mathbf{R}) + NPages(\mathbf{R}) = \underline{30{,}030}$
  3. **block nested loops join**, $\text{opt}_2(\mathbf{S})$ using **index scan**
     $cost = 10{,}500 + NPages(\mathbf{S}) \cdot NPages(\mathbf{R}) = \underline{110{,}500}$
  4. **merge join** (with two-way sort), $\text{opt}_2(S)$ using **index scan**
     $cost = 10{,}500 + 2 \cdot NPages(\mathbf{R}) + NPages(\mathbf{R}) = \underline{10{,}530}$
- $\text{opt}_i(\mathbf{S})$ with **B** as inner relation needs **cross-product** and gets **pruned**

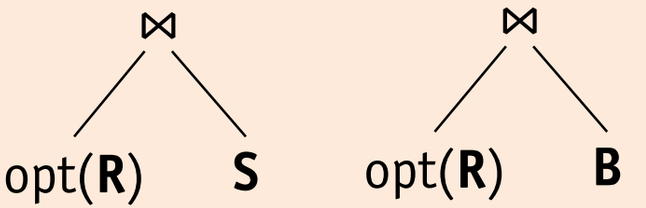
$\text{opt}_1(\mathbf{R}, \mathbf{S})$

Note that opt(**R**, **S**) exploits an interesting order of a non-optimal sub-plan

Dynamic Programming

Step-by-step example: Pass 2 (2-relation plans, cont'd)

Plans with opt(**R**) as outer relation



- opt(**R**) with **S** as inner relation
  1. **block nested loops join**, **heap scan** on **S**
     cost = 10 + 10 · *NPages*(**S**) = 100,010
  2. **index nested loops join**, **index access** on **S**
     cost = 10 + *NTuples*(**R**) · (*height*(**I**) + 1) = 410
  3. **merge join** (with two-way sort), **heap scan** on **S**
     cost = 10 + *NPages*(**S**) + 2 · (10 + *NPages*(**S**)) = 30,300
  4. **merge join** (with two-way sort), **index scan** on **S**
     cost = 10 + 2 · 10 + 10,500 = 10,530
- opt(**R**) with **B** as inner relation
  5. **block nested loops join**
     cost = 10 + 10 · *NPages*(**B**) = 110
  6. **merge join** (with two-way sort)
     cost = 10 + *NPages*(**B**) + 2 · (10 + *NPages*(**B**)) = 60

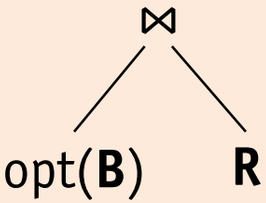
$\text{opt}_2(\mathbf{R}, \mathbf{S})$

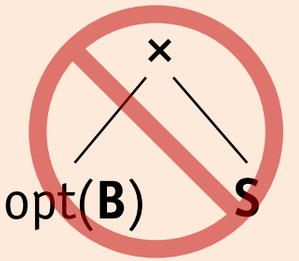
$\text{opt}_1(\mathbf{B}, \mathbf{R})$

Dynamic Programming

Step-by-step example: Pass 2 (2-relation plans, cont'd)

Plans with opt(**B**) as outer relation





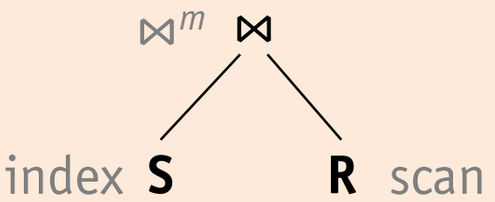
- opt(**B**) with **R** as inner relation
  1. **block nested loops join**
     cost = 10 + 10 · *NPages*(**R**) = 110
  2. **merge join** (with two-way sort)
     cost = 10 + *NPages*(**R**) + 2 · (10 + *NPages*(**R**)) = 110
- opt(**B**) with **S** as inner relation needs **cross-product** and gets **pruned**

opt_2(**B**, **R**)

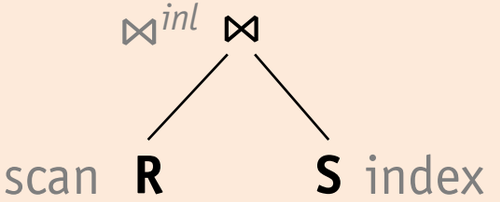
Dynamic Programming

Step-by-step example: Summary of Pass 2

- **R** ⋈ **S**

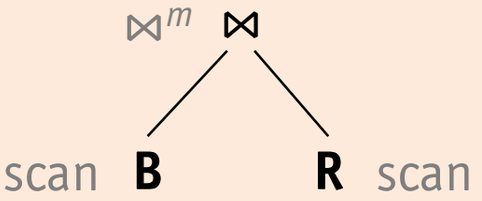


opt_1(**R**, **S**)
cost = 10,530
order on **`sid`**

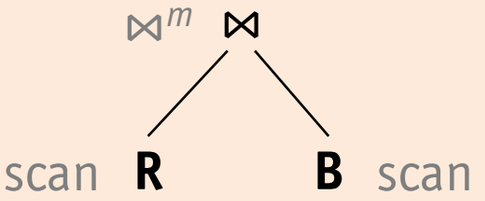


opt_2(**R**, **S**)
cost = 410
no order

- **B** ⋈ **R**



opt_1(**B**, **R**)
cost = 60
order on **`bid`**



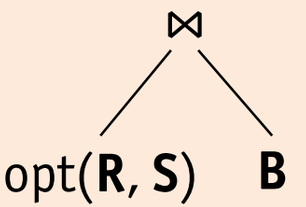
opt_2(**B**, **R**)
cost = 60
order on **`bid`**

Plans opt_2(**R**, **S**) *and* opt_1(**B**, **R**) *or* opt_2(**B**, **R**) are kept for the next pass. Note that the order in opt_1(**R**, **S**) is **not** interesting for subsequent join(s).

Dynamic Programming

Step-by-step example: Pass 3 (3-relation plans)

Plans with opt(**R**, **S**) as outer relation



- opt(**R**, **S**) with **B** as inner relation
  1. **block nested loops join**
     cost = 410 + *NPages*(**R** ⋈ **S**) · *NPages*(**B**) = 510
  2. **merge join** (with two-way sort)
     cost = 410 + *NPages*(**B**) + 2 · (*NPages*(**R** ⋈ **S**) + *NPages*(**B**)) = 460

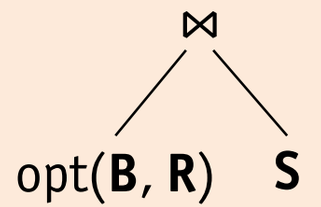
opt_1(**B**, **R**, **S**)

Observe that **R.sid** is a foreign key pointing to **S.sid**. Therefore, every tuple in **R** matches **exactly one** tuple in **S**. Since there are 100 **R**-tuples, *NTuples*(**R** ⋈ **S**) = 100. As 10 (**R** ⋈ **S**)-tuples fit in one page, NPages(**R** ⋈ **S**) = 10.

Dynamic Programming

Step-by-step example: Pass 3 (3-relation plans)

Plans with opt(**B**, **R**) as outer relation



- opt(**B**, **R**) with **S** as inner relation
  1. **block nested loops join**, **heap scan** on **S**
     cost = 60 + *NPages*(**B** ⋈ **R**) · *NPages*(**S**) = 100,060
  2. **index nested loops join**, **index access** on **S**
     cost = 60 + *NTuples*(**B** ⋈ **R**) · (*height*(**I**) + 1) = 460
  3. **merge join** (with two-way sort), **heap scan** on **S**
     cost = 60 + *NPages*(**S**) + 2 · (*NPages*(**B** ⋈ **R**) + *NPages*(**S**)) = 30,080
  4. **merge join** (with two-way sort), **index scan** on **S**
     cost = 60 + 2 · *NPages*(**B** ⋈ **R**) + 10,500 = 10,580

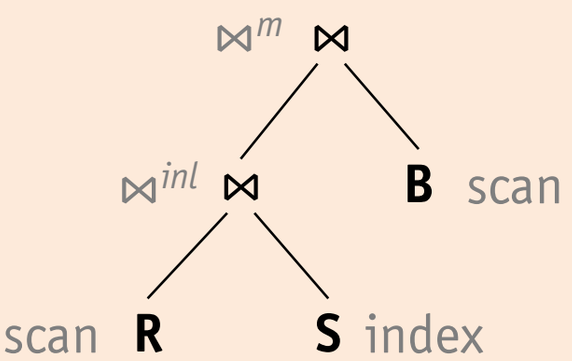
$\text{opt}_2(\mathbf{B}, \mathbf{R}, \mathbf{S})$

Observe that **R.bid** is a foreign key pointing to **B.bid**. Therefore, every tuple in **R** matches **exactly one** tuple in **B**. Since there are 100 **R**-tuples, *NTuples*(**B** ⋈ **R**) = 100. As 10 (**B** ⋈ **R**)-tuples fit in one page, *NPages*(**B** ⋈ **R**) = 10.

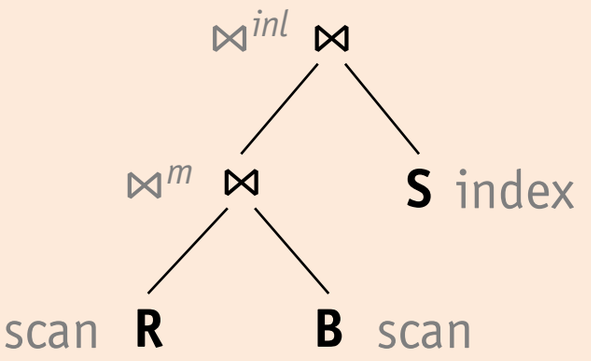
Dynamic Programming

Step-by-step example: Summary of Pass 3

- **B** ⋈ **R** ⋈ **S**



opt_1(**B**, **R**, **S**)
cost = 460



opt_2(**B**, **R**, **S**)
cost = 460

Observations

- best plans mix join algorithms and exploits indexes
- cost of worst plan > 100,000 (exact cost unknown due to pruning)
- optimization yielded ≈ 1000-fold improvement over the worst plan!

Dynamic Programming

Algorithm to find an optimal n-way left-deep join tree

function optimize($q(R_1, \ldots, R_n)$**)**
  for $i = 1$ **to** n **do** *(Pass 1)*
    $optPlan(\{R_i\}) \leftarrow$ **accessPlans(**R_i**) ;**
    prunePlans($optPlan(\{R_i\})$**) ;**
  for $i = 2$ **to** n **do** *(Pass 2 to n)*
    foreach $S \subseteq \{R_1, \ldots, R_n\}$ such that $|S| = i$ **do**
      $optPlan(S) \leftarrow \emptyset$ **;**
      foreach $R \subseteq \{R_1, \ldots, R_n\}$ **do**

$$optPlan(S) \leftarrow optPlan(S) \cup \textbf{possibleJoins}\left[\bowtie\left(optPlan(S \backslash R),\ optPlan(R)\right)\right];$$

      prunePlans($optPlan(S)$**) ;**
  return $optPlan(\{R_1, \ldots, R_n\})$**;**
end

Dynamic Programming

- Subroutines in dynamic programming algorithm
  - **`accessPlans(`***R*`)`** enumerates all access plans for a single relation *R*
  - **`possibleJoins(`***R* ⋈ *S*`)`** enumerates the possible joins between relations *R* and *S*, e.g., nested loops join, sort-merge join, hash join, etc.
  - **`prunePlans(`***set*`)`** discards all but the best plans from *set*
- Function **`optimize()`** draws it advantage from **filtering** candidate plans early in the process
  - Pass 2 of the example keeps 2 out of 12 plans
  - Pass 3 of the example keeps 1 out of 6 plans
- Heuristics are used to reduce the search space and to balance **plan quality** and **optimizer runtime**

Joining Many Relations

Optimization time for chain and star queries in the Columbia optimizer

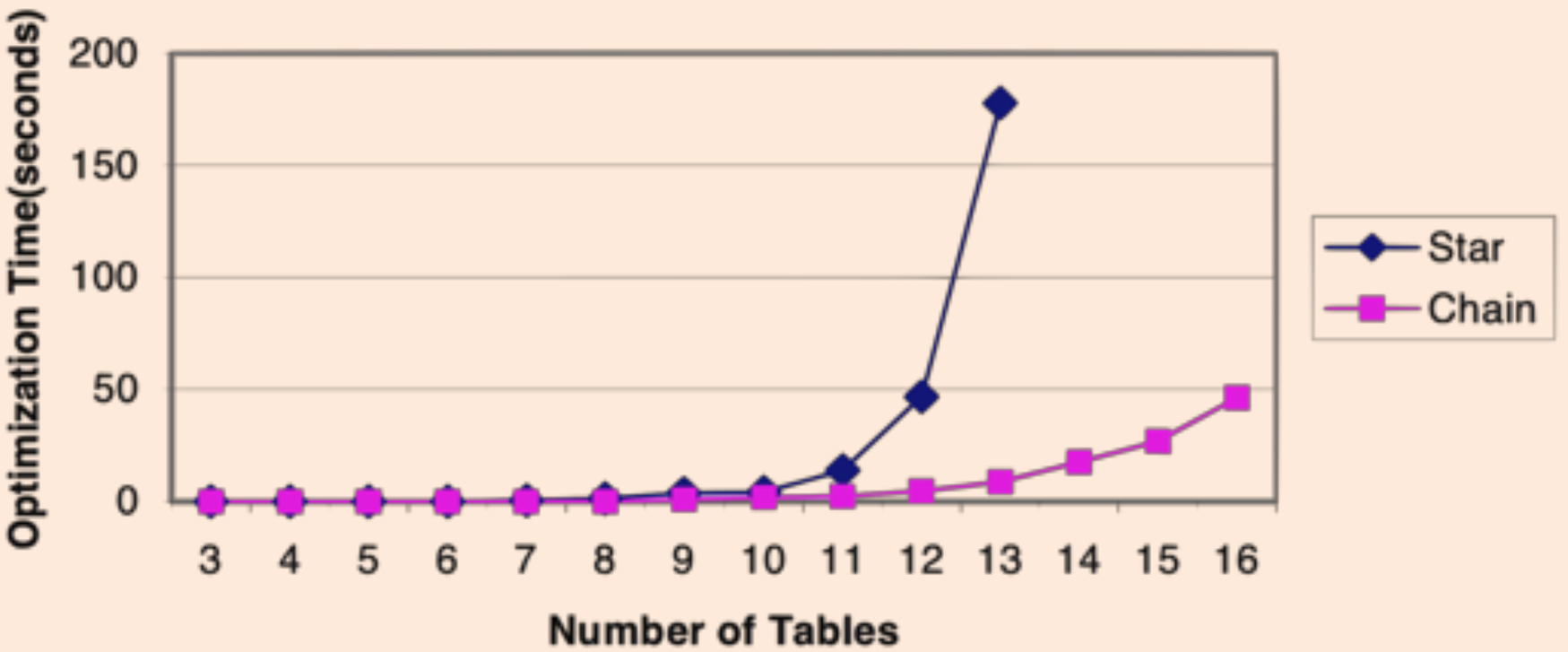


Figure Credit: Yongwen Xu, "Efficiency in the Columbia Database Query Optimizer", Portland State University, 1998.

