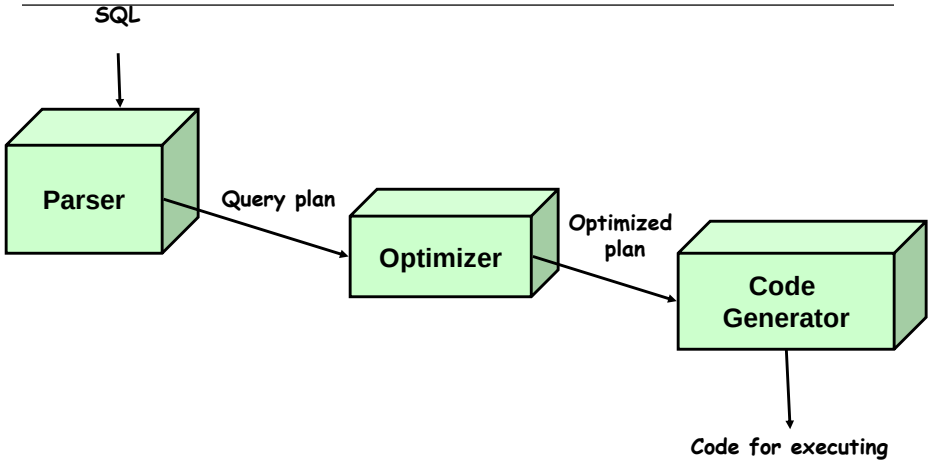
Query Processing

Vu Tuyet Trinh
trinhvt@soict.hust.edu.vn

Department of Information Systems
SoICT-HUST

Processus



Outline

- Standard relational operators
- Query plans & exec strategies
- Query Optimization

Relational Algebra

- Set of operations on relations
- Basic of the implementation of most current database systems.

- Operation
  - Input: one or two relation
  - Output: a relation

- Relational algebra expression or algebraic expression
  - a sequence of relational algebra operations

Relational Algebra Operations

- Projection
- Selection
- Join
- Division

- Union
- Intersection
- Difference
- Cartesian product

5

Example

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

x

Sport

SportID	Sport
05	Swimming
09	Dancing

Student_Sport

Id	Name	Suburb	SportID	Sport
1108	Robert	Kew	05	Swimming
3936	Glen	Bundoora	05	Swimming
8507	Norman	Bundoora	05	Swimming
8452	Mary	Balwyn	05	Swimming
1108	Robert	Kew	09	Dancing
3936	Glen	Bundoora	09	Dancing
8507	Norman	Bundoora	09	Dancing
8452	Mary	Balwyn	09	Dancing

Projection

- Choose some attributes.

$$\prod_{A1,A2,\ldots}(R)$$

- Example

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

$$\prod_{name}(Student)$$

Result

Name
Robert
Glen
Norman
Mary

7

Selection

- choose from R each tuple where the condition holds.

$$\sigma_{<condition>}(R)$$

- Example

$$\sigma_{suburb="Bundoora}(Student)$$

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

Result

Id	Name	Suburb
3936	Glen	Bundoora
8507	Norman	Bundoora

8

Example

$$\prod_{name}(\sigma_{suburb="Bundoora}Student)$$

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

Result

Name
Glen
Norman

Join

- Combine attributes from 2 tables

$$R_1 \rhd\lhd_{<join\_condition>} R_2$$

- Example

$$Student \rhd\lhd_{Id=SID} Enrol$$

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

Enrol

SID	Course
3936	101
1108	113
8507	101

Result

SID	Id	Name	Suburb	Course
1108	1108	Robert	Kew	113
3936	3936	Glen	Bundoora	101
8507	8507	Norman	Bundoora	101

Example

$$\prod_{name,Course}(\sigma_{suburb="Bundoo}(Student \triangleright\!\triangleleft_{Id=SID} Enrol))$$

Student

Id	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

Enrol

SID	Course
3936	101
1108	113
8507	101

Result

Name	Course
Glen	101
Norman	101

11

Natural Join

- Special join operation with equal join condition on their common attributes
- Example