Joining Many Relations

- Join enumeration still has **exponential** resource requirements
  - time complexity: $O(3^n)$
  - space complexity: $O(2^n)$
- This approach may still be too expensive
  - for joins involving many relations (~10-20 and more)
  - for simple queries over well-indexed data (where the right plan choice should be easy to make)
- The **greedy join enumeration** algorithm aims to close this gap

See: Kiyoshi Ono and Guy M. Lohman: "Measuring the Complexity of Join Enumeration in Query Optimization", In *Proc. VLDB, 1990.*

Joining Many Relations

Greedy join enumeration algorithm for an n-way join

```
function optimize-greedy(q(R_1, ..., R_n))
  worklist ← ∅ ;
  for i = 1 to n do
    worklist ← worklist ∪ bestAccessPlans(R_i) ;
  for i = n downto 2 do                                  (worklist = {P_1, ..., P_i})
    find P_j, P_k ∈ worklist ∧ ⋈* such that cost(P_j ⋈* P_k) is minimal ;
    worklist ← worklist \ {P_j, P_k} ∪ {(P_j ⋈* P_k)} ;
  return single plan left in worklist ;                  (worklist = {P_1})
end
```

- In each iteration, choose the **cheapest** join that can be made over the remaining sub-plans at that time (greedy part)
- Observe that `optimize-greedy()` operates similar to finding the optimum binary tree for **Huffman coding**

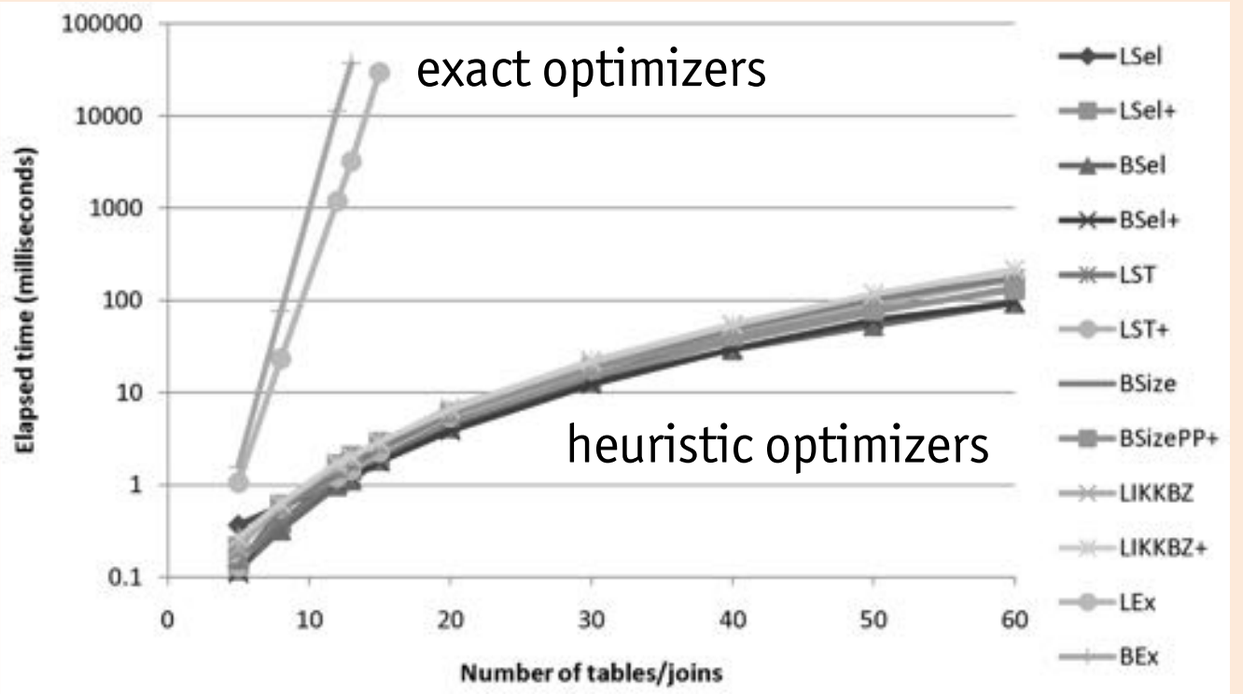
Slide Credit: Torsten Grust, University of Tübingen, Germany

Joining Many Relations

- Greedy join enumeration algorithm has $O(n^3)$ time complexity
  - each loop has n iterations: $O(n)$
  - each iteration looks at all remaining pairs of plans in *worklist*: $O(n^2)$
- Greedy algorithm considers all types of join trees
  - balances risk of returning a really bad plan
  - opens opportunities for additional (rewrite) heuristics
- There are also other join enumeration techniques
  - **randomized algorithms**: randomly rewrite the join tree on rewrite as a time, using **hill-climbing** or **simulated annealing** strategy to find optimal plan
  - **genetic algorithms**: explore plan space by **combining** plans ("offspring") and **altering** some plans randomly ("mutations")

Joining Many Relations

Performance of different optimization alternatives



- Naming scheme
  - Search space: **L** (linear) or **B** (bushy)
  - Ranking: **Sel** (*minSel*), **Size** (*minSize*), or **ST** (*smallestTable*)
  - Merging: **+** (*Switch-HJ* and *Switch- Idx*) or **PP** (*Pull/Push*)

Figure Credit: Nicolas Bruno, Cesar Galindo-Legaria, and Milind Joshi, "Polynomial Heuristics for Query Optimization", In *Proc. ICDE, 2010.*

Cardinality Estimation

- Recall the **cost model** for single-relation plans used so far
- Cost estimation of a multi-relation plan additionally involves **cardinality estimation** of intermediate query results
  - cost of a plan is dominated by page I/O operations
  - cost of implementation algorithms is determined by size of inputs
- Cardinality estimates are also used to allocate buffer pages or to determine blocking factor b for blocked I/O

Cost model for access methods on relation *R*

Access method	Cost
access primary index **I**	$\begin{cases} height(\mathbf{I}) + 1 & \text{if } \mathbf{I} \text{ is B+ tree} \\ 1.2 + 1 & \text{if } \mathbf{I} \text{ is hash index} \end{cases}$
clustered index **I** matching predicate p	$(NPages(\mathbf{I}) + NPages(\mathbf{R})) \cdot sel(p)$
unclustered index **I** matching predicate p	$(NPages(\mathbf{I}) + NTuples(\mathbf{R})) \cdot sel(p)$
sequential scan	$NPages(\mathbf{R})$

Cardinality Estimation

- A **database profile** is one of two principal approaches to query result cardinality estimation
  - **base relations:** maintain statistical information, e.g., number and sizes of tuples, distribution of attribute values, etc. in database catalog
  - **intermediate query results:** derive this information based on a simple statistical model during query optimization
- Remarks
  - statistical model typically assumes **uniformity** and **independence**
  - both are typically **not valid**, but they allow for simple calculations
  - system can record **histograms** that approximate value distributions more accurately in order to provide better cardinality estimations

Cardinality Estimation

- Alternatively, **sampling techniques** can be used to estimate query result set cardinality
  - run query on a **small sample** of the database
  - **gather necessary statistics** about query plan
  - **extrapolate** to full input size at query execution time
- Remarks
  - it is crucial to find the right balance between sample size (performance) and the resulting accuracy of estimation

Simple Database Profiles

- Keep profile information in the database catalog
  - update information whenever database updates are issued
  - derive a (**very simple**) statistical model using **primitive assumptions**

Typical database profile for relation *R*

NTuples(**R**)	number of tuples in relation **R**
NPages(**R**)	number of disk pages allocated for relation **R**
s(**R**)	average record size (width) of relation **R**
b	block size, alternative to *s*(**R**)
	$NPages(N) = NTuples(R) / \lfloor {}^{b}/_{s(R)} \rfloor$
V(**A**, **R**)	number of distinct values of attribute **A** in relation **R**
High(**A**, **R**)/*Low*(**A**, **R**)	maximum and minimum value of attribute **A** in relation **R**
MCV(**A**, **R**)	most common value(s) of attribute **A** in relation **R**
MVF(**A**, **R**)	frequency of most common value(s) of attribute **A** in relation **R**
⋮	*possibly many more*

Simple Database Profiles

- In order to obtain a simple and tractable cardinality estimation formulae, assume **one of the following**

Assumptions

1. **Uniformity and independence assumption**
   All values of an attribute uniformly appear with the same probability (or even distribution). Values of different attributes are independent of each other.
   ↳ *Simple, yet rarely realistic assumption*
2. **Worst case assumption**
   No knowledge about relation contents available at all. In case of a selection σ_p, assume that all records will satisfy predicate p.
   ↳ *Unrealistic assumption, can only be used for computing upper bounds*
3. **Perfect knowledge assumption**
   Details about the exact distribution of values are known. Requires huge catalog or prior knowledge of incoming queries.
   ↳ *Unrealistic assumption, can only be used for computing lower bounds*

Simple Database Profiles

- Find formulae describing properties of intermediate query results for all (logical) operators
  - express formulae in terms of profile information
  - database systems typically assume uniformity and independence

Selection query $Q := \sigma_{\mathtt{A}=c}(\mathbf{R})$

Selectivity $sel(\mathtt{A} = c)$	$\begin{cases} MCF(\mathtt{A}, \mathbf{R})[c] & \text{if } c \in MCF(\mathtt{A}, \mathbf{R}) \\ 1/V(\mathtt{A}, \mathbf{R}) & (\textbf{uniformity assumption}) \end{cases}$
Cardinality $\lvert Q \rvert$	$sel(\mathtt{A} = c) \cdot NTuples(\mathbf{R})$
Record size $s(Q)$	$s(\mathbf{R})$
Number of attribute values $V(\mathtt{A}', Q)$	$\begin{cases} 1, & \text{for } \mathtt{A}' = \mathtt{A} \\ c(\lvert\mathbf{R}\rvert, V(\mathtt{A}, \mathbf{R}), \lvert Q \rvert) & \text{otherwise} \end{cases}$

Simple Database Profiles

- Number $c(|\mathbf{R}|, V(\mathtt{A}, \mathbf{R}), |Q|)$ of distinct values in attribute **`A`**‘ after selection, is estimated using a well-known statistics formula
- Number c of distinct colors obtained by drawing r balls from a bag of n balls in m different colors is $c(n, m, r)$

$$c(n,m,r) = \begin{cases} r, & \text{for } r < \frac{m}{2} \\ \frac{r+m}{3}, & \text{for } \frac{m}{2} \le r < 2m \\ m, & \text{for } r \ge 2m \end{cases}$$

Simple Database Profiles

Selection query $Q := \sigma_{A=B}(R)$

Equality between attributes, e.g., $\sigma_{A=B}(\mathbf{R})$ can be approximated by

$$sel(\mathbf{A} = \mathbf{B}) = 1/\max(V(\mathbf{A}, \mathbf{R}), V(\mathbf{B}, \mathbf{R}))$$

This formula assumes that each value of the attribute with fewer distinct values has a corresponding match in the other attribute (**independence assumption**).

Selection query $Q := \sigma_{A>c}(R)$

If $Low(\mathbf{A}, \mathbf{R}) \leq c \leq High(\mathbf{A}, \mathbf{R})$, range selections, e.g., $\sigma_{A>c}(\mathbf{R})$ can be approximated by

$$sel(\mathbf{A} > c) = \frac{High(\mathbf{A}, \mathbf{R}) - c}{High(\mathbf{A}, \mathbf{R}) - Low(\mathbf{A}, \mathbf{R})}$$

This formula uses the **uniformity assumption**.

Selection query $Q := \sigma_{A \text{ IN } (\ldots)}(R)$

Element tests, e.g., $\sigma_{A \text{ IN } (\ldots)}(\mathbf{R})$ can be approximated by multiplying the selectivity for an equality selection $sel(\mathbf{A} = c)$ with the number of elements in the list of values.

Simple Database Profiles

- Estimating the number of result tuples of a **projection** query is difficult due to effect of duplicate elimination

Projection query $Q := \pi_L(R)$

Cardinality $\lvert Q \rvert$	$V(\mathbf{A}, \mathbf{R})$,	for $L = \{\mathbf{A}\}$
	$\lvert \mathbf{R} \rvert$	if keys of $\mathbf{R} \in L$
	$\lvert \mathbf{R} \rvert$	no duplicate elimination
	$\min(\lvert \mathbf{R} \rvert, \prod_{\mathbf{A}i \in L} V(\mathbf{A}_i, \mathbf{R}))$	otherwise
Record size $s(Q)$	$\sum_{\mathbf{A}i \in L} s(\mathbf{A}_i)$	
Number of attribute values $V(\mathbf{A}_i, Q)$	$V(\mathbf{A}_i, \mathbf{R})$	for $\mathbf{A}_i \in L$

Simple Database Profiles

Union query $Q := R \cup S$

$$\begin{aligned} |Q| &\leq |\mathbf{R}| + |\mathbf{S}| \\ s(Q) &= s(\mathbf{R}) = s(\mathbf{S}) \\ V(\mathtt{A}, Q) &\leq V(\mathtt{A}, \mathbf{R}) + V(\mathtt{A}, \mathbf{S}) \end{aligned} \qquad \text{sch}(\mathbf{R}) = \text{sch}(\mathbf{S})$$

Difference query $Q := R - S$

$$\max(0, |\mathbf{R}| - |\mathbf{S}|) \leq |Q| \leq |\mathbf{R}|$$
$$\begin{aligned} s(Q) &= s(\mathbf{R}) = s(\mathbf{S}) \\ V(\mathtt{A}, Q) &\leq V(\mathtt{A}, \mathbf{R}) \end{aligned}$$

Cross-product query $Q := R \times S$

$$\begin{aligned} |Q| &= |\mathbf{R}| \cdot |\mathbf{S}| \\ s(Q) &= s(\mathbf{R}) + s(\mathbf{S}) \\ V(\mathtt{A}, Q) &= \begin{cases} V(\mathtt{A}, \mathbf{R}), & \text{if } \mathtt{A} \in sch(\mathbf{R}) \\ V(\mathtt{A}, \mathbf{S}), & \text{if } \mathtt{A} \in sch(\mathbf{S}) \end{cases} \end{aligned}$$

Simple Database Profiles

- Cardinality estimation for the general **join** case is challenging
  - there are, however, a few special simple cases
  - a very common simple case is the **foreign key relationship**

Special cases of join queries $Q := \mathbf{R} \bowtie_p \mathbf{S}$

- no common attributes ($sch(\mathbf{R}) \cap sch(\mathbf{S}) = \emptyset$) or join predicate $p = \text{true}$

$$\mathbf{R} \bowtie_p \mathbf{S} = \mathbf{R} \times \mathbf{S}$$

- join attribute, say $\mathtt{A}$, is key in one of the relations, e.g., in $\mathbf{R}$, and assuming the inclusion dependency $\pi_{\mathtt{A}}(\mathbf{S}) \subseteq \pi_{\mathtt{A}}(\mathbf{R})$

$$|Q| = |\mathbf{R}|$$

this inclusion dependency is guaranteed by a foreign key relationship between $\mathbf{R}.\mathtt{A}$ and $\mathbf{S}.\mathtt{A}$ in $\mathbf{R} \bowtie_{\mathbf{R}.\mathtt{A}=\mathbf{S}.\mathtt{A}} \mathbf{S}$.

Simple Database Profiles

General join queries $Q := R \bowtie_{R.A=S.B} S$

Assuming inclusion dependencies, the cardinality of a general join query Q can be estimated as

Cardinality $|Q|$

$$\begin{cases} \dfrac{|\mathbf{R}| \cdot |\mathbf{S}|}{V(\mathbf{A}, \mathbf{R})}, & \text{for } \pi_{\mathbf{B}}(\mathbf{S}) \subseteq \pi_{\mathbf{A}}(\mathbf{R}) \\ \dfrac{|\mathbf{R}| \cdot |\mathbf{S}|}{V(\mathbf{B}, \mathbf{S})}, & \text{for } \pi_{\mathbf{A}}(\mathbf{R}) \subseteq \pi_{\mathbf{B}}(\mathbf{S}) \end{cases}$$

Typically, the smaller of these two estimates is used

Cardinality $|Q|$

$$\frac{|\mathbf{R}| \cdot |\mathbf{S}|}{\max(V(\mathbf{A}, \mathbf{R}), V(\mathbf{B}, \mathbf{S}))}$$

Record size $s(Q)$ $\quad s(\mathbf{R}) + s(\mathbf{S}) - \sum s(\mathbf{A}_i)$ for all common $\mathbf{A}_i$ of a **natural join**

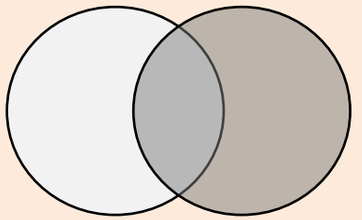
Number of attribute values $V(\mathbf{A}', Q) \leq \begin{cases} \min(V(\mathbf{A}', \mathbf{R}), V(\mathbf{A}', \mathbf{S})), & \mathbf{A}' \in sch(\mathbf{R}) \cap sch(\mathbf{S}) \\ V(\mathbf{A}', X), & \mathbf{A}' \in sch(X) \end{cases}$

Simple Database Profiles

- Estimating selectivity of selections with **composite predicates**
  - compute selectivity of each individual condition separately
  - combine results under the independence assumption

Selections with composite predicates

- **conjunctive** predicates, e.g., $Q := \sigma_{\mathbf{A}=c1 \wedge \mathbf{B}=c2}(\mathbf{R})$



$$sel(\mathbf{A} = c_1 \wedge \mathbf{B} = c_2) = sel(\mathbf{A} = c_1) \cdot sel(\mathbf{B} = c_2)$$

which gives $|Q| = \dfrac{|\mathbf{R}|}{V(\mathbf{A}, \mathbf{R}) \cdot V(\mathbf{B}, \mathbf{R})}$

- **disjunctive** predicates, e.g., $Q := \sigma_{\mathbf{A}=c1 \vee \mathbf{B}=c2}(\mathbf{R})$

$$sel(\mathbf{A} = c_1 \vee \mathbf{B} = c_2) = sel(\mathbf{A} = c_1) + sel(\mathbf{B} = c_2) - sel(\mathbf{A} = c_1) \cdot sel(\mathbf{B} = c_2)$$

which gives $|Q| = \dfrac{|\mathbf{R}|}{V(\mathbf{A}, \mathbf{R}) + V(\mathbf{B}, \mathbf{R}) - V(\mathbf{A}, \mathbf{R}) \cdot V(\mathbf{B}, \mathbf{R})}$

Histograms

- In realistic database instances, values are **not uniformly distributed** across the active domain of an attribute
- To keep track of non-uniform value distribution of attribute **A**, maintain a **histogram** to approximate the actual distribution
  1. divide the active domain of **A** into **adjacent intervals** (so-called **buckets**) by selecting boundary values $b_i \in \mathrm{dom}(\mathbf{A})$
  2. collect **statistical parameters** for each interval such as the number of tuples $b_{i-1} < \mathrm{t}[\mathbf{A}] \leq b_i$ or the number of distinct **A**-values in that interval
- Histograms allow for more exact estimates of both **equality** and **range selections**

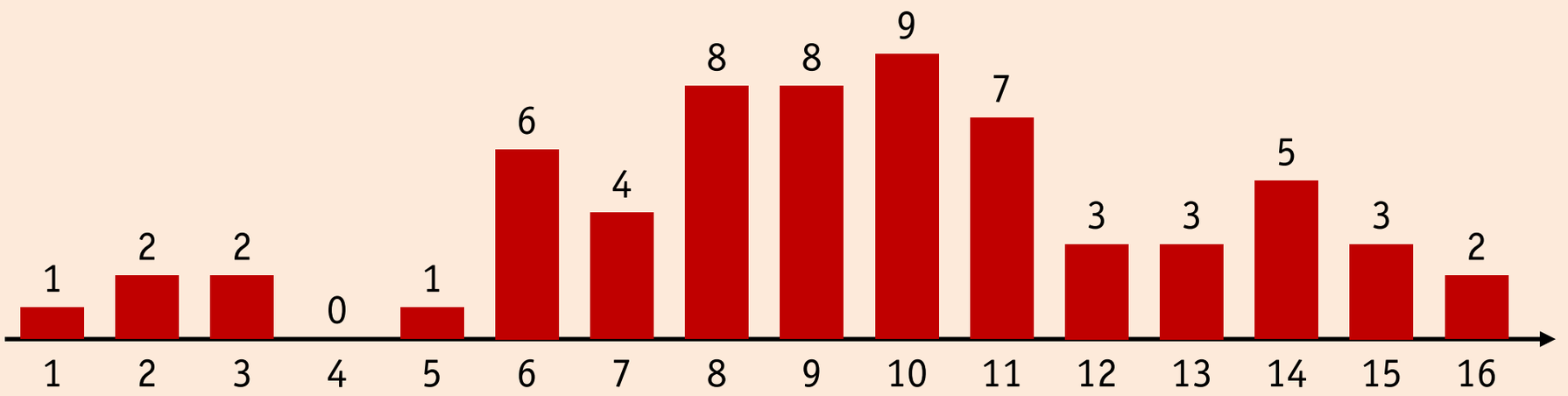
Histograms

- Two types of histograms are widely used
  1. **Equi-Width Histograms**
     All histogram buckets have the **same with**, i.e., boundary $b_i = b_i + w$, for some fixed width w
  2. **Equi-Depth Histograms**
     All histogram buckets contain the **same number of tuples**, i.e., their width is varying
- Equi-depth histograms are better able to adapt to data skew (high uniformity)
- Number of histogram buckets can be used to control the **trade-off** between
  - **histogram resolution**, which defines estimation quality
  - **histogram size** (space in database catalog is limited)

Histograms

Example: actual value distribution

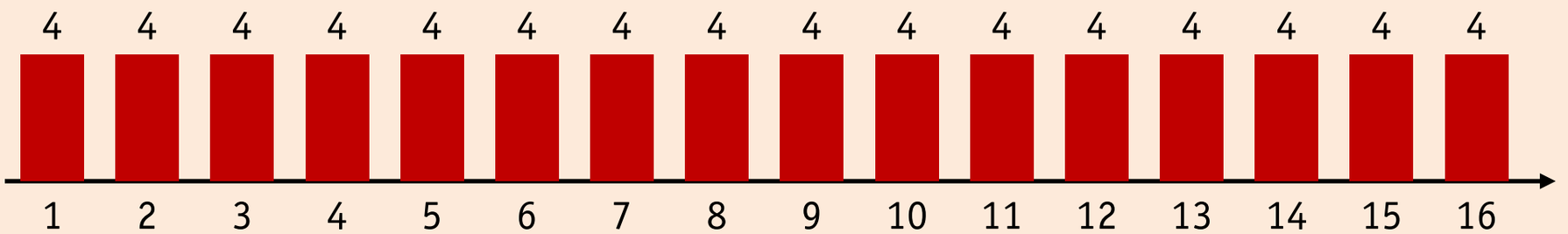
Assume a column **A** of SQL type **`INTEGER`** (domain {..., **`-2`**, **`-1`**, **`0`**, **`1`**, **`2`**, ...}) with the following actual non-uniform value distribution in a relation **R**.



Histograms

Example: uniform distribution assumption

For skewed, i.e., non-uniform, data distributions, working without histograms gives **bad** estimates. Working under the uniformity assumption, the value distribution from the previous slide is approximated as follows.



As a consequence, the selectivity *sel*(**A** < 6) would be computed as $64/16 \cdot 5 = 20$, which is **far from correct**. The actual number of tuples that qualify is 6.

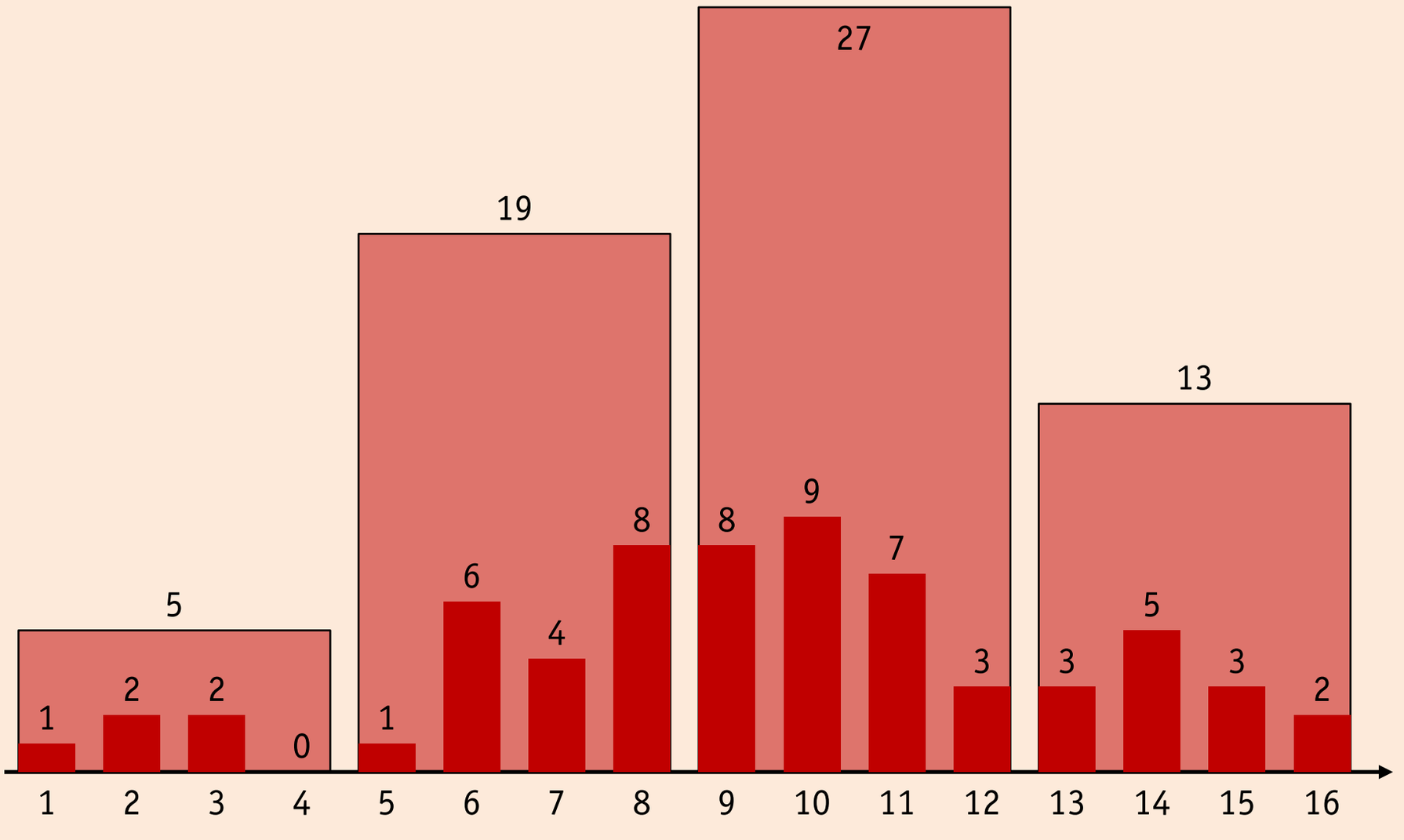
The error in estimation using the uniformity assumption is especially large for those values, which occur often in the database. This situation is particularly bad, since for those values the best estimates are actually required.

Histograms

- Construct an **equi-width histogram** on attribute **A** of relation **R**
  1. divide **active domain** of attribute **A** into B buckets of equal **bucket width** w, such that
  $$w = \frac{High(\mathbf{A}, \mathbf{R}) - Low(\mathbf{A}, \mathbf{R}) + 1}{B}$$
  2. bucket boundary $b_0 = 0$ and $b_i = b_{i-1} + w$
  3. while scanning **R** once sequentially, maintain B **running sums of value frequencies**, one for each bucket
- If scanning **R** is prohibitive, scan sample $\mathbf{R}_{sample} \subseteq \mathbf{R}$, then scale counters by $|\mathbf{R}|/|\mathbf{R}_{sample}|$
- To maintain histogram under insertions and deletions, simply increment or decrement frequency counter in affected bucket
- To estimate result cardinality, use uniformity assumption **within each bucket**

Histograms

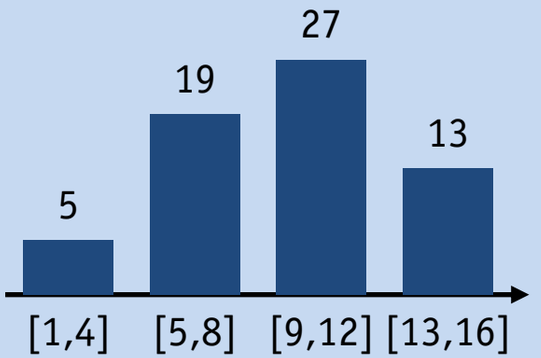
Example: equi-width histogram ($B = 4$)



Histograms

Cardinality estimates with equi-width histograms

What are the cardinality estimates for the following selections based on this **equi-width histogram**? How do they compare to the actual value and the value estimated under the uniformity assumption?



Equality selection query $Q := \sigma_{\mathbf{A}=5}(\mathbf{R})$

Range selection query $Q := \sigma_{\mathbf{A}<6}(\mathbf{R})$

Range selection query $Q := \sigma_{\mathbf{A}>6 \wedge \mathbf{A}<14}(\mathbf{R})$

Histograms

- In order to construct an **equi-depth histogram** on attribute **A** in relation **R**
  1. divide **active domain** of attribute **A** into B buckets of equal **bucket depth** d, such that $d = NTuples(\mathbf{R})/B$
  2. sort **R** by sort criterion on attribute **A**, e.g., natural order
  3. bucket boundary $b_0 = Low(\mathbf{A}, \mathbf{R})$, then determine b_i by dividing the sorted relation **R** into blocks of size ~d

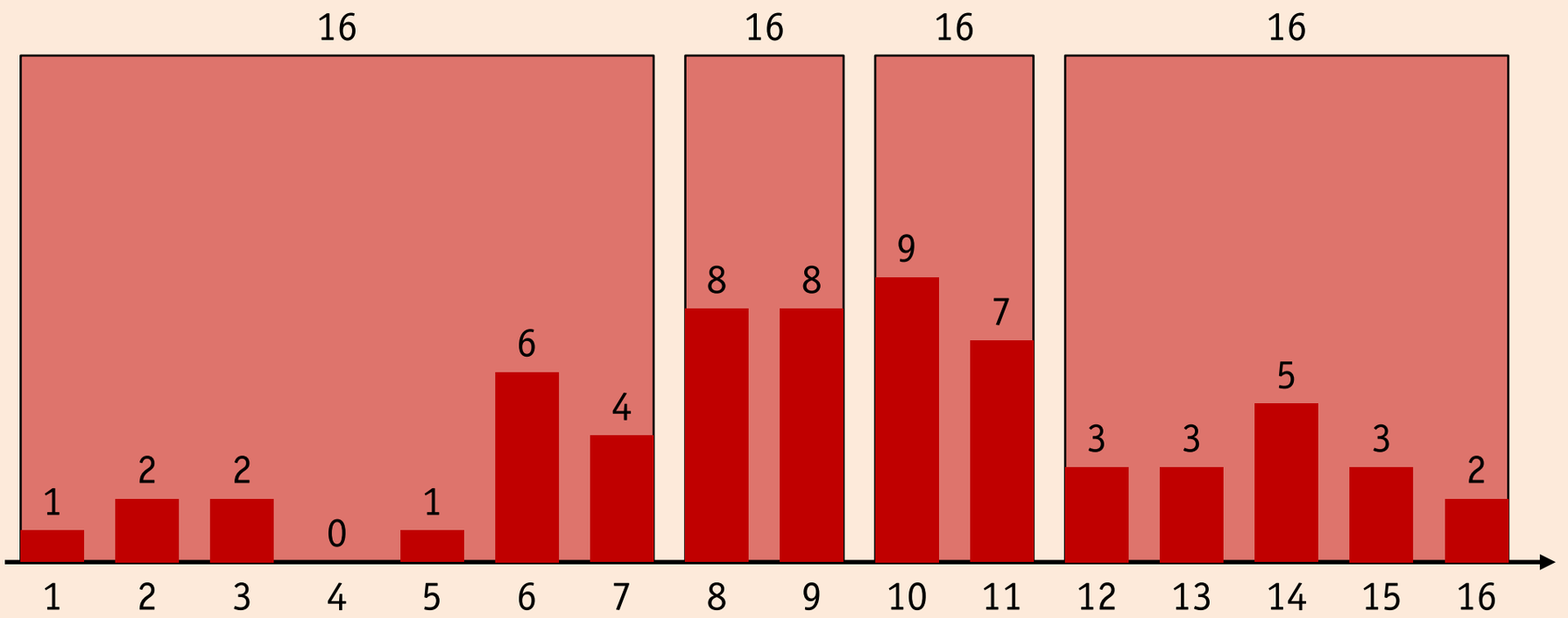
Example: equi-width histogram ($B = 4$, $|R| = 64$)