Takes

SID	SNO
1108	21
1108	23
8507	23
8507	29

*

Enrol

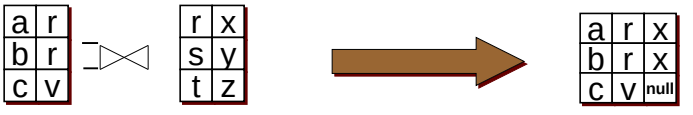
SID	Course
3936	101
1108	113
8507	101

SID	SNO	Course
1108	21	113
1108	23	113
8507	23	101
8507	29	101

12

Outer-Join

- Left Outer Join



- Right Outer Join

13

Example

Student

ID	Name	Suburb
1108	Robert	Kew
3936	Glen	Bundoora
8507	Norman	Bundoora
8452	Mary	Balwyn

Enrol

SID	Course
3936	101
1108	113
8507	101

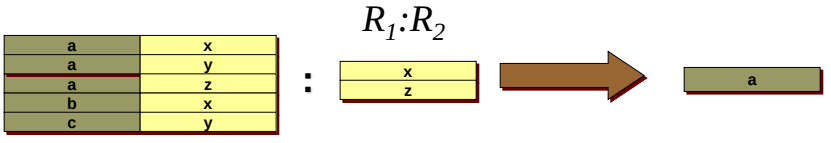
Result

ID	Name	Suburb	Course
1108	Robert	Kew	113
3936	Glen	Bundoora	101
8507	Norman	Bundoora	101
8452	Mary	Balwyn	null

14

Division

- The division operator divides a dividend relation R1 or degree m+n by a divisor relation R2 of degree n, and produces a quotient relation of degree m.



- Example

Subject

Name	Course
Systems	BCS
Database	BCS
Database	MCS
Algebra	MCS

:

Course

Course
BCS
MCS

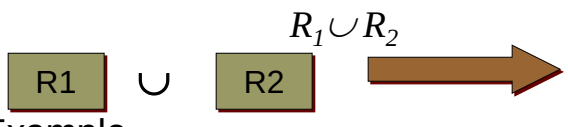
Results

Name
Database

15

Union

- combining the tuples from two input union-compatible relations (having the same set of attributes)



- Example

Subject

Name	Course
Systems	BCS
Database	BCS
Database	MCS
Algebra	MCS

∪

Subject2

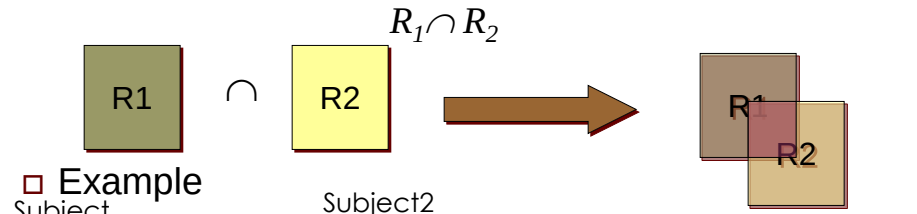
Name	Course
DataMining	MCS
Writing	BCS

Result

Name	Course
Systems	BCS
Database	BCS
Database	MCS
Algebra	MCS
DataMining	MCS
Writing	BCS

Intersection

- Keeping only common tuples from 2 input union-compatible relation



- Example

Subject

Name	Course
Systems	BCS
Database	BCS
Database	MCS
Algebra	MCS

∩

Subject2

Name	Course
DataMining	MCS
Database	MCS
Systems	BCS
Writing	BCS

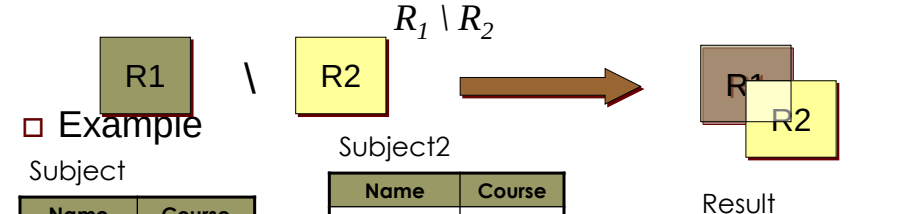
Kết quả

Name	Course
Systems	BCS
Database	MCS

17

Difference

- Difference of two union compatible relations is a relation containing tuples occurred in the first relation but not in the second.