1. d = 64/4 = 16
2. sorted relation **R** (on attribute **A**)
   {1,2,2,3,3,5,6,6,6,6,6,6,7,7,7,7,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,10,10, ...}
3. boundaries of d-sized blocks in sorted relation **R**
   {1,2,2,3,3,5,6,6,6,6,6,6,7,7,7,7,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,10,10, ...}

$b_1 = 7$ $b_2 = 9$

Histograms

Example: equi-width histogram ($B = 4$, $d = 16$)

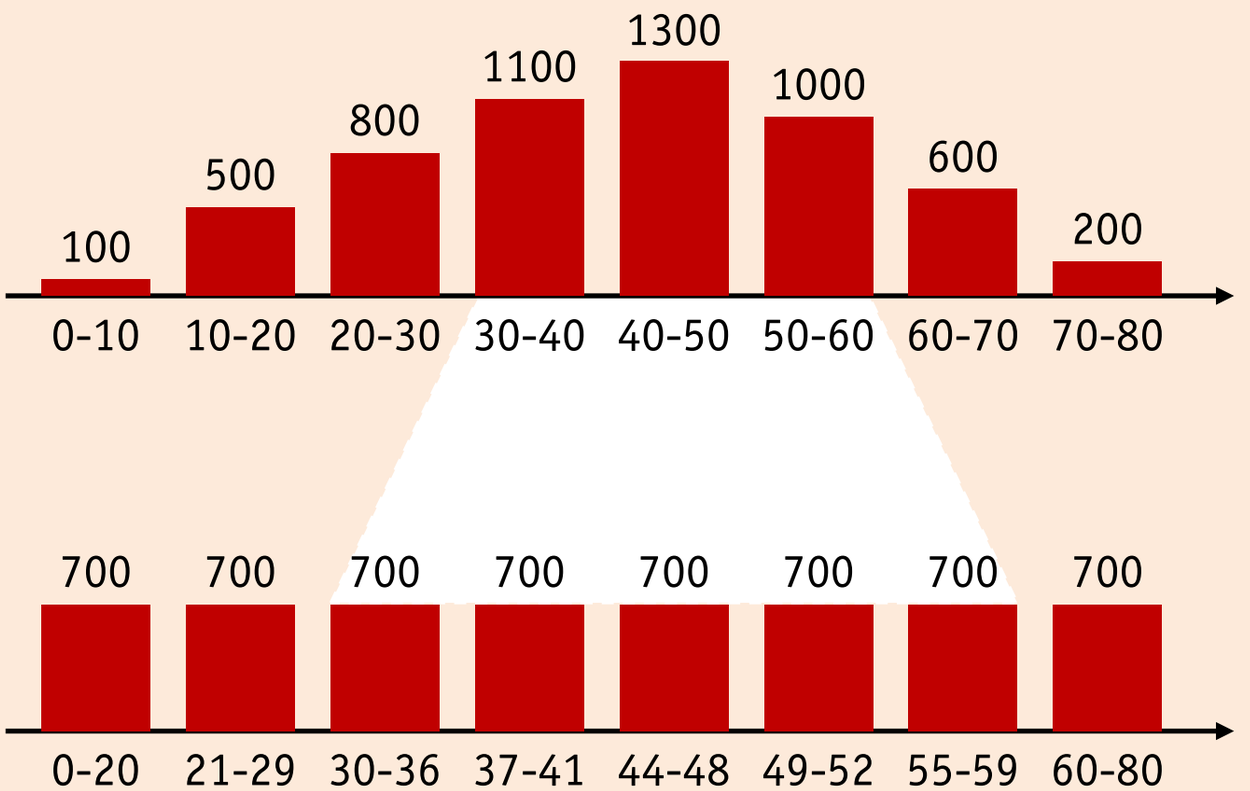


- Intuition
  - high value frequencies are **more important** than low frequencies
  - resolution of histogram **adapts** to skewed value distributions

Histograms

Example: equi-width vs. equi-depth histogram on person age ($B = 8$, $|R| = 5600$)

Equi-depth histogram "invests" bytes in the densely populated person age region between 30 and 59

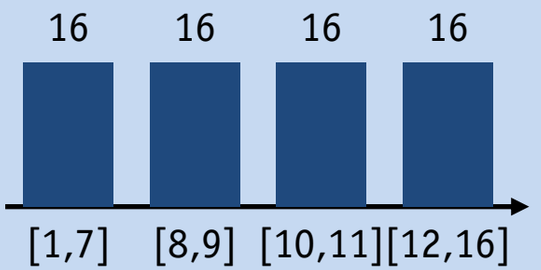


Histograms

Cardinality estimates with equi-width histograms

What are the cardinality estimates for the following selections based on this **equi-depth histogram**? How do they compare to the actual value and the value estimated using the equi-width histogram?

Equality selection query $Q := \sigma_{\mathbf{A}=5}(\mathbf{R})$



Range selection query $Q := \sigma_{\mathbf{A}<6}(\mathbf{R})$

Range selection query $Q := \sigma_{\mathbf{A}>6 \wedge \mathbf{A}<14}(\mathbf{R})$

Sampling

- Maintaining a database profile can be **costly** and **error-prone**
  - frequent database updates introduce overhead
  - provided estimates may be far-off, in particular for complex queries
- In such cases, it might be better to use **sampling** instead
  1. run complex query on a small sample of the data to collect statistics
  2. extrapolate statistics to full data set and optimize query accordingly
- Parameters of sampling
  - **sample size**: small enough to be executed efficiently, large enough to obtain useful characteristics
  - **extrapolation precision**: good predictions require a large (enough) sample
- Select a value for either one of those parameters and derive the other one

Sampling

- Three approaches are typically found in the literature
  - **adaptive sampling** tries to achieve a given precision with minimal sample size
  - **double (two-phase) sampling** obtains a course picture from a very small sample first and then computes the necessary sample size for a “useful” sample in a second step
  - **sequential sampling:** uses a sliding (continuous) calculation of characteristics and stops once the estimated precision is high enough

Example Query Optimizers

- **System R Optimizer**
  - bottom-up
  - dynamic programming (plan generation)
  - interesting orders
- **Starburst**
  - bottom-up
  - rule-based (plan transformation and generation)
  - e.g., IBM DB2
- **Cascades**
  - top-down
  - rule-based (plan transformation)
  - e.g., Microsoft SQL Server

Example Query Optimizers

- Starting point of optimization process is one way to classify query optimizers
- **Bottom-up:** start with individual relations of query
  - global cost of a plan **must** be computed bottom-up as the cost of each operator depends on the cost of its input(s)
  - dynamic programming **requires** breadth-first enumeration to pick the best plan
  - impossible to pick best plan until its cost has been computed
- **Top-down:** start with entire expression of query
  - operators may **require** certain properties (e.g., order or partitioning)
  - limit exploration based upon context of use
  - prune based on upper and lower bound

Based on: Guy M. Lohman: "The DB2 Universal Database Optimizer", 2007.

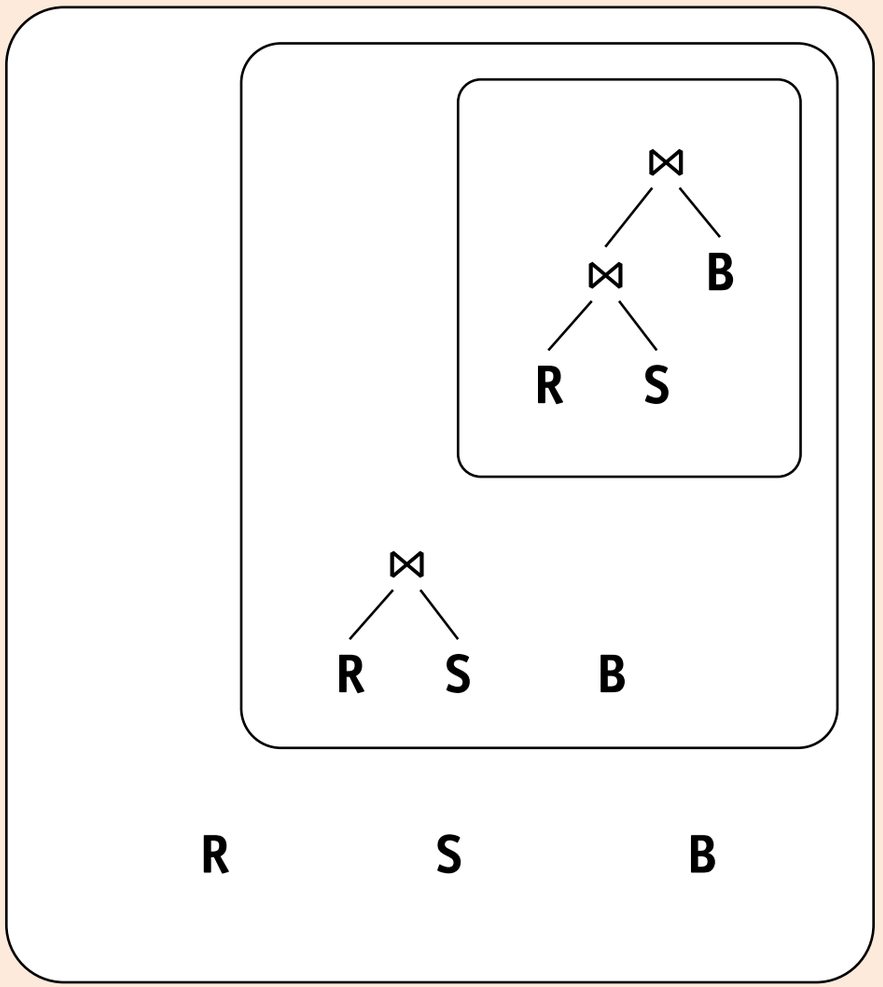
Example Query Optimizers

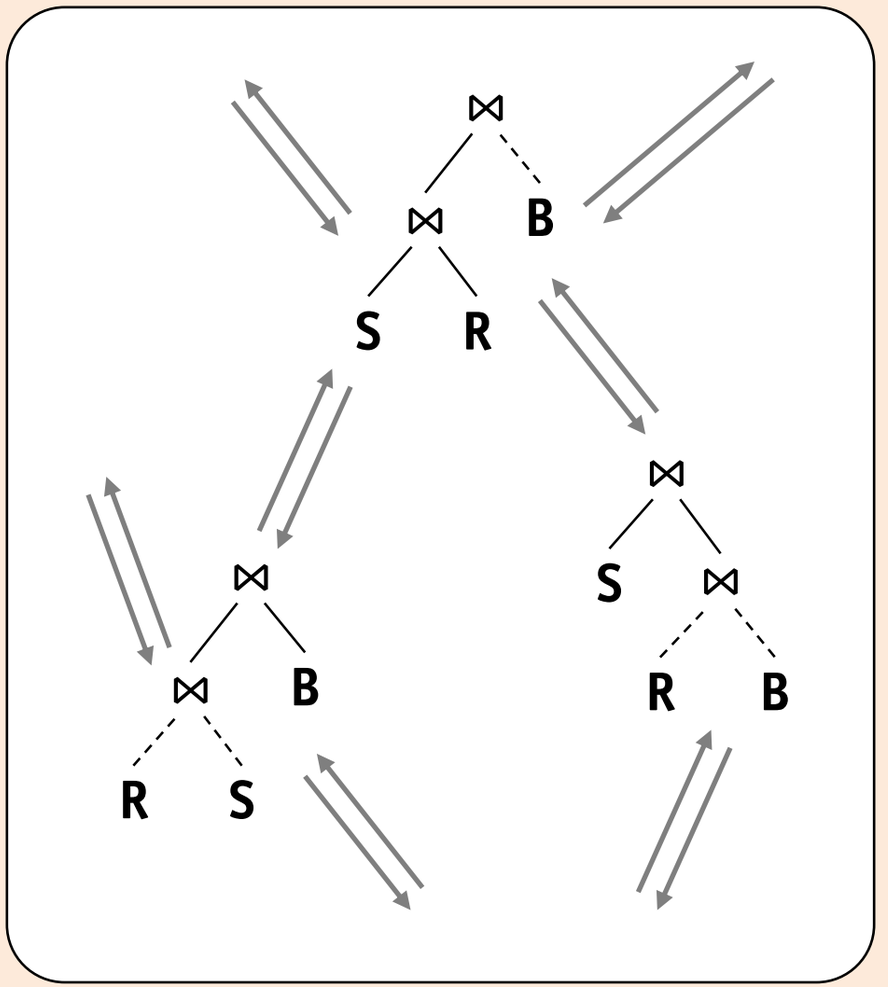
- Another way to classify query optimizers is what approach is used for plan enumeration
- **Generation**
  - enumerates different plans by assembling building blocks and adding one operator after another, until a complete plan has been produced
  - e.g., dynamic programming
- **Transformation**
  - enumerates different plans by transforming one plan into another equivalent plan
  - e.g., application of algebraic equivalences
- In the generation-based approach, plans can only be executed once all building blocks and operators have been assembled

See: Guido Moerkotte: "Building Query Compilers", 2009 (http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf).

Example Query Optimizers

Plan generation vs. transformation





See: Guido Moerkotte, "Building Query Compilers", 2009 (http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf).

System R Optimizer

- Many ideas pioneered by System R have already been discussed
- Summary of important design choices in optimizer of System R
  1. **Cost Model**: account for both I/O cost and CPU cost
  2. **Use of Statistics**: estimate cost of a query evaluation plan based on statistics about database instance
  3. **Select-Project-Join Queries**: focus on query blocks without nesting, process nested queries in a relatively ad hoc way
  4. **Delay Duplicate Elimination**: only perform duplicate elimination for projections as a final step, if required by a `DISTINCT` clause
  5. **Left-Deep Plans**: to reduce number of alternative plans, consider only plans with binary joins, where inner relation is a base relation
  6. **Dynamic Programming**: build plans bottom up and prune search space after every step
  7. **Interesting Orders**: keep sub-plans that produce tuples in an order required by the query or other operators