- Example

Subject

Name	Course
Systems	BCS
Database	BCS
Database	MCS
Algebra	MCS

\

Subject2

Name	Course
DataMining	MCS
Database	MCS
Systems	BCS
Writing	BCS

Result

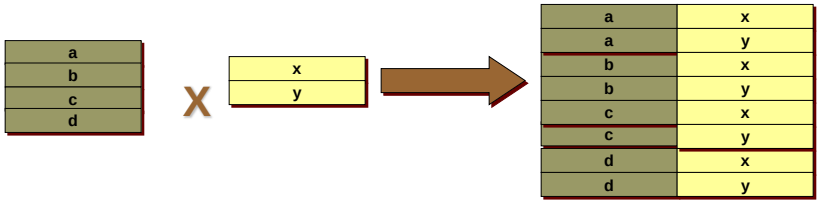
Name	Course
Database	BCS
Algebra	MCS

18

Cartesian Product

- the concatenation of every tuple of one relation with every tuple of the other relation.

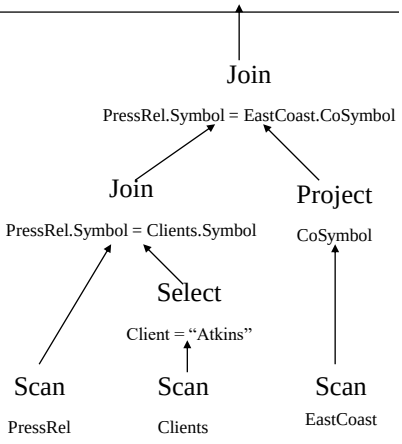
$$R_1 \times R_2$$



19

Query Plans

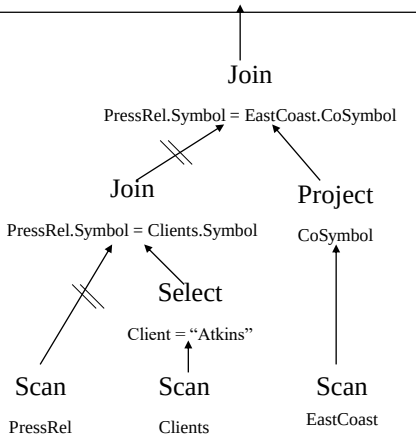
- Data-flow graph of relational algebra operators
- *Typically:* determined by optimizer




```
SELECT *
FROM PressRel p, Clients C
WHERE p.Symbol = c.Symbol
 AND c.Client = 'Atkins'
 AND c.Symbol IN
(SELECT CoSymbol FROM EastCoast)
```

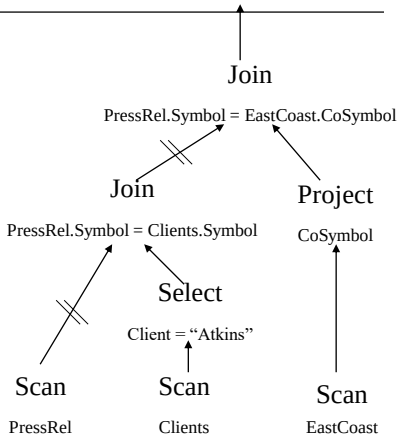
Execution Strategy Issues

- Granularity & parallelism:
  - Pipelining vs. blocking
  - Materialization



Iterator-Based Query Execution

- Execution begins at root
  - *open, next, close*
  - Propagate calls to children
    May call multiple child *next*s
- ✓ Efficient scheduling & resource usage



Basic Principles

- Many DB operations require reading tuples, tuple vs. previous tuples, or tuples vs. tuples in another table
- Techniques generally used:
  - *Iteration*: for/while loop comparing with all tuples on disk
  - *Index*: if comparison of attribute that's indexed, look up matches in index & return those
  - *Sort*: iteration against presorted data (*interesting orders*)
  - *Hash*: build hash table of the tuple list, *probe* the hash table
- *Must be able to support larger-than-memory data*

Basic Operators

- One-pass operators:
  - Scan
  - Select
  - Project
- Multi-pass operators:
  - Join
    - Various implementations
    - Handling of larger-than-memory sources
  - Semi-join
  - Aggregation, union, etc.

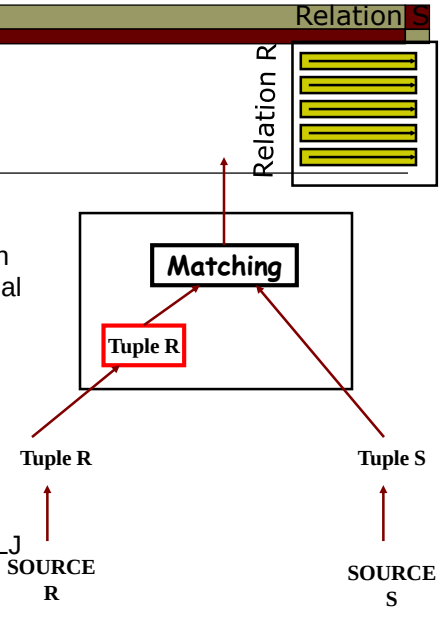
1-Pass Operators: Scanning a Table

- Sequential scan: read through blocks of table
- Index scan: retrieve tuples in index order

- Cost in page reads -- b(T) blocks, r(T) tuples
  - b(T) pages for sequential scan
  - Up to r(T) for index scan if unclustered index
  - Requires memory for one block

Multi-pass Operators: Nested-Loops Join

- Principle
  - reading the external relation R & iterating over the internal relation S
  - one-and-haft pass, non-blocking
- Variations
  - Tuple-based NLJ, block-based NLJ, index-based NLJ



Two-Pass Algorithms

Sort-based

Need to do a multiway sort first (or have an index)

Approximately linear in practice, 2 b(T) for table T

Hash-based

Store one relation in a hash table

(Sort-)Merge Join

- Requires data sorted by join attributes

```
Merge and join sorted files, reading sequentially a
  block at a time
```

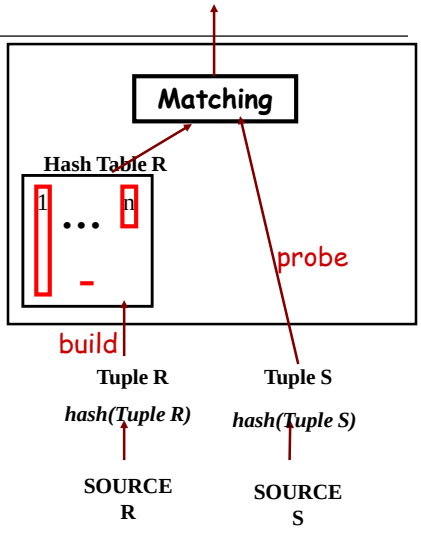
  - Maintain two file pointers
    - While tuple at R < tuple at S, advance R (and vice versa)
    - While tuples match, output all possible pairings
  - Preserves sorted order of "outer" relation
- ✓ Very efficient for presorted data
- ✓ Can be "hybridized" with NL Join for range joins
- × May require a sort before (adds cost + delay)
- Cost: b(R) + b(S) plus sort costs, if necessary
  In practice, approximately linear, 3 (b(R) + b(S))

Hash-Based Joins

- Allows partial pipelining of operations with equality comparisons
- Sort-based operations block, but allow range and inequality comparisons
- Hash joins usually done with static number of hash buckets
  - Generally have fairly long chains at each bucket
  - What happens when memory is too small?