Starburst

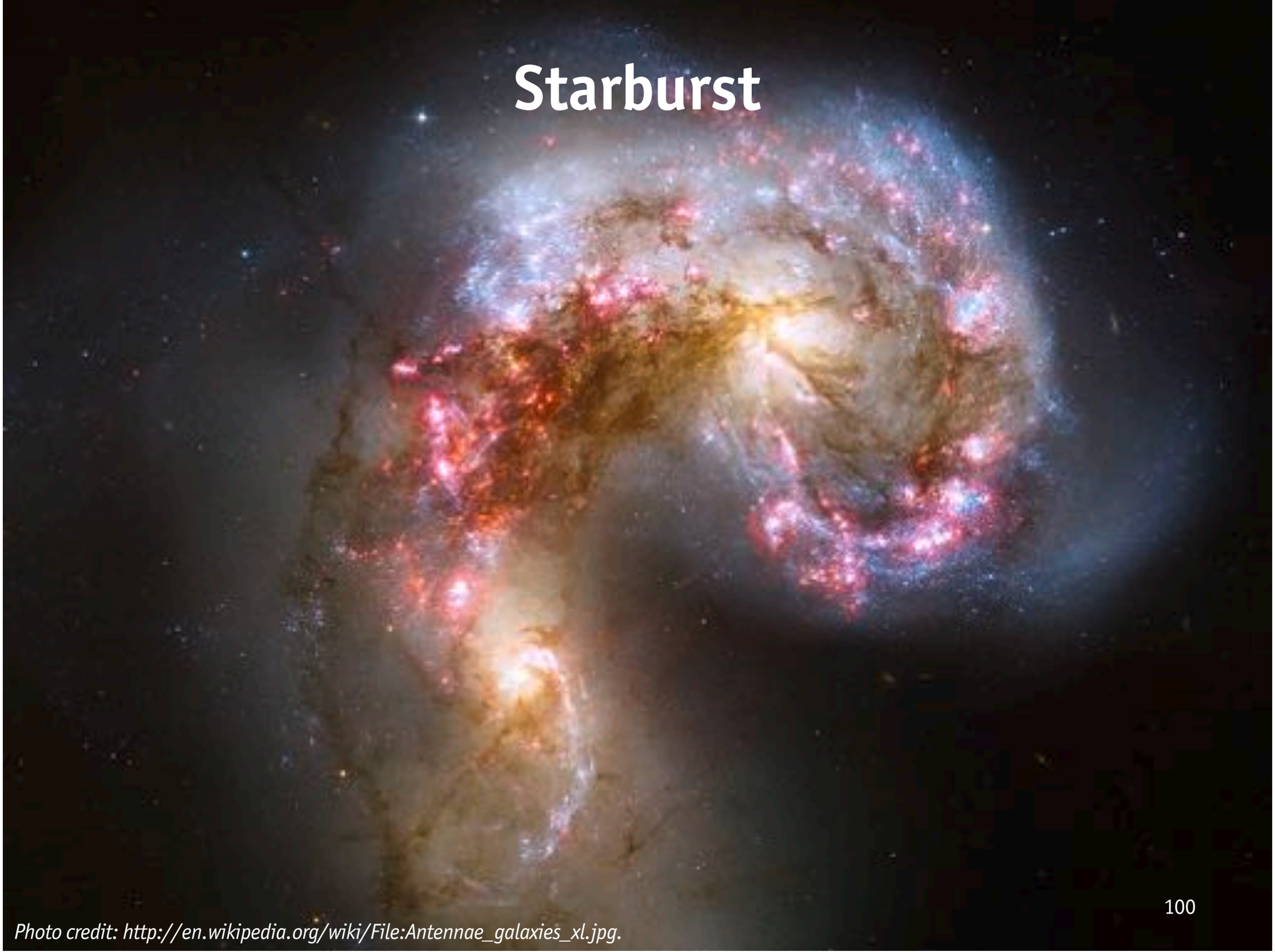


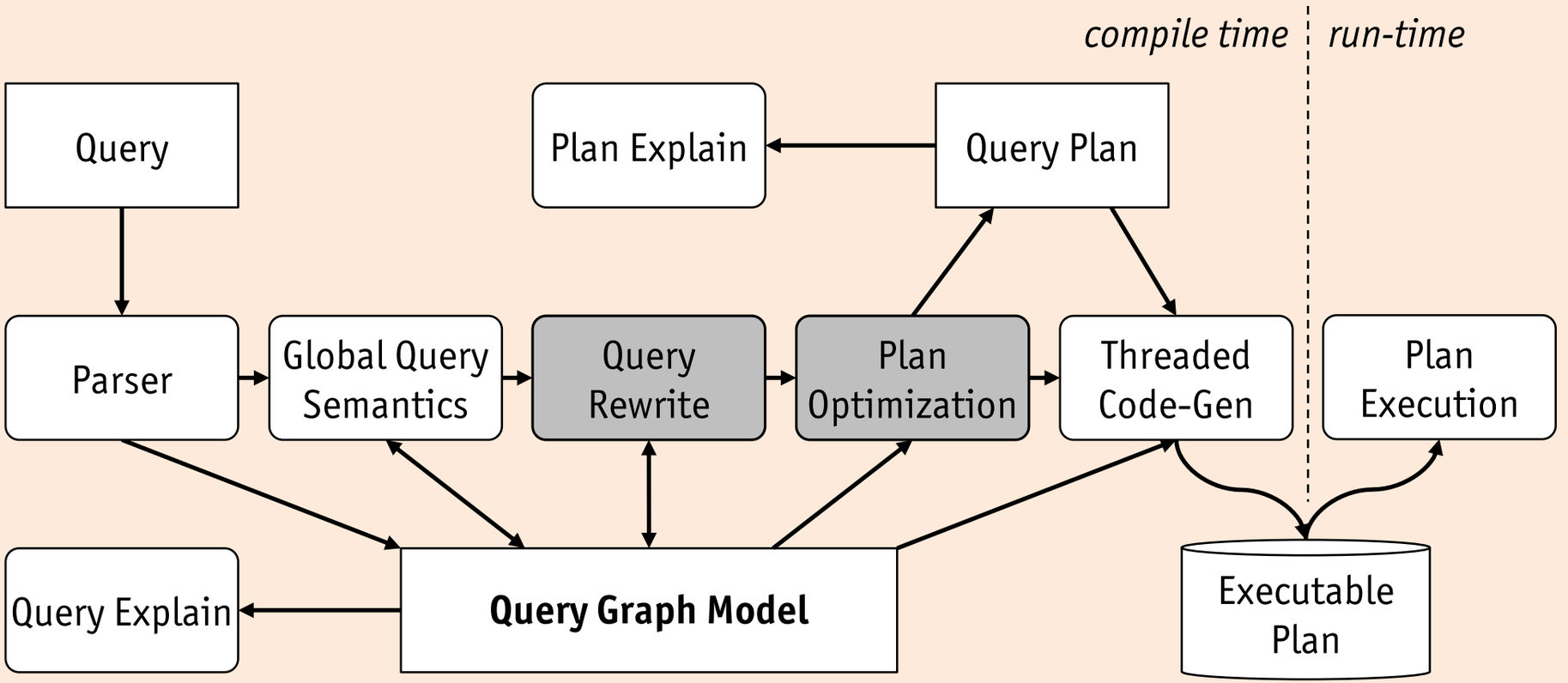
Photo credit: http://en.wikipedia.org/wiki/File:Antennae_galaxies_xl.jpg.

Starburst

- Starburst optimizer extends System R approach to support greater extensibility
- Queries are represented using the **query graph model** (QGM) throughout entire optimization lifecycle
- Query optimization has **two phases**
  1. **Query Rewrite** uses rules to transform a QGM representation of a query into another equivalent QGM representation
  2. **Plan Optimization** derives a query execution plan from QGM representation of a query bottom up using production rules
- First phase performs rewrite (heuristic) optimization, whereas second phase uses cost-based optimization

Starburst

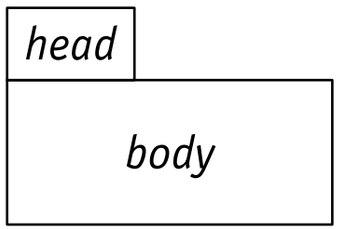
Query compiler overview



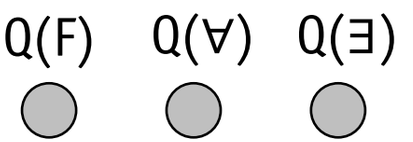
See: Laura M. Haas et al., "Extensible Query Processing in Starburst", In Proc. SIGMOD, 1989.

Starburst

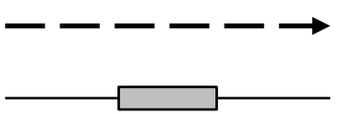
- Query Graph Model (QGM)
  - captures entire semantics of query to be compiled
  - used in all phases of query processing
- Boxes represent operations
  - **head** describes schema of output table
  - **body** contains a graphical representation of operation
  - select, insert, update, delete, union, and intersection



- Vertices represent iterators
  - **set formers** (F) contribute to result generation
  - **quantifiers** (∀ and ∃) are used to restrict a result



- Edges
  - **range edges** connect vertices to table or other operations
  - **qualifier edges** represent conjuncts of a predicate

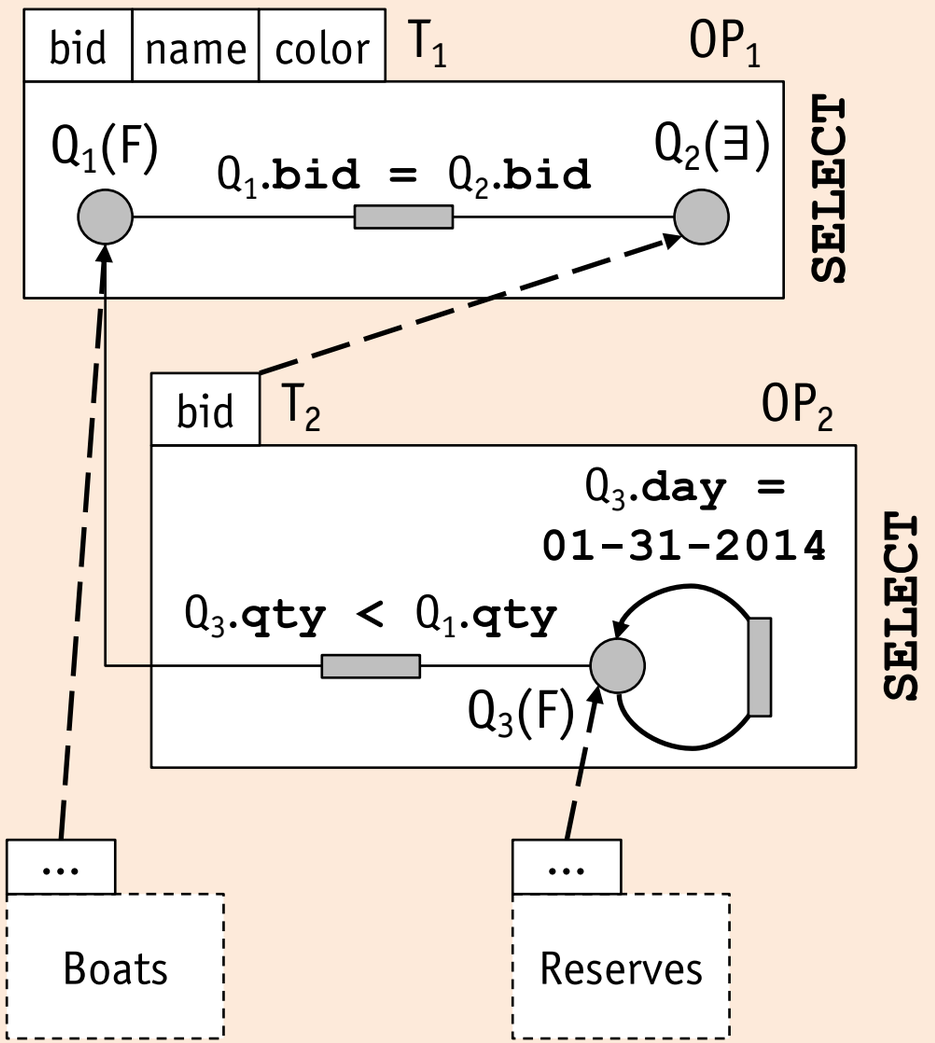


See: Laura M. Haas et al., "Extensible Query Processing in Starburst", In Proc. SIGMOD, 1989.

Starburst

Example: query graph model

```
SELECT bid, name, color
  FROM Boats B
 WHERE B.bid IN
  (SELECT R.bid
     FROM Reserves R
    WHERE R.qty < B.qty
      AND R.day = 01-31-2014)
```



See: Laura M. Haas et al., "Extensible Query Processing in Starburst", In Proc. SIGMOD, 1989.

Starburst

- Query rewrite phase
  - uses a rule-based engine to **transform** a QGM into a more efficient QGM
  - terminates when no rules are eligible or (time) budget is exceeded
- Transformation rules
  - consist of a **condition** and an **action**, both implemented as a C function
  - **access** and **manipulate** QGM representation of query directly
  - rules can be grouped into classes, e.g., for **conflict resolution**
- Classes of rules
  - **predicate migration** pushes predicates into lower level operations
  - **projection push-down** avoids retrieval of unused table and view columns
  - **operation merging** combines two QGM operations into a single one
- Starburst avoids need for a general rule interpreter with pattern matching by using C as a rule language

Starburst

Example: query rewrite phase

Rule 1 (Subquery to join)

IF $OP_1.type$ = **SELECT** $\wedge$ $Q_2.type = \exists \wedge$ (*at each evaluation of the existential predicate at most one tuple of* T_2 *satisfies the predicate)*

THEN

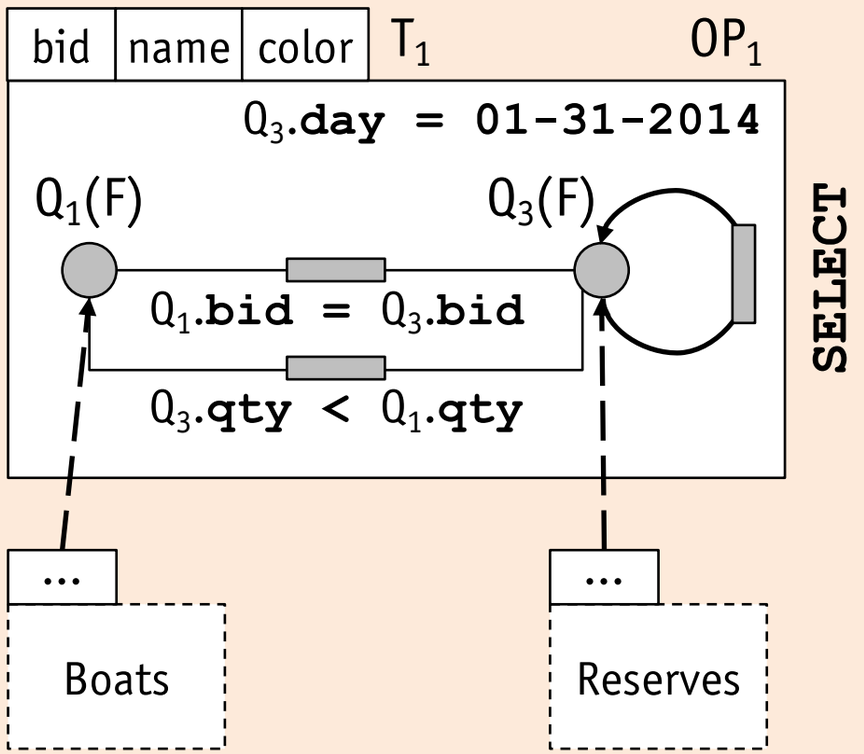
Q_2.type := F ;

Rule 2 (Operation merging)

IF $OP_1.type$ = **SELECT** $\wedge$
$OP_2.type$ = **SELECT** $\wedge$ $Q_2.type$ = F $\wedge$
(**NOT**($T_1.distinct$ = **FALSE** $\wedge$
$OP_2.dupelim$ = **TRUE**))

THEN

merge OP_2 into OP_2 ;
IF $OP_2.dupelim$ = **TRUE**
THEN $OP_1.dupelim$:= **TRUE** ;



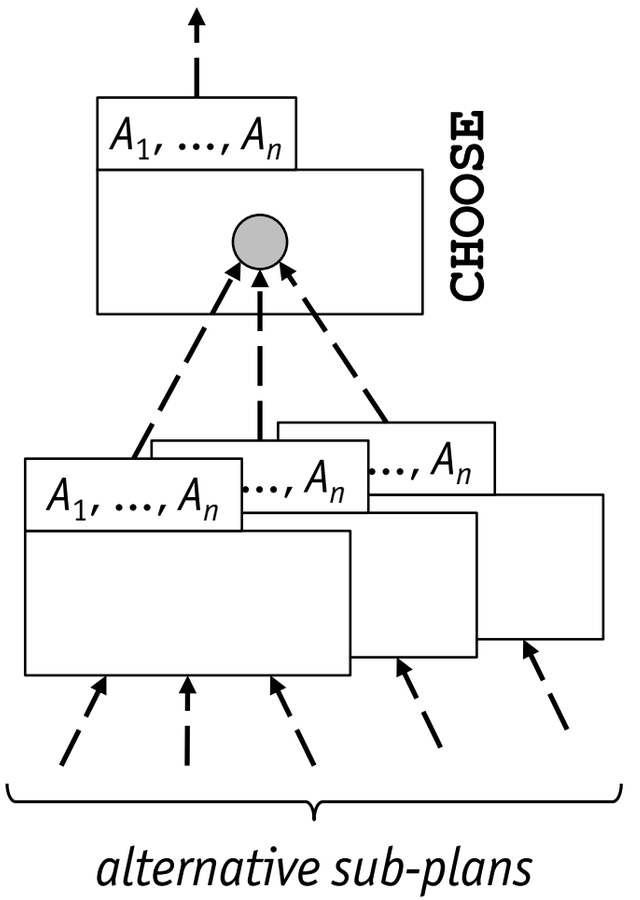
See: Laura M. Haas et al., "Extensible Query Processing in Starburst", In Proc. SIGMOD, 1989.