Hash Join (HJ)

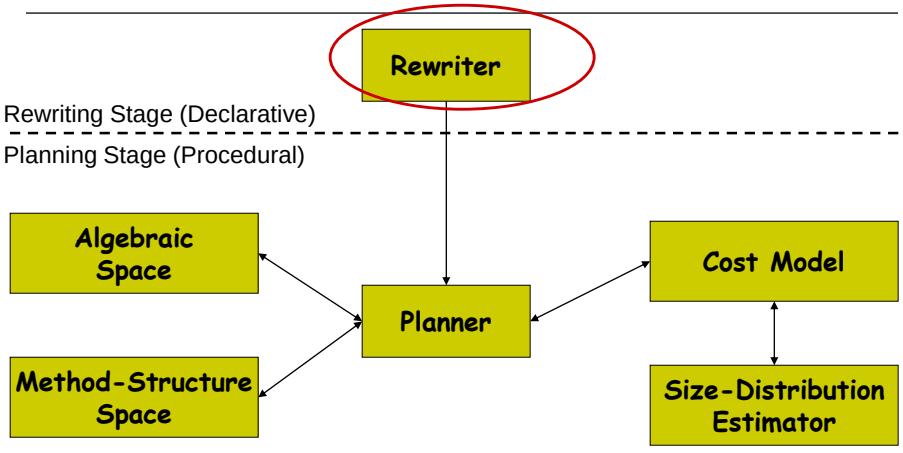
- Principle
  - using a hash table (HT) of R
  - two-pass, blocking algorithm
- Process
  - build the HT of R
  - probe every tuple of S with corresponding tuples in the HT of R



Overview of Query Optimization

- *Plan*: *Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan.
  Practically: Avoid worst plans!

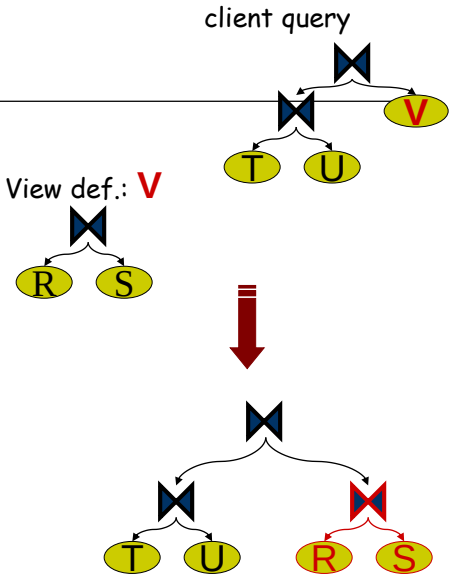
Query optimization [Ioa00]



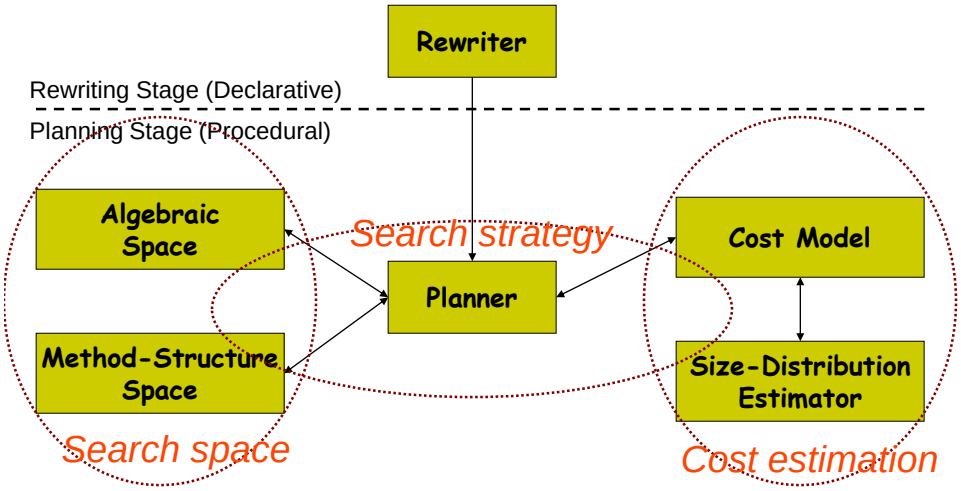
Rewriting

- Principle [Lev01]
  - reformulating a client query to use available views

  →replacing views by their definition



Query optimization [Ioa00]