Starburst

- Rule engine applies rules using **forward chaining** method
  - start with available data, i.e., QGM sub-graphs
  - apply transformation rules until goal is reached
- Several **control strategies** are available to manage rules
  - **sequential:** process all rules sequentially
  - **priority:** give a chance to rules with a higher priority first
  - **statistical:** next rule is chosen randomly based on a user-defined probability distribution
- Search strategy
  - identifies QGM **sub-graphs** to which rules can be applied
  - supports **depth-first** (top down) and **breadth-first** search

Starburst

- Cost estimates are **not known** during query rewrite phase
  - both phases are executes strictly sequentially
  - without cost estimates plans cannot be pruned
- New QGM operator **CHOOSE**
  - links together alternative equivalent plans
  - cost-based optimization selects best plan later
- Number of alternatives may be very large
  - predicate migration
  - view merging vs. view materialization
- However, selection of best plan may even be deferred until query execution time



Starburst

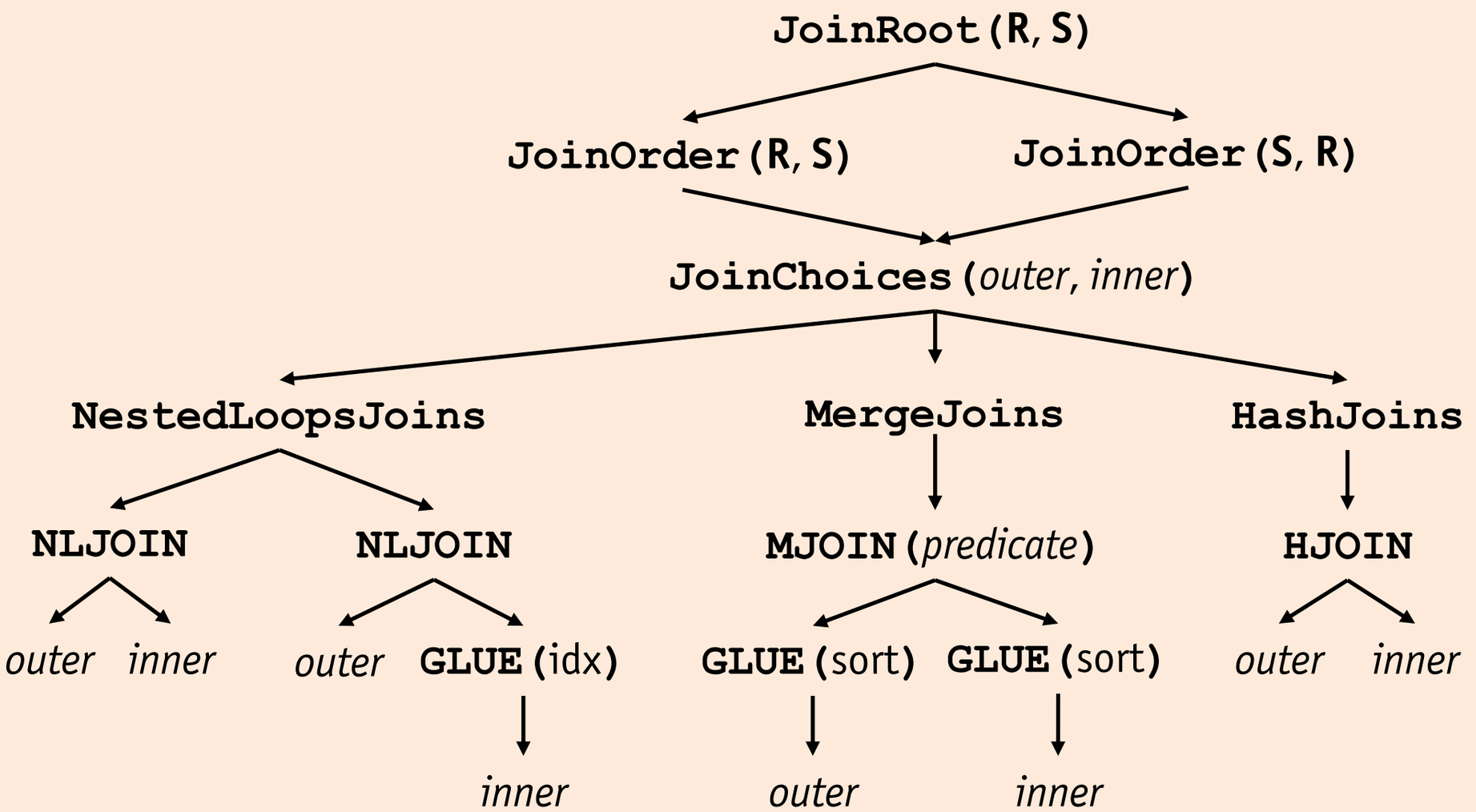
- Plan optimization phase
  - rule-driven generator derives query execution plans bottom-up
  - plan has a **relational** description, estimated **cost**, and **physical** properties
  - prunes plans with same logical and physical properties but higher cost
- Plan optimizer is a **select-project-join** optimizer
  - join enumerator is similar to enumeration scheme of System R
  - plan generator (*see next slide*)
- Join enumerator
  - uses feasibility criteria (mandatory and heuristic) to limit number of joins
  - enumeration algorithm can be replaced due to modular design

Starburst

- Plan generator is rule-based
- Strategy alternative rule (STAR)
  - set of parameterized rules is similar to a language grammar
  - define higher-level **non-terminal** from low-level **terminal** symbols
  - priority queue controls order in which STARs are evaluated
- Low-level plan operator (LOLEPOP)
  - terminal symbol of grammar language
  - concrete relational operators and physical operators (fetch, scan, etc.)
- Glue STAR
  - enforce required physical properties, e.g., sort order for merge join
  - may add LOLEPOP to make existing plan meet requirements

Starburst

Example: generation of join alternatives using STARs



See: Guy M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives", In Proc. SIGMOD, 1988.

Cascades

Photo credit: http://commons.wikimedia.org/wiki/File:Cascades_Mountains_Range.jpg.

Cascades

- Transformation-based, top-down approach
  - **depth-first search:** beginning with original query, consider subqueries, and optimize them
  - **goal-driven:** no need to maintain bottom-up interesting orders
- Fully cost-based
  - no separation into phases, i.e., heuristic and cost-based optimization
- Flexible and extensible concepts
  - **operators:** expressions consisting of logical, physical, and item operators are used to represent query trees and execution plans
  - **rules:** plan search space is defined by a set of transformation, implementation, and enforcer rules
  - **strategy:** search space exploration strategy is guided by sequence of optimization tasks

Cascades

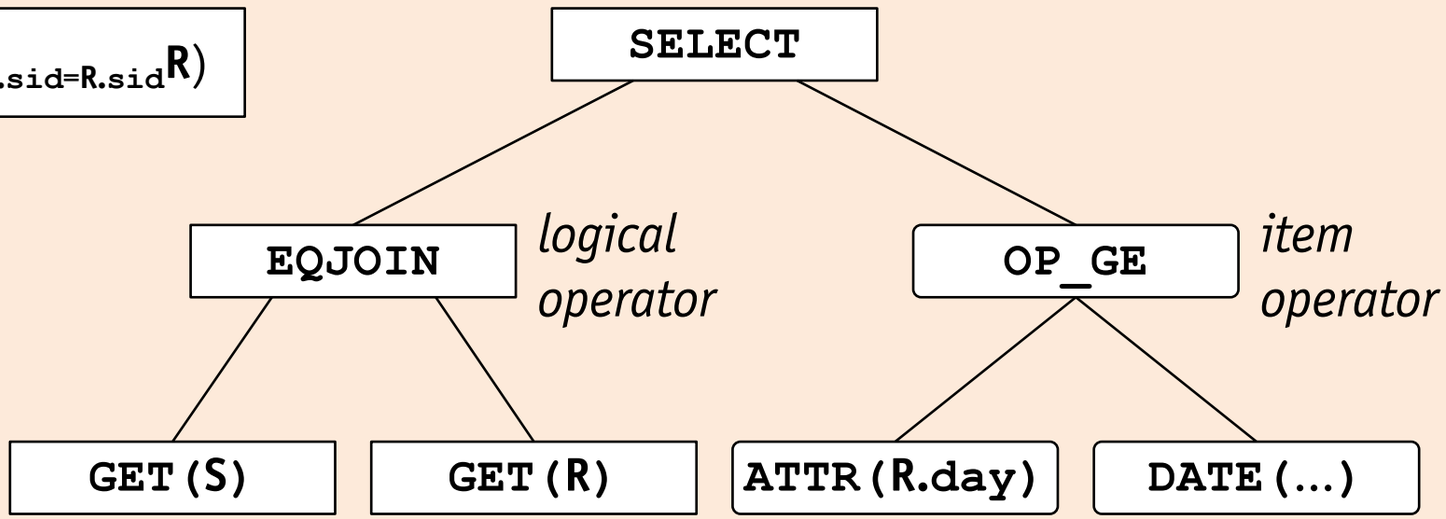
- Query trees and execution plans are represented as **expressions**
- Logical operators
  - matching rule patterns to expressions
  - hashing for detecting duplicate expressions
  - deriving logical properties (e.g., schema) from input
- Physical operators
  - computing cost from operator algorithm and input cost
  - checking cost limit between optimization of two inputs
  - translating optimization goals to input operators
- Item operators
  - represent selection predicates for easy manipulation by rules

Cascades

Example: query expression tree

```
SELECT *
  FROM Sailors S, Reserves R
 WHERE S.sid = R.sid
   AND R.day >= 01-01-2014
```

$\sigma_{R.day \geq 01\text{-}01\text{-}2014}(S \bowtie_{S.sid=R.sid} R)$



Cascades

- A **group** is a set of equivalent logical and physical expressions
  - records **logical properties** of all expressions
  - maintains **lower bound**, such that every expression in group has a cost greater than this lower bound
  - current best plan, i.e., **winner**, is stored for each group

Example: expression group

[ABC]	lower bound: 927

Logical Expressions

$(A \bowtie B) \bowtie C$, $(B \bowtie C) \bowtie A$, $(A \bowtie C) \bowtie B$, $A \bowtie (B \bowtie C)$, $C \bowtie (A \bowtie B)$, $B \bowtie (A \bowtie C)$,
$(B \bowtie A) \bowtie C$, $(C \bowtie B) \bowtie A$, $(C \bowtie A) \bowtie B$, $A \bowtie (C \bowtie B)$, $C \bowtie (B \bowtie A)$, $B \bowtie (C \bowtie A)$

Physical Expressions

$(A_F \bowtie_{NL} B_F) \bowtie_{NL} C_F$, $(A_F \bowtie_{M} B_F) \bowtie_{M} C_F$, $(B_F \bowtie_{NL} C_F) \bowtie_{NL} A_F$, $(A_F \bowtie_{NL} C_F) \bowtie_{NL} B_F$, ...

Winner

$$(B_F \bowtie_{M} C_F) \bowtie_{INL} A_{IDX}$$

Cascades

- Use of **multi-expressions** saves space required for a group
  - consist of logical and physical operators, but take groups as input
  - reduce number of expressions by representing all equivalent expressions

Example: multi-expression group

[ABC]	lower bound: 927

Logical Expressions

$[AB] \bowtie [C],\ [BC] \bowtie [A],\ [AC] \bowtie [B],\ [A] \bowtie [BC],\ [C] \bowtie [AB],\ [B] \bowtie [AC]$

Physical Expressions

$[AB] \bowtie_{NL} [C],\ [AB] \bowtie_{M} [C],\ [BC] \bowtie_{NL} [A],\ [AC] \bowtie_{NL} [B],\ \ldots$

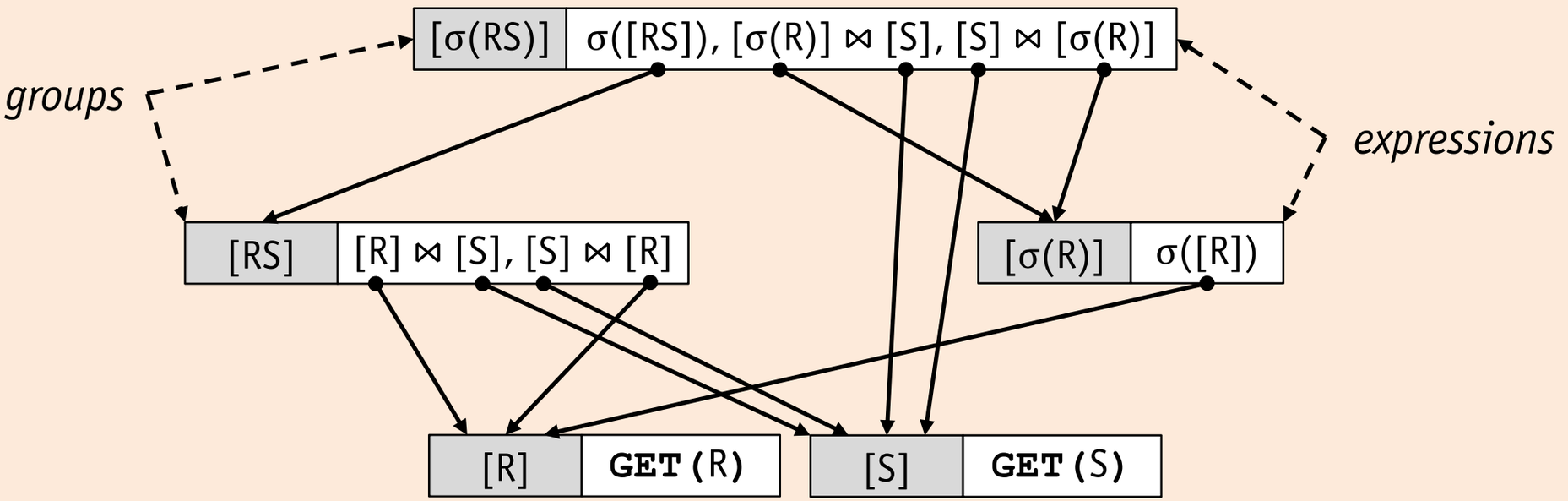
Winner

$(B_F \bowtie_M C_F) \bowtie_{INL} A_{IDX}$

Cascades

Example: query expression tree

```
SELECT *
  FROM Sailors S, Reserves R
 WHERE S.sid = R.sid
   AND R.day >= 01-01-2014
```



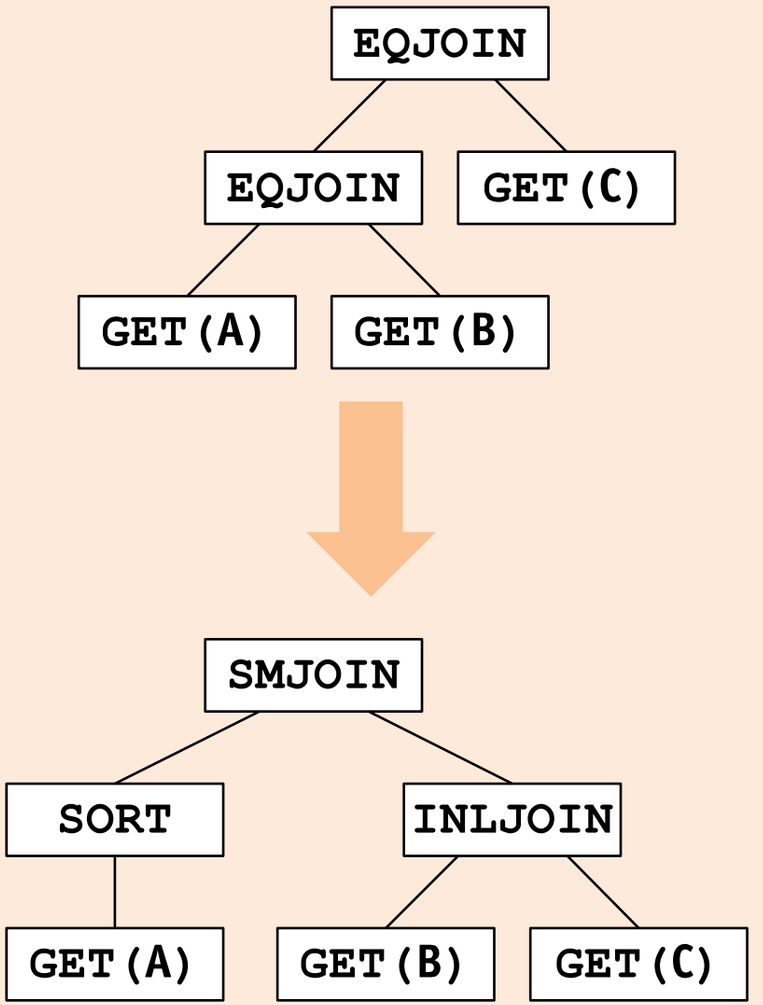
Cascades

- Possible search space is defined by **rules**
  - rules transform an expression into a **logically equivalent** expression
  - **pattern** defines structure of logical expression to which applies
  - **condition** checks whether rule can be applied
  - **substitute** defines structure of result after rule application
- Cascades distinguishes three types of rules
  - **transformation rule:** substitute is another logical expression
  - **implementation rule:** substitute is a physical expression
  - **enforcer rule:** substitute contains an enforcer, i.e., an operator that guarantees certain physical properties, e.g., sort
- Rules can have a **promise** value that guides exploration order
  - rules that require a specific physical property typically have promise 0
  - transformation rules typically have promise 1
  - implementation rules typically have promise 2