Search Space

- Algebraic space
  - operators execution orders that are to be considered by the Planner for each query sent to it
- Method-structure space
  - implementation choices that exist for the execution of each specified ordered series of operators

Relational Algebra Equivalences

- Allowing to choose different join orders and to `push' selections and projections ahead of joins.
- *Selections*: $\sigma_{c1\wedge\ldots\wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ (*Cascade*)

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ *(Commute)*
- *Projections*: $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ *(Cascade)*
- *Joins*: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(Associative)*

  $(R \bowtie S) \equiv (S \bowtie R)$ *(Commute)*

  ➢ Show that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with join (i.e., σ (R $\bowtie$ S) $\equiv$ σ(R) $\bowtie$ S)
- Similarly, if a projection follows a join R $\bowtie$ S, we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

Enumeration of Alternative Plans

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

Cost estimation

- Cost factors
  - CPU cost + I/O cost + Communication cost
  - Size of (intermediate) relations
- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
  - Must also estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

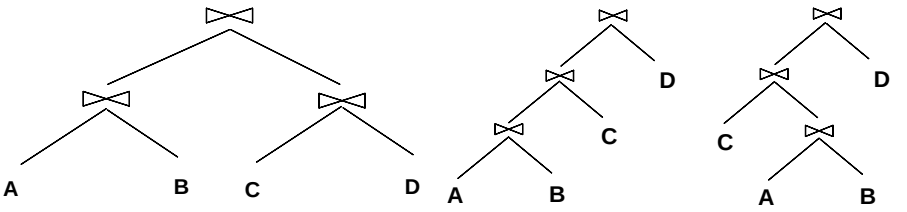
Search strategy

- Objective
  - exploring the set of alternative execution plans and finding the "*cheapest*" one
- Taxonomy
  - Polynomial vs. Combinatorial
  - Heuristics vs. Systematic
  - Deterministic vs. Randomized
  - Transformative vs. Constructive

orthogonal but related

Queries Over Multiple Relations

- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).

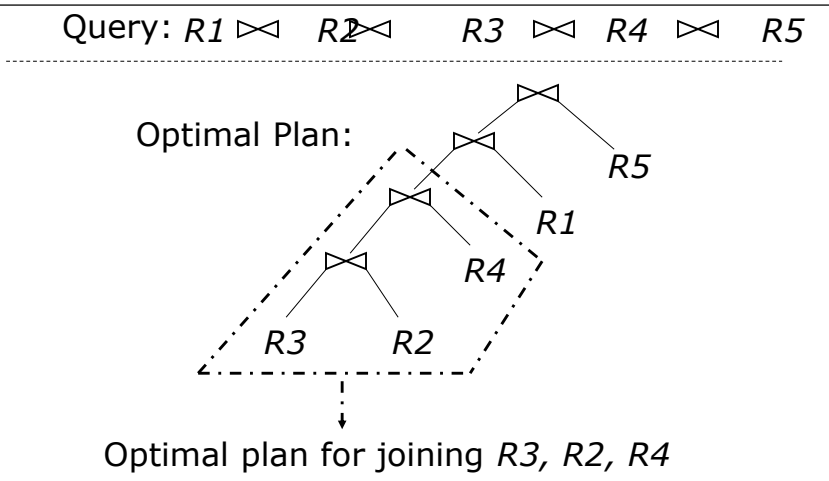


Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.

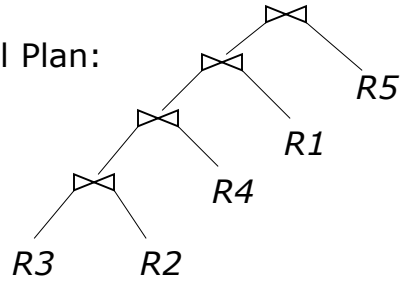
Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an addional sorting operator.
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up.
  - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables.