Cascades

Example: rules

- **Transformation rules**
  - EQJOIN_LTOR
    Pattern: $(Leaf(X) \bowtie Leaf(Y)) \bowtie Leaf(Z)$
    Substitute: $Leaf(X) \bowtie (Leaf(Y) \bowtie Leaf(Z))$
- **Implementation rules**
  - EQ_TO_SM
    Pattern: $Leaf(X) \bowtie Leaf(Y)$
    Substitute: $Leaf(X) \bowtie_{SM} Leaf(Y)$
  - EQ_TO_INL
    Pattern: $Leaf(X) \bowtie Leaf(Y)$
    Condition: ∃ index on join attribute
    Substitute: $Leaf(X) \bowtie_{INL} Leaf(Y)$
- **Enforcer rules**
  - SORT
    Pattern: $Leaf(X)$
    Substitute: **SORT** $Leaf(X)$

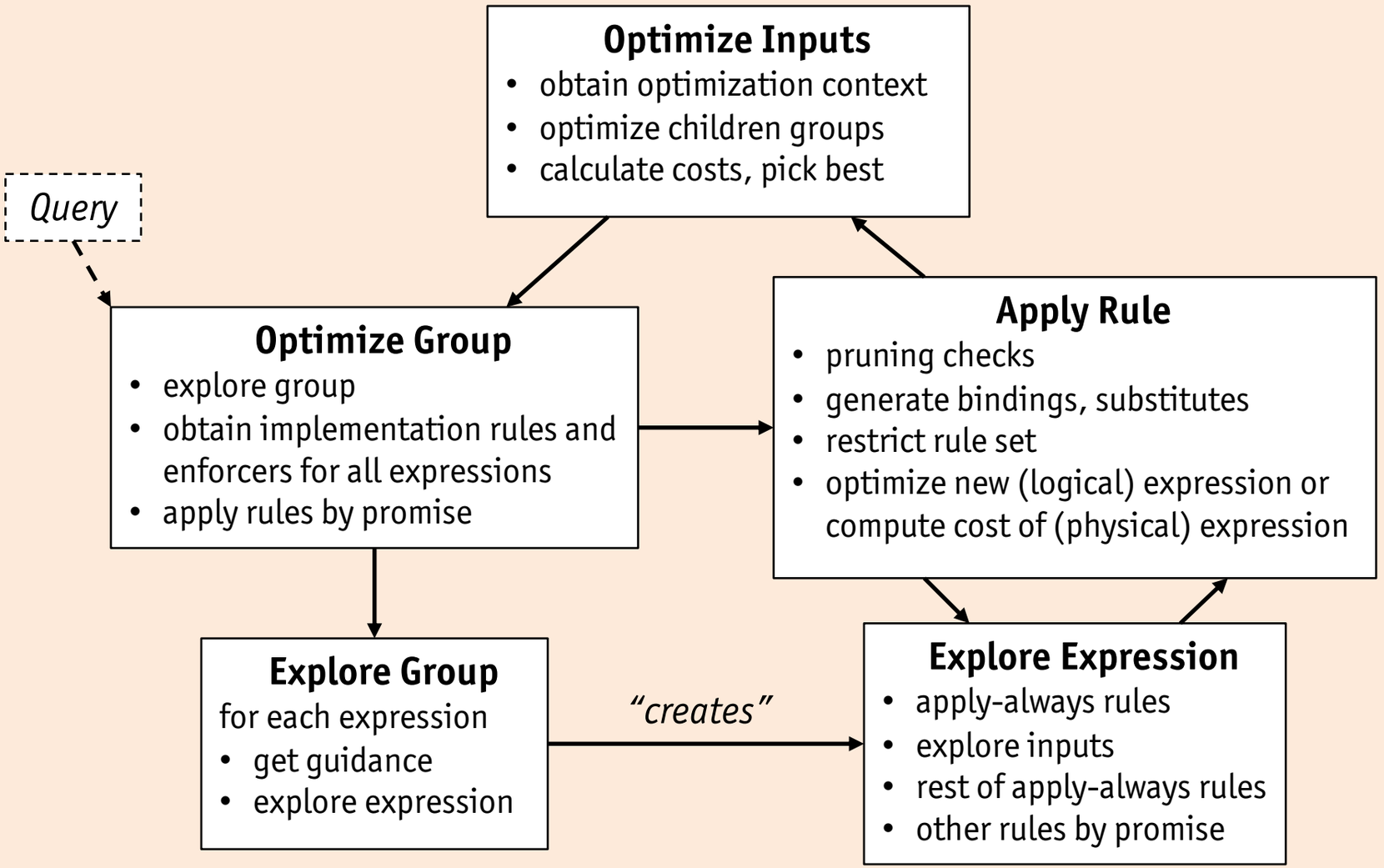


Cascades

- Optimization algorithm is broken down into **tasks**
  - original task is to optimize entire query
  - tasks create and schedule each other
  - when no undone tasks remain, optimization terminates
- Tasks are managed in a **task structure**
  - last-in-first-out stack, dependency graph, etc.
  - tasks can be reordered for heuristic guidance
- An **optimization goal** combines a group or expression and a cost limit with required and excluded physical properties
  - pursuing an optimization goal results either in a plan or in failure
  - dynamic programming and memoization are used to ensure that the same optimization goal is not pursued multiple times

Cascades

Optimization tasks



Cascades

Main optimization loop

```
function optimize(query)

   Stack<OptTask> tasks = new Stack<OptTask>();

   tasks.push(new OptGroup(query));          (start optimization at top group)

   while tasks is not empty do               (perform undone tasks until there are no more left)
      task ← tasks.pop();                                    (get next task)
      task.perform();                                        (perform next task)

end
```

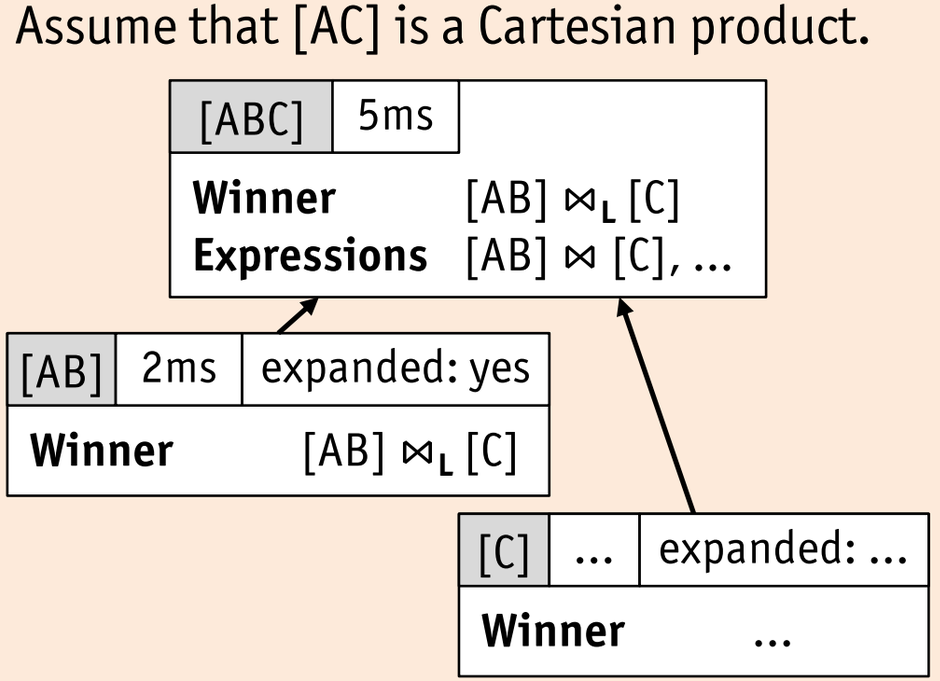

See: Yongwen Xu, "Efficiency in the Columbia Database Query Optimizer", Portland State University, 1998.

Cascades

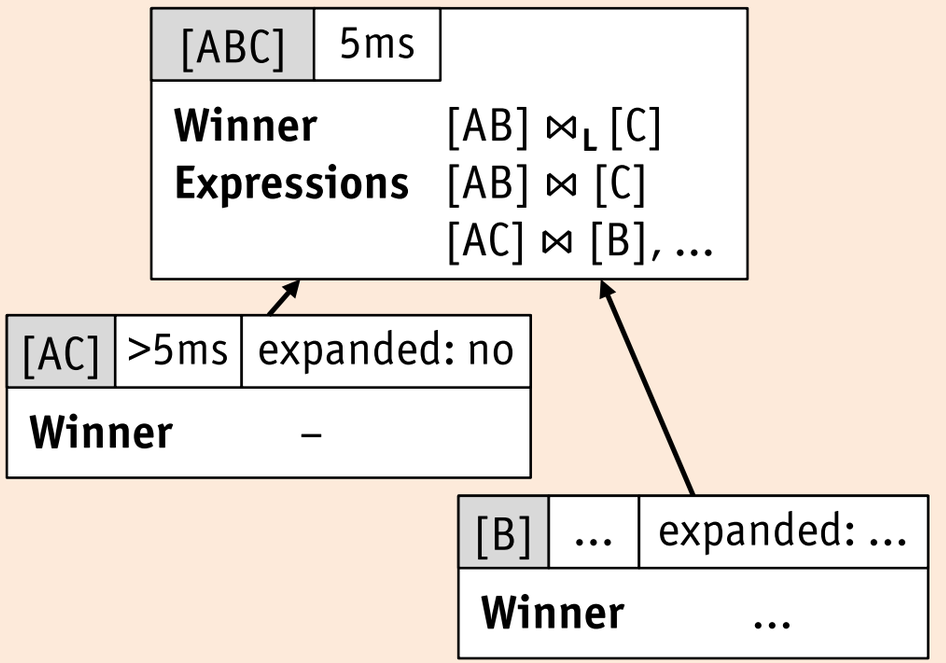
- Plan **pruning** is performed in the optimize input task
  - each time an input has been optimized, lowest cost so far is recorded
  - this cost serves as limit for optimizing next inputs

Example: pruning

Assume that [AC] is a Cartesian product.



After optimizing [AB] ⋈ [C]



After optimizing [AC] ⋈ [B]

Starburst vs. Cascades

- Optimization phases
  - Starburst optimizes queries in **two phases**, a (heuristic) query rewrite phase and a (cost-based) plan optimization phase
  - Cascades only uses **one phase** since all transformations are algebraic and cost-based
- Mapping logical to physical operators
  - Starburst expands expressions **step-by-step** as grammar-like set of plan production rules implies hierarchy of intermediate levels (non-terminals)
  - Cascades accomplishes this mapping in **one single step**
- Rule application
  - Starburst applies rules by **forward chaining** in query rewrite phase
  - Cascades performs **goal-driven** application of rules

Nested Subqueries

- Recall, **unit of optimization** in typical systems is a query block
  - a query with nested subqueries consists of **multiple** query blocks
  - focus on optimization of nested **subqueries in the `WHERE` clause**
- Optimizer has several options to deal with such queries
  - evaluate subquery **once**, reuse result for all tuples of top-level query
  - evaluate subquery **multiple times**, once for every tuple of top-level query
  - try to **rewrite** query to an equivalent query without nesting
- Available optimization options depend on
  - form of **`WHERE` predicate**
  - presence or absence of **aggregates** in subquery
  - whether subquery is **uncorrelated** or **correlated**
  - other characteristics

Nested Subqueries

Canonical form

A SQL query is in **canonical form** if its `FROM` clause only contains table names and its `WHERE` clause only contains simple and join predicates.

Uncorrelated and correlated subqueries

A nested subquery is said to **correlated** if it references a table or tuple variable of the top-level query. Otherwise, the nested subquery is said to be **uncorrelated**.

Uncorrelated subquery

```
SELECT *
  FROM A
 WHERE A.a IN
       (SELECT B.a
          FROM B
         WHERE B.b = 1)
```

top-level or outer query

nested or inner query

Correlated subquery

```
SELECT *
  FROM A
 WHERE A.a IN
          (SELECT B.a
             FROM B
            WHERE B.b = A.b)
```

correlation

Nested Subqueries

- Types of nesting
  - **Type A:** uncorrelated, aggregated subquery
  - **Type N:** uncorrelated, not aggregated subquery
  - **Type J:** correlated, not aggregated subquery
  - **Type JA:** correlated, aggregated subquery
  - **Type D:** correlated, division subquery
- Due to the removal of the **`CONTAINS`** keyword in SQL-2 and higher, Type D nesting is no longer possible

See: Won Kim: "On Optimizing an SQL-like Nested Query", In *ACM TOIS 7(3):443-469, 1982.*

Nested Subqueries

Extended example database schema

```
CREATE TABLE Sailors (
  sid      INTEGER,
  sname    STRING,
  rating   INTEGER,
  city     STRING,
  PRIMARY KEY (sid)
)

CREATE Table Reserves (
  sid      INTEGER,
  bid      INTEGER,
  qty      INTEGER,
  day      DATE,
  harbor   STRING,
  PRIMARY KEY (sid, bid),
  FOREIGN KEY sid REFERENCES Sailors(sid),
  FOREIGN KEY bid REFERENCES Boats(bis)
)
```

```
CREATE TABLE Boats (
  bid      INTEGER,
  bname    STRING,
  color    STRING,
  qty      INTEGER,
  harbor   STRING,
  PRIMARY KEY (bid)
)
```

Nested Subqueries

Example: Type A nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE S.rating = (SELECT MAX(S2.rating)
                     FROM Sailors S2)
```

- Using **tuple iteration semantics** to evaluate this simple query is very inefficient
  - nested subquery is evaluated for **every tuple** of the top-level query
  - result of nested subquery is **constant**
- Better evaluation strategy
  - evaluate nested subquery just **once**, yielding a single value
  - rewrite top-level query to incorporate this value into it **`WHERE`** clause, e.g., **`S.rating = 8`**

Nested Subqueries

Example: Type N nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE S.sid IN (SELECT R.sid
                   FROM Reserves R
                  WHERE R.bid = 103)
```

- Evaluation strategy
  - again, evaluate nested subquery just **once**, yielding a collection of **`sid`**s
  - for each tuple of the top-level query, check if its **`sid`** value is in computed collection of **`sid`**s, i.e., a join of ***Sailors*** and the computed collection
  - in principle, the full range of join methods is available, e.g., assuming an index on ***Sailors*.`sid`**, an index nested loops join could be used
- Typically, optimizers **cannot** find such strategies and revert to a nested loops join with computed collection as inner relation

Nested Subqueries

Example: Type J nesting

```
SELECT S.sname
  FROM Sailors S
 WHERE EXISTS (SELECT *
                 FROM Reserves R
                WHERE R.bid = 103 AND
                      S.sid = R.sid)
```

Example: Type JA nesting

```
SELECT B.bname
  FROM Boats B
 WHERE B.bid = (SELECT MAX(R.bid)
                  FROM Reserves R
                 WHERE R.harbor = B.harbor)
```

Nested Subqueries

- Correlated subqueries cannot be evaluated just once
  - typical nested subquery is evaluated for each tuple of top-level query
  - this evaluation strategy is very inefficient
- Optimizer is **likely** to do a poor job using this limited approach to nested query optimization
  - if the same value appears in the correlation field of several tuples of the top-level query, the **same subquery** is evaluated multiple times
  - approach is **not set-oriented**, i.e., join method is effectively index nested loops (with evaluation of nested query as "index access") and other join methods (sort-merge join, hash join) cannot be considered
  - even if index nested loops join is appropriate, optimizer **cannot leverage indexes** on relations of nested queries as there is no real join predicate
  - nesting imposes an **implicit ordering on query evaluation**, which leads to missed opportunities for finding a good evaluation plan

Nested Subqueries

- Try to rewrite queries with nested subqueries to **canonical form** in order to open up more optimization opportunities
  - instead of computed collections **use database tables directly**, if possible
  - otherwise, create **appropriate temporary tables** from nested subqueries
- Based on types of nested subqueries, two algorithms have been defined that follow this idea
  - algorithm **NEST-N-J:** transforms nested subqueries of Type N and J to a query in canonical form
  - algorithm **NEST-JA:** uses temporary tables to remove one level of nesting for subqueries of Type JA
- Subqueries of Type A are still dealt with as described before

See: Won Kim: "On Optimizing an SQL-like Nested Query", In *ACM TOIS 7(3):443-469, 1982.*

Nested Subqueries

- Algorithm **NEST-N-J** for Type N or Type J nested subqueries
  1. combine **`FROM`** clauses of all query blocks into one **`FROM`** clause
  2. combine **`WHERE`** clauses of all query blocks using **`AND`**, replacing element test (**`IN`**) with equality (**`=`**)
  3. retain **`SELECT`** clause of the outermost query block
- Resulting canonical query is **equivalent** to original nested query

Algorithm NEST-N-J

```
SELECT Ri.Ck
  FROM Ri
 WHERE Ri.Cm IN
       (SELECT Rj.Cn
          FROM Rj
         WHERE Ri.Cx = Rj.Cy)
```



```
SELECT DISTINCT Ri.Ck
  FROM Ri, Rj
 WHERE Ri.Cm = Rj.Cn AND
       Ri.Cx = Rj.Cy
```


See: Won Kim: "On Optimizing an SQL-like Nested Query", In ACM TOIS 7(3):443-469, 1982.

Nested Query

Rewriting Type N and Type J nested subqueries

Rewrite the following query with nested subqueries to a canonical query using algorithm NEST-N-J.

```
SELECT S.sname
  FROM Sailors S
 WHERE S.sid IN (
   SELECT R.sid
     FROM Reserves R
    WHERE S.city = R.harbor
      AND R.bid IN (
      SELECT B.bid
        FROM Boats B
        WHERE B.bname = "Laser"
      )
   )
```

Nested Subqueries

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Algorithm **NEST-JA** for Type JA nested subqueries
  1. create a temporary relation $R_{tmp}(C_1, \ldots, C_n, C_{n+1})$ from relation R_2, such that $R_{tmp}.C_{n+1}$ is the result of applying aggregate function **AGG** on the C_n columns of R_2 that have matching values for $C_1, \ldots, C_n$ in R_1
  2. transform inner query block of the original query by changing all references to R_2 columns in join predicates that also reference R_1 to corresponding R_{tmp} columns
- Result of algorithm NEST-JA is a Type J nested query that can be transformed to its canonical equivalent by algorithm NEST-N-J

Nested Subqueries

Algorithm NEST-JA (Step 1)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT AGG(R2.Cn+1)
                    FROM R2
                   WHERE R1.C1 = R2.C1
                     AND R1.C2 = R2.C2
                         ...
                     AND R1.Cn = R2.Cn)
```



```
CREATE TABLE Rtmp(C1, ..., Cn, Cn+1) AS
  SELECT C1, ..., Cn, AGG(Cn+1)
    FROM R2
   GROUP BY C1, ..., Cn
```

Nested Subqueries

Algorithm NEST-JA (Step 2)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT Rtmp.Cn+1
                    FROM Rtmp
                   WHERE R1.C1 = Rtmp.C1
                     AND R1.C2 = Rtmp.C2
                     ...
                     AND R1.Cn = Rtmp.Cn)
```

The Real World

Many real-world RDBMS have functionality to support **temporary tables.**

- **Microsoft SQL Server**
  - **local** temporary tables (name prefixed with **#**) are only visible in current session
  - **global** temporary tables (name prefixed with **##**) are visible to all sessions
  - temporary tables are **automatically dropped** when they go out of scope

Nested Query

Rewriting Type JA nested subqueries

Rewrite the following query with a nested subquery to a canonical query using algorithm NEST-JA.

```
SELECT B.bname
  FROM Boats B
 WHERE B.bid = (SELECT MAX(R.sid)
                  FROM Reserves R
                 WHERE R.harbor = B.harbor)
```

Nested Subqueries

- Algorithm NEST-JA may give **incorrect results** for nested subqueries that contain the **COUNT** aggregation function

Example: Problem with COUNT

What is the result of the following query?

```
SELECT B.bid
  FROM Boats B
 WHERE B.qty =
  (SELECT COUNT(R.day)
     FROM Reserves R
    WHERE B.bid = R.bid AND
          R.day <= 31-12-2010)
```

Result

bid
10
8

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
10	Finn	Konstanz	1
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

See: Werner Kiessling: "SQL-like and Quel-like Correlation Queries with Aggregates Revisited", UCB/ERL Memo 84/75, UC Berkeley, 1984.

Nested Subqueries

Example: Problem with COUNT (cont'd)

Step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP1 AS
  SELECT bid,
         COUNT(day) AS numday
    FROM Reserves
   WHERE day <= 31-12-2010
   GROUP BY bid
```

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

Step 2 of algorithm NEST-JA rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B
 WHERE B.qty =
  (SELECT T.numday
     FROM TMP1 T
    WHERE B.bid = T.bid)
```

TMP1

bid	numday
3	2
10	1

Result

bid
10

Nested Subqueries

- For nested subqueries using the **`COUNT`** aggregation function, algorithm NEST-JA needs to include an **outer join** in step 1
- Full outer join **A** ⟗ **B** retains all records from **A** and **B**, even if there is no match, and inserts **`NULL`** values where necessary

Full outer join (⟗)

bid	bname
3	Laser
10	Finn
8	49erFX
9	Narca 17

⟗

bid	qty	day
3	4	07-03-2009
5	1	06-22-2008
10	2	08-10-2011
8	5	05-07-2013

=

bid	bname	qty	day
3	Laser	4	07-03-2009
5	**`NULL`**	1	06-22-2008
10	Finn	2	08-10-2011
8	49erFX	5	05-07-2013
9	Narca 17	**`NULL`**	**`NULL`**

See: Richard A. Ganski and Harry K. T. Wong: "Optimization of Nested SQL Queries Revisited", 1987.

Nested Subqueries

Using outer join to fix problem with COUNT

Adapted step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP2 AS
  SELECT B.bid, COUNT(R.day) AS numday
    FROM Boats B FULL OUTER JOIN Reserves R
      ON B.bid = R.rid
   WHERE R.day <= 31-12-2010
   GROUP BY B.bid
```

Adapted step 2 of algorithm NEST-JA rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B, TMP2 T
 WHERE B.qty = T.numday
   AND B.bid = T.bid
```

Nested Subqueries

Using outer join to fix problem with COUNT

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
10	Finn	Konstanz	1
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	10	2	08-10-2011
4	8	5	05-07-2013

Boats $\bowtie \sigma_{day \leq 31\text{-}12\text{-}2010}$(***Reserves***)

bid	bname	harbor	qty	sid	qty	day
3	Laser	Konstanz	6	2	4	07-03-2009
3	Laser	Konstanz	6	1	2	10-01-2008
10	Finn	Konstanz	1	1	1	06-08-2008
8	49erFX	Kreuzlingen	0	NULL	NULL	NULL

TMP2

bid	numday
3	2
10	1
8	0

Result

bid
10
8

Nested Subqueries

- Points to observe to obtain correct results when using outer join to deal with nested subqueries that contain **`COUNT`**
- Evaluation order
  - selection predicate, e.g., *day* ≤ 31-12-2010, has to be applied **before** outer join
  - if necessary split creation of temporary table into two steps to force evaluation order
- Subqueries with **`COUNT(*)`**:
  - such subqueries will always return a result > 0 because of the outer join
  - Change `*` to a column name from relation of the nested subquery, typically the join column
- Duplicates in the join column
  - *see next slide*

Nested Subqueries

Problem with duplicates in join column of outer join

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	6
3	Laser	Kreuzlingen	2
10	Finn	Konstanz	1
10	Finn	Kreuzlingen	0
8	49erFX	Kreuzlingen	0

Reserves

sid	bid	qty	day
2	3	4	08-14-2007
1	3	2	11-11-2008
1	10	1	06-22-2006

Boats $\bowtie \sigma_{day \leq 31\text{-}12\text{-}2010}$(***Reserves***)

bid	bname	harbor	qty	sid	qty	day
3	Laser	Konstanz	6	2	4	07-03-2009
3	Laser	Konstanz	6	1	2	10-01-2008
3	Laser	Kreuzlingen	2	2	4	07-03-2009
3	Laser	Kreuzlingen	2	1	2	10-01-2008
10	Finn	Konstanz	1	1	1	06-08-2008
10	Finn	Kreuzlingen	0	1	1	06-08-2008
8	49erFX	Kreuzlingen	0	NULL	NULL	NULL

TMP2

bid	numday
3	4
10	2
8	0

Result

bid
8

Actual Result

bid
3
10
8

Nested Subqueries

Problem with duplicates in join column of outer join

Solution

1. During the creation of the temporary table, remove duplicates **before** computing the outer join.

```
CREATE TABLE TMP3 AS
  SELECT DISTINCT bid
    FROM Boats
```

TMP3

bid
3
10
8

2. In any join required to build the temporary table, use this projection **instead** of the original outer relation

```
CREATE TABLE TMP4 AS
  SELECT T.bid, COUNT(R.day) AS numday
    FROM TMP3 T FULL OUTER JOIN Reserves R
      ON T.bid = R.rid
   WHERE R.day <= 31-12-2010
   GROUP BY T.bid
```

TMP4

bid	numday
3	2
10	1
8	0

Nested Subqueries

- Another problem arises with algorithm NEST-JA, if join predicate of nested subquery uses **inequality**

Example: Problem with inequality in join predicates of nested subquery

What is the result of the following query?

```
SELECT B.bid
  FROM Boats B
 WHERE B.qty =
  (SELECT MAX(R.qty)
     FROM Reserves R
    WHERE R.bid < B.bid AND
          R.day <= 31-12-2010)
```

Result

bid
8

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	0
10	Finn	Konstanz	4
8	49erFX	Kreuzlingen	4

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	9	5	03-02-2009

Nested Subqueries

Example: Problem with inequality in join predicates of nested subquery (cont'd)

Step 1 of algorithm NEST-JA creates the following temporary table.

```
CREATE TABLE TMP5 AS
  SELECT bid,
         MAX(qty) AS maxqty
    FROM Reserves
   WHERE day <= 31-12-2010
   GROUP BY bid
```

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	0
10	Finn	Konstanz	4
8	49erFX	Kreuzlingen	4

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	9	5	03-02-2009

Step 2 of algorithm NEST-JA followed by application of algorithm NEST-N-J rewrites the query as follows.

```
SELECT B.bid
  FROM Boats B, TMP5 T
 WHERE B.qty = T.maxqty AND
       T.bid < B.bid
```

TMP5

bid	maxqty
3	4
10	1
9	5

Result

bid
10
8

Nested Subqueries

- Reason for this problem
  - **`GROUP BY`** calculates the **`MAX`** for each distinct value of the join column of temporary relation (**`bid`**)
  - **`MAX`** should be computed for a **set of join column values** in inner relation that are all smaller than a given join column value in outer relation
- Solution
  - **push** join predicate into computation of temporary relation, i.e., switch the evaluation order of join and group-by/aggregate
  - join is now performed **before** group-by/aggregate, causing temporary table to include aggregate value over proper ranges of join column values
  - as before, join predicate in outer query needs to be **changed to equality**

Nested Subqueries

Problem with inequality in join predicates of nested subquery

Perform inequality join **before** group-by/aggregate.

```
CREATE TABLE TMP6 AS
  SELECT B.bid,
         MAX(R.qty) AS maxqty
    FROM Boats B, Reserves R
   WHERE R.day <= 31-12-2010
     AND R.bid < B.bid
   GROUP BY bid
```

Boats

bid	bname	harbor	qty
3	Laser	Konstanz	0
10	Finn	Konstanz	4
8	49erFX	Kreuzlingen	4

Reserves

sid	bid	qty	day
2	3	4	07-03-2009
1	3	2	10-01-2008
1	10	1	06-08-2008
3	9	5	03-02-2009

Transform query to use temporary table in **equality** join.

```
SELECT B.bid
  FROM Boats B, TMP6 T
 WHERE B.qty = T.maxqty
   AND B.bid = T.bid
```

TMP6

bid	maxqty
10	5
8	4

Result

bid
8

Nested Subqueries

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Modified algorithm **NEST-JA2** for Type JA nested subqueries
  1. project join column of R_1 and restrict it with any simple predicates applying to R_1
  2. create temporary relation R_{tmp} by joining R_1 and R_2 using same operator as join predicate in original query
     - if aggregate function is **`COUNT`**, join must be outer join
     - if aggregate function is **`COUNT(*)`**, compute aggregate over join column
     - include join column(s) and aggregated column in **`SELECT`** clause
     - include join column(s) in **`GROUP BY`** clause
  3. join R_1 to R_{tmp} according to transformed version of original query by changing join predicate to equality (=)
- Result of algorithm NEST-JA2 is again a Type J nested query

Nested Subqueries

- So far, only nested predicates with **scalar** and **set inclusion** (**`IN`**) operators where considered
- SQL supports additional **set comparisons**
  - **`EXISTS`** and **`NOT EXISTS`**
  - **`ANY`** and **`ALL`**
- Extensions to transformation algorithms are required to support these comparisons
  - rewrite predicates using general set comparisons to predicates containing scalar or set inclusion operators only
  - process rewritten query using same algorithms as before

Nested Subqueries

EXISTS and NOT EXISTS

```
… WHERE EXISTS
    (SELECT A
       FROM R
      WHERE B = C)
```



```
… WHERE 0 <
    (SELECT COUNT(A)
       FROM R
      WHERE B = C)
```

```
… WHERE NOT EXISTS
    (SELECT A
       FROM R
      WHERE B = C)
```



```
… WHERE 0 =
    (SELECT COUNT(A)
       FEOM R
      WHERE B = C)
```

Nested Subqueries

ANY and ALL

```
... WHERE const < ANY
      (SELECT A
         FROM R
        WHERE B = C)
```



```
... WHERE const <
      (SELECT MAX(A)
         FROM R
        WHERE B = C)
```

And ...**> ANY (SELECT A**... is transformed to ...**> SELECT MIN(A)**..., for reasons of symmetry.

```
... WHERE const < ALL
      (SELECT A
         FROM R
        WHERE B = C)
```



```
... WHERE const <
      (SELECT MIN(A)
         FEOM R
        WHERE B = C)
```

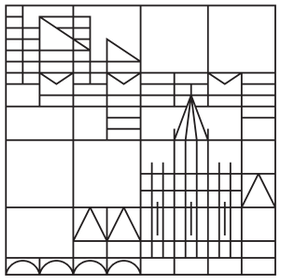
And ...**> ALL (SELECT A**... is transformed to ...**> SELECT MAX(A)**..., for reasons of symmetry.

Finally, ...**= ANY**... is transformed to ...**IN**..., whereas ...**<> ANY**... is transformed to ...**NOT IN**....

Nested Subqueries

Recursive algorithm to process a general nested query

```
function nest-g(outer)                                   (outer: outer query block)
  foreach p ∈ outer.WHERE do                             (p: predicate)
    if p.inner ≠ ∅ then                                  (p contains an inner query block, i.e., p is nested)
      nest-g(p.inner) ;
      if p.inner.SELECT contains aggregation function then
        if p.inner.WHERE is correlated then              (Type JA nesting)
          nest-ja2(p.inner) ;
          nest-n-j(outer, p.inner) ;
        else                                             (Type A nesting)
          nest-a(p.inner) ;
      else                                               (Type N or Type J nesting)
        nest-n-j(outer, p.inner) ;
end
```



TO BE CONTINUED...