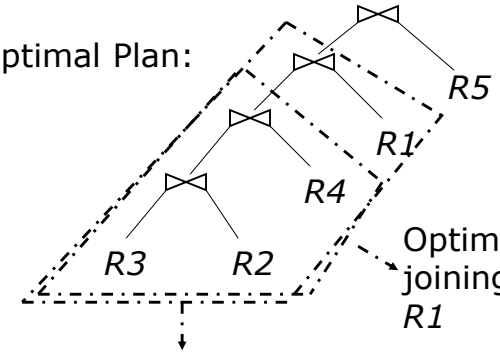
Query: *R1* ⋈ *R2* ⋈ *R3* ⋈ *R4* ⋈ *R5*

Optimal Plan:



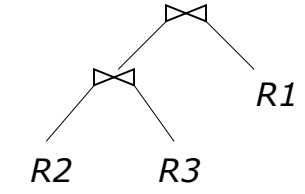
Query: *R1* ⋈ *R2* ⋈ *R3* ⋈ *R4* ⋈ *R5*

Optimal Plan:

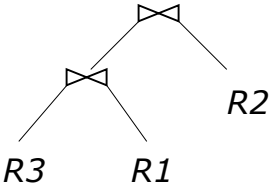


Optimal plan for joining *R3, R2, R4, R1*

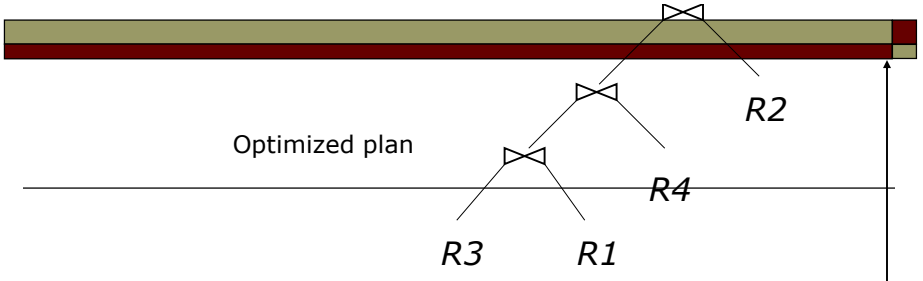
Optimal plan for joining *R3, R2, R4*

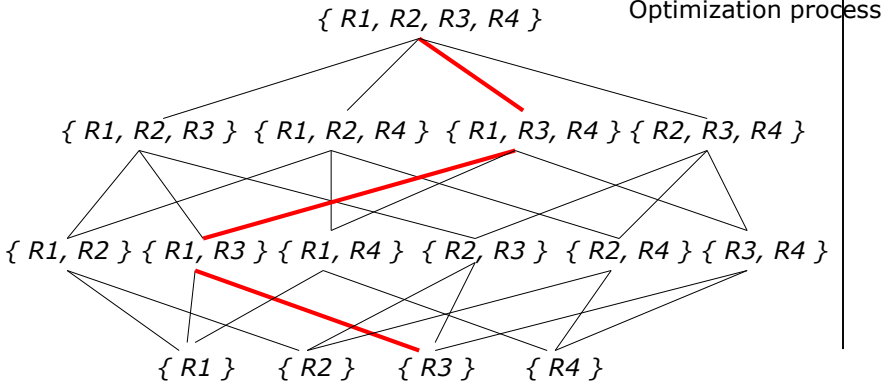


Optimal
for joining *R1, R2, R3*



Sub-Optimal
for joining *R1, R2, R3*





Performance issues

- Disk access
- Index utilization
- Database schema
- Nested queries, nested transaction
- …

Example

- Employee(**ssnum**, name, manager, dept, salary, numfriends)
  - Clustering index : ssnum
  - Non clustering indexes (i) name (ii) dept (iii) salary
- Student(**ssnum**, name, degree_sought, year)
  - Clustering index :ssnum
  - Non clustering index :name
- Tech(**dept**, manager, location)
  - Clustering index : dept

Rewriting queries

- Using Index
- Eliminating DISTINCTs
- Nested queries
- Join queries
- Having clause
- Using view
- Using materialized views

Using index

- Unenabling the use of index
  - Math expression

    WHERE salary/12 >= 4000;
  - String function

    SELECT * FROM employee
    WHERE SUBSTR(name, 1, 1) = ‘G’;
  - Comparison with different data type
  - Comparison with null value

DISTINCTs

- Example

```
SELECT distinct ssnum
FROM employee
WHERE dept = ‘information systems’
```

- Necessary ?

- Example

```
SELECT DISTINCT ssnum
FROM employee, tech
WHERE employee.dept = tech.dept
```

- DISTINCT or not?

Employee(**ssnum**, name, manager, dept, salary, numfriends)
Student(**ssnum**, name, degree_sought, year)
Tech(**dept**, manager, location)

- SELECT ssnum
  FROM employee, tech
  WHERE employee.manager = tech.manager

- SELECT ssnum, tech.dept
  FROM employee, tech
  WHERE employee.manager = tech.manager

- SELECT student.ssnum
  FROM student, employee, tech
  WHERE student.ssnum = employee.ssnum
  AND employee.dept = tech.dept;

55

Nested queries

- SELECT ssnum FROM employee
  WHERE salary > (select avg(salary)
  from employee)

- SELECT ssnum FROM employee
  WHERE dept in (select dept from tech)

- SELECT ssnum
  FROM employee e1
  WHERE salary =
  (SELECT avg(e2.salary)
  FROM employee e2,tech
  WHERE e2.dept = e1.dept AND
  e2.dept = tech.dept)

Nested queries

SELECT ssnum
FROM employee
WHERE dept in (select dept
from tech)

➔

SELECT ssnum
FROM employee, tech
WHERE employee.dept = tech.dept

Nested queries with aggregation function

- Example
  - SELECT avg(salary)
    FROM employee
    WHERE manager in (select manager from tech)
  - SELECT avg(salary)
    FROM employee, tech
    WHERE employee.manager = tech.manager
- Any difference ?

Nested queries vs. join queries

- Example

```
SELECT ssnum
FROM employee e1
WHERE salary = (SELECT avg(e2.salary
                FROM employee e2, tech
                WHERE e2.dept = e1.dept
                      and e2.dept = tech.dept);
```

➔

- Create view temp as
  SELECT avg(salary) as avsalary, employee.dept
  FROM employee, tech
  WHERE employee.dept = tech.dept
  GROUP BY employee.dept;

- SELECT ssnum
  FROM employee, temp
  WHERE salary = avsalary
  AND employee.dept = temp.dept

Employee(**ssnum**, name, manager, dept, salary, numfriends)
Student(**ssnum**, name, degree_sought, year)
Tech(**dept**, manager, location)

```
SELECT ssnum
FROM employee e1
WHERE numfriends = COUNT(SELECT e2.ssnum
                         FROM employee e2, tech
                         WHERE e2.dept = tech.dept
                               AND e2.dept = e1.dept);
```

➔

- INSERT INTO temp
  SELECT COUNT(ssnum) as numcolleagues,
  employee.dept
  FROM employee, tech
  WHERE employee.dept = tech.dept
  GROUP BY employee.dept;

- SELECT ssnum
  FROM employee, temp
  WHERE numfriends = numcolleagues
  AND employee.dept = temp.dept;

Join condition

- Join over clustering indexes.

- Join expression vs. data type (numeric, string,…)

Using view

- CREATE VIEW techlocation
  AS
  SELECT ssnum, tech.dept,
  location
  FROM employee, tech
  WHERE
  employee.dept =
  tech.dept;
- SELECT location
  FROM techlocation
  WHERE ssnum = 43253265;

- SELECT location
  FROM employee, tech
  WHERE
  employee.dept =
  tech.dept
  AND ssnum =
  43253265;

Materialized view

- Materialized views:

```
CREATE MATERIALIZED VIEW VendorOutstanding
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT orders.vendor, sum(orders.quantity*item.price)
FROM orders,item
WHERE orders.itemnum = item.itemnum
group by orders.vendor;
```

  - Parameters
    - BUILD immediate/deferred
    - REFRESH complete/fast
    - ENABLE QUERY REWRITE
  - Advantages
    - Automatically update
    - Used by optimizer